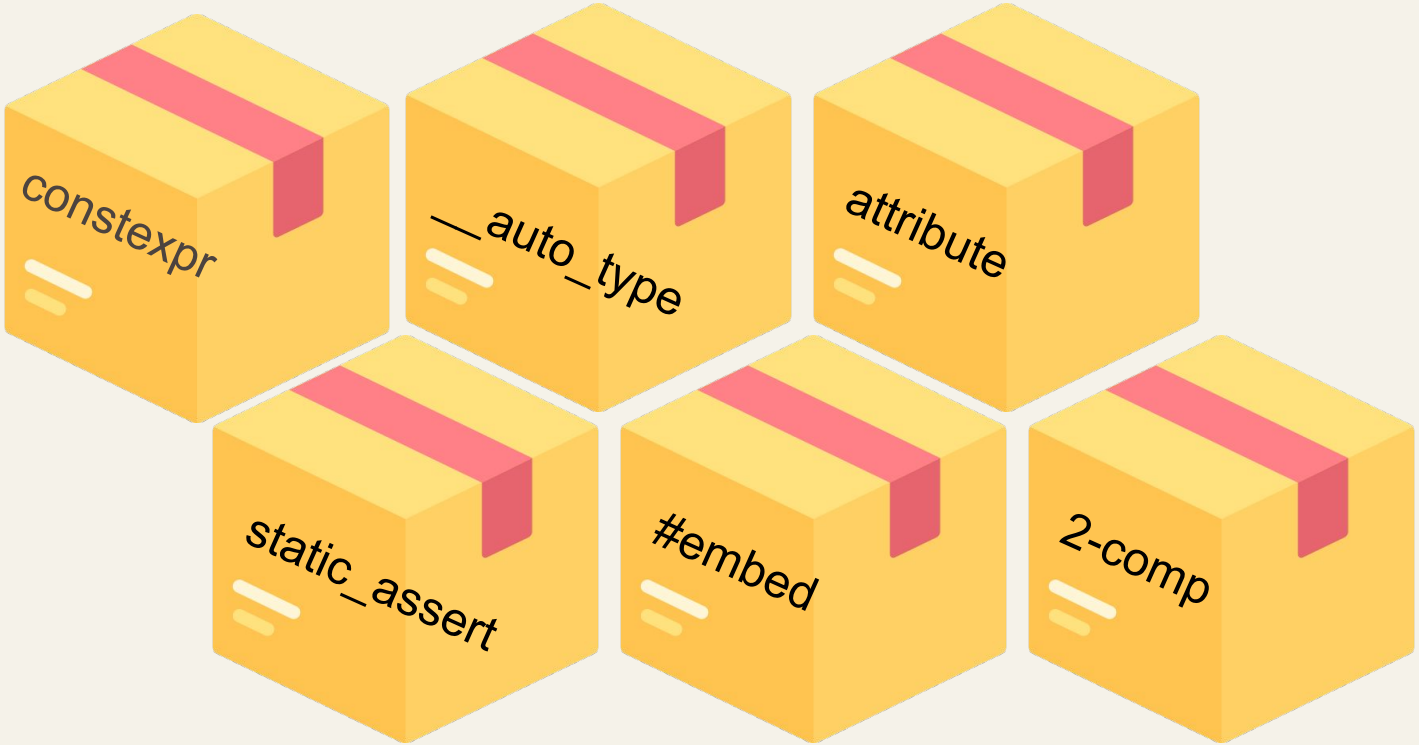
constexpr

The evolution of modern C++.

scc@teamt5.org

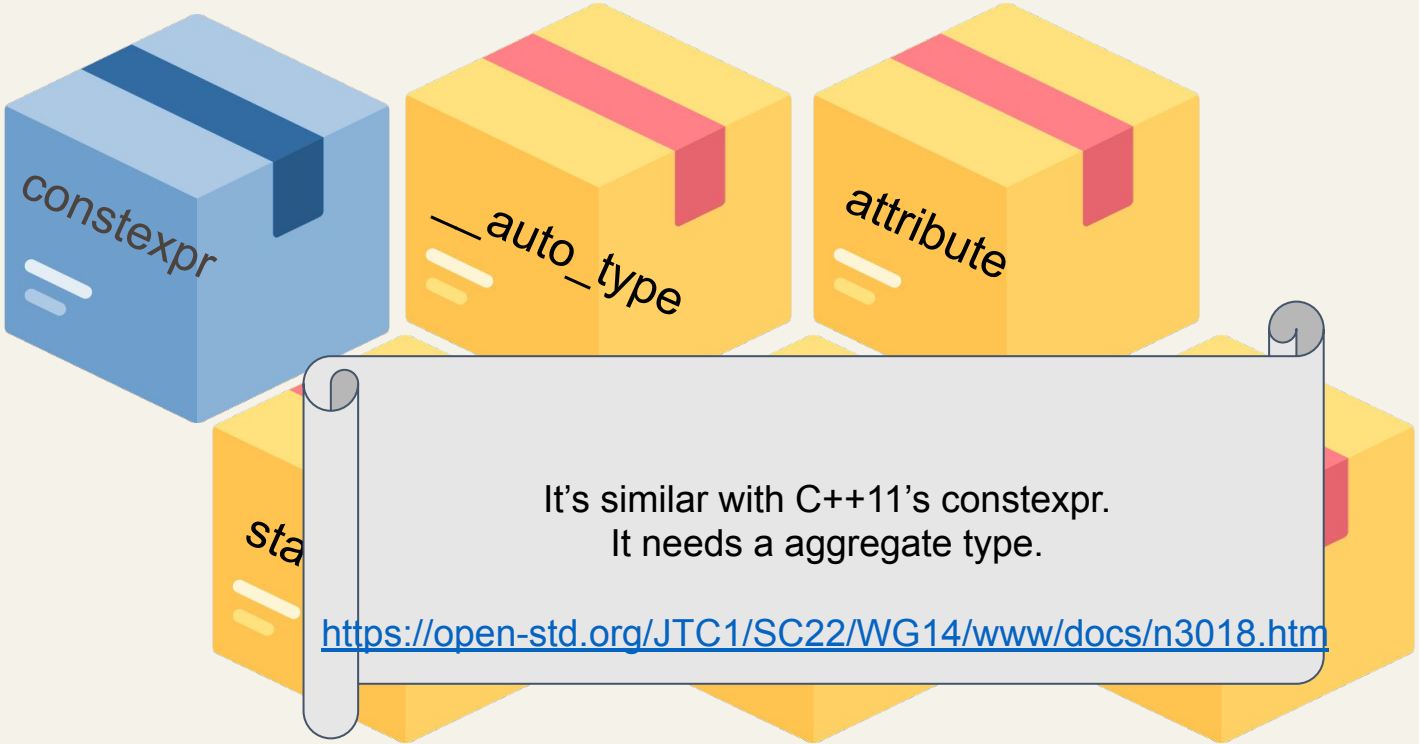
The C23

TEAMT5
杜 浦 數 位 安 全



https://en.cppreference.com/w/c/23

The C23



https://en.cppreference.com/w/c/23

C++ is much more powerful.

First coming of const. expr.

Doc. no. N1521=03-0104
Date: September 21, 2003
Reply-To: Gabriel Dos Reis
gdr@acm.org

Generalized Constant Expressions

Abstract

We suggest to generalize the notion of *constant expressions* to include calls to *constant-valued* functions. The purpose is to push their expressive power even further, to remove embarassments in the standard library, to provide some support for meta programming and convenient notation for expressions that are morally constant.

Introduction

This paper proposes to generalize the definition of constant expressions to include calls to suitable simple functions with constant expressions — which abstractly are constant expressions with named sub-patterns. It aims at providing better type-safety support for components of the standard library (or general libraries), to remove embarrassments, and to enhance the expressive power of constant expressions. The suggestions contained in this paper are not intended to be final wordings. Rather, they are initial basis for discussions and improvements.

1 The problems

This section describes examplars of problems the idea of generalizaing constant expressions, as proposed in §2.2, is trying to solve. At the end of this section, we recall the definition of constant expressions as currently in use.

1.1 Non-portable, non-compile time bitmasks

The Standard Library introduces the notion of *bitmask types* in its introductory clause (see [1, §17.3.2.1.2]):

https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1521.pdf

First coming of const. expr.

```
enum fmtflags {
  boolapha, dec, fixed, hex, internal, left, oct, right,
   // ...
};

inline fmtflags
operator|(fmtflags a, fmtflags b)
{ return fmtflags(int_type(a) | int_type(b)); }
// ...

const fmtflags
  adjustfield = left | right | internal // NOT a constant
```

Not a “compile time” constant

https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1521.pdf

First coming of const. expr.

```
enum fmtflags {
  boolalpha, dec, fixed, hex, internal, left, oct, right,
   // ...
};

inline fm
operator|
{ return
// ...

const fmtflags
  adjustfield = left | right | internal // NOT a constant
```

1.2 Embarassments with numeric constants

The Standard Library defines a traits (`numeric_limits`) that provides C++ programs with information about various properties of the implementation's representation of fundamental arithmetic types. For example, `numeric_limits<T>::is_signed` is an integral constant expression that evaluates to `true` when the type `T` is signed. If `T` is an integer type, then associated macros `XXX_MIN` and `XXX_MAX` (defined in `<climit>`) are integral constant expressions that denotes the minimum and maximum values of `T`. The same values are available as calls to functions `numeric_limits<T>::min()` and `numeric_limits<T>::max()`, except that they are no longer integral constant expressions, mostly because of a restrictive definition of constant expression.

???????

Not a "compile time" constant

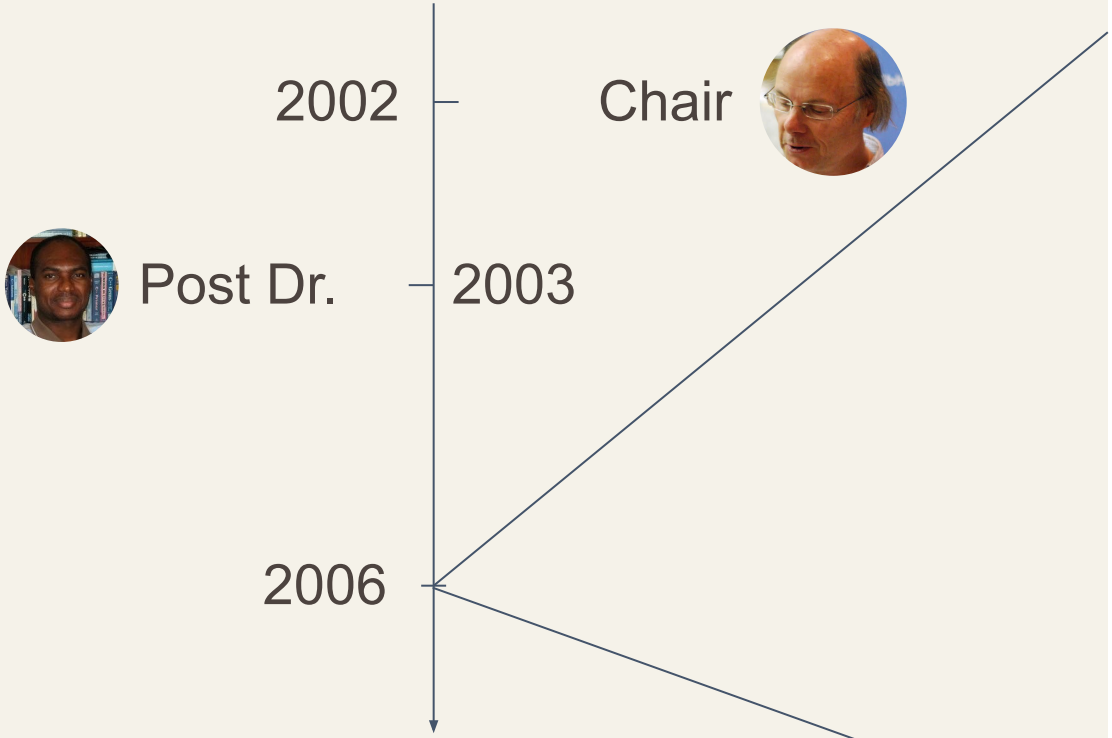
https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1521.pdf

N1521 is similar to C23's constexpr

TEAMT5
杜 浦 數 位 安 全

1. an *integral constant expression*,
2. a *null pointer value*,
3. a *null member pointer value*,
4. an *arithmetic constant expression*,
5. an *address constant expression*,
6. a *reference constant expression*,
7. an address constant expression for a complete object type, plus or minus an integral constant expression, or
8. a *pointer to member constant expression*.

Generalized Constant Expressions — Revision 2



https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n1972.pdf

Doc no: N1972=06-0042
Date: 2006-02-26
Reply-To: Gabriel Dos Reis
gdr@cs.tamu.edu

Generalized Constant Expressions — Revision 2

Gabriel Dos Reis
Texas A&M University

Bjarne Stroustrup
Texas A&M University
and
AT&T Labs Research

Abstract

This paper proposes to generalize the notion of *constant expressions* to include *constant-expression functions* and *user-defined literals*. In addition, some floating-point constant expressions are allowed. The goal is to improve support for generic programming, systems programming, and library building, and to increase C99 compatibility. The proposal allows us to remove long-standing embarassments from some Standard Library components (notably `<limits>`).

Introduction

This paper generalizes the notion of constant expressions to include calls to "sufficiently simple" functions (*constant-expression functions*) and objects of user-defined types constructed from "sufficiently simple" constructors (*constant-expression constructors*.) The proposal aims to

- improve type-safety and portability for code requiring compile time evaluation;
- improve support for systems programming, library building, generic programming; and
- remove embarassments from existing Standard Library components.

The suggestions in this proposal directly build on previous work — in particular *Generalized Constant Expressions* [DR03] and *Literals for user-defined types* [Str03] — and discussions at committee meetings — in particular in Kona (October 2003), Redmond (October 2004) and Mont Tremblant (October 2005).

C++11 constexpr

C++11 constexpr

TEAMT5
杜 浦 數 位 安 全

Generalized Constant Expressions — Revision 5

The generalization we propose are articulated in three steps:

- First, we introduce ***constant-expression functions*** and use those to generalize constant expressions.
- Second, we introduce “literals for user-defined type” based on the notion of ***constant expression constructors***.
- Finally, we describe floating-point constant expressions.

https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2235.pdf

constexpr function

TEAMT5
杜 浦 數 位 安 全

- it returns a value (i.e., has non-void return type)
- its body consists of a single statement of the form **`return`** ***expr***;
- it is declared with the keyword constexpr.

constexpr function

TEAMT5
杜 浦 數 位 安 全

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

- it returns a value (i.e., has non-void return type)
- its body consists of a single statement of the form `return` *expr*;
- it is declared with the keyword constexpr.

constexpr function

TEAMT5
杜 浦 數 位 安 全

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

- it returns a value (i.e., has non-void return type)
- its body consists of a single statement of the form `return` *expr*;
- it is declared with the keyword constexpr.

constexpr function

TEAMT5
杜 浦 數 位 安 全

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

- it returns a value (i.e., has non-void return type)
- its body consists of a single statement of the form `return` *expr*;
- it is declared with the keyword constexpr.

constexpr function

TEAMT5
杜 浦 數 位 安 全

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

- it returns a value (i.e., has non-void return type)
- its body consists of a single statement of the form `return` *expr*;
- it is declared with the keyword constexpr.

Constexpr ctors

Constructor
Destructor
Copy-constructor

Constexpr ctors

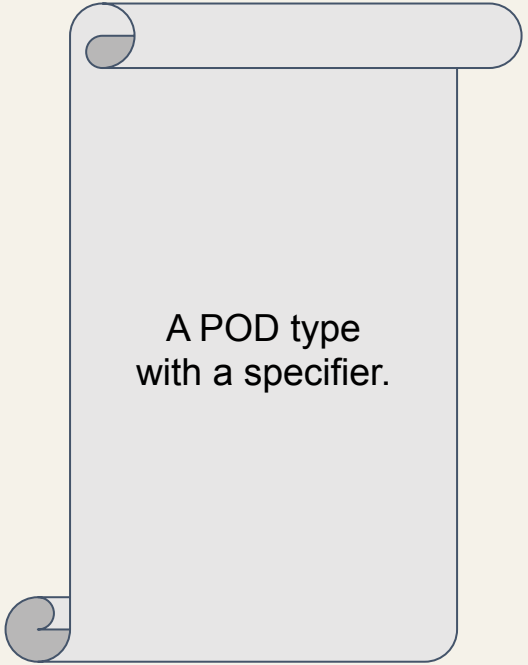
Constructor
- declared with the constexpr specifier
- with member-initializer part involving only potential constant-expressions
- and the body of which is empty

Destructor
Copy-constructor

Constexpr ctors

Constructor
- declared with the constexpr specifier
- with member-initializer part involving only potential constant-expressions
- and the body of which is empty

Destructor
Copy-constructor

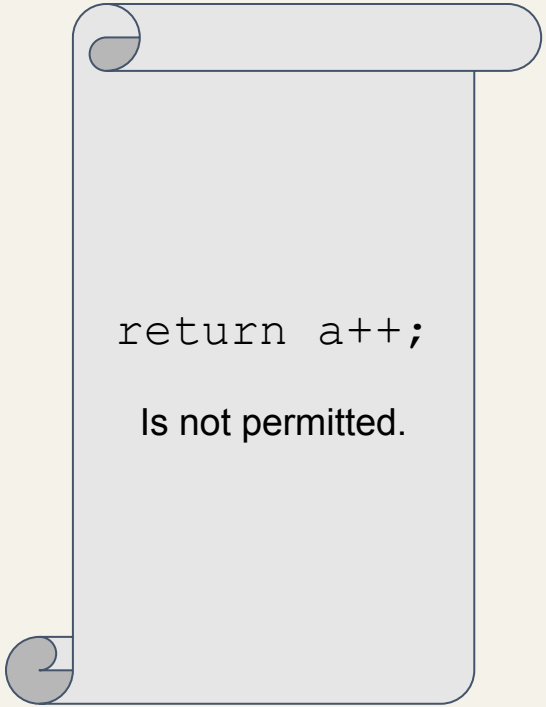


Constexpr ctors

TEAMT5
杜 浦 數 位 安 全

Constructor
Destructor
- Zero side-effect

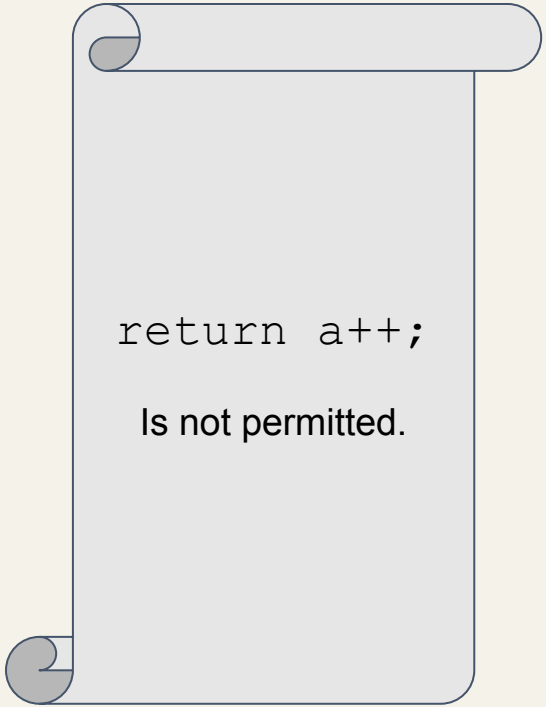
Copy-constructor



Constexpr ctors

Constructor
Destructor
- Zero side-effect

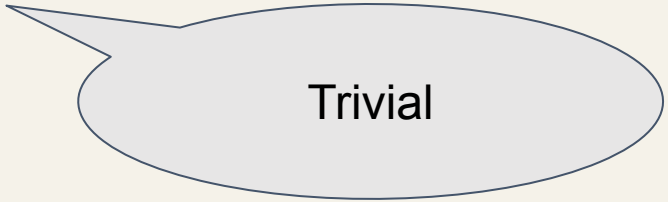
Copy-constructor



It’s another issue that called pure functions.

Constexpr ctors

Constructor
Destructor
Copy-constructor



with potential constant expression arguments, if any.

A trivial copy constructor is also considered a constexpr constructor.

Constexpr ctors

TEAMT5
杜 浦 數 位 安 全

Constructor
Destructor
Copy-constructor

Trivial

In simple words a "trivial" special member function literally means a member function that does its job in a very **straightforward** manner.

with potential constant expression arguments, if any.

A trivial copy constructor is also considered a constexpr constructor.

Constexpr ctors

Constructor
Destructor
Copy-constructor

Trivial

Copy-constructor:
Same as memcpy

with potential constant expression arguments, if any.

A trivial copy constructor is also considered a constexpr constructor.

So, C++11 constexpr

TEAMT5
杜 浦 數 位 安 全

```
constexpr int factorial(int n) {
    return n <= 1 ? 1 : (n * factorial(n - 1));
}
```

So, C++11 constexpr

TEAMT5 杜浦數位安全

```
class conststr {
    const char* p;
    std::size_t sz;
public:
    template<std::size_t N>
    constexpr conststr(const char(&a)[N]): p(a), sz(N - 1) {}

    // constexpr functions signal errors by throwing exceptions
    // in C++11, they must do so from the conditional operator ?:
    constexpr char operator[](std::size_t n) const {
        return n < sz ? p[n] : throw std::out_of_range("");
    }

    constexpr std::size_t size() const { return sz; }
};
```

So, C++11 constexpr

TEAMT5 杜浦數位安全

```
class conststr {
    const char* p;
    std::size_t sz;
public:
    template<std::size_t N>
    constexpr conststr(const char(&a)[N]): p(a), sz(N - 1) {}
```

Reference type

New paragraph Insert after §3.9/10:

11 A type is a *literal type* if

— it is a scalar type; or
— it is a class type (9) with
- trivial copy constructor,
- trivial destructor,
- at least one constexpr constructor other than the copy constructor,
- no virtual base classes, and
- all non-static data members and base classes of literal types; or

— it is an array of literal type.

Not in purpose

10 A type is a *literal type* if it is:

— a scalar type; or

§3.9/10

— a reference type referring to a literal type; or

— an array of literal type; or

— a class type (Clause 9) that has all of the followin

But in standard

```
};
```

C++14 constexpr

Introduce

TEAMT5
杜 浦 數 位 安 全

- N3652 Extended constexpr
- N3302 constexpr for <complex>
- N3469 constexpr for <chrono>
- N3470 constexpr for <array>
- N3471 constexpr for <initializer_list>, <utility> and <tuple>

 N3652 Extended constexpr

It comes from N3597, but not all feature are selected.

Create another paper(ticket) 3652 to pick features.

 N3652 Extended constexpr

TEAMT5 杜浦數位安全

The *for*-statement in constexpr function:

```
constexpr int sum(int begin, int end) {
    int s = 0;
    for (int i = begin; i < end; i++)
        s += i;
    return s;
}
```

N3652 Extended constexpr

TEAMT5 杜浦數位安全

The *for*-statement in constexpr function:

```
constexpr int sum(int begin, int end) {
    int s = 0;
    for (int i = begin; i < end; i++)
        s += i;
    return s;
}
```

- its body consists of a single statement of the form `return` *expr*;

N3652 Extended constexpr

TEAMT5 杜浦數位安全

The *for*-statement in constexpr function:

```
constexpr int sum(int begin, int end) {
    int s = 0;
    for (int i = begin; i < end; i++)
        s += i;
    return s;
}
```

Side-effects

- its body consists of a single statement of the form `return` *expr*;

N3652 Extended constexpr

TEAMT5 杜浦數位安全

The `for` statement in constexpr function:

```
constexpr int sum(int begin, int end) {
    return (begin == end)
            ? 0
            : begin + sum(begin + 1, end);
}
```

- its body consists of a single statement of the form `return` *expr*;

N3652 Extended constexpr

TEAMT5 杜浦數位安全

The *for*-statement in constexpr function:

```
constexpr int sum(int begin, int end) {
    int s = 0;
    for (int i = begin; i < end; i++)
        s += i;
    return s;
}
```

N3652 Extended constexpr

TEAMT5 杜浦數位安全

§7.1.5/8 A **`constexpr`** specifier for a non-static member function that is not a constructor declares that member function to be **`const`**.

N3652 Extended constexpr

TEAMT5 杜浦數位安全

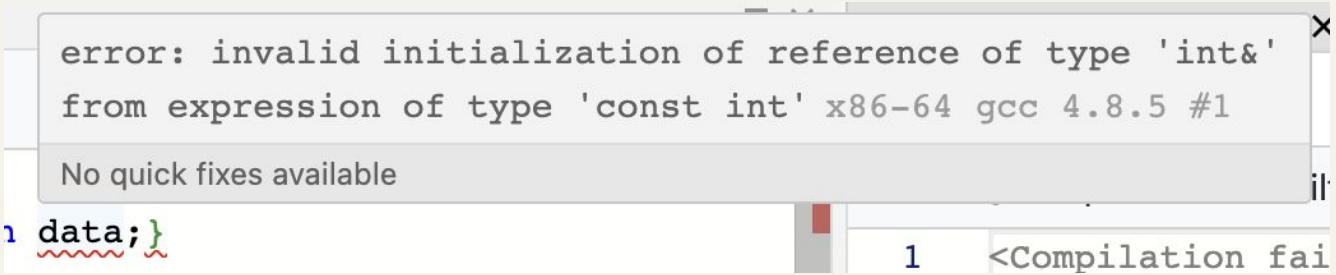
§7.1.5/8 A **constexpr** specifier for a non-static member function that is not a constructor declares that member function to be **const**.

```
struct A {

    constexpr int &get_data() {return data;}

private:

    int data;

};
```

N3652 Extended constexpr

§7.1.5/8 A **`constexpr`** specifier for a non-static member function that is not a constructor declares that member function to be **`const`**.

```
struct A {
    constexpr int &get_data() {return data;}
private:
    int data;
};
```

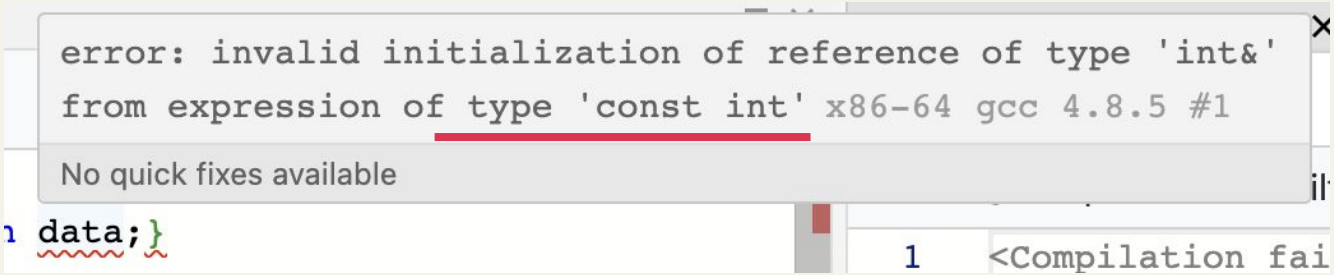


N3652 Extended constexpr

TEAMT5 杜浦數位安全

§7.1.5/8 A **`constexpr`** specifier for a non-static member function that is not a constructor declares that member function to be **`const`**.

```
struct A {
    constexpr int &get_data() {return data;}
private:
    int data;
};
```



N3652 Extended constexpr

TEAMT5 杜浦數位安全

§7.1.5/8 A **constexpr** specifier for a non-static member function that is not a constructor declares that member function to be **const**.

```
struct A {

    constexpr int &get_data() {return data;}

private:

    int data;

};
```

From 98 to 11.
Fixed in 14.

So, C++14 constexpr

TEAMT5
杜 浦 數 位 安 全

```
constexpr int product() {
    return 1;
}

template<typename T, typename... Args>
constexpr T product(T first, Args... args) {
    return first * product(args...);
}

int main() {
    constexpr int result = product(2, 3, 4, 5);
    std::cout << "Product: " << result << std::endl;

    return 0;
}
```

So, C++14 constexp

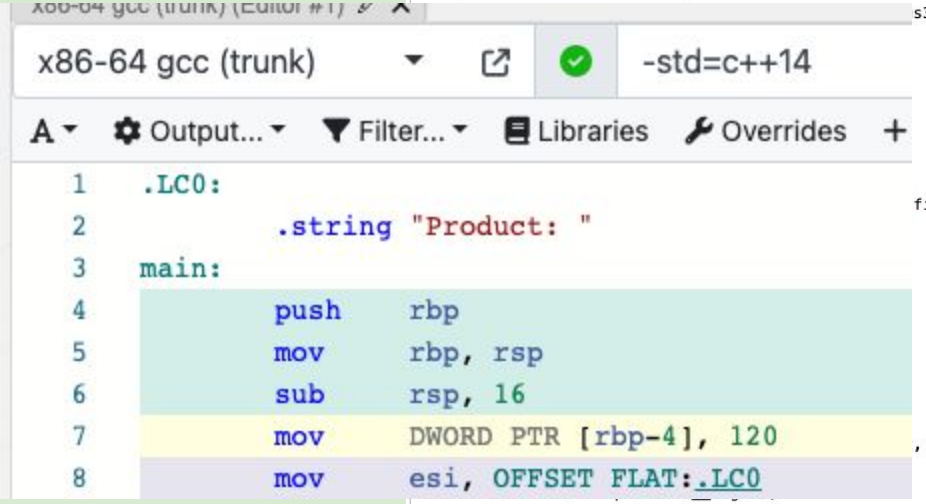
```
constexpr int product() {
    return 1;
}

template<typename T, typename..
constexpr T product(T first, Ar
    return first * product(args.
}

int main() {
    constexpr int result = produ
    std::cout << "Product: " <<

    return 0;
}
```

```
 9 template<typename T, typename ... Args>
10 inline constexpr T product(T first, Args... args)
11 {
12   return first * product(args... );
13 }
14
15
16 /* First instantiated from: insights.cpp:13 */
17 #ifdef INSIGHTS_USE_TEMPLATE
18 template<>
19 inline constexpr int product<int, int, int, int>(int first, int __args1, int __args2, int __args3)
20 {
21   return first * product(__args1, __args2, __args3);
22 }
23 #endif
24
25
26 /* First instantiated from: insights.cpp:9 */
27 #ifdef INSIGHTS_USE_TEMPLATE
28 template<>
29 inline constexpr int product<int, int, int>(int first, int __args1, int __args2)
30 {
31   return first * product(__args1, __args2);
32 }
33 #endif
34
35
36 /* First instantiated from: insights.cpp:9 */
37 #ifdef INSIGHTS_USE_TEMPLATE
38 template<>
39 inline constexpr int product<int, int>(int first, int __args1)
40 {
41   return first * product(__args1);
42 }
43 #endif
44
45
46 /* First instantiated from: insights.cpp:9 */
47 #ifdef INSIGHTS_USE_TEMPLATE
48 template<>
49 inline constexpr int product<int>(int first)
50 {
51   return first * product();
52 }
53 #endif
```

Cpp Insight

So, C++14 constexp

```
constexpr int product() {
    return 1;
}

template<typename
constexpr T produ
    return first *
}

int main() {
    constexpr int result = produ
    std::cout << "Product: " <<

    return 0;
}
```




```
template<typename T, typename ... Args>
inline constexpr T product(T first, Args... args)
{
  return first * product(args... );
}


/* First instantiated from: insights.cpp:13 */
#ifdef INSIGHTS_USE_TEMPLATE
template<>
inline constexpr int product<int, int, int, int>(int first, int __args1, int __args2, int __args3)
{
  return first * product(__args1, __args2, __args3);
}
#endif


/* First instantiated from: insights.cpp:9 */
#ifdef INSIGHTS_USE_TEMPLATE
template<>
inline constexpr int product<int>(int first)
{
  return first * product();
}
#endif
```

Cpp Insight

C++17 constexpr

Introduce

TEAMT5
杜 浦 數 位 安 全

📝P0170R1 constexpr lambda expressions

📝P0292R2 constexpr if statements

And other standard libraries with default constexpr.

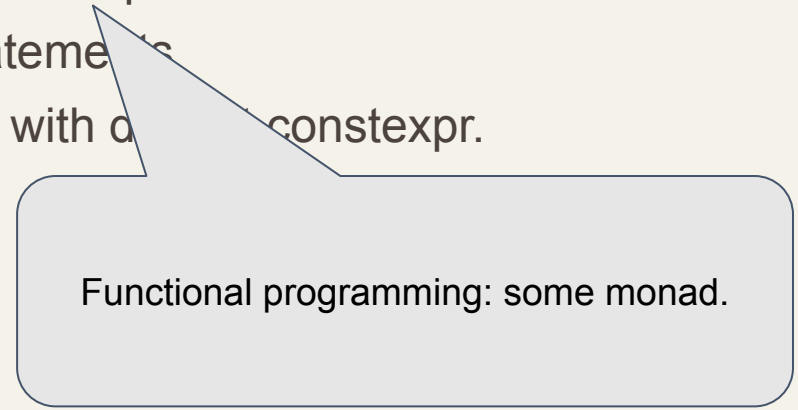
Introduce

TEAMT5
杜 浦 數 位 安 全

📝P0170R1 constexpr lambda expressions

📝P0292R2 constexpr if statements

And other standard libraries with constexpr.



constexpr if

TEAMT5
杜 浦 數 位 安 全

```
template<typename T>
auto get_value(T t) {
    if constexpr (std::is_pointer_v<T>)
        return *t; // deduces return type to int for T = int*
    else
        return t;  // deduces return type to int for T = int
}
```

constexpr if

TEAMT5
杜 浦 數 位 安 全

```
template<typename T>
auto get_value(T t) {
    if constexpr (std::is_pointer_v<T>)
        return *t; // deduces return type to int for T = int*
    else
        return t;  // deduces return type to int for T = int
}
```

Should be evaluated in compile time

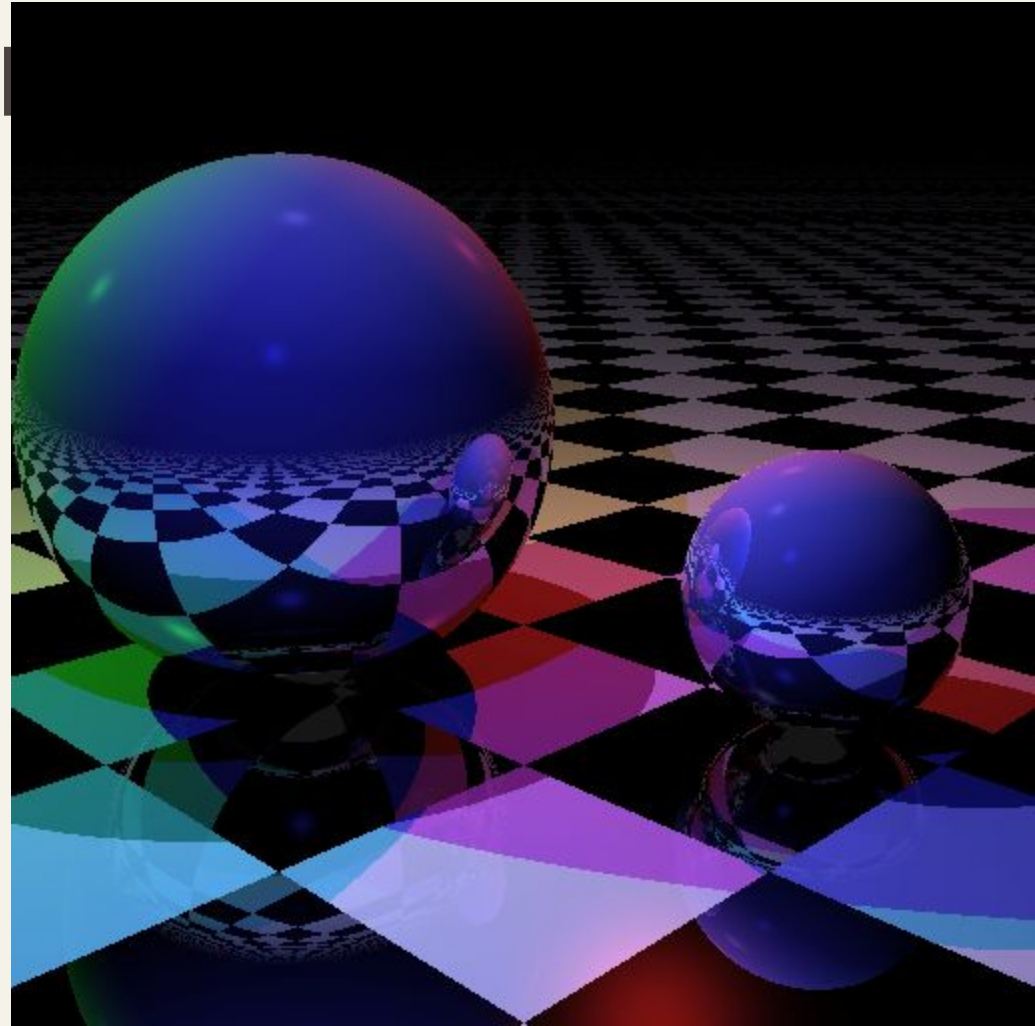
constexpr if -> SFINAE

TEAMT5 杜浦數位安全

```
template<typename T>
auto get_value(T t)
       -> std::enable_if_t<std::is_pointer_v<T>, decltype(*t)>
{
   return *t;
}

template<typename T>
auto get_value(T t)
       -> std::enable_if_t<!std::is_pointer_v<T>, T>
{
   return t;
}
```

So, C++17 constexp

It can do ray tracing in compile time.



https://github.com/tcbrindle/raytracer.hpp

CTRE

TEAMT5
杜 浦 數 位 安 全

```
#define RE_RAW "^(\\d{4})/(\\d{1,2}+)/(\\d{1,2}+)$"
constexpr std::optional<date> extract_date(std::string_view s)
noexcept {
    using namespace ctre::literals;
    if (const auto &[whole, year, month, day] =
                    RE_RAW ""_ctre.match(s); whole)
        return date{year, month, day};
    else
        return std::nullopt;
}
```

https://github.com/hanickadot/compile-time-regular-expressions
https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1149r0.html

CTRE

TEAMT5 杜浦數位安全

```
#define RE_RAW "^(\\d{4})/(\\d{1,2}+)/(\\d{1,2}+)$"
constexpr std::optional<date> extract_date(std::string_view s)
noexcept {
    using namespace ctre::literals;
    if (const auto &[whole, year, month, day] =
                    RE_RAW ""_ctre.match(s); whole)
        return date{year, month, day};
    else
        return std::nullopt;
}
```

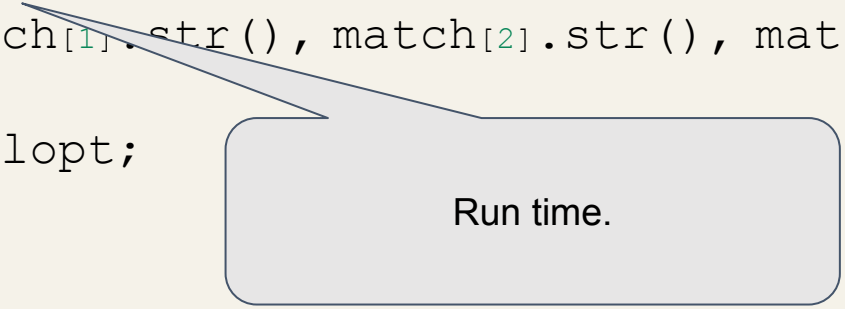
Compile time evaluation.

https://github.com/hanickadot/compile-time-regular-expressions
https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1149r0.html

CTRE

TEAMT5 杜浦數位安全

```
#define RE_RAW "^(\\d{4})/(\\d{1,2}+)/(\\d{1,2}+)$"
std::optional<date> std_extract_date(std::string s) noexcept {
    const std::regex re(RE_RAW);
    std::smatch match;

    if (std::regex_match(s, match, re))
        return date{match[1].str(), match[2].str(), match[3].str()};
    else
        return std::nullopt;
}
```



https://github.com/hanickadot/compile-time-regular-expressions
https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1149r0.html

CTRE

```
int main() {
    const std::string date_str = "2023/11/11";

    double extract_time = eval_time(date_str, extract_date);
    double std_extract_time = eval_time(date_str, std_extract_date);

    std::cout << "Custom Extract Time: " << extract_time << "
nanoseconds\n";
    std::cout << "std::regex E
std_extract_time << " nanosec

    return 0;
}
```

```
ASM generation compiler returned: 0
Execution build compiler returned: 0
Program returned: 0
    Custom Extract Time: 100 nanoseconds
    std::regex Extract Time: 722078 nanoseconds
```

https://godbolt.org/z/sjexKvEGK

CTRE

```
int main() {
    const std::s

    double extra
    double std_e

    std::cout <<
nanoseconds\n";
    std::cout <<
std_extract_tim

    return 0;
}
```


```
ract_date);
, std_extract_date);

act_time << "
```

```
rned: 0
urned: 0

nanoseconds
std::regex Extract Time: 722078 nanoseconds
```

https://godbolt.org/z/sjexKvEGK

But constexpr

TEAMT5
杜 浦 數 位 安 全

```
1   #include <cstdio>
2
3   constexpr int foo(int n) {
4       int res = 0;
5       for (int i = 0; i < n; ++i) {
6           res += i * i;
7       }
8       return res;
9   }
10
11  int main() {
12      printf("%d\n", foo(25));
13  }
14
```

https://godbolt.org/z/aM8G3j4hx

But constexpr

```cpp
#include <cstdio>

constexpr int foo(int n) {
    int res = 0;
    for (int i = 0; i < n; ++i) {
        res += i * i;
    }
    return res;
}

int main() {
    printf("%d\n", foo(25));
}
```

x86-64 clang 12.0.0 --std=c++17

A Output... Filter... Libraries Overrides + Add new... Add tool...

```
main:                                   # @main
        push    rbp
        mov     rbp, rsp
        mov     edi, 25
        call    foo(int)
        mov     esi, eax
        movabs  rdi, offset .L.str
        mov     al, 0
        call    printf
        xor     eax, eax
        pop     rbp
        ret
foo(int):                               # @foo(int)
        push    rbp
        mov     rbp, rsp
        mov     dword ptr [rbp - 4], edi
        mov     dword ptr [rbp - 8], 0
        mov     dword ptr [rbp - 12], 0
.LBB1_1:                                # =>This Inner Loop Header: Depth=1
        mov     eax, dword ptr [rbp - 12]
        cmp     eax, dword ptr [rbp - 4]
        jge     .LBB1_4
        mov     eax, dword ptr [rbp - 12]
        imul    eax, dword ptr [rbp - 12]
        add     eax, dword ptr [rbp - 8]
        mov     dword ptr [rbp - 8], eax
        mov     eax, dword ptr [rbp - 12]
        add     eax, 1
        mov     dword ptr [rbp - 12], eax
        jmp     .LBB1_1
.LBB1_4:
        mov     eax, dword ptr [rbp - 8]
        pop     rbp
        ret
.L.str:
        .asciz  "%d\n"
```

https://godbolt.org/z/aM8G3j4hx

But constexpr

```
1  #include <cstdio>
2
3  const
4      i
5      f
6
7      }
8      r
9  }
10
11 int m
12     p
13 }
14
```

Some compilers are lazy.

§8.20/1 Constant expressions can be evaluated during translation.

x86-64 clang 12.0.0 --std=c++17

Output... Filter... Libraries Overrides Add new... Add tool...

```
1  main:                                   # @main
2          push    rbp
3          mov     rbp, rsp
4          mov     edi, 25
5          call    foo(int)
6          mov     esi, eax
7          movabs  rdi, offset .L.str
8          mov     al, 0
9          call    printf
10         xor     eax, eax
11         pop     rbp
                                                    oth=1
28         add     eax, 1
29         mov     dword ptr [rbp - 12], eax
30         jmp     .LBB1_1
31 .LBB1_4:
32         mov     eax, dword ptr [rbp - 8]
33         pop     rbp
34         ret
35 .L.str:
36         .asciz  "%d\n"
```

https://godbolt.org/z/aM8G3j4hx

C++20 constexpr

Introduce

TEAMT5
杜 浦 數 位 安 全

📝P0859R0 Specify when constexpr function definitions are needed for constant evaluation
📝P1064R0 constexpr virtual function
📝P1002R1 constexpr try-catch blocks
📝P1073R3 Immediate functions (consteval)
📝P1330R0 Changing the active member of a union inside constexpr
📝P0784R7 constexpr container operations
📝P1331R2 Trivial default initialization in constexpr functions
📝P1668R1 Unevaluated asm-declaration in constexpr functions
📝P1143R2 constinit

Introduce

TEAMT5 杜浦數位安全

- P0202R3 constexpr for <algorithm> and <utility>
- P0415R1 More constexpr for <complex>
- P0858R0 ConstexprIterator requirements
- P0879R0 constexpr for std::swap() and swap related functions
- P1023R0 constexpr comparison operators for std::array
- P1006R1 constexpr in std::pointer_traits
- P1032R1 Misc constexpr bits
- P0784R7 constexpr std::allocator and related utilities
- P0980R1 constexpr std::string
- P1004R2 constexpr std::vector
- P1065R2 constexpr std::invoke() and related utilities
- P0883R2 constexpr default constructor of std::atomic and std::atomic_flag
- P1645R1 constexpr for numeric algorithms

Introduce

TEAMT5 杜浦數位安全

📝P0859R0 Specify when constexpr function definitions are needed for constant evaluation

- 📝P1064R0 constexpr virtual function
- 📝P1002R1 constexpr try-catch blocks

📝P1073R3 Immediate functions (consteval)

- 📝P1330R0 Changing the active member of a union inside constexpr
- 📝P0784R7 constexpr container operations
- 📝P1331R2 Trivial default initialization in constexpr functions
- 📝P1668R1 Unevaluated asm-declaration in constexpr functions

📝P1143R2 constinit

Introduce

TEAMT5 杜浦數位安全

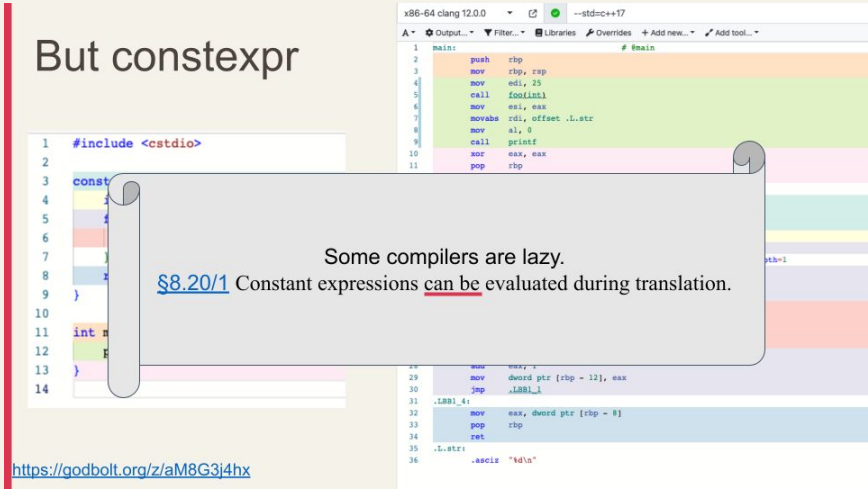
📝P0859R0 Specify when constexpr function definitions are needed for constant evaluation

📝P1064R0 constexpr virtual function
📝P1002R1 constexpr try-catch blocks

📝P1073R3 Immediate functions (consteval)

📝P1330R0 Changing the active member of a union inside constexpr
📝P0784R7 constexpr container operations
📝P1331R2 Trivial default initialization in constexpr functions
📝P1668R1 Unevaluated asm-declaration in constexpr functions

📝P1143R2 constinit



📝P0859R0 Specify when constexpr function definitions are needed for constant evaluation

TEAMT5 杜浦數位安全

```cpp
#include <cstdio>

constexpr int foo(int n) {
    int res = 0;
    for (int i = 0; i < n; ++i) {
        res += i * i;
    }
    return res;
}

int main() {
    int a = foo(25);
    printf("%d\n", a);
    printf("%d\n", foo(25));
}
```

```asm
main:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     DWORD PTR [rbp-4], 4900
        mov     eax, DWORD PTR [rbp-4]
        mov     esi, eax
        mov     edi, OFFSET FLAT:.LC0
        mov     eax, 0
        call    printf
        mov     edi, 25
        call    foo(int)
        mov     esi, eax
        mov     edi, OFFSET FLAT:.LC0
        mov     eax, 0
        call    printf
        mov     eax, 0
        leave
        ret
```

Introduce

TEAMT5 杜浦數位安全

📝P0859R0 Specify when constexpr function definitions are needed for constant evaluation

- 📝P1064R0 constexpr virtual function
- 📝P1002R1 constexpr try-catch blocks

📝P1073R3 Immediate functions (consteval)

- 📝P1330R0 Changing the active member of a union inside constexpr
- 📝P0784R7 constexpr container operations
- 📝P1331R2 Trivial default initialization in constexpr functions
- 📝P1668R1 Unevaluated asm-declaration in constexpr functions

📝P1143R2 constinit

constexpr -> 3 parts

TEAMT5 杜浦數位安全

P0859R0 Specify when constexpr function definitions ed for constant evaluation

P1064R0 constexpr virtual

P1002R1 constexpr try-c

P1073R3 I tions (consteval)

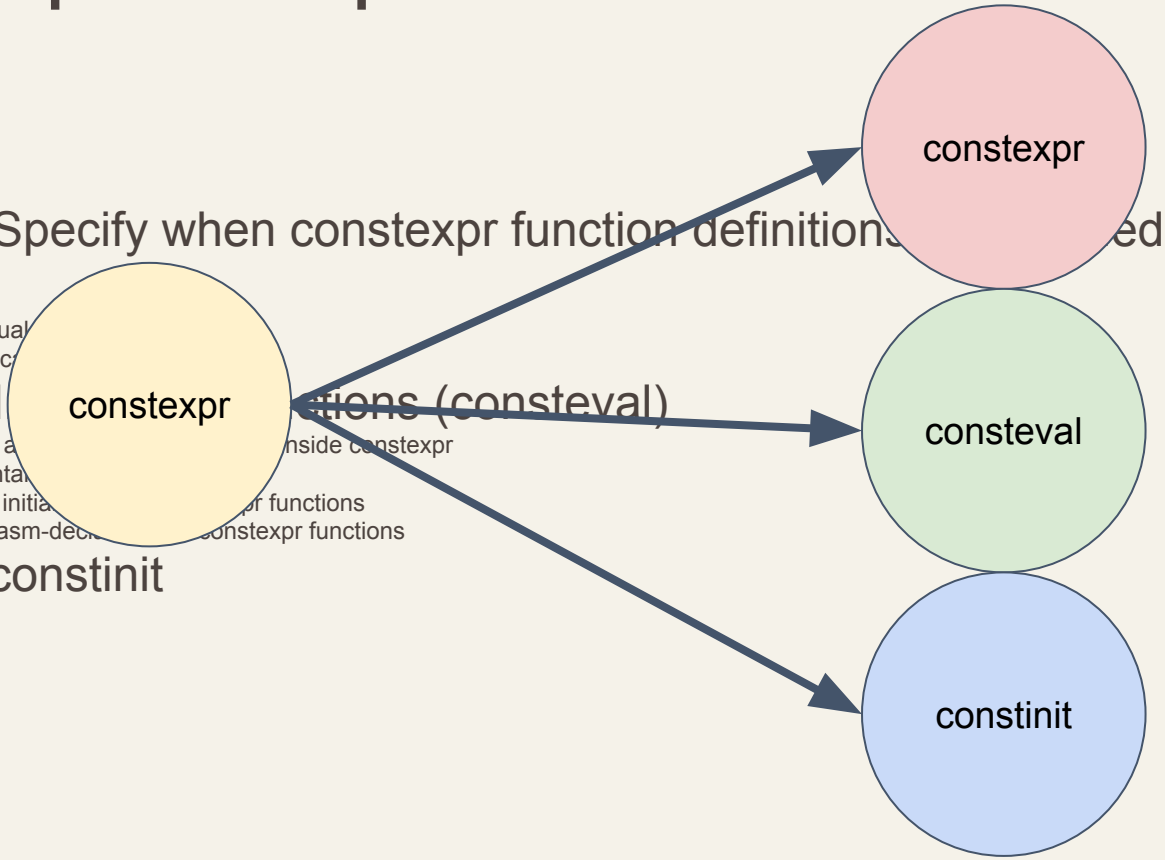
P1330R0 Changing the a nside constexpr

P0784R7 constexpr conta

P1331R2 Trivial default initia r functions

P1668R1 Unevaluated asm-dec onstexpr functions

P1143R2 constinit



sonar::expected

Toward zero-exception

scc

TEAMT5
杜浦數位安全
Persistent Cyber Threat Hunters

constexpr container operations

- P0784R7 constexpr container operations
- P0784R7 constexpr std::allocator and related utilities
- P0980R1 constexpr std::string
- P1004R2 constexpr std::vector

constexpr container operations

TEAMT5 杜浦數位安全

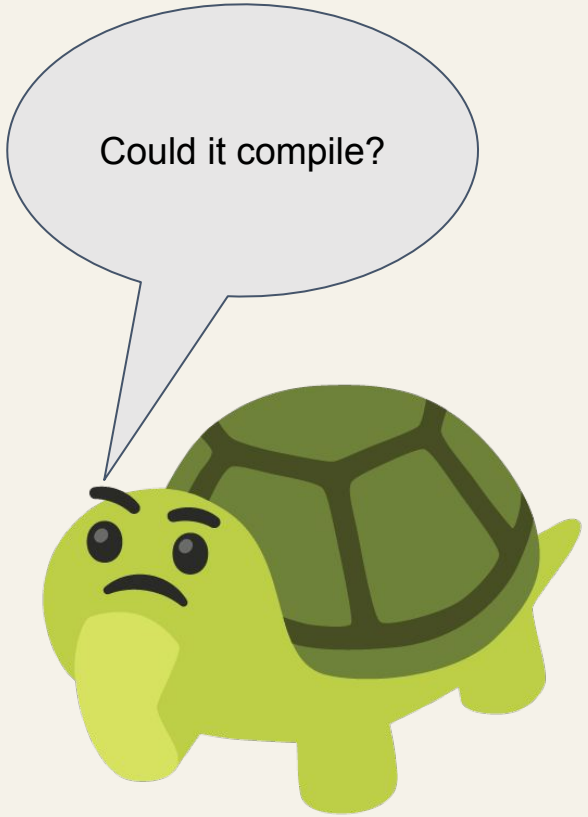
- P0784R7 constexpr container operations
- P0784R7 constexpr std::allocator and related utilities
- P0980R1 constexpr std::string
- P1004R2 constexpr std::vector

> §9.2.5/10 A **`constexpr`** variable shall have constant destruction.

Guess!

TEAMT5
杜 浦 數 位 安 全

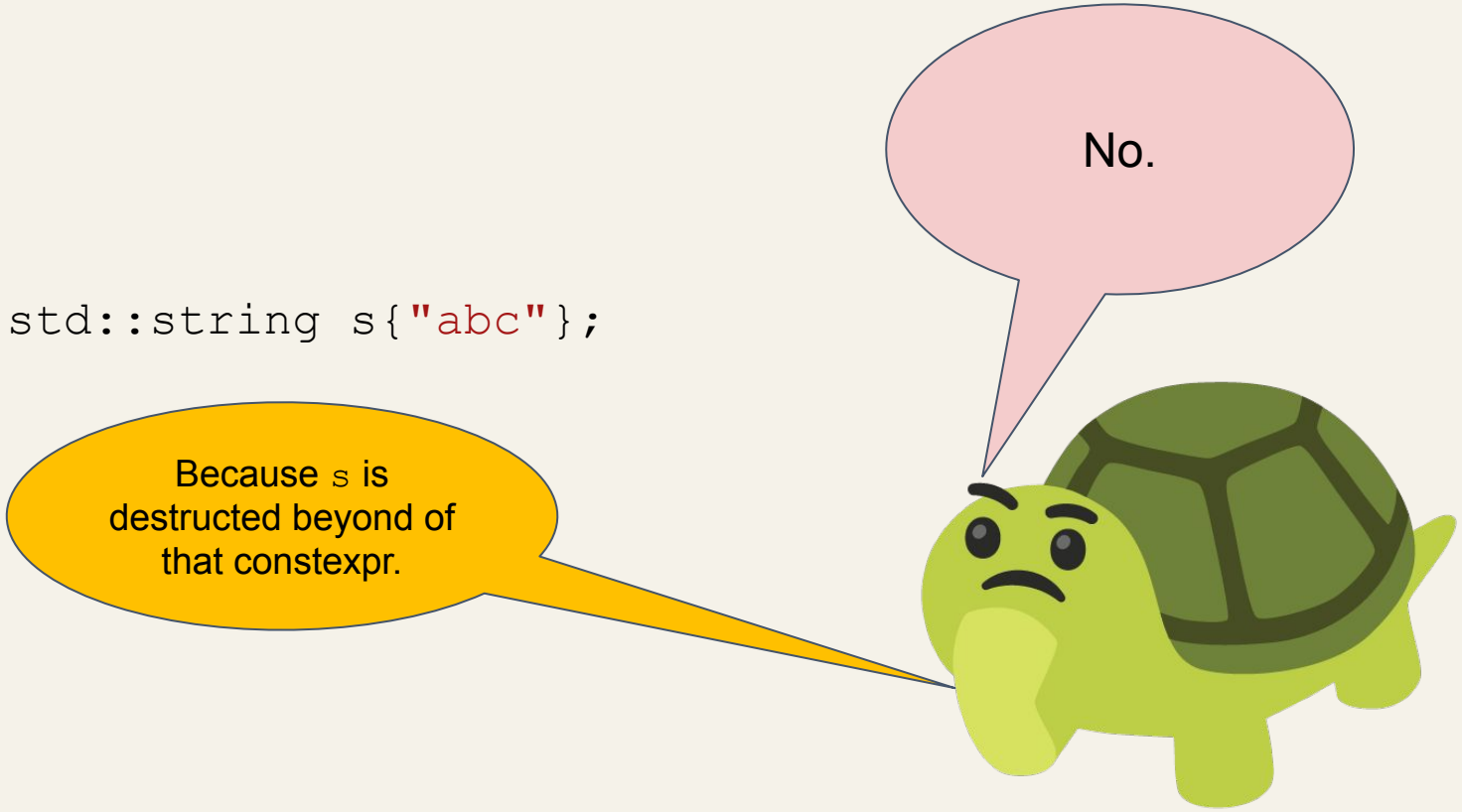
```
int main() {
    constexpr std::string s{"abc"};
    return 0;
}
```



Guess!

TEAMT5
杜 浦 數 位 安 全

```
int main() {
    constexpr std::string s{"abc"};
    return 0;
}
```

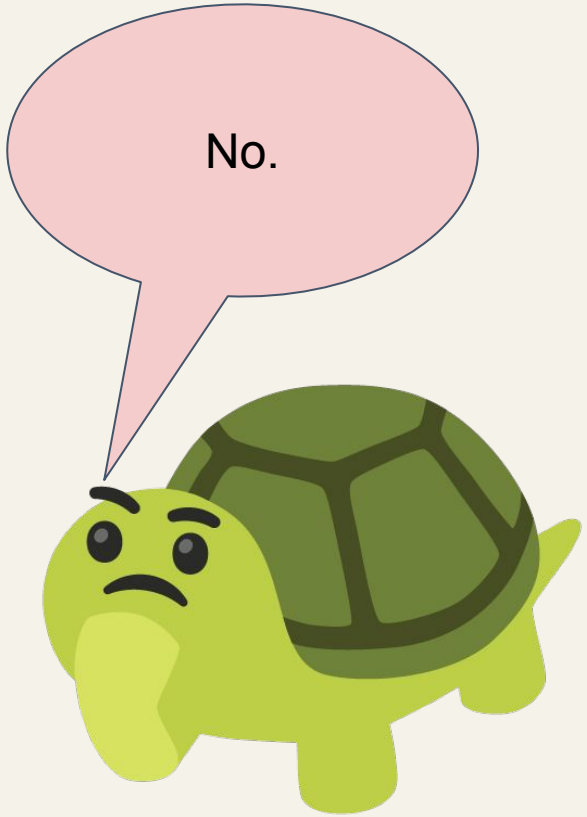


Guess!

TEAMT5
杜 浦 數 位 安 全

```
int main() {
    constexpr std::string s{"abc"};
    return 0;
}
```

Actually, it's here with run time deallocation.



constexpr std::string

TEAMT5
杜 浦 數 位 安 全

```
consteval auto hello(std::string_view name) {
    auto me = std::string("hello ") + std::string(name);
    return to_array(me.data(), me.size());
}

int main() {
    constexpr auto greet = hello("scc");
    for (const auto& c : greet)
        std::cout << c;
}
```

constexpr std::string

TEAMT5 杜浦數位安全

```
consteval auto hello(std::string_view name) {
    auto me = std::string("hello ") + std::string(name);
    return to_array(me.data(), me.size());
}

int main() {
    constexpr auto greet = hello("scc");
    for (const auto& c : greet)
        std::cout << c;
}
```

You need to freeze it in array.

constexpr std::string

TEAMT5 杜浦數位安全

```
consteval
    auto me
    return
}

int main()
    constex
    for (co
        sto
}
```

x86-64 gcc (trunk) -std=c++20 -O0

A Output... Filter... Libraries Overrides + Add new... Add tool...

```
11          mov     DWORD PTR [rdx], eax
12          add     rdx, 4
13          mov     BYTE PTR [rdx], al
14          add     rdx, 1
15          mov     BYTE PTR [rbp-288], 104
16          mov     BYTE PTR [rbp-287], 101
17          mov     BYTE PTR [rbp-286], 108
18          mov     BYTE PTR [rbp-285], 108
19          mov     BYTE PTR [rbp-284], 111
20          mov     BYTE PTR [rbp-283], 32
21          mov     BYTE PTR [rbp-282], 115
22          mov     BYTE PTR [rbp-281], 99
23          mov     BYTE PTR [rbp-280], 99
```

104 = 0x68 = 1.45735040e-43f = 'h'

```
ASM generation compiler returned: 0
Execution build compiler returned: 0
Program returned: 0
   hello scc
```

So, C++20 constexpr

Compile time memory allocation.

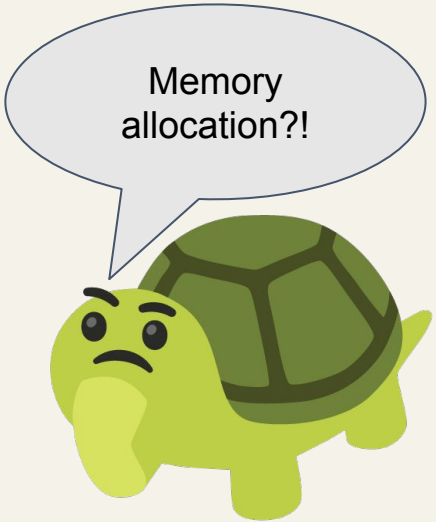
constexpr, consteval and constinit.

So, C++20 constexpr

TEAMT5
杜 浦 數 位 安 全

Compile time memory allocation.

constexpr, consteval and constinit.



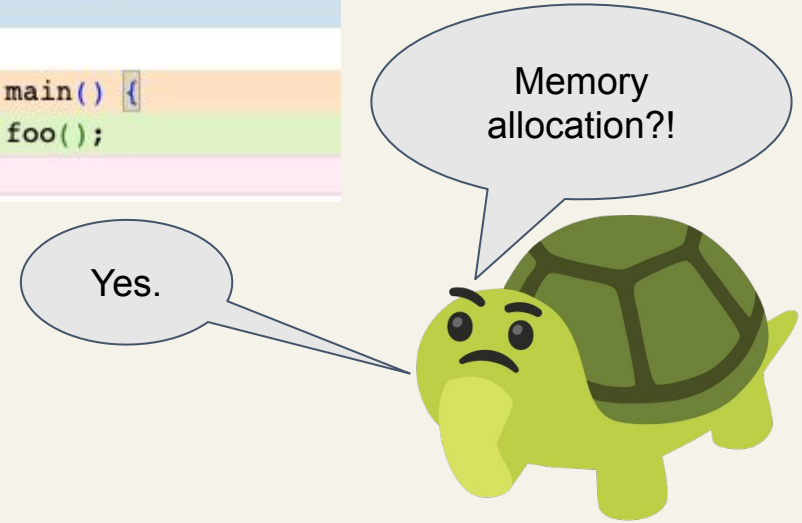
So, C++20 constexpr

TEAMT5
杜 浦 數 位 安 全

Compile time memory allocation.

constexpr, consteval and constinit.

```
3   constexpr void foo() {
4       auto ptr = new int;
5       *ptr = 42;
6       delete ptr;
7   }
8
9   int main() {
10      foo();
11  }
```



So, C++20 constexpr

TEAMT5 杜浦數位安全

```
1  struct Base {
2      constexpr virtual int hello() {return 1;}
3  };
4
5  struct Derived : Base {
6      constexpr virtual int hello() override {return 2;}
7  };
8
9  int main() {
10     static_assert(Derived{}.hello() != Base{}.hello());
11     return 0;
12 }
```

CRTP -> constexpr virtual

Compile time memory allocation.

constexpr, consteval and constinit.

Some try-catch(block only), and virtual functions.

C++23 constexpr

Introduce

TEAMT5 杜浦數位安全

- P1938R3 if consteval
- P1401R5 Narrowing contextual conversions in static_assert and constexpr if
- P2242R3 Non-literal variables (and labels and gotos) in constexpr functions
- P2448R2 Relaxing some constexpr restrictions
- P2647R1 Permitting static constexpr variables in constexpr functions
- P2564R0 consteval needs to propagate up
- P1328R1 constexpr type_info::operator==()
- P0533R9 constexpr for <cmath> and <cstdlib>
- P2273R3 constexpr std::unique_ptr
- P2291R3 constexpr for integral overloads of std::to_chars() and std::from_chars().
- P2417R2 constexpr std::bitset
- P2231R1 constexpr for std::optional and std::variant

if consteval

TEAMT5 杜浦數位安全

- P1938R3 if consteval
- P1401R5 Narrowing contextual conversions in static_assert and constexpr if
- P2242R3 Non-literal variables (and labels and gotos) in constexpr functions
- P2448R2 Relaxing some constexpr restrictions
- P2647R1 Permitting static constexpr variables in constexpr functions
- P2564R0 consteval needs to propagate up
- P1328R1 constexpr type_info::operator==()
- P0533R9 constexpr for <cmath> and <cstdlib>
- P2273R3 constexpr std::unique_ptr
- P2291R3 constexpr for integral overloads of std::to_chars() and std::from_chars().
- P2417R2 constexpr std::bitset
- P2231R1 constexpr for std::optional and std::variant

```
constexpr bool is_constant_evaluated() noexcept
{
    if consteval { return true; }
    else { return false; }
}
```

Non-literal relaxing

TEAMT5 杜浦數位安全

- P1938R3 if consteval
- P1401R5 Narrowing contextual conversions in static_assert and constexpr if
- P2242R3 Non-literal variables (and labels and gotos) in constexpr functions
- P2448R2 Relaxing some constexpr restrictions
- P2647R1 Permitting static constexpr variables in constexpr functions
- P2564R0 consteval needs to propagate up
- P1328R1 constexpr type_info::operator==()
- P0533R9 constexpr for <cmath> and <cstdlib>
- P2273R3 constexpr std::unique_ptr
- P2291R3 constexpr for integral overloads of std::to_chars() and std::from_chars().
- P2417R2 constexpr std::bitset
- P2231R1 constexpr for std::optional and std::variant

✅ goto
✅ void
function

Introducing const static data

- P1938R3 if consteval
- P1401R5 Narrowing contextual conversions in static_assert and constexpr if
- P2242R3 Non-literal variables (and labels and gotos) in constexpr functions
- P2448R2 Relaxing some constexpr restrictions
- P2647R1 Permitting static constexpr variables in constexpr functions
- P2564R0 consteval needs to propagate up
- P1328R1 constexpr type_info::operator==()
- P0533R9 constexpr for <cmath> and <cstdlib>
- P2273R3 constexpr std::unique_ptr
- P2291R3 constexpr for integral overloads of std::to_chars() and std::from_chars().
- P2417R2 constexpr std::bitset
- P2231R1 constexpr for std::optional and std::variant

It can use static variable
only if there is no side-effect.

Compile time memory ownership

- P1938R3 if consteval
- P1401R5 Narrowing contextual conversions in static_assert and constexpr if
- P2242R3 Non-literal variables (and labels and gotos) in constexpr functions
- P2448R2 Relaxing some constexpr restrictions
- P2647R1 Permitting static constexpr variables in constexpr functions
- P2564R0 consteval needs to propagate up
- P1328R1 constexpr type_info::operator==()
- P0533R9 constexpr for <cmath> and <cstdlib>
- P2273R3 constexpr std::unique_ptr
- P2291R3 constexpr for integral overloads of std::to_chars() and std::from_chars().
- P2417R2 constexpr std::bitset
- P2231R1 constexpr for std::optional and std::variant

constexpr std::allocation was introduced since C++20.
It guaranteed the ownership in C++23.

C++26 constexpr

(2c)

TEAMT5
杜 浦 數 位 安 全

C++26 constexpr

TEAMT5
杜 浦 數 位 安 全

- P2738R1 constexpr cast from void*
- P2562R1 constexpr stable sorting
- P1383R2 More constexpr for <cmath> and <complex>

Evolution history with proposal

Design and evolution of constexpr in C++

- C++98 and C++03: Ranks among const variables
- 10-x: Constant evaluator in compiler
  - Clang and LLVM
- 2003: No need for macros
  - Where are proposals located and what do they consist of?
- 2006-2007: When it all becomes clear
- 2007: First constexpr for data structures
- 2008: Recursive constexpr methods
- 2010: "const T&" as arguments in constexpr methods
- 2011: static_assert in constexpr methods
- 2012: (Almost) any code in constexpr functions
- 2013: (Almost) any code allowed in constexpr functions ver 2.0 Mutable Edition
- 2013: Legendary const methods and popular constexpr methods
- 2015-2016: Syntactic sugar for templates
- 2015: Constexpr lambdas
  - Proto-lambda for [](int x) { std::cout << x << std::endl; }
- 2017-2019: Double standards
- 2017-2019: We need to go deeper
- 2017: The evil twin of the standard library
- 2017-2019: Constexpr gains memory
- 2018: Catch me if you can
- 2018: I said constexpr!
- 2018: Too radical constexpr
- 2020: Long-lasting constexpr memory
- 2021: Constexpr classes
- 2019-∞: Constant interpreter in the compiler
- What else to look?

https://pvs-studio.com/en/blog/posts/cpp/0909/

Thank you

scc@teamt5.org

TEAMT5

杜 浦 數 位 安 全

Persistent Cyber Threat Hunters