Спецификаторы, квалификаторы и шаблоны

Михаил Матросов

Align Technology

Prefs | Filter | 1h 8m | Mark all comments as read | Blame | Keyboard shortcuts

/Common/.../ForceDesignerAlgorithms/FaceSplittingTest.cpp Added **33 — 4 hidden** FishEye

a09df82 d56a745 b75f240 5cdb834 469b8d7 16f1e39 461b493 9ccbec2

Leave Unread

Mikhail Matrosov

Make them static constexpr.

DEFECT Classification: Improvement desirable Ranking: Minor

Reply · Edit · Delete · Add to favourites · Create issue · 17 Sep

Turmets Makoev marked as RESOLVED 20 Sep Reopen

Viktor Sazhaev

Why **static constexpr**? Just **constexpr** without any **static**. Compile-time objects don't need storage class.

Reply · Mark as unread · Mark as needs resolution · Create issue · 18 Sep

Mikhail Matrosov

This is a little mind-blowing, I understand, but these are orthogonal. **constexpr** is related to compile time, **static** is related to runtime. Historically, static integral variables are known at compile time, thus you can use **static** without **constexpr** (but not vice versa) to achieve the same behavior as for **static constexpr**. But the latter better expresses your intent: **constexpr** to highlight the fact the value is known at compile time, **static** to highlight the fact single value is shared for all objects of the class.

> Compile-time objects don't need storage class

They do, since they are also available at runtime.

See also https://stackoverflow.com/a/41125798/261217

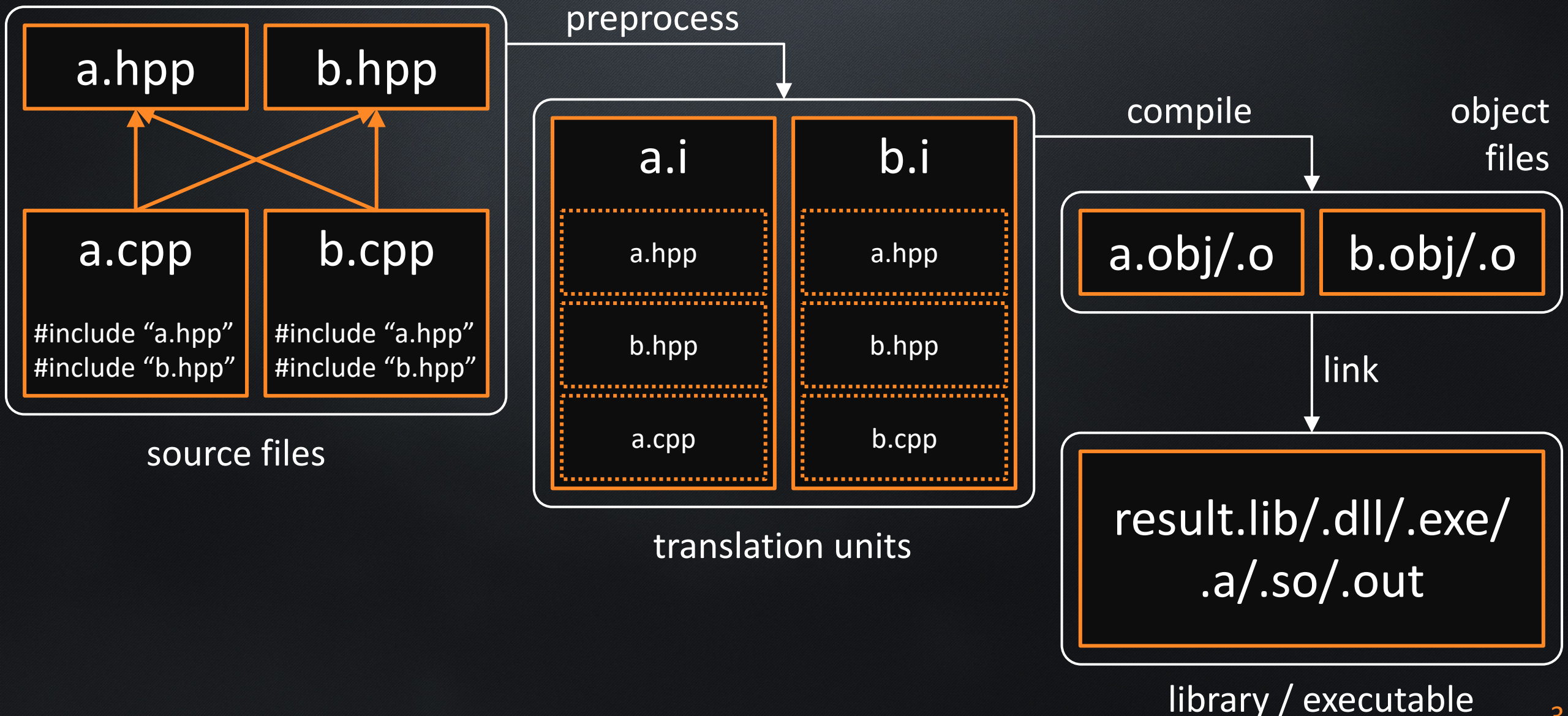
Reply · Edit · Delete · Mark as needs resolution · Create issue · 18 Sep

Viktor Sazhaev

We can use both words. They are formally orthogonal. Though **static** makes little sense with **constexpr**. Visibility can be different but storage class is always "the most global".
Usually I also separate such declarations from classes. But for conceptually static members that may be okay.

Reply · Mark as unread · Mark as needs resolution · Create issue · 24 Sep

Building a C/C++ program



Объявление и определение

```
// Function declaration
int sqr(int x);

// Function definition
int sqr(int x) { return x * x; }

// Variable definition
int n;

// Variable declaration
extern int n;
```

- Сколько угодно объявлений
- Ровно одно определение
  - В некоторых случаях допускается множество определений, но они все должны быть одинаковыми

Linkage

a.cpp

```
int sqr(int x) {
  return x * x;
}
```

external linkage

a’.cpp

```
static int sqr(int x) {
  return x * x;
}
```

internal linkage

b.cpp

```
int sqr(int x);

bool check(int a, int b, int c) {
  return sqr(a) + sqr(b) == sqr(c);
}
```

⇒ link time error: sqr not found in b.obj

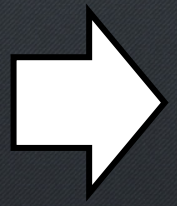
Linkage

	can be referred to from
external linkage	All translation units
internal linkage	Current translation unit
no linkage	Current scope

Storage duration

```
bool beetlejuice()
{
  int counter = 0;
  return ++counter >= 3;
}
```

automatic storage duration



```
bool beetlejuice()
{
  static int counter = 0;
  return ++counter >= 3;
}
```

static storage duration

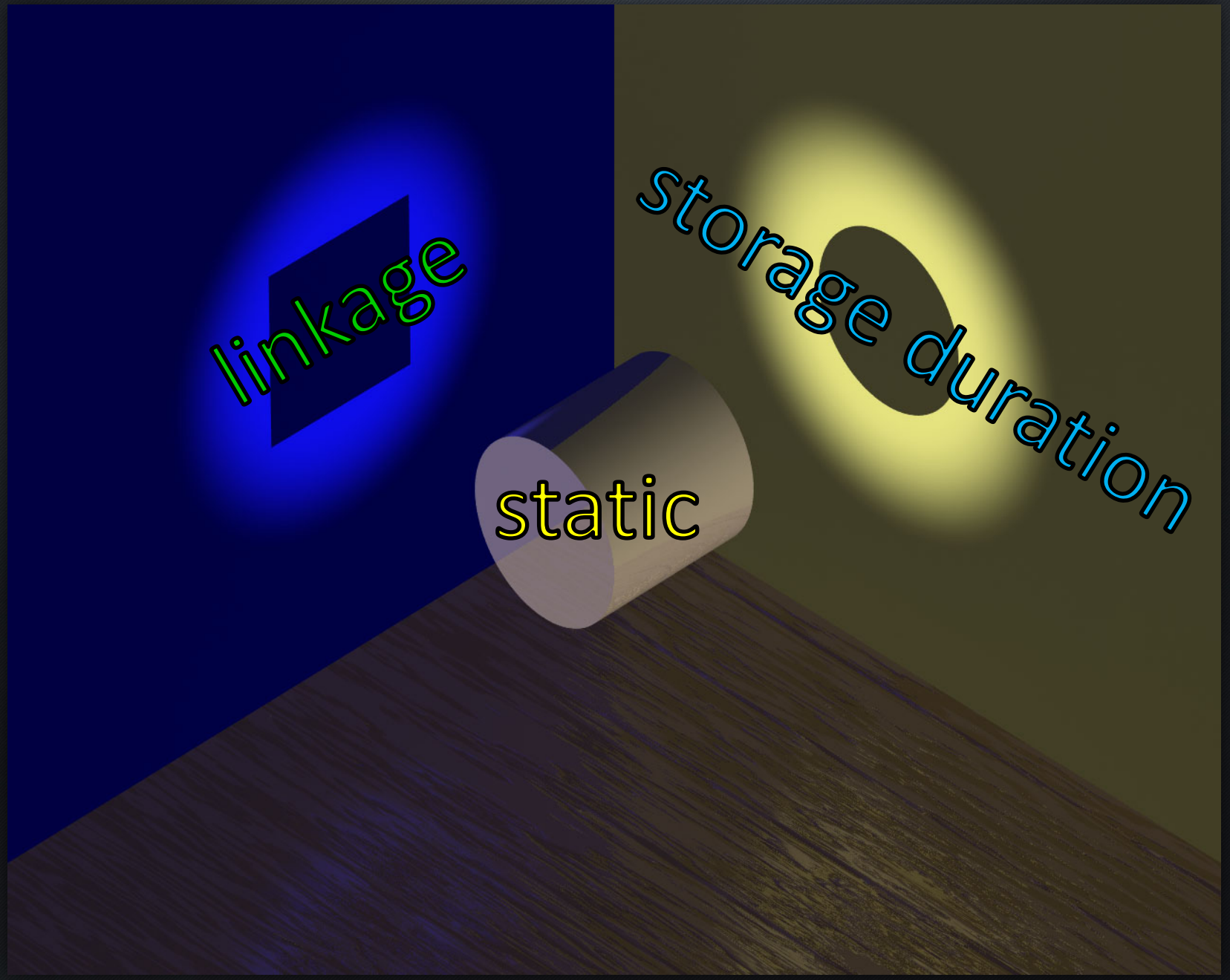
Storage duration

	storage allocated/deallocated
automatic	When entering/leaving the scope or when creating/destroying an object of a class
static	When the program begins/ends
thread	When the thread begins/ends
dynamic	When new/delete is called

- Applicable only to objects
- Time of initialization/destruction is more complicated
- Storage class specifiers: static, extern, thread_local, mutable

Что? Где? Когда?

- Что? – объект
- Где? – linkage
- Когда? – storage duration


linkage
storage duration
static

common.hpp

```
const double thickness = 0.65;
const char* name = "tooth";
```

internal linkage
external linkage

a.cpp

```
#include "common.hpp"
```

b.cpp

```
#include "common.hpp"
```

⇒ link time error: multiple definitions of symbol “name”

Any of the following names declared at namespace scope have internal linkage:

- *non-volatile non-template non-inline const-qualified variables (including constexpr) that aren't declared extern and aren't previously declared to have external linkage;*

common.hpp

```
constexpr double thickness = 0.65;
const std::string name = "tooth";
```

internal linkage
internal linkage

a.cpp

```
#include "common.hpp"
```

b.cpp

```
#include "common.hpp"
```

A constexpr specifier used in an object declaration implies const.

Any of the following names declared at namespace scope have internal linkage:

- *non-volatile non-template non-inline const-qualified variables (including constexpr) that aren't declared extern and aren't previously declared to have external linkage;*

common.hpp

```
const std::string name = "tooth";
const char* getName();
```

internal linkage

a.cpp

```
#include "common.hpp"
#include <iostream>

bool dumbCmp(const char* s1, const char* s2) {
  return s1 == s2;
}

int main() {
  std::cout << std::boolalpha
            << dumbCmp(name.data(), getName());
}
```

b.cpp

```
#include "common.hpp"

const char* getName() {
  return name.data();
}
```

⇒ false

common.hpp

```cpp
inline const std::string name = "tooth";
const char* getName();
```

external (weak) linkage

a.cpp

```cpp
#include "common.hpp"
#include <iostream>

bool dumbCmp(const char* s1, const char* s2) {
  return s1 == s2;
}
```

b.cpp

```cpp
#include "common.hpp"

const char* getName() {
  return name.data();
}
```

Any of the following names declared at namespace scope have internal linkage:

- *non-volatile non-template non-inline const-qualified variables (including constexpr) that aren't declared extern and aren't previously declared to have external linkage;*

common.hpp

```
int sqr(int x) {
  return x * x;
}
```

external linkage

a.cpp

```
#include "common.hpp"
```

b.cpp

```
#include "common.hpp"
```

⇒ link time error: multiple definitions of symbol “sqr”

common.hpp

```
constexpr int sqr(int x) {
  return x * x;
}
```

external (weak) linkage

a.cpp

```
#include "common.hpp"
```

b.cpp

```
#include "common.hpp"
```

A constexpr specifier used in a function or static member variable declaration implies inline.

main.cpp

```cpp
void other();


struct Local {
  static void foo() {
    std::cout << "main ";
  }
};


int main() {
  Local::foo();
  other();
}
```

external linkage

other.cpp

```cpp

struct Local {
  static void foo() {
    std::cout << "other ";
  }
};


void other() {
  Local::foo();
}
```

external linkage

⇒ main main

Every program shall contain exactly one definition of every non-inline function or variable that is odr-used in that program outside of a discarded statement (9.4.1); no diagnostic required.

main.cpp

```cpp
void other();

namespace {
  struct Local {
    static void foo() {
      std::cout << "main ";
    }
  };
}

int main() {
  Local::foo();
  other();
}
```

internal linkage

other.cpp

```cpp
namespace {
  struct Local {
    static void foo() {
      std::cout << "other ";
    }
  };
}

void other() {
  Local::foo();
}
```

internal linkage

⇒ main other

In addition, all names declared in unnamed namespace or a namespace within an unnamed namespace, even ones explicitly declared extern, have internal linkage.

Собираем в кучу

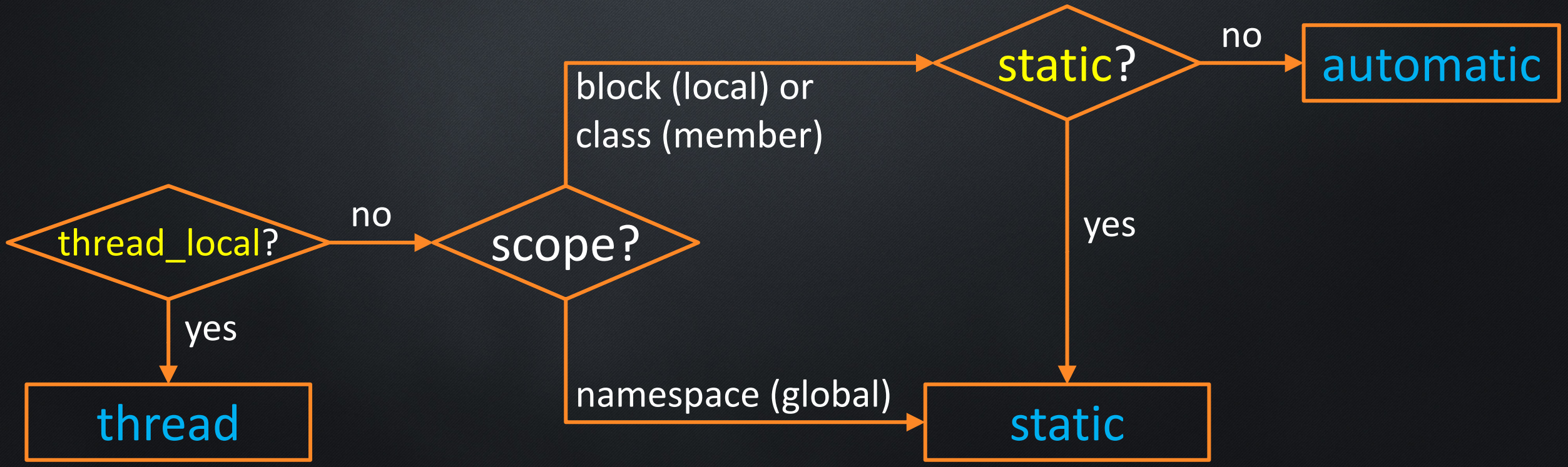
Цвета имеют значение

- static keyword
- static storage duration
- internal linkage

Допустимые комбинации storage duration и linkage

Linkage / Storage duration	no linkage	internal linkage	external linkage
automatic	local variable		
static	static local variable	static global variable	global variable
thread	thread_local local variable	thread_local static global variable	thread_local global variable
dynamic			

How to identify storage duration of an object?



Properties of entities with static and thread storage duration in C++17

Apply in order

#	Property \ Entity	local variable	global variable	member variable	global function	member function
1	no linkage					
2	external linkage					
3	constexpr ⇒ const					
4	constexpr ⇒ inline					
5	const ⇒* internal linkage				N/A	
6	inline ⇒ external (weak) linkage	N/A				
7	static ⇒ internal linkage	Required		Required		Required
8	anonymous (unnamed) namespace ⇒ internal linkage					

* if not volatile and not template


static
inline
const
namespace{};

А как же extern?..

common.hpp

```
const std::string name = "tooth";
const char* getName();
```

internal linkage

a.cpp

```
#include "common.hpp"
#include <iostream>



bool dumbCmp(const char* s1, const char* s2) {
  return s1 == s2;
}

int main() {
  std::cout << std::boolalpha
            << dumbCmp(name.data(), getName());
}
```

b.cpp

```
#include "common.hpp"

const char* getName() {
  return name.data();
}
```

⇒ false

common.hpp

```
extern const std::string name;
const char* getName();
```

external linkage (declaration)

a.cpp

```
#include "common.hpp"
#include <iostream>

const std::string name = "tooth";

bool dumbCmp(const char* s1, const char* s2) {
  return s1 == s2;
}

int main() {
  std::cout << std::boolalpha
            << dumbCmp(name.data(), getName());
}
```

b.cpp

```
#include "common.hpp"

const char* getName() {
  return name.data();
}
```

⇒ true

extern

- Свойства:
  - Применим только к глобальным функциям и переменным
  - Не совместим со static
  - Не имеет смысла с constexpr и с inline
  - Значение не видно в точке объявления (обычно недостаток)
- Недостатки:
  - Необходимо вручную сделать определение
- Достоинства:
  - Может позволить оптимизировать время сборки
- Рекомендации:
  - Вместо него лучше использовать inline

Properties of entities with static and thread storage duration in C++17

Apply in order

#	Property \ Entity	local variable	global variable	member variable	global function	member function
1	no linkage					
2	external linkage					
3	constexpr ⇒ const					
4	constexpr ⇒ inline					
5	const ⇒* internal linkage				N/A	
6	inline ⇒ external (weak) linkage	N/A				
7	static ⇒ internal linkage	Required		Required		Required
8	extern ⇒ external linkage (declaration)	N/A		N/A	Redundant	N/A
9	anonymous (unnamed) namespace ⇒ internal linkage					

* if not volatile, not template, and not previously declared extern

Practice time!

```
struct A
{
  double x1;
  static double x2;
  static const double x3;
  static inline const double x4 = 4.0;
  static constexpr double x5 = 5.0;
};
```

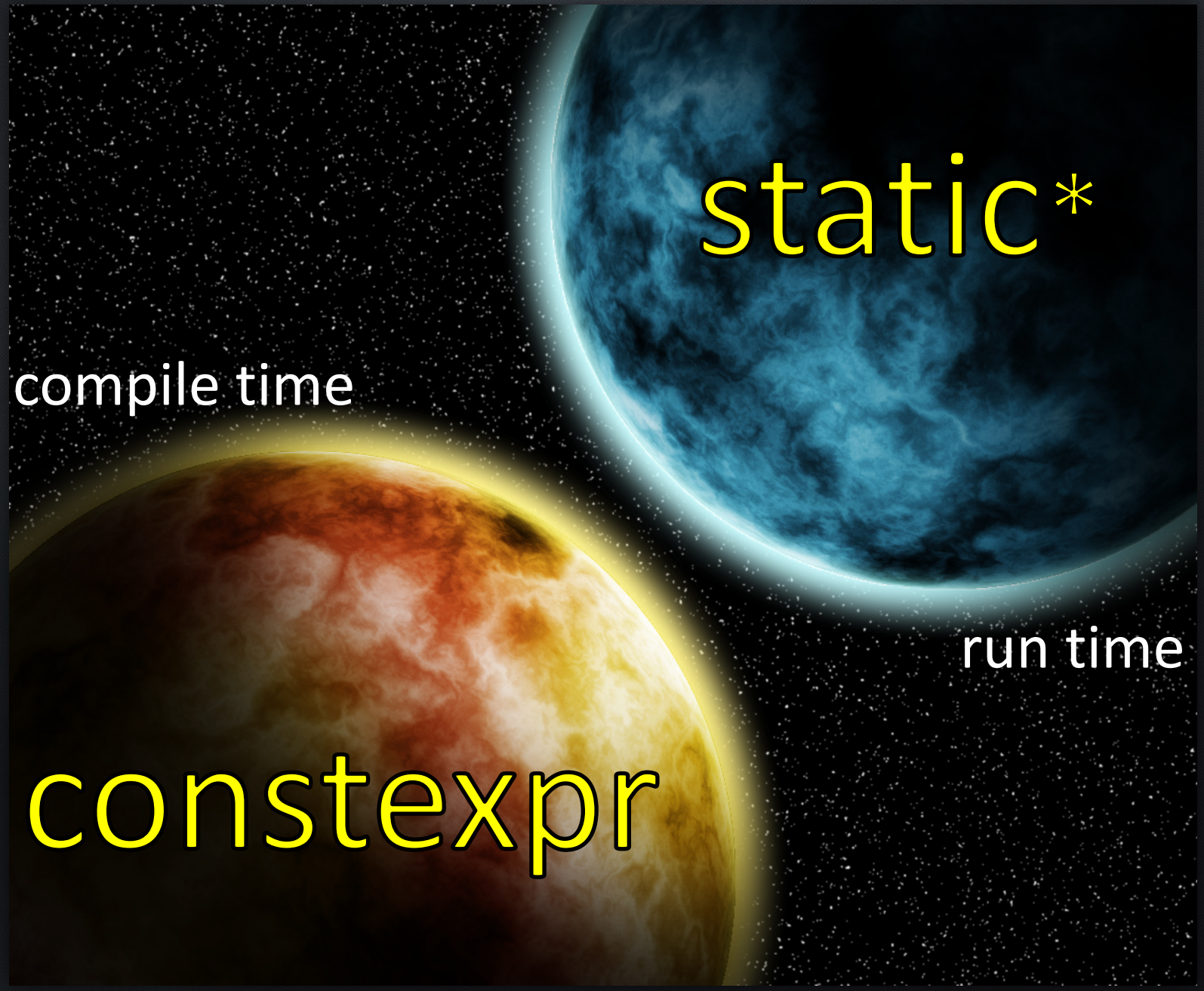
Properties of entities with static and thread storage duration in C++17

Apply in order

#	Property \ Entity	local variable	global variable	member variable	global function	member function
1	no linkage					
2	external linkage					
3	constexpr ⇒ const					
4	constexpr ⇒ inline					
5	const ⇒* internal linkage				N/A	
6	inline ⇒ external (weak) linkage	N/A				
7	static ⇒ internal linkage	Required		Required		Required
8	extern ⇒ external linkage (declaration)	N/A		N/A	Redundant	N/A
9	anonymous (unnamed) namespace ⇒ internal linkage					

* if not volatile, not template, and not previously declared extern

```
                                          //   storage |
struct A                                  //  duration | linkage
{                                         // ----------x----------------
  double x1;                              // automatic | no linkage
  static double x2;                       //    static | external
  static const double x3;                 //    static | external
  static inline const double x4 = 4.0;    //    static | external (weak)
  static constexpr double x5 = 5.0;       //    static | external (weak)
};
```



* when used to modify storage duration

const – можем ли *мы* менять объект?

volatile – может ли *кто-то другой* менять (и читать) объект?

Шаблоны

- Релевантные сущности: функции, классы, переменные
- Не бывает шаблонных сущностей (template entity),
  есть шаблоны сущностей (entity template)
- Сущность получается из шаблона в момент инстанциации
- Удобно думать, что спецификаторы применяются к *инстанцированной сущности*, а не к *шаблону*
- Компоновщик сам позаботится об одинаковых неявных инстанциациях в разных модулях трансляции.
  Их linkage не так важен, и даже не всегда понятен.

I'm taking addresses of the following instantiated variable templates from two translation units:

```
template<class T> bool b = true;
template<class T> const bool cb = true;
template<class T> inline const bool icb = true;
```

I'm printing addresses of `b<int>`, `cb<int>` and `icb<int>`. Here is what clang says:

```
0x6030c0 0x401ae4 0x401ae5  // first translation unit
0x6030c0 0x401ae4 0x401ae5  // second translation unit
```

All addresses are the same, kind of expected. And here is what gcc says:

```
0x6015b0 0x400ef5 0x400ef4  // first translation unit
0x6015b0 0x400ef6 0x400ef4  // second translation unit
```

The address of `cb<int>` changes. Huh? Is this a bug? If not, could someone please explain this effect to me?

This, to me, appears to be related to CWG Issue 1713:

> **Linkage of variable template specializations**
>
> Given a namespace-scope declaration like
>
> ```
> template<typename T> T var = T();
> ```
>
> should `T<const int>` have internal linkage by virtue of its const-qualified type? Or should it inherit the linkage of the template?
>
> **Notes from the February, 2014 meeting**:
>
> CWG noted that linkage is by name, and a specialization of a variable template does not have a name separate from that of the variable template, thus the specialization will have the linkage of the template.

Clang seems to be following it. The template name has external linkage, and so does the variable spun from it.

Ultimately, the intended linkage of variable template specialization is not too well specified currently by the standard itself. It is specified for regular variables, but the templates are a different beast.

inline никогда не подведёт

```
template<class T>
inline constexpr bool is_const_v = is_const<T>::value;
```

inline function templates

- Нет смысла объявлять шаблон функции inline, но может иметь смысл указать его для специализации

```
template<class T>
void swap(T& a, T& b) {
  T t = std::move(a);
  a = std::move(b);
  b = std::move(t);
}

template<>
inline void swap(std::filesystem::path& lhs, std::filesystem::path& rhs) {
  lhs.swap(rhs);
}
```

Объявление явной инстанциации (explicit instantiation declaration)

header.hpp

```
template<class T>
int complicatedTemplateFunction(const T& x) {
  // Some complicated stuff
}

extern template int complicatedTemplateFunction(const std::string& x);
```

source.cpp

```
template int complicatedTemplateFunction(const std::string& x);
```

- Может ускорить время сборки
- Позволяет полностью спрятать тело шаблона

Long road to const

How to declare a constant before C++17?

```
// header.hpp

#define n 42  // Cannot contain expressions; simply evil

const int n = 42;  // Duplicated in each translation unit

extern const int n;  // Initializer is not visible; need manual definition

inline int n() {  // Needs parens to be used; returns rvalue
  return 42;
}

enum {
  n = 42  // Works only for integers
};
```

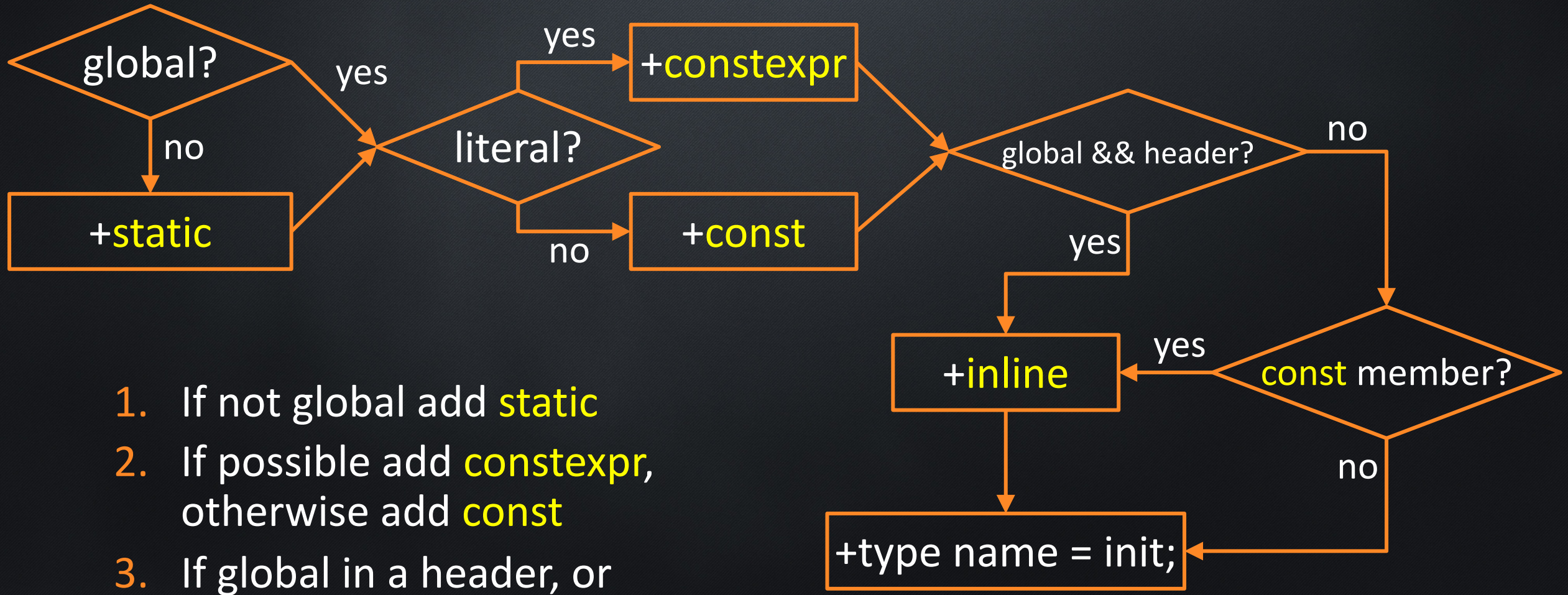
How to declare a constant with an initializer?

```
// header.hpp
inline constexpr int n1 = 1;          // Default choice
inline const std::string s2 = "2";  // If not a literal type

// source.cpp or module.ixx
constexpr int n3 = 3;          // Default choice; implicitly static
const std::string s4 = "4";  // If not a literal type; implicitly static

// Anywhere
struct A {
  static constexpr int n = 5;                // Default choice; implicitly inline
  static inline const std::string s = "6";  // If not a literal type
};

void f() {
  static constexpr int n = 7;          // Default choice
  static const std::string s = "8";  // If not a literal type
}
```

How to declare a constant with an initializer *in an ideal world*?

```
// headers are no longer used
// whatever possible is constexpr


// module.ixx
constexpr int n3 = 3;


// Anywhere
struct A {
  static constexpr int n = 5;

};

void f() {
  static constexpr int n = 7;

}
```

How to declare a constant with an initializer?



1. If not global add static
2. If possible add constexpr, otherwise add const
3. If global in a header, or const member, add inline

Пример

```
template<class T>
static inline thread_local constexpr const volatile T x = {};
```

Implied by constexpr

```
template<class T>
static inline thread_local constexpr volatile T x = {};
```

Overridden by static

```
template<class T>
static thread_local constexpr volatile T x = {};
```

internal thread storage duration

linkage

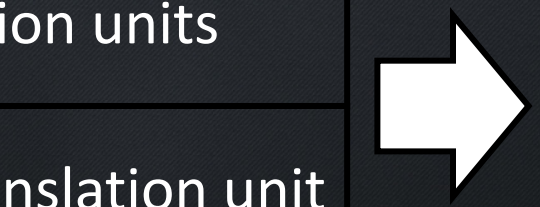
constexpr volatile variable template with thread storage duration and internal linkage

C++20

Modules

C++17

	can be referred to from
external linkage	All translation units
internal linkage	Current translation unit
no linkage	Current scope



C++20

	can be referred to from
external linkage	All modules
module linkage	All translation units within this module
internal linkage	Current translation unit
no linkage	Current scope

consteval specifier

- Функция обязана возвращать константу
- consteval ⇒ constexpr (⇒ inline)

```
constexpr int sqr(int n) {
  return n * n;
}

int arr[sqr(2)];  // OK

int x = std::rand();
int r = sqr(x);  // OK
```

```
consteval int sqr(int n) {
  return n * n;
}

int arr[sqr(2)];  // OK

int x = std::rand();
int r = sqr(x); // Error: Call does not
                // produce a constant
```

consteval specifier

- Удобно думать, что consteval (a.k.a. immediate) функция
  - недоступна на этапе компоновки и выполнения
  - не генерирует символа в объектном файле
  - своего рода функциональный макрос
- На самом деле это не совсем так. Стандарт ничего не знает про компиляцию, компоновку и выполнение.

constinit specifier

```
constexpr int sqr(int n) {
  return n * n;
}

constinit int r = sqr(10);  // OK

int x = std::rand();
constinit int r2 = sqr(x);  // Error: sqr(x) is not a
                            // constant expression
void foo() {
  r = 20;  // Works: constinit does not imply const
}
```

- Применим только к переменным
- constexpr ⇒* constinit ⇏ const

* напрямую это не утверждается, но constexpr накладывает более строгие ограничения, чем constinit

constinit specifier

- Переменная должна иметь static или thread storage duration
- Позволяет избежать static initialization order fiasco

Core constant expressions

A *core constant expression* is any expression whose evaluation *would not* evaluate any one of the following:

1. the `this` pointer, except in a `constexpr` function or a `constexpr` constructor that is being evaluated as part of the expression
2. a function call expression that calls a function (or a constructor) that is not declared constexpr

```
constexpr int n = std::numeric_limits<int>::max(); // OK: max() is constexpr
constexpr int m = std::time(nullptr); // Error: std::time() is not constexpr
```

3. a function call to a `constexpr` function which is declared, but not defined
4. a function call to a `constexpr` function/constructor template instantiation where the instantiation fails to satisfy constexpr function/constructor requirements.
5. (since C++20) a function call to a constexpr virtual function, invoked on an object not *usable in constant expressions* (see below) and whose lifetime began outside this expression.
6. an expression that would exceed the implementation-defined limits
7. an expression whose evaluation leads to any form of core language undefined behavior (including signed integer overflow, division by zero, pointer arithmetic outside array bounds, etc). Whether standard library

18. (until C++20) a `typeid` expression applied to a glvalue of polymorphic type
19. a new-expression or a delete-expression
20. an equality or relational operator when the result is unspecified
21. a throw expression
22. inside a lambda-expression, a reference to `this` or to a variable defined outside that lambda, if that reference would be an odr-use

Properties of entities with static and thread storage duration in C++20**

Apply in order

#	Property \ Entity	local variable	global variable	member variable	global function	member function
1	no linkage					
2	external linkage					
3	consteval ⇒ constexpr	N/A	N/A	N/A		
4	constexpr ⇒ const && constinit					
5	constexpr ⇒ inline					
6	const ⇒* internal linkage				N/A	
7	inline ⇒ external (weak) linkage	N/A				
8	static ⇒ internal linkage	Required		Required		Required
9	extern ⇒ external linkage (declaration)	N/A		N/A	Redundant	N/A
10	anonymous (unnamed) namespace ⇒ internal linkage					

* if not volatile, not template, and not previously declared extern

** ignoring modules

constexpr const

constexpr const

```
constexpr const char* foo() {
  return "cppconf";
}
```

constexpr const

```
struct Params {
  double param1;
  double param2;
};

constexpr Params GlobalParamsWithDescriptiveName = { 3.14, 2.72 };

void foo() {
  constexpr Params& params = GlobalParamsWithDescriptiveName;
}
```

⇒ error: binding reference of type ‘Params&’ to ‘const Params’ discards qualifiers

constexpr const

```
struct Params {
  double param1;
  double param2;
};

constexpr Params GlobalParamsWithDescriptiveName = { 3.14, 2.72 };

void foo() {
  constexpr const Params& params = GlobalParamsWithDescriptiveName;
}
```

constexpr references

```
Params p;                       // Value;
const Params* pcp = &p;         // Pointer to const
Params* const cpp = &p;         // Const pointer
const Params& rcp = p;          // Reference to const
Params& const crp = p;          // Const reference (does not compile)

static_assert(!std::is_const_v<decltype(rcp)>);

void foo() {
  constexpr const Params& params = GlobalParamsWithDescriptiveName;
}
```

Applies to object

Applies to reference itself

AAA

```
void foo() {
  constexpr auto& params = GlobalParamsWithDescriptiveName;
}
```

Заключение

Рекомендации

- Помещайте всё в анонимное пространство имён если возможно
  - Рассмотрите вариант полностью отказаться от static для глобальных сущностей (его хотели сделать deprecated)
- Предпочитайте inline вместо extern
- Предпочитайте constexpr вместо const
- Старайтесь использовать переменные со static и thread storage duration только для констант

MIPT Искусство разработки на современном C++ Specialization

Enrolled

About How It Works **Courses** Instructors Enrollment Options FAQ

COURSE 4

Основы разработки на C++: коричневый пояс

★★★★½ 5.0 67 ratings • 10 reviews

Основная цель этого курса — научить идиомам языка C++, то есть показать, как с помощью различных возможностей языка создавать элегантные, эффективные и надёжные блоки кода. В совокупности со знаниями, полученными на «Красном поясе», это позволит вам

SHOW ALL

COURSE 5

Основы разработки на C++: черный пояс

★★★★★ 5.0 7 ratings • 2 reviews

Во-первых, в «Чёрном поясе» будут изложены темы, без которых ваше представление о C++ будет неполным, — это таблицы виртуальных методов, виртуальные деструкторы, неопределённое поведение, шаблоны с произвольным числом аргументов и forwarding-ссылки. Во-

SHOW ALL

Подвал

constexpr trap

- constexpr implies inline for a function
- constexpr functions with same signature and different bodies might interfere!
  - Note that return type is not part of a mangled name!
  - Different behavior depending on optimization!
- If the functions weren’t inline, the linker would warn you

Описание

Уже в C++98 у нас были const, volatile, static, extern, inline, и, конечно, шаблоны. В C++11 добавились thread_local, constexpr, а также extern для шаблонов. В C++14 добавились шаблоны переменных. В C++17 – inline переменные. В C++20 обещают подвезти consteval и constinit. А вы когда-нибудь задумывались, что такое template static inline thread_local constexpr volatile переменная?..

В этом докладе я попытаюсь разложить по полочкам всё это многообразие ключевых слов. Мы вспомним про linkage, storage duration и инстанциации шаблонов. Разберёмся, какая связь между template и inline, между static и constexpr. Поймём, зачем нам extern, когда у нас есть inline. И осознаем, как нам потребовалось почти 20 лет, чтобы научиться нормально объявлять константы.

План доклада

С последовательностью изложения я до конца не определился. Но содержимое примерно такое:

- Сначала небольшой теоретический экскурс: модули трансляции, linkage, storage duration. Слово-перевёртыш static – в разных контекстах значит совершенно разные вещи.
- Чуть более подробный экскурс про constexpr. Если linkage определяет область видимости, то constexpr определяет время видимости (compile/run time). Как сочетаются static и constexpr.
- Может быть скажу, зачем нужен constexpr const для переменных (например, необходим, чтобы объявить constexpr ссылку).
- Кратко про const и volatile – двух братьев близнецов. Один из которых уродец, и про него никто не вспоминает.
- Что такое extern? Он превращает определение в объявление. Неявно присутствует у функций.
- Поэтапно про константы: макросы, const, getter-function, constexpr, inline.
- Отличие extern и inline переменных. Исследуем объектные файлы. Компоновщик либо убирает лишнее, либо выбирает единственное доступное определение.
- В чём связь template и inline функций? И те и другие могут приводить к множественным определениям. Но это разные вещи. Может быть упомяну, когда необходимо помечать template функцию как inline (при явной специализации).
- Когда имеет смысл делать extern template. Как себя в этом случае ведут неявные инстанциации. Что при этом видит компоновщик. Если успею, сделаю замеры на реальном проекте.
- Взгляд в C++20: consteval и constinit
- Итог: в каких сценариях какие комбинации следует указывать. Самый практичный – как раз глобальные константы.

В процессе детальной подготовки может оказаться, что тем слишком много. В этом случае я планирую сократить глубину, но не ширину доклада. Т.к. идея именно в том, чтобы создать некоторую общую картину, в которую будут укладываться все сущности C++ (ну, хотя бы C++17). Пусть даже без подробностей и нюансов.

Out of scope

- virtual/override/final
- mutable – хотя является “storage class specifier”
- noexcept
- extern for language linkage
- dynamic libraries
- pre-C++-17 approaches and hacks
- different declarations/definitions in different places
  - exception: consider inline functions to understand how linker works
  - exception: const for function parameter in function definition
  - exception: extern

Random

- An inline function or variable with external linkage must be declared inline in every translation unit
  - However, works for gcc/clang (clang –c; nm, weak symbol)
  - Indeed does not work for VC++ (dumpbin /symbols)

Things to clarify

- https://github.com/ericniebler/range-v3/blob/9930c89e005e203ff8294f461e5ac99277d3b088/include/range/v3/iterator/diffmax_t.hpp#L430 – why need inline definitions for integral static members?
- Does inline slow down linker?
- linkage of template instantiations
- inline and extern, https://stackoverflow.com/a/14017155/261217

Переводы

- implies – влечёт
- linking – компоновка