

КОНСТАНТНЫЕ ВЫРАЖЕНИЯ

Переменные, функции и объекты времени компиляции.
Пользовательские суффиксы.

К. Владимиров, Syntacore, 2024
mail-to: konstantin.vladimirov@gmail.com

➤ Константность

- ❑ Функции времени компиляции

- ❑ ООП времени компиляции

- ❑ Снова о метапрограммировании

Обсуждение

- В чём смысл следующей конструкции и где она может быть применима?

```
uint8_t const volatile * const p_latch_reg = (uint8_t *) 0x42;
```

const означает readonly

- В чём смысл следующей конструкции и где она может быть применима?

```
uint8_t const volatile * const p_latch_reg = (uint8_t *) 0x42;
```

- Это проводок с заданным адресом, с которого можно считать данные но не изменить их.
- При этом сами данные могут непредсказуемо измениться, так что доступ к ним нельзя оптимизировать.

```
data = *p_latch_reg; // считали значение
```

```
.....
```

```
data = *p_latch_reg; // снова считали значение
```

Плавающая константность

- Известность чего-либо на этапе компиляции может плавать в зависимости от организации кода:

```
struct S2 { static const int sz; };  
const int page_sz2 = 4 * S2::sz;  
const int S2::sz = 256;  
int arr2[page_sz2]; // error: not CT constant
```

- Но при этом

```
struct S1 { static const int sz = 256; };  
const int page_sz1 = 4 * S1::sz;  
int arr1[page_sz1]; // ok, CT constant
```

Что известно на этапе компиляции

- Литералы (в том числе введенные через макросы) и члены enum.
- Параметры шаблонов и результаты sizeof над типами.
- constexpr переменные.

```
struct S2 { static const int sz; };  
  
// error: S2::sz not initialized  
constexpr int page_sz2 = 4 * S2::sz;  
  
constexpr int S2::sz = 256;  
int arr2[page_sz2];
```

Ограничение на constexpr переменные

- constexpr переменная должна иметь **литеральный тип**.
- Коротко говоря, литеральный тип это тип у которого есть литералы.
Пример: 1, "hello", 'c', 1.0, 1ull.
- Использовать constexprs с плавающей точкой можно, но не рекомендуется.

```
constexpr float ct = 1.0f / 3.0f;  
assert(x == 1.0f && y == 3.0f);
```

```
float rt = x / y;  
assert(rt == ct); // ONLY?
```

CONSTEXPR означает CONST?

- Следующий случай может быть несколько не очевиден:

```
constexpr int arr[] = {2, 3, 5, 7, 11};
```

```
constexpr int * x = &arr[3]; // всё хорошо?
```

- Тут зависит от того, к чему относится constexpr во второй строчке. Варианта, собственно, два.

1. `constexpr int * x` → `const int * x`

2. `constexpr int * x` → `int * const x`

- Обсуждение: давайте проголосуем?

CONSTEXPR означает CONST?

- Следующий случай может быть несколько не очевиден:

```
constexpr int arr[] = {2, 3, 5, 7, 11};
```

```
constexpr const int * x = &arr[3]; // теперь всё хорошо
```

- Тут зависит от того, к чему относится constexpr во второй строчке. Варианта, собственно, два.

1. `constexpr int * x` → `const int * x`

2. `constexpr int * x` → `int * const x`

- Обсуждение: давайте проголосуем?
- Второй вариант семантически консистентен: мы объявили constexpr pointer.

Контрольный вопрос

- Как вы думаете, имеют ли смысл нестатические constexpr данные внутри класса?

```
struct Foo { constexpr int x; }; // ok?
```

- Например const в этой же записи имеет вполне осмысленную семантику.

Ответ

- Как вы думаете, имеют ли смысл нестатические constexpr данные внутри класса?

```
struct Foo { constexpr int x; }; // fail
```

- Например const в этой же записи имеет вполне осмысленную семантику.
- Увы, по определению, нестатические поля неизвестны на этапе компиляции.

C++17: constexpr control flow

- Возможность использования выражений времени компиляции делает интересным вопрос переключения по ним.

```
constexpr bool b = true;
```

```
if (b) { тут много кода } else { тут ещё больше кода }
```

- В этом случае компилятор будет вынужден транслировать код, находящийся под ложным условием
- Хотелось бы **ленивого поведения** для такого control flow

C++17: constexpr control flow

- Возможность использования выражений времени компиляции делает интересным вопрос переключения по ним.

```
constexpr bool b = true;
```

```
if constexpr (b) {  
    // тут много кода  
}
```

```
else {  
    // теперь совершенно не важно что тут  
}
```

- Начиная с C++17 такое ленивое поведение предоставляет `if constexpr`

Альтернатива SFINAE

- На прошлой лекции мы *хлебнули лиха* с правильным sfinae-outing функций

```
template <typename T> enable_if_t<(sizeof(T) > 4)>  
foo (T x) { сделать что-то с x }
```

```
template <typename T> enable_if_t<(sizeof(T) <= 4)>  
foo (T x) { сделать что-то ещё с x }
```

- Кажется, теперь появился иной вариант...

```
template <typename T> void  
foo (T x) {  
    if constexpr (sizeof(T) > 4) { сделать что-то с x }  
    else { сделать что-то ещё с x }  
}
```

Ограничения if constexpr

- Мы должны очень хорошо помнить: if constexpr выкидывает ветки шаблонных инстанцирований.

```
template <typename T> void  
foo (T x) {  
    if constexpr (sizeof(T) > 4) {  
        тут проверка первой фазы и может быть второй  
    }  
    тут проверка первой и второй фазы  
}
```

Обсуждение

- Что вы думаете насчёт идеи `for constexpr` для замены шаблонной рекурсии в метапрограммировании?

❑ Константность

➤ Функции времени компиляции

❑ ООП времени компиляции

❑ Снова о метапрограммировании

Снова о метапрограммах

- Простая задача: возведение в квадрат времени компиляции

```
template <size_t n> square: integral_constant <size_t, n * n>;  
int arr[square<5>{}]; // arr[25]
```

- Тут угадать, что `square` на самом деле функтор – довольно сложно

Снова о метапрограммах

- Простая задача: возведение в квадрат времени компиляции

```
template <size_t n> square: integral_constant <size_t, n * n>;  
int arr[square<5>{}]; // arr[25]
```

- Тут угадать, что square на самом деле функтор – довольно сложно

```
constexpr int square(int x) { return x * x; }  
int arr[square(5)]; // ok, arr[25]
```

- Теперь очевидно, что мы вызываем функцию времени компиляции
- Стандарт накладывает некоторые ограничения на тела таких функций

Ограничения в C++11

- Параметры и возвращаемые значения должны быть литеральными типами
- Тело это одно выражение { return expression; } без побочных эффектов и с вызовом внутри только constexpr функций

```
constexpr size_t static_log_helper (size_t N, size_t pos) {  
    return ((N & (1ull << pos)) == (1ull << pos) || pos == 0) ?  
        (N == (1ull << pos) ? pos : pos + 1) :  
        static_log_helper (N, pos - 1);  
}
```

```
constexpr size_t static_log (size_t N) {  
    return static_log_helper (N, sizeof(size_t) * CHAR_BIT - 1);  
}
```

Обработка ошибок

- Запрет на побочные эффекты имеет приятные побочные эффекты.
- Части constexpr функции включаются лениво, что порождает throw idiom.

```
constexpr size_t static_log_helper (size_t N, size_t pos) {  
    // тут всё как раньше  
}  
  
constexpr size_t static_log (size_t N) {  
    return (N != 0) ?  
        static_log_helper (N, sizeof(size_t) * CHAR_BIT - 1) :  
        throw "N == 0 not supported";  
}
```

Ограничения в C++14

- `new` и `delete`
- Генерация исключений через `throw`
- Вызов не-constexpr функций
- Использование `goto`
- Лямбда выражения
- Преобразования `const_cast` и `reinterpret_cast`
- Преобразования `void*` в `object*`
- Модификация нелокальных объектов
- Неинициализированные данные
- Сравнения с `unspecified` результатом
- Вызов `typeid` для полиморфных классов и `dynamic_cast`
- Блоки `try` для обработки исключений
- Операции с `undefined behavior`
- Инлайн ассемблер во всех разновидностях
- Большая часть операций с `this`

Пример: логарифм в C++14

```
constexpr size_t int_log (size_t N) {  
    size_t pos = sizeof(size_t) * CHAR_BIT, mask = 0;  
    if (N == 0) throw "N == 0 not supported";  
    do {  
        pos -= 1;  
        mask = 1ull << pos;  
    } while ((N & mask) != mask);  
    if (N != mask) pos += 1;  
    return pos;  
}
```

Самостоятельное исследование

- Что скомпилируется быстрее: этот логарифм или его брат-близнец написанный на шаблонах?

Не всегда constexpr

- Логичный вопрос: можно ли перегрузить функцию по constexpr, чтобы иметь и статический и нестатический вариант static_log?
- Ответ немного удивителен: статический вариант может быть использован с неизвестным на этапе компиляции аргументом.

```
cin >> x;
```

```
cout << static_log(x) << endl;
```

- Поэтому constexpr не входит в тип функции и не может аннотировать параметры.

Обсуждение: гарантии

- Можем ли мы каким-то образом гарантировать, что constexpr функция выполнялась во время компиляции?

```
int t = static_log(5);
```

- Законных оснований надеяться на это здесь у нас нет.

Старый вариант: через constexpr

- Можем ли мы каким-то образом гарантировать, что constexpr функция выполнялась во время компиляции?
- Решение: использовать в compile-time контексте (положить в constexpr переменную, сделать размером массива, параметризовать шаблон).

```
constexpr int logval = static_log(5);
```

```
int t = logval;
```

- Теперь мы уверены, что вызов состоялся на этапе компиляции.

C++20, введение constexpr

- Функции, помеченные constexpr обязаны быть выполнены именно и конкретно на этапе компиляции.

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

```
constexpr int r = ctsqr(100); // OK
```

```
int x = 100; int r2 = ctsqr(x); // Ошибка: не const
```

- Поэтому:

```
constexpr size_t int_log_ce(size_t N) {  
    // тут наш логарифм, если он плох на этапе выполнения  
}
```

C++20, введение `constinit`

- Для того чтобы гарантировать только константную инициализацию `constexpr` наоборот слишком сильная гарантия и достаточно `constinit`

```
constinit int x = 1000; // запрещено для локальных переменных  
++x; // OK
```

Обсуждение

- Мы хотели бы изнутри СЕ функции понимать находимся мы на этапе компиляции или нет.

```
constexpr size_t int_log(size_t N) {  
    если мы на этапе компиляции {  
        return int_log_ce(N);  
    } иначе {  
        return __builtin_ctz(N);  
    }  
}
```

- Как бы вы этого добились в ранних версиях стандарта (до 23-го года)?

C++23: if consteval

- Мы хотели бы изнутри CE функции понимать находимся мы на этапе компиляции или нет.

```
constexpr size_t int_log(size_t N) {  
    if consteval {  
        return int_log_ce(N);  
    } else {  
        return __builtin_ctz(N);  
    }  
}
```

- Это почти волшебство.

Не везде constexpr

- Двойная природа constexpr функций имеет обратную сторону

```
template <typename T>
constexpr size_t ilist_sz(initializer_list<T> init) {
    constexpr size_t init_sz = init.size();
    return init_sz;
}
```

- Это ошибка. Компилятор тут не может дать **гарантию** константности для переменной (хотя сама функция и constexpr).
- Как вы думаете, изменится ли ситуация если я заменю на consteval?
- А если я убегу отмеченное красным?

Не везде constexpr

- Двойная природа constexpr функций имеет обратную сторону

```
template <typename T>
constexpr size_t ilist_sz(initializer_list<T> init) {
    size_t init_sz = init.size();
    return init_sz;
}
```

- Теперь всё хорошо. И, более того, теперь результат может жить в constexpr переменной!

```
constexpr size_t s = ilist_sz({1, 2, 3}); // ok!
```

Ещё один показательный случай

- Мы всегда должны смотреть является ли **выражение** constexpr.

```
constexpr bool negate(bool x) { return !x; }
```

```
template <typename P> constexpr int f(P pred) {  
    if constexpr(pred(true)) // не является  
        return 1;  
    return 0;  
}
```

```
constexpr int x = f(negate);
```

- Компилятор ориентируется на синтаксис.

Случаи UB для constexpr функций

- В constexpr функциях на этапе компиляции запрещено UB.

```
template <typename FwdIt, typename Value>
constexpr FwdIt static_find (FwdIt it, FwdIt fin, Value v) {
    while ((v != *it) && (it != fin)) ++it
    return it;
}
```

```
int a[] = {1};
```

```
static_find(a, a + 1, 4); // array subscript value '1' is
                           outside the bounds of array
```

- Это до некоторой степени делает constexpr функции UB-санитайзером.

Более интересный случай

- Следующий пример UB в C++17 но не в C++20 и constexpr это отслеживает.

```
constexpr int djb2(char const *str) {  
    int hash = 5381;  
    int c = 0;  
  
    while ((c = *str++))  
        hash = ((hash << 5) + hash) + c;  
  
    return hash;  
}  
  
constexpr int x = djb2("hello you :)"); // overflow in shift
```

Case study: триты

- Тритами называются цифры сбалансированной системы счисления по основанию 3, т.е. $\{-1, 0, 1\}$. Обозначим -1 как j . Тогда:
- $10j = 8$
- $j01 = -8$
- $11j0.jj = ???$

Пример: триты

- Тритами называются цифры сбалансированной системы счисления по основанию 3, т.е. $\{-1, 0, 1\}$. Обозначим -1 как j . Тогда:
- $10j = 8$
- $j01 = -8$
- $11j0.jj = 32\frac{5}{9}$
- $11j0 = 33$
- Триты имеют ряд привлекательных свойств: отрицание числа это просто флип с 1 на -1 , а отбрасывание дробной части всегда округляет к ближайшему целому

Триты в вашей программе

- Было бы здорово оперировать тритовыми константами как будто это обычные числа времени компиляции

```
constexpr int u = ct_trit<int>("10j");
```

```
constexpr int v = ct_trit<int>("j01");
```

```
cout << u << " : " << v << endl;
```

- Тут на экране должно быть 8 и -8
- В этом случае функция `ct_trit` должна быть полноценным парсером времени компиляции

Триты в вашей программе

- Её упрощённая реализация на C++14 (тут многого не хватает)

```
template <typename T = int> constexpr T ct_trit(const char* t) {  
    T x = 0;  
    size_t pos = 0;  
  
    for (size_t i = 0; t[i] != '\0'; ++i)  
        switch (t[i]) {  
            case '0': x = (x * 3); ++pos; break;  
            case '1': x = (x * 3) + 1; ++pos; break;  
            case 'j': x = (x * 3) - 1; ++pos; break;  
            default: throw "Only '0', '1', and 'j' may be used";  
        }  
}
```

- Мы не особо заботимся о переполнении.

Задача

- Определитель размера массива

```
int keyVals[] = {2, 3, 5, 7, 11, 13};
```

```
int mapped[arraySize(keyvals)]; // тот же размер
```

- Сишный вариант с макросом (так себе)

```
#define arraySize(x) (sizeof(x) / sizeof(x[0]))
```

Задача

- Определитель размера массива

```
int keyVals[] = {2, 3, 5, 7, 11, 13};
```

```
int mapped[arraySize(keyvals)]; // тот же размер
```

- Сишный вариант с макросом (так себе)

```
#define arraySize(x) (sizeof(x) / sizeof(x[0]))
```

- Вариант Майерса

```
template <typename T, size_t N>  
constexpr size_t arraySize (T(&)[N]) { return N; }
```

Использование в switch

- Константно-выраженные функции могут порождать результаты используемые в switch-cases и более того могут быть операторами

```
enum class bitmask { b0 = 0x1, b1 = 0x2, b2 = 0x4, и т.д. };
```

```
constexpr bitmask operator | (bitmask v0, bitmask v1) {  
    return bitmask(int(v0) | int(v1));  
}
```

```
int foo (bitmask b) {  
    switch(b) {  
        case bitmask::b0 | bitmask::b1 : сделать нечто
```

```
..... и так далее .....
```

Обсуждение

- Имеют ли смысл нестатические `constexpr` методы в классах?

Core constant expressions

- Всё, что касается constexpr, полно сложных и странных сюрпризов.

```
struct S {  
    int n_;  
    S(int n) : n_(n) {}  
    constexpr int get() { return 42; }  
};
```

```
int main() {  
    S s{2};  
    constexpr int k = s.get();  
}
```

Обсуждение

- Имеют ли ещё какой-то смысл нестатические constexpr методы в классах?

- ❑ Константность

- ❑ Функции времени компиляции

- ООП времени компиляции

- ❑ Снова о метапрограммировании

Пользовательские литеральные типы

- Чтобы сделать пользовательский тип литеральным, ему нужен `constexpr` конструктор (который и делает осмысленными прочие нестатические `constexpr` методы)

```
struct Complex{  
    constexpr Complex(double r, double i) : re(r), im(i) {}  
    constexpr double real() const { return re;}  
    constexpr double imag() const { return im;}  
private:  
    double re, im;  
};
```

```
constexpr Complex c(0.0, 1.0); // это литеральное значение
```


Арифметика

- Для таких объектов становится возможной арифметика времени компиляции

```
constexpr Complex& Complex::operator+= (Complex rhs) {  
    re += rhs.re; im += rhs.im; return *this;  
}
```

```
constexpr Complex operator+ (Complex lhs, Complex rhs){  
    lhs += rhs; return lhs;  
}
```

- Использование:

```
constexpr Complex c(0.0, 1.0), d(1.0, 2.0);  
constexpr Complex e = c + d;
```

Контейнеры

- Многие стандартные контейнеры имеют constexpr конструкторы и их можно использовать на этапе компиляции

```
constexpr std::array<size_t, 5> arr = {0, 4, 2, 1, 3};  
constexpr size_t four = arr[3] + arr[4];  
static_assert(four == 4); // ok
```

- Упражнение: написать функцию cycle_elems которая берёт array и число N и после этого N раз выдаёт все элементы этого массива в результирующий

```
constexpr array<size_t, 5> arr = {0, 4, 2, 1, 3};  
constexpr auto cycled = cycle_elems<3>(arr);  
  
// cycled == {0, 4, 2, 1, 3, 0, 4, 2, 1, 3, 0, 4, 2, 1, 3}
```

Контейнеры

- Очевидное решение:

```
template <size_t N, typename T, size_t Size>
constexpr array <T, N * Size>
cycle_elems (array<T, Size> a) {
    array <T, N * Size> result {};
    for (T i = 0; i < N * Size ; ++i)
        result[i] = a[i % Size];
    return result;
}
```

- Работает только для C++17, но в случае C++14 выдаёт ошибку:

call to non-constexpr function 'array<_Tp, _Nm>::operator[]'

Контейнер своими руками

```
template <typename T, size_t N> class array_result {  
    constexpr static size_t size_ = N;  
    T data_[N] {};  
public:  
    template <typename ... Ts>  
    constexpr array_result (Ts ... ints) : data_ {ints ...} {}  
    constexpr size_t size() const { return N; }  
    constexpr T& operator[](size_t n) { return data_[n]; }  
    using iterator = const T*;  
    constexpr iterator begin() const { return data_; }  
    constexpr iterator end() const { return data_ + N; }  
};
```

Обсуждение

- Мы явно сделали нечто странное: non-const constexpr метод

```
template <typename T, size_t N> class array_result {  
    T data_[N] {};  
public:  
    constexpr T& operator[](size_t n) { return data_[n]; }  
  
..... и так далее .....  
};
```

- Он изменяет состояние на этапе компиляции.

Конкатенация строк

- Внезапно мы можем смешивать указатели в том числе на внутренности контейнера.

```
template <size_t N1, size_t N2> constexpr auto
concat(char const (&a)[N1], char const (&b)[N2]) {
    std::array<char, N1 + N2 - 1> result = {};
    char *next = strncpy(result.data(), a, N1 - 1);
    char *nextdst = next ? next : &result[N1 - 1];
    strncpy(nextdst, b, N2);
    result.back() = '\0';
    return result;
}
```

Обсуждение

- Почему идея виртуальной функции на этапе компиляции нас так раздражает?

Виртуальные constexpr функции

- Почему идея виртуальной функции на этапе компиляции нас так раздражает?
- Потому что виртуальные функции это механизм стирания типов

```
struct Base { constexpr Base() = default; };
```

```
struct Derived : Base { constexpr Derived() = default; };
```

- Очевидно взятие адреса от constexpr объекта не является constant expression

```
constexpr Derived d;
```

```
constexpr const Base *pb = &d; // error
```

- И зачем же тогда это всё?

Виртуальные constexpr функции

- Ради использования внутри одного constexpr контекста

```
struct Base // нет constexpr ctor
{ virtual constexpr int data() const { return 1; } };

struct Derived : Base
{ constexpr int data() const override { return 2; } };

constexpr int foo() {
    const Base b;
    const Derived d;
    const Base *bases[] = {&b, &d};
    // обычное использование внутри функции
};
```

Обсуждение

- Таким образом введённые в C++20 virtual constexpr функции это псевдо-стирание типов.

```
const Base *bases[] = {&b, &d};
```

- Если эта функция и впрямь выполняется во время компиляции, компилятор отлично запомнит типы.

```
for (auto pb : bases)  
    sum += pb->data();
```

- Этот вызов **не будет** вызовом через vtable.
- Ловкость рук, никакого мошенничества.



Больше фокусов: constexpr new

- [expr.const] [E не является constexpr если его частью является] a new-expression, unless the selected allocation function is a replaceable global allocation function and the allocated storage is deallocated within the evaluation of E.
- То есть следующее будет работать и работает.

```
constexpr int test() {  
    int *p = new int(2); // почему бы не std::vector?  
    int k = *p;  
    delete p;  
    return k;  
}
```

Снова комплексные числа

- Ранее был рассмотрен класс для комплексных чисел времени компиляции

```
struct Complex{  
    constexpr Complex(double r, double i) : re(r), im(i) { }  
    constexpr double real() const { return re;}  
    constexpr double imag() const { return im;}  
    constexpr operator+= (Complex rhs);  
    // ..... и так далее .....  
};
```

- Но константы такого класса выглядят как `Complex(1.0, 1.0)` вместо более привычной формы `1.0 + 1.0i`

Пользовательский суффикс

- Ранее был рассмотрен класс для комплексных чисел времени компиляции.

```
struct Complex{  
    constexpr Complex(double r, double i) : re(r), im(i) { }  
    // и так далее  
};  
  
constexpr Complex operator "" _i (long double arg){  
    return Complex (0.0, arg);  
}  
  
constexpr Complex c = 0.0 + 1.0_i; // ok, arg_i → ""_i(arg)
```

- Здесь суффикс определён с параметром типа double.

Внезапная проблема

- Допустим, хочется переопределить суффикс `_binary` для бинарных констант.
- Но уже даже довольно маленькая константа: `1010101010101_binary` не влазит в `unsigned long long` параметр.

Вариабельные суффиксы

- Допустим, хочется переопределить суффикс `_binary` для бинарных констант
- Решение: переменный суффикс

```
template<char... Chars>  
constexpr unsigned long long operator "" _binary() {  
    return binparser(0ull, Chars...);  
}
```

- Как может выглядеть и что делать binparser?

Binparser: constexpr вариант

```
template<typename ... Ts> constexpr unsigned long long
binparser (unsigned long long accum, char c, Ts ... cs) {
    unsigned digit = (c == '1') ? 1 :
                     (c == '0') ? 0 : throw "out of range";
    if constexpr (sizeof...(Ts) != 0)
        return binparser(accum * 2 + digit, cs...);
    return accum * 2 + digit;
}

template<char... Chars>
constexpr unsigned long long operator "" _binary() {
    return binparser(0ull, Chars...);
}
```


Ещё одна проблема

- Увы, этот подход не работает для тритов.

Попытка применить: `10j01_trit` заставляет компилятор рассматривать `j01_trit` как суффикс.

- Более общий подход: строковые литералы.

```
constexpr long long  
operator "" _trit(char const *s, size_t len) {  
    return ct_trit<long long>(s);  
}
```

```
constexpr long long n = "10j01"_trit;
```

- Функция `ct_trit` уже была написана ранее.

Обсуждение

- Предложите ваши варианты использования пользовательских литералов.

Физические величины

- Цель: работа с физическими величинами с контролем размерности на этапе компиляции.

```
Speed sp1 = 100_m/9.8_s;           // ok
Speed sp2 = 100_m/9.8_s2;          // ошибка (m/s2 это ускорение)
Speed sp3 = 100/9.8_s;              // ошибка (1/s это частота)
Acceleration acc = sp1/0.5_s;      // ok
```

- Идея для решения: единица измерения как enum.

```
template<int M, int K, int S> struct Unit {
    enum { m=M, kg=K, s=S };
};
```

Физические величины

- Тогда собственно физическая величина это число плюс единица измерения.

```
template<typename Unit> struct Value {  
    double val;  
    explicit Value(double d) : val(d) {}  
};
```

- Например, распространённые величины.

```
using Meter = Unit<1,0,0>;  
using Second = Unit<0,0,1>;  
using Second2 = Unit<0,0,2>;  
using Speed = Value<Unit<1,0,-1>>;  
using Acceleration = Value<Unit<1,0,-2>>;
```

Физические величины

- Разумеется литералы для удобства работы

```
constexpr Value<Meter> operator"" _m(long double d)
{ return Value<Meter>(d); }
```

```
constexpr Value<Second> operator"" _s(long double d)
{ return Value<Second>(d); }
```

```
constexpr Value<Second2> operator"" _s2(long double d)
{ return Value<Second2>(d); }
```

- И вот теперь настало время определить арифметику над этими величинами

Проблема умножения

- Попробуем написать умножение

```
template <typename D1, typename D2>
auto operator*(Value<D1> a, Value<D2> b) {
    using D = ???;
    return Value<D>{double(a) * double(b)};
}
```

- Теперь видно в чём дело: нам нужно итерировать по $\text{Unit}\langle l_1, m_1, t_1 \rangle$ и по $\text{Unit}\langle l_2, m_2, t_2 \rangle$ и сложить всё что мы там встретим
- А потом вывести это как тип.

- ❑ Константность

- ❑ Функции времени компиляции

- ❑ ООП времени компиляции

- Снова о метапрограммировании

Квадранты вычислений

Вычисления времени
компиляции

Вычисления времени
выполнения

Преобразования над
типами

Смешанные
вычисления

Вычисления над типами

- Пример вычислений над типами: MPL

```
template <typename T> struct add_pointer {  
    using type = T*;  
};
```

```
using types = mpl::vector<int, char, float, void>;  
using pointers = mpl::transform<types,  
    add_const_pointer<mpl::_1>>::type;
```

- Какие мы видим тут недостатки?

Представление типов значениями

- Благодаря constexpr функциям в языке мы можем работать со значениями.

```
template <typename T> struct type {};
```

```
template <typename T>  
constexpr type<T*> add_pointer(type<T>) { return {};
```

- Эти идеи легли в основу boost::hana

```
auto types = hana::tuple_t<int, char, float, void>;
```

```
auto pointers = hana::transform(types, [] (auto t) {  
    return add_pointer(t);  
});
```

Index sequences

- Удивительно полезный класс `integer_sequence`:

```
template <class T, T... Ints> class integer_sequence;
```

- Его синоним если нам нужны только индексы:

```
template <size_t... Ints>  
using index_sequence = std::integer_sequence<size_t, Ints...>;
```

- Мы можем писать `std::make_index_sequence<3>{}`
- Типом этого выражения является `integer_sequence<size_t, 1, 2, 3>`

Идея для ST mapping

- По сути индексная последовательность позволяет кодировать теги.

```
// using MyCommonTags = mpl::vector_c<int, 11, 35, 42, 10916>;
```

```
using MyCommonTags =  
    std::integer_sequence<int, 11, 35, 42, 10916>;  
FixMsg<MyCommonTags> fixMsg;
```

```
auto& tagEntity = fixMsg.getTagEntity<11>();  
tagEntity.isValid = true;  
tagEntity.value = "someval";
```

- Я надеюсь все ещё помнят этот пример с MPL?

Используем constexpr

- Чтобы сочетать время исполнения и компиляции, сделаем обычный метод в классе без constexpr ctor.

```
template<int Tag> TagEntity& getTagEntity() {  
    constexpr ct_ints<CommonTagsCount> vec(CommonTags{});  
    constexpr int idx = vec.find(Tag);  
    static_assert(всё то же самое);  
    return commonTags_[idx];  
}
```

- Осталось написать не такой сложный метод find.

Заменяем mpl::find

```
template <int N> struct ct_ints {  
    std::array<int, N> arr_;  
  
    template <int ... Is>  
    constexpr ct_ints(std::integer_sequence<int, Is...>) :  
        arr_{Is...} {}  
  
    constexpr int find(int tag) const {  
        for(int i = 0; i < N; ++i)  
            if (arr_[i] == tag)  
                return i;  
        return N + 1;  
    }  
};
```

Смешанные вычисления

- Пример смешанных вычислений над типами и значениями в boost fusion.

```
auto to_string = [](auto t) {  
    std::stringstream ss; ss << t; return ss.str();  
};
```

```
fusion::vector<int, std::string, float> seq{1, "abc", 3.4f};
```

```
auto strings = fusion::transform(seq, to_string);
```

- Hana предоставляет похожий интерфейс.

```
auto seq = hana::make_tuple(1, "abc"s, 3.4f);
```

```
auto strings = hana::transform(seq, to_string);
```

Физические величины в boost::hana

```
template <typename Unit> struct Value {  
    double val;  
    explicit constexpr Value(double d) : val(d) {}  
    explicit constexpr operator double() const { return val; }  
  
    // template <typename Other> Value(Value<Other> other) ?  
};
```

- Сами величины теперь hana tuples

```
using Meter = decltype(hana::tuple_c<int, 1, 0, 0>);  
using Second = decltype(hana::tuple_c<int, 0, 0, 1>);  
using Second2 = decltype(hana::tuple_c<int, 0, 0, 2>);
```


Физические величины в boost::hana

```
template <typename Unit> struct Value {  
    double val;  
    explicit constexpr Value(double d) : val(d) {}  
    explicit constexpr operator double() const { return val; }  
  
    template <typename Other>  
    explicit constexpr Value(Value<Other> other) : val(other.val) {  
        static_assert(Unit{} == Other{}, "Dimensions incompatible");  
    }  
};
```

- Осталось написать операторы вроде умножения и деления

Композиция величин

- Самый сложный шаг теперь довольно прост

```
template <typename D1, typename D2>
auto operator*(Value<D1> a, Value<D2> b) {
    using D = decltype(hana::zip_with(std::plus<>{}, D1{}, D2{}));
    return Value<D>{double(a) * double(b)};
}
```

- Особое внимание на zip_with
- D1 и D2 это кортежи {d11 d1n} и {d21 d2n}
- D это {d11 + d21,, d1n + d2n}

Задачи, часть 1. Constant expressions

- В следующем коде clang и gcc ведут себя по разному.

```
struct S {  
    int : *new int{0};  
};
```

```
enum E {  
    V = *new int{1},  
};
```

- Проанализируйте кто прав и почему.

Задачи, часть 2. Core CE

- Следующий код также по разному себя ведёт в gcc vs clang.

```
struct Type {  
    int a; const int& b;  
    constexpr Type() : a(1), b(a) {}  
};  
  
constexpr auto get() { return Type(); }  
  
constexpr Type t2 = get(); // gcc error, clang ok  
  
constexpr int c2 = t2.a;
```

Задачи, часть 3: физические величины

- Без использования какой-либо библиотеки метапрограммирования, решите задачу умножения для физических величин.

```
template <typename D1, typename D2>  
auto operator*(Value<D1> a, Value<D2> b) {  
    using D = ???;  
    return Value<D>{double(a) * double(b)};  
}
```

- Возможно в процессе вы напишете свою?

Задачи, часть 4: отсортировать кортеж

- На входе вашей метапрограммы *кортеж* из типов.
- Я специально не уточняю что за кортеж, это не обязательно `std::tuple`. Главное чтобы список типов можно было как-то задать.
- На выходе -- он же, но отсортированный по возрастанию размера.
- Сортировка должна быть стабильной.
- Ни у одного человека, делающего это задание, не должна повторяться использованная библиотека метапрограммирования.

Литература

- ISO/IEC, "Information technology -- Programming languages – C++", ISO/IEC 14882:2017, 2017
- Bjarne Stroustrup, The C++ Programming Language (4th Edition)
- Scott Meyers, Effective Modern C++, O'Reilly, 2014
- Bjarne Stroustrup, "Software Development for Infrastructure", 2012
- Scott Shurr, "Constexpr Introduction" and "Constexpr Applications", CppCon'15
- Dietmar Kuhl, "Constant fun", CppCon'16
- Ben Deane, Jason Turner, "Constexpr all the things", CppCon'17
- Arne Mertz, "Constexpr Additions in C++17", 2017