

Вычисления на этапе компиляции

Проблема

Известно, что в качестве размера C-style массива или параметра шаблона могут выступать только константы, значения которых известны на этапе компиляции.
(константные выражения)

```
int main() {
    const int n = 100;
    int c_style[n]; // Ok
    std::array<n * n + 32> arr; // Ok
    return 0;
}
```

Проблема

Константное выражение (*constant expression*) - выражение, которое **может быть** вычислено на этапе компиляции.

Например, в качестве размера C-style массива или параметра шаблона могут выступать только константные выражения.

```
int main() {
    const int n = 100;
    int c_style[n]; // Ok
    std::array<int, n * n + 32> arr; // Ok
    return 0;
}
```

Проблема

А что если хочется передать не константу, а значение функции от константы?

```
int64_t Pow(int64_t x, int n); // x^n

int main() {
    const int n = 10;
    int c_style[Pow(3, n)]; // CE
    std::array<int, Pow(5, n)> arr; // CE
    return 0;
}
```

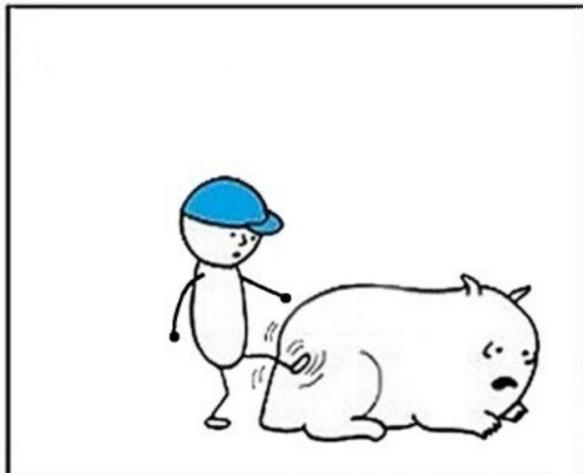
```
main.cpp: error: ISO C++ forbids variable length array 'c_style' [-Werror=vla]
    int c_style[Pow(3, n)]; // CE
main.cpp: error: call to non-'constexpr' function 'int Pow(int, int)'
    std::array<int, Pow(5, n)> arr; // CE
```

Проблема

А что если хочется передать не константу, а значение функции от константы?

```
main.cpp: error: ISO C++ forbids variable length array 'c_style' [-Werror=vla]
    int c_style[Pow(3, n)]; // CE
main.cpp: error: call to non-'constexpr' function 'int Pow(int, int)'
    std::array<int, Pow(5, n)> arr; // CE
```

Как заставить компилятор вычислять самостоятельно?



Классические compile-time вычисления (C++03)

Подготовка

Заведем специальный шаблонный класс, который хранит в себе определенную целочисленную константу:

```
template <class Integer, Integer Value>
struct IntegralConstant {
    static const Integer kValue = Value;
    // ...
};

int main() {
    int c_style[IntegralConstant<int, 100>::kValue];
    std::array<int, IntegralConstant<int, 100>::kValue> arr;
}
```



Подготовка

А лучше воспользуемся готовым из стандартной библиотеки

```
#include <type_traits> // std::integral_constant

int main() {
    int c_style[std::integral_constant<int, 100>::value];
    std::array<int, std::integral_constant<int, 100>::value> arr;
}
```



Идея

Что гарантированно происходит на этапе компиляции?

Правильно! - Подстановка шаблонных параметров.

То есть компилятор обязан вычислить значение шаблонного параметра, иначе у него не получится инстанцировать шаблонный класс.

```
template <int N>
struct Square : std::integral_constant<int, N * N> {};
// теперь Square хранит в поле value значение квадрата N (известное на этапе компиляции!)
```

```
int main() {
    int c_style[Square<100>::value]; // ok
    std::array<int, Square<100>::value> arr; // ok
}
```

Пример (факториал)

```
template <size_t N>
struct Factorial
: std::integral_constant<size_t, N * Factorial<N - 1>::value> {};
```

Видите ли вы здесь проблемы?

Пример (факториал)

```
template <size_t N>
struct Factorial
: std::integral_constant<size_t, N * Factorial<N - 1>::value> {};
```

Проблема 1: отсутствие конца рекурсии.

Решение: добавим специализацию шаблонаю

```
template <>
struct Factorial<0> : std::integral_constant<size_t, 1> {};
```

Пример (факториал)

Проблема 2: `Factorial<N>::value` выглядит слишком громоздко. Заведем шаблонную переменную для хранения этого значения:

```
template <size_t N>
struct Factorial; // forward declaration

template <size_t N>
inline const size_t kFactorialV = Factorial<N>::value; // template variable

template <size_t N>
struct Factorial : std::integral_constant<size_t, N * kFactorialV<N - 1>> {};

template <>
struct Factorial<0> : std::integral_constant<size_t, 1> {};
```

```
int main() {
    int c_style[kFactorialV<8>];
    std::array<int, kFactorialV<10>> arr;
}
```

Пример (Фибоначчи)

```
template <size_t N>
struct Fibonacci; // forward declaration

template <size_t N>
inline const size_t kFibonacciV = Fibonacci<N>::value;

template <size_t N>
struct Fibonacci
: std::integral_constant<size_t, kFibonacciV<N - 1> + kFibonacciV<N - 2>> {};

template <>
struct Fibonacci<0> : std::integral_constant<size_t, 0> {};

template <>
struct Fibonacci<1> : std::integral_constant<size_t, 1> {};
```

Есть ли тут проблемы?

Пример (Фибоначчи)

```
template <size_t N>
struct Fibonacci; // forward declaration

template <size_t N>
inline const size_t kFibonacciV = Fibonacci<N>::value;

template <size_t N>
struct Fibonacci
: std::integral_constant<size_t, kFibonacciV<N - 1> + kFibonacciV<N - 2>> {};

template <>
struct Fibonacci<0> : std::integral_constant<size_t, 0> {};

template <>
struct Fibonacci<1> : std::integral_constant<size_t, 1> {};
```

Вычисляется за $O(N)$ на этапе компиляции! Так как шаблон с одним и тем же набором параметров не инстанцируется дважды.

Во время исполнения $O(1)$, так как просто подставляется готовое значение.

Пример (быстрое возведение в степень)

```
template <int64_t X, size_t N>
struct Pow;

template <int64_t X, size_t N>
inline const int64_t kPowV = Pow<X, N>::value;

template <int64_t X, size_t N>
struct Pow
: std::integral_constant<int64_t, N % 2 == 0 ? kPowV<X * X, N / 2> :
X * kPowV<X * X, (N - 1) / 2>> {};

template <int64_t X>
struct Pow<X, 0> : std::integral_constant<int64_t, 1> {};
```

```
int main() {
    int c_style[kPowV<3, 10>];
    std::array<int, kPowV<5, 10>> arr;
}
```

Современные compile-time вычисления (C++11)



`constexpr` функции

`constexpr` функция - функция, которая может быть вычислена на этапе компиляции.

```
constexpr size_t Factorial(size_t n) {
    size_t res = 1;
    for (; n > 1; --n) {
        res *= n;
    }
    return res;
}

int main() {
    int c_style[Factorial(5)]; // Ok
    std::array<int, Factorial(3)> arr; // Ok
}
```

`constexpr` функции

Свойства `constexpr` функций из C++17:

- Автоматически являются `inline` функциями
- Не могут быть виртуальными (до C++20)
- Не могут иметь try-catch блок (до C++20)
- Не могут содержать ассемблерные вставки (до C++20)
- Не могут использовать операции с динамической памятью (до C++20)
- Не могут содержать `goto` (до C++23)
- Могут определять только нестатические литеральные переменные (до C++23)
- При вычислении на этапе компиляции UB = CE, исключение = CE

constexpr функции

```
constexpr size_t Fibonacci(size_t n) {
    if (n <= 1) {
        return n;
    }
    size_t prev = 0;
    size_t res = 1;
    for (; n > 1; --n) {
        res += prev;
        prev = res - prev;
    }
    return res;
}

constexpr int64_t Pow(int64_t x, size_t n) {
    if (n == 0) {
        return 1;
    }
    return (n % 2 == 0 ? 1 : x) * Pow(x * x, n / 2); // рекурсия допустима
}
```

constexpr функции

В C++20 можно и так!

```
constexpr size_t Fibonacci(size_t n) {
    auto fib = new size_t[n + 1];
    fib[0] = 0;
    fib[1] = 1;
    for (size_t i = 2; i <= n; ++i) {
        fib[i] = fib[i - 1] + fib[i - 2];
    }
    auto res = fib[n];
    delete[] fib; // если забудете это, то будет СЕ!
    return res;
}
```

```
constexpr size_t Fibonacci(size_t n) {
    std::vector<size_t> v(n);
    v[0] = 0;
    v[1] = 1;
    // ...
}
```

Литеральные типы

В `constexpr` функции можно определять только переменные литературального типа.

Литеральный тип (неформально) - либо примитивный тип, либо тип с `constexpr` конструктором (кроме конструктора копирования и перемещения) и деструктором, либо агрегат из литературальных типов.

```
int x = 0; // Ok

struct S {
    int a;
    double b;
};

S y{}; // Ok

std::array<int, 5> z{}; // Ok

std::vector<int> t; // CE (Ok в C++20)
```

consteval (C++20)

Если `constexpr` функция вызывается в контексте, где ожидается константное выражение, то ее значение будет вычислено на этапе компиляции.

Но! В прочих ситуациях тот факт, что функция **может быть** вычислена на этапе компиляции, не гарантирует, что она **будет вычислена** во время компиляции.

```
constexpr int64_t Pow(int64_t x, size_t n);

int main() {
    int c_style[Pow(3, 5)]; // Будет вычислено на этапе компиляции
    std::cout << Pow(5, 3) << '\n'; // Скорее всего, будет вызвана во время исполнения
    std::cout << Pow(std::rand(), 2) << '\n'; // Будет вызвана во время исполнения
}
```



consteval (C++20)

Чтобы гарантировать вычисление на этапе компиляции, можно воспользоваться `consteval`.

```
consteval int64_t Pow(int64_t x, size_t n);

int main() {
    int c_style[Pow(3, 5)]; // Будет вычислено на этапе компиляции
    std::cout << Pow(5, 3) << '\n'; // Будет вычислено на этапе компиляции
    std::cout << Pow(std::rand(), 2) << '\n'; // СЕ
}
```



`constexpr` переменные

constexpr переменные

constexpr переменная - константная переменная, значение которой известно на этапе компиляции.

```
constexpr int64_t x = 3;
constexpr auto x5 = Pow(x, 5);

x = 5; // CE

constexpr int y; // CE
constexpr int z = std::rand(); // CE
```

`constexpr` переменные

Свойства `constexpr` переменных:

- переменная должна быть литерального типа
- она должна быть проинициализирована константным выражением
- являются константами

constexpr переменные

А что если хочется проинициализировать константой времени компиляции, но при этом не делать саму переменную константой?

```
constexpr int64_t Pow(int64_t x, size_t n);  
// ...  
auto x = Pow(3, 5); // не гарантируется вычисление Pow в compile-time
```

constinit переменные (C++20)

`constinit` гарантирует, что начальное значение переменной будет вычислено на этапе компиляции. При этом сама переменная константой не считается.

```
constexpr int64_t Pow(int64_t x, size_t n);  
// ...  
  
constinit auto x = Pow(3, 5); // гарантируется вычисление Pow в compile-time  
x = 10; // Ok  
std::cin >> x; // Ok
```

