Не бойтесь метапрограммирования -

наслаждайтесь им!

Что такое метапрограммирование

- В широком смысле - создание кода, управляющего созданием кода.
- Трансляторы как “программирующие программы”.
- Различие между обобщённым (generic) программированием и метапрограммированием - определяется прагматикой.
  - Воплощая обобщённую функцию наподобие поиска в контейнере, программист обычно не воспринимает это как метапрограммирование.
- Нестрогий, но разумный критерий: при метапрограммировании от компилятора ожидают принятия нетривиальных решений.
  - Например, какой из нескольких возможных алгоритмов применить, в зависимости от типов данных.
  - Выбрать алгоритм сортировки или поиска в зависимости от того, поддерживает ли контейнер произвольный доступ.

Метапрограммирование на constexpr

Метапрограммирование без шаблонов

- Заставить компилятор выполнить нетривиальное вычисление.
- Результат этого вычисления должен быть “жёстким”.
- Настолько жёстким, чтобы от него зависело наличие или отсутствие целых участков исполняемого кода.
- Удовлетворяет неформальному критерию метапрограммирования.
- Подходящее средство - constexpr:
  - для переменной означает, что значение вычисляется на этапе компиляции;
  - для функции означает, что она *может* отработать на этапе компиляции;
- Начиная с C++20 - также
  - consteval - функция *обязана* вычисляться только на этапе компиляции;
  - constinit - обязывает инициализировать переменную статически.

Спецификатор constexpr и переменные

- Наследие языка C:
  ```
  #define DATA_SIZE 100
  ```
- Более изящно:
  ```
  std::size_t const data_size = 100;
  ```
- В C++11 появился спецификатор `constexpr`
  ```
  constexpr std::size_t data_size = 100;
  ```
- `const` означает запрет менять значение после инициализации.
- Значение `const` может вычисляться на этапе выполнения:
  ```
  int f(std::vector<int> const& a) {
      auto const n = a.size(); // …
  ```
- `constexpr` требует вычисления на этапе компиляции.

Спецификатор constexpr и функции

- Значение функции должно быть вычислено на этапе компиляции, *если известны значения аргументов*.
- Если значение аргумента неизвестно при компиляции, функция отработает на этапе выполнения.

```
constexpr int chunk = 512;
constexpr int header = 8;
constexpr int get_packed_size(int raw)
  { return ((raw + chunk - 1) / chunk) * header + raw; }

std::array<char, get_packed_size(800)> buffer;
```

Ограничения constexpr

- Тип constexpr-переменной - только литеральный:
  - скалярный;
  - указатель;
  - массив элементов скалярного типа;
  - класс, удовлетворяющий условиям:
    - деструктор по умолчанию;
    - нестатические члены - литеральных типов;
    - хотя бы один constexpr-конструктор или без конструкторов.
- Функция constexpr:
  - не может быть виртуальной (до C++20);
  - должна возвращать значение литерального типа;
  - (кроме конструкторов) ровно один оператор return;
  - и ряд других, см. справочник.

Дополнительные замечания о constexpr

- Поначалу сопровождался жёсткими ограничениями.
- Постепенно ограничения ослабляются:
  - разрешены операторы присваивания, `if`, `switch`, `for` (C++ 14);
  - тип `void*` стал считаться литеральным (C++ 14);
  - уже разрешены блоки `try` (C++ 20);
  - уже могут быть virtual (C++ 20);
- Запрещены по-прежнему:
  - операторы goto и метки;
  - вызов функций, не имеющих спецификатора `constexpr`;
  - объявление локальных переменных нелитерального типа;
  - объявление переменных со статическим или потоковым временем жизни.

constexpr if

- Введён в C++ 17.
- Оператор if, у которого выбор ветки гарантированно может быть осуществлён на этапе компиляции.
- Условие представляет собой выражение constexpr.
- Одно из применений: обработка частных случаев в шаблонах:

```
template <size_t N>
void handle_message(std::uint8_t (&bytes)[N]) {
  if constexpr (N == 4) { // to a single 32-bit int
```

- Позволяет различать архитектуру

```
if constexpr (sizeof(int*) == 8) // 64-bit
```

Примеры функций constexpr

```
constexpr unsigned long long factorial_old(int const n)
    { return n == 0 ? 1 : n * factorial_old(n - 1); }

constexpr unsigned long long factorial_new(int n) {
    unsigned long long r = 1;
    while (n != 0) {
      r *= n;
      --n;
    }
    return r;
}
```

Контрольная сумма строки на этапе компиляции!

```
constexpr std::uint32_t adler32_helper(
    unsigned char const* s,
    uint32_t a,
    uint32_t b)
{
  static constexpr std::uint16_t prime = 65521;
  auto const c = static_cast<std::uint32_t>(*s);
  return 0 == c
    ? (a | (b << 16))
    : adler32_helper(s+1, (a + c) % prime, (a + b + c) % prime);
}
```

Контрольная сумма строки на этапе компиляции!

```
constexpr std::uint32_t adler32(char const* s) {
  return adler32_helper(
    reinterpret_cast<unsigned char const*>(s),
    1,
    0);
}

constexpr auto checksum = adler32("Constexpr checksum");
```

Реальный пример: поиск строки в списке

- Пусть каждая функция в приложении пишет сообщение в лог.
- Первым аргументом в функцию log передаётся имя функции…
- … известное на этапе компиляции.
- Некоторые функции считаются “интересными”.
- Список имён интересных функций жёстко задан в коде…
- … т.е. известен во время компиляции.
- Логгер должен игнорировать сообщения от неинтересных функций.
- Поиск в контейнере “интересных” имён на этапе выполнения - долго.
- Нужно сделать распознаватель вхождения строки в контейнер, работающий на этапе компиляции.

Сравнение строк

```
constexpr bool str_eq(
    char const* const s,
    char const* const t)
{
  return (*s == *t)
      && (*s == 0 || str_eq(s + 1, t + 1));
}

constexpr char const* s1 = "A this is a string";
constexpr char const* s2 = "B this is a string";
static_assert(str_eq(s1 + 2, s2 + 2), "Must be recognized equal");
```

Линейный поиск в массиве строк

```
constexpr bool str_in(
    char const* const s,
    char const* const* const ps,
    char const* const* const ps_end)
{
  return (ps != ps_end)
      && (str_eq(s, *ps) || str_in(s, ps + 1, ps_end));
}
```

Фасад

```
template <std::size_t N>
constexpr bool str_in(
    char const* const s,
    char const* const (&ps)[N])
{
  return str_in(s, &(ps[0]), &(ps[N]));
}
```

Демонстрация использования

```
constexpr char const* const names[] { "Baldr", "Heimdall" };

template <typename T, T X>
constexpr T ensure_constexpr() { return X; }

int main() {
    std::cout
        << ensure_constexpr<bool, str_in("Baldr",  names)>()
        << std::endl
        << ensure_constexpr<bool, str_in("Hoenir", names)>()
        << std::endl;
```

Метапрограммирование на шаблонах

Разрешение перегрузки

- У шаблона класса, помимо основного определения, может быть ещё несколько специализаций.
- Для шаблонов функций допускаются *полные* специализации.
- У одного имени функции может быть несколько перегруженных объявлений (как шаблонных, так и нешаблонных).
- Компилятору нужно выбрать наиболее подходящее определение для перегруженного имени (функции или класса).
- Увидев перегруженное шаблонное имя, компилятор составляет список всех известных определений и отбрасывает заведомо неподходящие.
- Из подходящих отбирается единственный наилучший кандидат.

Разрешение перегрузки: пример

```
void f(int, std::vector<int>);                          // 1
void f(int, int);                                       // 2
void f(double, double);                                 // 3
void f(int, int, char, std::string, std::vector<int>); // 4
void f(std::string);                                    // 5
void f(...);                                            // 6
template<typename T>  void f(T, T);                     // 7


f(1, 2); // что выбрать?
```

Задача

- Сделать распознаватель категории выражения: (l|x|pr)value
- Может быть оформлен как угодно:
  - шаблон функции;
  - шаблон класса;
  - макрос;
  - …
- Должен отрабатывать на этапе компиляции.
- Должен “вытаскивать” из выражения-аргумента его категорию.

Ожидаемая форма использования

```
int x;
int &&r = 0;

std::cout
  << EXPR_CATEG( std::cout    )::name << std::endl
  << EXPR_CATEG( 0            )::name << std::endl
  << EXPR_CATEG( x            )::name << std::endl
  << EXPR_CATEG( &x           )::name << std::endl
  << EXPR_CATEG( x++          )::name << std::endl
  << EXPR_CATEG( ++x          )::name << std::endl
  << EXPR_CATEG( r            )::name << std::endl
  << EXPR_CATEG( std::move(r) )::name << std::endl;
```

Ожидаемый результат

```
int x;
int &&r = 0;

std::cout
  << EXPR_CATEG( std::cout    )::name << std::endl   // l
  << EXPR_CATEG( 0            )::name << std::endl   // pr
  << EXPR_CATEG( x            )::name << std::endl   // l
  << EXPR_CATEG( &x           )::name << std::endl   // pr
  << EXPR_CATEG( x++          )::name << std::endl   // pr
  << EXPR_CATEG( ++x          )::name << std::endl   // l
  << EXPR_CATEG( r            )::name << std::endl   // l
  << EXPR_CATEG( std::move(r) )::name << std::endl;  // x
```

```
struct categ_pr
    { static constexpr char const* name = "pr"; };
struct categ_l
    { static constexpr char const* name = "l";  };
struct categ_x
    { static constexpr char const* name = "x";  };

template <class T> struct category_of
    { using which = categ_pr; };
template <class T> struct category_of<T&>
    { using which = categ_l; };
template <class T> struct category_of<T&&>
    { using which = categ_x; };

#define EXPR_CATEG(e) category_of<decltype((e))>::which
```

Анализ решения

- Типы - не для того, чтобы работать с объектами данных, а в качестве данных для компилятора.
- Логика метакода построена на правилах выбора перегрузки и/или специализации.
- Рядом с перегрузкой/специализацией, отвечающей за частный случай, должно находиться определение, к которому компилятор сможет обратиться, если частный случай применить не удаётся.
- Определение для общего случая должно быть “хуже” частного с точки зрения алгоритма выбора специализации или перегрузки.

Вычисления на этапе компиляции

- На практике бесполезно, но уж очень занимательно :-)
- Числа моделировать **типами**.
  - Например, числам 3 и 7 соответствуют типы, которые здесь обозначим _3 и _7.
- Операции над числами - моделировать операциями **над типами**.
  - А операция над типами - это шаблон.
  - Шаблон принимает типы в качестве параметров и строит новый тип.
  - Например, шаблон add из типов _3 и _7 построит тип add<_3, _7>, который должен соответствовать числу 10.
- Операции должны включать в себя +, -, * и факториал.
- Решение - на основе аксиом Пеано и рекурсии (на этапе компиляции)

Аксиомы Пеано

1. Существует натуральное число 0, называемое *нулём*.
2. За каждым натуральным числом n непосредственно следует однозначно определённое натуральное число n’, называемое *непосредственно следующим за n*.
3. Число 0 не следует ни за каким натуральным числом.
4. Каждое натуральное число непосредственно следует не более чем за одним натуральным числом.
5. Любое подмножество M множества натуральных чисел N, содержащее 0, и вместе с каждым числом из M содержащее следующее за ним число, совпадает с множеством N.

Рекуррентные определения операций

- $x + 0 = x$. сложение: база
- $x + y' = (x + y)'$. сложение: шаг
- $x \times 0 = 0$. умножение: база
- $x \times y' = (x \times y) + x$. умножение: шаг

	$2 + 2$	расшифровать через 0 и x’
=	$0'' + 0''$	0’’ можно представить в виде y’ при y=0’
=	$(0'' + 0')'$	снова второе слагаемое есть y’ при y=0
=	$(0'' + 0)''$	дошли до базы рекурсии
=	$(0'')''$	скобки можно убрать, получится 0’’’’
=	4	после перевода в привычную запись

Арифметика на типах: числа как типы

```
struct zero
{
  static constexpr value_t value = 0;
  using type = zero;
};

template <typename T>
struct succ
{
    static constexpr value_t value = 1 + T::value;
    using pred = typename T::type;
    using type = succ<typename T::type>;
};
```

Арифметика на типах: операции над типами

```
template <typename S, typename T>
struct add: succ<typename add<S, typename T::pred>::type>::type {};

template <typename S>
struct add<S, zero>: S::type {};

template <typename S, typename T>
struct mul: add<typename mul<S, typename T::pred>::type, S>::type {};

template <typename S>
struct mul<S, zero>: zero {};

template <typename S>
struct factorial: mul<S, typename factorial<typename S::pred>::type>::type {};

template <>
struct factorial<zero>: succ<zero>::type {};
```

Арифметика на типах: использование

```
// 5 * 7 == 35
constexpr auto a = mul<_5, _7>::value;

// 2 * 7 + 3 * (4 + 5) == 41
constexpr auto b = add<
        mul<_2, _7>,
        mul<  _3,
              add<_4, _5>
        >
    > ::value;

// and now factorial: 5! == 120
constexpr auto c = factorial<_5>::value;
```

```
// constants

using _0 = zero;
using _1 = succ<_0>::type;
using _2 = succ<_1>::type;
using _3 = succ<_2>::type;
using _4 = succ<_3>::type;
using _5 = succ<_4>::type;
using _6 = succ<_5>::type;
using _7 = succ<_6>::type;
using _8 = succ<_7>::type;
using _9 = succ<_8>::type;
```

S F I N A E

SFINAE

- Substitution Failure Is Not An Error: неудача подстановки не есть ошибка.
- Если не получается рассчитать окончательные типы аргументов (провести подстановку шаблонных параметров) перегруженной шаблонной функции, компилятор не выбрасывает ошибку, а ищет другую подходящую перегрузку.
- Ошибка будет в таких случаях:
  - Не нашлось ни одной подходящей перегрузки.
  - Нашлось несколько таких перегрузок, и невозможно решить, какая лучше.
  - Перегрузка нашлась, она оказалась шаблонной, и при инстанцировании шаблона случилась ошибка.

Разрешение перегрузки и SFINAE: пример

```
void f(int, std::vector<int>);                                  // 1
void f(int, int);                                               // 2
void f(double, double);                                         // 3
void f(int, int, char, std::string, std::vector<int>);          // 4
void f(std::string);                                            // 5
void f(...);                                                    // 6
template<typename T>  void f(T, T);                             // 7
template<typename T>  void f(T, typename T::iterator);          // 8

f(1, 2); // что выбрать?
```

SFINAE и рефлексия

- Рефлексия - возможность для программы анализировать и\или менять собственное устройство.
- Рефлексия может отрабатывать во время выполнения или на этапе компиляции.
- Рефлексия - частный случай метапрограммирования.
- Механизм SFINAE позволяет во время компиляции определять, обладает ли тип T (параметр шаблона) теми или иными свойствами…
- …и, в зависимости от этого, выбирать подходящие для его обработки алгоритмы.

SFINAE-рефлексия средствами C++03

```
template<typename T>
class DetectFind
{
  struct HasFind { int find; };
  struct InheritFind : T, HasFind { };
  template<typename U, U> struct Check;
  typedef char Size1[1];
  typedef char Size2[2];
  template<typename U> static Size2& func(Check<int HasFind::*, &U::find> *);
  template<typename U> static Size1& func(...);
public:
  enum { value = sizeof(func<InheritFind>(0)) == sizeof(Size1) };
};

// DetectFind<std::set<int> >::value
```

SFINAE: извлечение уроков

- Функции фиктивны - не вызываются на этапе выполнения, …
- … а нужны лишь для того, чтобы было куда на этапе компиляции подставлять фиктивные типы.
- Логика метакода построена на *невозможности* вызвать функцию или инстанцировать шаблон класса.
- Некоторый тип в перегрузке или специализации имеет смысл только при определённых условиях, иначе - вообще не является типом.
- Рядом с такой перегрузкой/специализацией должно находиться запасное определение, к которому компилятор всегда сможет перейти, когда первое определение потерпит неудачу.
- Запасное определение должно быть “хуже” проверочного.

SFINAE: извлечение уроков

- Логика метакода совершенно неочевидна.
- Требуется хорошее знание правил разрешения перегрузки с учётом неявного преобразования типов, аргументов по умолчанию.
- Предыдущий пример компилируется даже в C++03.
- В стандартах начаная с C++11 появляются средства для удобства метапрограммирования на SFINAE:
  - decltype,
  - std::declval,
  - std::void_t;
  - std::enable_if,
  - <type_traits>.

decltype и std::declval

- Если e - выражение, то `decltype(e)` - тип этого выражения.
- Если выражение e не имеет смысла, а тип `decltype(e)` используется в сигнатуре перегруженной и/или шаблонной функции, к нему применяется принцип SFINAE.
- Иными словами, `decltype(e)` можно использовать смело, не заботясь о том, осмыслено ли выражение e.
- В худшем случае перегрузка будет отброшена.
- Шаблон функции std::declval<T>() фиктивен - может использоваться только на этапе компиляции.
- “Строит” пример значения типа T.

decltype и std::declval

```
template <typename T>
decltype(std::declval<T>() + 1) grow(T x, int d) { /*...*/ }
```

- Пусть T - некоторый неизвестный тип.
- Пусть дано некоторое значение x этого типа.
- В предположении, что имеет смысл выражение x+1, …
- … взять тип этого выражения.

std::void_t

- C++ Russia 2018: Ivan Čukić, 2020: A void_t odyssey
  - https://youtu.be/dZyH01tyIsA
- Шаблон типа, который на любых типах-аргументах имеет результатом тип void.
- Если шаблоны типов рассматривать как функции над типами, то void_t подобен функции, тождественно равной 0 на любых аргументах.
- Не бесполезно ли это?

std::void_t

- C++ Russia 2018: Ivan Čukić, 2020: A void_t odyssey
  - https://youtu.be/dZyH01tyIsA
- Шаблон типа, который на любых типах-аргументах имеет результатом тип void.
- Если шаблоны типов рассматривать как функции над типами, то void_t подобен функции, тождественно равной 0 на любых аргументах.
- Не бесполезно ли это?
- Отнюдь: void_t позволяет проверять *отсутствие* типов, т.е. бессмысленность имён.
  - Позволяет реализовать рефлексию: если у типа есть член f, имеет смысл его тип.

void_t: распознать возможность присваивания

```
template <typename T, typename = std::void_t<> >
struct is_assignable_t : std::false_type {};

template <typename T>
using assign_t = decltype(std::declval<T&>() = std::declval<T const&>());

template <typename T>
struct is_assignable_t<T, std::void_t<assign_t<T> > > : std::true_type {};

template <typename T>
bool constexpr is_assignable() {
  return is_assignable_t<T>::value;
}
```

void_t: каким методом сортировать контейнер

```
template <typename I>
using random_access_t = decltype(std::declval<I>() + 1);

template <typename I, typename = std::void_t<> >
struct collection_utils {
  static void sort(I from, I to) {
    std::cerr << "Bubble sort\n";
  }
};

template <typename I>
struct collection_utils<I, std::void_t<random_access_t<I> > > {
  static void sort(I from, I to) {
    std::cerr << "Heapsort\n";
  }
};
```

void_t: каким методом сортировать контейнер

```
template <typename I>
using random_access_t = decltype(std::declval<I>() + 1);

template <typename I, typename = std::void_t<> >
struct collection_utils {
  static void sort(I from, I to) {
    std::cerr << "Bubble sort\n";
  }
};

template <typename I>
struct collection_utils<I, std::void_t<random_access_t<I> > > {
  static void sort(I from, I to) {
    std::cerr << "Heapsort\n";
  }
};
```

```
template <typename I>
void sort(I from, I to) {
  collection_utils<I>::sort(from, to);
}
```

std::enable_if: базовые сведения

- `template <bool B, class T = void> struct std::enable_if;`
- Если B есть `true`, то `std::enable_if<B, T>::type` есть тип T.
  - В противном случае имя `std::enable_if<B, T>::type` **не имеет смысла**.
- Используется для включения или отключения определений функций в зависимости от статических свойств типов.

```
template <bool B, class T = void>
struct enable_if {};

template <class T>
struct enable_if<true, T> { typedef T type; };
```

std::enable_if: пример применения

- Шаблона функции f сериализует объекты разных типов.
- Можно реализовать оптимизированные версии для частных случаев.
- Тип T есть POD (plain old data).
  - В типе не должно быть виртуальных функций, нетривиальных конструкторов и деструкторов, также ряд других условий, см. справочник.
  - Объекты типа T можно сериализовать в поток байт и восстановить из такого потока.
- Тип T есть POD и его размер не превышает размера целого числа.
  - Побайтное представление объекта можно упаковать в единственное значение целого типа, а затем однозначно восстановить.
  - Для ввода-вывода объекта можно воспользоваться встроенными средствами для ввода-вывода чисел.

std::enable_if: пример применения

```
template <typename T>
typename std::enable_if<
    (sizeof(T) <= sizeof(std::uint64_t) && std::is_pod<T>::value)>::type
f(T const&) {/*...*/}

template <typename T>
typename std::enable_if<
    (sizeof(T) > sizeof(std::uint64_t)) && std::is_pod<T>::value>::type
f(T const&) {/*...*/}

template <typename T> // общий случай
typename std::enable_if<!std::is_pod<T>::value>::type
f(T const&) {/*...*/}
```

std::enable_if: пример применения

```
f(0);                 // int    : POD and not larger than 8 bytes

f(3.14);              // double : POD and not larger than 8 bytes

f("abcdefg");         // char[8]: POD and not larger than 8 bytes

f("abcdefgh");        // char[9]: POD and larger than 8 bytes

f(std::set<int>{}); // not a POD
```

std::enable_if: итоги

- Обычные средства позволяют определять частные случаи для конкретных типов: общее определение для произвольного типа T и частные случаи для int, void*, char[N] и т.д.
- SFINAE позволяет определять частные случаи, основываясь на интерфейсных свойствах типа T: например, для контейнеров, допускающих произвольный или последовательный доступ; для типов, допускающих присваивание;
- enable_if позволяет определять частные случаи на основе любых свойств типа T, какие возможно распознать.
- Средства распознавания свойств собраны в <type_traits>.

<type_traits> и рефлексия на этапе компиляции

- is_void
- is_integral
- is_floating_point
- is_array
- is_enum
- is_function
- is_pointer
- is_member_function_pointer
- is_same
- is_base_of
- is_const
- is_trivially_copyable
- is_standard_layout
- is_empty
- is_polymorphic
- is_abstract
- is_default_constructible
- is_copy_constructible
- has_virtual_destructor

Задача о конверторе

Конвертор: постановка задачи

- Десериализовать объект из потока байт и сериализовать в строку.
- Конвертор из двоичного формата в строковый через промежуточный тип.
- Пользователь определяет конверсию для нужных типов.
- Для остальных типов должна работать реализация по умолчанию.

```
template <typename T>
std::string to_text(std::uint8_t const* p, std::size_t n)
```

- У функции `to_text` нет параметра типа T.
- Несколько различных реализаций.
- https://github.com/vadimvinnik/non_arg_template

Конвертор: вариант 1

```
template <typename T>
struct implicit_from {
  implicit_from(T const&) {}
};

template <typename T>
std::string to_text(implicit_from<std::uint8_t const*>, std::size_t) {/*...*/}

template <typename S, typename T>
using string_if_same = std::enable_if_t<std::is_same_v<S, T>, std::string>;

template <typename T>
string_if_same<T, std::uint64_t>
to_text(std::uint8_t const* bytes, std::size_t size) {/*...*/}
```

Конвертор: вариант 1

```
template <typename S, typename T>
using string_if_same = std::enable_if_t<std::is_same_v<S, T>, std::string>;

template <typename T>
string_if_same<T, std::uint64_t>
to_text(std::uint8_t const* bytes, std::size_t size) {/*...*/}
```

Конвертор: вариант 2

```
template <typename T>
struct to_text_helper {
  static std::string to_text(std::uint8_t const*, std::size_t) {/*...*/}
};

template <>
struct to_text_helper<std::uint64_t> {
  static std::string to_text(std::uint8_t const* b, std::size_t n) {/*...*/}
};

template <typename T>
std::string to_text(std::uint8_t const* bytes, std::size_t size) {
  return to_text_helper<T>::to_text(bytes, size);
}
```

Конвертор: вариант 3

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

template <typename T>
std::string to_text_helper(
    std::uint8_t const* bytes, std::size_t size, type_id<T>) {/* ... */}
std::string to_text_helper(
    std::uint8_t const* bytes, std::size_t size, type_id<std::uint64_t>);
std::string to_text_helper(
    std::uint8_t const* bytes, std::size_t size, type_id<std::string>);

template <typename T>
std::string to_text(std::uint8_t const* bytes, std::size_t size){
  return to_text_helper(bytes, size, type_id<T>{});
}
```

Конвертор: вариант 4

```
template <typename T>
std::string to_text(std::uint8_t const*, std::size_t)
{
  return "[unknown_type]";
}

// function templates allow full specialisations

template <>
std::string to_text<std::uint64_t>(std::uint8_t const* p, std::size_t n);

template <>
std::string to_text<std::string>(std::uint8_t const* p, std::size_t n);
```

Вариадические шаблоны

Базовые сведения

- Шаблон функции или класса, имеющий заранее неизвестное число параметров.
- Кортеж из произвольного числа (в том числе 0) разнородных параметров называется пакетом параметров.
- Вариадический шаблон обладает одним *или несколькими* пакетами параметров.
- Обозначается многоточием (...):
  - в заголовке шаблона - объявляет пакет параметров;
  - слева от имени параметра функции - объявление набора параметров неизвестной длины;
  - в теле функции справа от имени параметра - разворачивает пакет в аргументы.

Объявление и инстанцирование

- Объявление вариадического шаблона класса:
  - ```
    template <typename... Args>
    class variadic_demo { /*. . . */ };
    ```
- Экземпляры вариадического шаблона:
  - `using t0 = variadic_demo<>;`
  - `using t1 = variadic_demo<int>;`
  - `using t2 = variadic_demo<int, std::string>;`
- Вопрос на засыпку: различаются ли между собой тип t1 и следующие:
  - `using v1 = variadic_demo<void>;`
  - `using v2 = variadic_demo<void, void>;`
    ```

Что делать с пакетом параметров?

- Получить число переданных аргументов: `sizeof...(args)`
- `(const args&...)`
  `// (T1 const& arg1, T2 const& arg2, ...)`
- `((f(args) + g(args))...)`
  `// (f(arg1) + g(arg1), f(arg2) + g(arg2), ...)`
- `(f(args...) + g(args...))`
  `// (f(arg1, arg2,...) + g(arg1, arg2, ...))`
- `(std::make_tuple(std::forward<Args>(args)...))`
  `// (std::make_tuple(`
  `//    std::forward<T1>(arg1),`
  `//    std::forward<T2>(arg2), ...))`

Что делать с пакетом параметров?

- Разбирать по одному рекурсивно!
- Рекурсия на этапе компиляции, а не выполнения!
- База рекурсии: с одним параметром или без параметров.
- У шаблонов функций частичной специализации не бывает!
- Для функции базой будет нешаблонная перегрузка.
- Задача. Создать вариадическую функцию, которая все свои аргументы переводит в текстовое представление и собирает в строку.
- Пример использования:

```
std::string log_message = string_helper::collect(
  "completed, code=", error_code, ", time=", duration_ms, "ms");
```

```
template <class Stream>
void append_to_stream(Stream &stream) {}

template <class Stream, class Head, class... Tail>
void append_to_stream(Stream &stream, Head const& head, Tail&& ...tail) {
    stream << head;
    append_to_stream(stream, std::forward<Tail>(tail)...);
}

template <class ...Args>
std::string collect(Args&&... args) {
  std::ostringstream stream;
  append_to_stream(stream, std::forward<Args>(args)...);
  return stream.str();
}
```

Произведение типов: постановка задачи

- Создать вариадический шаблон класса, который ведёт себя как гетерогенный контейнер.
- В объекте фиксированное число полей данных, типы которых заданы параметрами шаблона.
- Подобен struct или std::tuple.
- С точки зрения теории категорий - **произведение** типов.
- Пример использования:

```
xtypes::product<int, bool, std::string> r;
r.get<2>() = "abc";
```

- https://github.com/vadimvinnik/xtypes

Произведение типов: простейшее решение

```
template <typename... Args>
struct product {};

template <
    typename Head,
    typename... Tail>
struct product<Head, Tail...> {
  using first_t = Head;
  using others_t = product<Tail...>;
  first_t first;
  others_t others;
};
```

Произведение типов: простейшее решение

```
template <typename... Args>
struct product {};

template <
    typename Head,
    typename... Tail>
struct product<Head, Tail...> {
  using first_t = Head;
  using others_t = product<Tail...>;
  first_t first;
  others_t others;
};
```

```
xtypes::product<int, bool, std::string> r;
r.first = 9;
r.others.first = true;
r.others.others.first = "a";
```

Произведение типов: улучшение

```
template <std::size_t I, typename P>
struct product_helper {
  using next_t = product_helper<I-1, typename P::others_t>;
  using item_t = typename next_t::item_t;
  static item_t& get(P& p) noexcept {return next_t::get(p.others);}
};

template <typename P>
struct product_helper<0, P> {
  using item_t = typename P::first_t;
  static item_t& get(P& p) noexcept { return p.first; }
};
```

Произведение типов: улучшение

```
template <std::size_t I, typename P>
struct product_helper {
  using next_t = product_helper<I-1, typename P::others_t>;
  using item_t
  static item_t
};

template <typen
struct product_helper<0, P> {
  using item_t = typename P::first_t;
  static item_t& get(P& p) noexcept { return p.first; }
};
```

```
xtypes::product<int, bool, std::string> r;
xtypes::product_helper<0, record_t>::get(r) = 9;
xtypes::product_helper<1, record_t>::get(r) = true;
xtypes::product_helper<2, record_t>::get(r) = "a";
```

Произведение типов: окончательное решение

```
template <typename Head, typename... Tail>
struct product<Head, Tail...> {
  using first_t = Head;
  using others_t = product<Tail...>;
  using this_t = product<Head, Tail...>;

  template <std::size_t I>
  using helper_t = product_helper<I, this_t>;
  template <std::size_t I>
  using item_t = typename helper_t<I>::item_t;

  first_t first;
  others_t others;

  template <std::size_t I>
  item_t<I>& get() & noexcept { return helper_t<I>::get(*this); }
```

Произведение типов: окончательное решение

```
template <typename Head, typename... Tail>
struct product<Head, Tail...> {
  using first_t = Head;
  using others_t = product<Tail...>;
  using this_t = product<Head, T

  template <std::size_t I>
  using helper_t = product_helpe
  template <std::size_t I>
  using item_t = typename helper

  first_t first;
  others_t others;

  template <std::size_t I>
  item_t<I>& get() & noexcept { return helper_t<I>::get(*this); }
```

```
xtypes::product<int, bool, std::string> r;
r.get<0>() = 9;
r.get<1>() = true;
r.get<2>() = "a";
```

Стирание типов

Зачем стирать тип объекта?

- Через единый интерфейс работать с объектами различных типов.
- Забыть фактические типы объектов.
- Помнить лишь небольшой набор поддерживаемых ими операций.
- Пример std::function - единая обёртка над любыми сущностями, которые можно вызвать в виде $f(x_1, \ldots, x_n)$:
  - указатель на функцию U (*func)(T_1, …, T_n);
  - объект с перегруженным operator();
  - ... том числе λ;
  - указатель на функцию-член класса (указатель на объект передаётся дополнительным аргументом).

Стирание типа функционального объекта (1)

```
struct linear {
    int a;
    int b;
    int get(int x) const { return a * x + b; }

};

int apply(linear const* p, int x) { return p->get(x); }

using linear_f = std::function<int(linear const*, int)>;
```

Стирание типа функционального объекта (2)

```
linear l{ 3, 5 };
int d = 2;
linear_f f1 = &linear::get;
linear_f f2 = &apply;
linear_f f3 = [&d](linear const* p, int x)
    { return p->get(x + d); };

assert(f1(&l, 4) == 17);
assert(f2(&l, 4) == 17);
assert(f3(&l, 2) == 17);
```

Анализ примера

- Переменные f1, f2, f3 имеют одинаковый тип
  - `std::function<int(linear const*, int)>;`
- Проинициализированы значениями разных типов:
  - `int (linear::*)(int) const;`
  - `int (const linear*, int);`
  - compiler-defined functional object with `int operator()(const linear*, int)`.
- После инициализации первоначальный тип этих значений забывается.
- В частности, законны присваивания вида `f1 = f2;`
- Однако в момент вызова `f1(&l, 4)` тип завёрнутого значения вспоминается, чтобы правильно делегировать вызов.

Общий вывод

- Type erasure (стирание типов) обеспечивает полиморфизм времени выполнения.
- Этот полиморфизм не основан на наследовании от общего предка-интерфейса.
- Позволяет с объектами разных типов работать через универсальную обёртку единого типа.
- Обёртка предоставляет единый интерфейс.
- Для различных обёртываемых типов реализация этого интерфейса различается.
- Как этот механизм реализовать своими руками?

Нулевой вариант

- Никак не заворачивать значения.
- Фактический тип объекта совпадает с объявленным.
- Полиморфизм возможен только на этапе компиляции.
- Универсальные алгоритмы оформляются шаблонами.
- Определения - только в заголовочных файлах.
- Компилируется долго.
- Большой объём исполняемого кода.
- Широкий простор для оптимизаций - наибольшая скорость.
- И наименьшая гибкость.

Рудиментарный вариант:

- Значение произвольного типа передаётся через указатель void*;
- ```
  void qsort_r(
      void *base,
      size_t count,
      size_t size,
      int (*cmp)(const void *l, const void *r, void *st),
      void *st);
  ```
- Ответственность за контроль типов - на человеке.
- Наибольшая гибкость.
- Потеря производительности на косвенный вызов.
  ```

Простой вариант, завёртывание

```
struct any_wrapper {
public:
    template<typename T>
    any_wrapper(T& var):
        data { static_cast<void*>(&var) },
        type { typeid(var) }
    {}
public:
    void* data;
    std::type_index type;
};
```

Простой вариант, развёртывание

```
template<typename T>
T& unwrap(const any_wrapper& any) {

    if(any.type != typeid(T))
        throw std::logic_error("wrapped type mismatch");

    return *(static_cast<T*>(any.data));
}
```

Анализ решения:

- Указатель на фактические данные преобразуется к void*.
- Первоначальный тип с точки зрения компилятора забывается.
- На этапе выполнения можно вспомнить первоначальный тип.
- Контроль типов происходит во время выполнения.
- Развёртывание безопасно.
- Единая точка входа значений в обёртку - “полиморфный” конструктор.
- Нельзя завернуть копию значения, только указатель на объект.
- Осторожно: typeid неточен: typeid(T&) == typeid(T).
- Пользователь вынужден самостоятельно приводить тип к ожидаемому.
- Нет абстрагирования интерфейса от представления.

На пути к лучшему решению

- Отделить поведение (интерфейс) от представления (состояния).
- Представление можно передавать через void*.
- Поведение реализуется фунциями с аргументом void*, которые гарантированно правильно его интерпретируют.
- Гарантия правильности поведения - автоматическая генерация этих функций.
- Автогенерация обеспечивается метапрограммированием.

Более утончённое решение (начало)

```
class counter_ref {
private:
    void *repr_;
    void (*inc_)(void*);
    void (*dec_)(void*);

public:
    counter_ref& operator++() { inc_(repr_); return *this; }
    counter_ref& operator--() { dec_(repr_); return *this; }
    // see next slide...
};
```

Более утончённое решение (продолжение)

```
class counter_ref {
// see the previous slide
public:
    template <typename T>
    counter_ref(T& value) :
        repr_(static_cast<void*>(&value)),
        inc_([](void *r) { ++(*static_cast<T*>(r)); },
        dec_([](void *r) { --(*static_cast<T*>(r)); }
    {}
};
```

Анализ реализации

- Нет нужды хранить typeid и динамически проверять правильность типа.
- Вместо того, чтобы вытаскивать из обёртки значение ожидаемого типа,
- …поручаем обёртке самой выполнить нужную операцию над значением.
- Вместо typeid обёртка хранит только реализацию наперёд заданного интерфейса для значения со стёртым типом.
- Объект-обёртка не помнит фактический тип завёрнутого значения…
- …помнит лишь набор функций для работы с ним, которые внутри себя инкапсулируют знание этого типа.
- Как управлять временем жизни объекта?

Следующий шаг: управление временем жизни

```
class any_counter {
    void *repr_;
    void (*deleter_)(void*);
public:
    template <typename T>
    any_counter(T value) :
        repr_(static_cast<void*>(new T{ value })),
        deleter_([](void *r) { delete static_cast<T*>(r) }
    {}
    ~any_counter() { deleter_(repr_); }
};
```

Анализ решения

- Обёртка по-прежнему хранит указатель на значение.
- Этот указатель инициализируется копией исходного значения в куче.
- “Удаляемость” - это такой же элемент поведения, часть интерфейса обёртываемого типа.
- Поэтому для удаления заводится ещё одна функция, знающая тип завёрнутого значения и скрывающая этот тип от клиента.
- Дополнительное преимущество: можно использовать семантику перемещения (на слайде не показано).
- Каждый объект-обёртка должен хранить много указателей на функции.

Окончательное решение: идея

- Интерфейс (чисто абстрактный класс) с нужными методами.
  - Также включает метод клонирования - создать копию текущего объекта.
- Унаследованный от него шаблон прослойку для конкретного типа.
- Обёртка содержит (умный) указатель лишь на интерфейс…
  - и тем самым скрывает реализацию, включая конкретный тип прослойки.
- Конструктор создаёт объект-прослойку конкретного типа…
  - и забывает этот тип, приводя к базовому классу-интерфейсу.
- Методы для удобства: копирование, перемещение, присваивание.
- Практический пример: универсальная обёртка над любыми типами, поддерживающими сериализацию в поток `operator<<`.

Окончательное решение: интерфейс

```
struct ostreamable_impl_base;
using ostreamable_impl_base_ptr = std::unique_ptr<ostreamable_impl_base>;

struct ostreamable_impl_base {
  virtual ostreamable_impl_base_ptr clone() const = 0;
  virtual void to_stream(std::ostream &s) const = 0;

  virtual ~ostreamable_impl_base();
};
```

Окончательное решение: прослойка

```
template <typename T>
class ostreamable_impl : public ostreamable_impl_base {
public:
  template <typename U>
  explicit ostreamable_impl(U&& value) : value_{ std::forward<U>(value) }
  {}

  ostreamable_impl_base_ptr clone() const override
    { return std::make_unique<ostreamable_impl<T>>(value_); }
  void to_stream(std::ostream &s) const override { s << value_; }

private:
  std::decay_t<T> value_;
};
```

Окончательное решение: обёртка

```
class ostreamable {
public:
  template <typename T, typename = not_ostreamable<T>>
  ostreamable(T&& value) : impl_{ OSTREAMABLE_MAKE_IMPL(value) }
  {}
  // . . .
  void to_stream(std::ostream &s) const { if (impl_) impl_->to_stream(s); }
private:
  ostreamable_impl_base_ptr impl_;
};

std::ostream& operator<<(std::ostream &s, ostreamable const& x)
  { x.to_stream(s);  return s; }
```

Окончательное решение: вспомогательное

```
#define OSTREAMABLE_MAKE_IMPL(arg_)      \
  std::make_unique<ostreamable_impl<T>>( \
    std::forward<T>(arg_))

class ostreamable;

template <typename T>
using not_ostreamable = std::enable_if_t<
    !std::is_same_v<ostreamable, std::decay_t<T>>>;
```

Окончательное решение: использование

```
std::vector<ostreamable> v; // гетерогенный вектор!

v.emplace_back(1002);
v.emplace_back("abcde");
v.emplace_back(custom_streamable{});

for (auto const& x : v) {
    std::cout << x << std::endl;
}
```

Окончательное решение: анализ

- Управление временем жизни обёрнутого значения автоматизировано:
  - значение есть подобъект прослойки и уничтожается вместе с ней;
  - прослойка принадлежит обёртке через умный указатель.
- Интерфейс содержит метод клонирования (служебный) и прикладные методы, отражающие специфику предметной области.
- Объекты различных типов могут быть вписаны в прослойки, обладающие одинаковым интерфейсом.
- Типы прослоек по-прежнему различаются.
- Зато прослойки можно вложить в единый тип-обёртку.
- Накладные расходы: вызовы виртуальных методов.

Обработка списков на этапе компиляции

Постановка задачи

- Шаблоны позволяют вычислять одиночные значения-константы.
- Позволяет разгрузить этап выполнения.
- Можно гарантировать правильность вычисленных значений.
- Для приложений может понадобиться вычисление таблиц значений:
  - таблица простых чисел;
  - таблица тригонометрических функций (с фиксированной запятой);
  - таблица числа единиц в двоичном представлении 8-битных целых беззнаковых;
  - поразрядное представление длинного целого числа.
- Нужны инструменты обработки списков на этапе компиляции.
- Существующими библиотеками эта задача покрыта недостаточно.

Элементарные определения

```
template <typename T, T... Xs>
struct list {
  using item_type = T;
  static constexpr auto size = sizeof...(Xs);
};

template <typename U> // should be list<T, T... Xs>
using item_t = typename U::item_type;

template <typename U>
constexpr auto size_v = U::size;

template <typename U>
constexpr bool is_empty_v = (size_v<U> == 0);
```

Простейшие конструирующие операции

```
template <typename U, item_t<U> X>
struct append; // never reach this case

template <typename T, T... Xs, T X>
struct append<list<T, Xs...>, X> {
  using type = list<T, Xs..., X>;
}; // same for prepend

template <typename U, typename V>
struct concat; // never reach this case

template <typename T, T... Xs, T... Ys>
struct concat<list<T, Xs...>, list<T, Ys...>> {
  using type = list<T, Xs..., Ys...>;
};
```

Простейшие операции-анализаторы

```
template <typename U>
struct uncons; // never reach this case

template <typename T, T X, T... Xs>
struct uncons<list<T, X, Xs...>>
{
  static constexpr T head = X;
  using tail = list<T, Xs...>;
};
```

- Требуется непустой список: в противном случае ошибка компиляции.
- Операция uncons вычисляет сразу два результата: голову и хвост.

Первая сложность: деконструкция справа

- Вместо головы и хвоста - последний элемент и префикс.
- Встроенные механизмы языка не дают разобрать список с конца.
- Пакет параметров должен стоять последним среди параметров шаблона.
- Остаётся рекурсивное решение.

```
template <typename U>
struct unconsr; // never reach this case

template <typename U>
using init_t = typename unconsr<U>::init;

template <typename U>
constexpr item_t<U> last_v = unconsr<U>::last;
```

Первая сложность: деконструкция справа

```
template <typename T, T X>
struct unconsr<list<T, X>> {
  static constexpr T last = X;
  using init = list<T>;
};

template <typename T, T X, T... Xs>
struct unconsr<list<T, X, Xs...>> {
private:
  using remainder = list<T, Xs...>;
public:
  static constexpr T last = last_v<remainder>;
  using init = prepend_t<init_t<list<T, Xs...>>, X>;
};
```

- База рекурсии - список из одного элемента.
- Единственный элемент и есть последний.
- Префикс - пустой список.

- Шаг рекурсии - список произвольной длины…
- … отличной от 1 (перехватывается выше).
- Последний элемент хвоста есть последний элемент всего списка.
- Префикс списка есть префикс его хвоста с приписанным слева первым элементом.

Переворачивание списка

```
template <typename U>
struct revert
{
  using type = append_t<
      revert<tail_t<U>>::type,
      head_v<U>>;
};

template <typename T>
struct revert<list<T>>
{
  using type = list<T>;
};
```

- Шаг рекурсии - непустой список.
- Чтобы перевернуть непустой список, нужно:
  - отделить первый элемент;
  - перевернуть остаток;
  - поставить изъятый элемент в конец.

- База рекурсии - пустой список.
- Инверсия пустого списка есть пустой список.

Разделение по предикату

```
template <
  typename U,
  typename V,
  typename W,
  template <auto> typename P,
  auto X,
  bool B>
struct partition_loop_if
// only when B is true
{
  using step = partition_loop<
      U, append_t<V, X>, W, P>;
};
```

```
template <
  typename U,
  typename V,
  typename W,
  template <auto> typename P,
  auto X>
struct partition_loop_if<
    U, V, W, P, X, false>
{
  using step = partition_loop<
      U, V, append_t<W, X>, P>;
};
```

Разделение по предикату

```
template <
  typename U,
  typename V,
  typename W,
  template <auto> typename P>
struct partition_loop // only reached if U is not empty
{
private:
  using step = typename partition_loop_if<
    tail_t<U, V, W, P, head_v<U>, P<head_v<U>>::value>::step;
public:
  using left = typename step::left;
  using right = typename step::right;
};
```

Разделение по предикату

```
template <
  typename T,
  typename V,
  typename W,
  template <auto> typename P>
struct partition_loop<list<T>, V, W, P>
{
  using left = V;
  using right = W;
};

template <typename U, template <auto> typename P, typename T = item_t<U>>
using partition = partition_loop<U, list<T>, list<T>, P>;
```

Генератор простых чисел

```
template <typename T, unsigned N>
struct primes
{
  using item_type = T;
  using init = prepend_t<generate_t<iota<T, 3, N, 2>>, 2>;

  template <typename S>
  using can_proceed = std::bool_constant<!is_empty_v<S>>;

  template <typename S>
  using get_item = std::integral_constant<T, head_v<S>>;

// see next slide
```

Генератор простых чисел

```
// continued

  template <typename S>
  struct step {
  private:
    template <unsigned X>
    struct predicate : std::bool_constant<(X % head_v<S> == 0)> {};
  public:
    using result = typename partition<S, predicate>::right;
  };
};
```

Обзор библиотеки

- span
- partition
- sort
- fmap
- zip_with
- for_each
- foldl
- foldr
- unfoldr
- generate

Генераторы последовательностей:

- Целые числа из диапазона.
- Числа Фибоначчи.
- Простые числа
  (решето Эратосфена)

Тесты

- На этапе компиляции.
- Через static_assert.

Пример теста

```
using t_1 = list<int, 0>;
using t_2 = prepend_t<t_1, 1>;
using t_3 = append_t<t_2, 2>;
using t_4 = prepend_t<t_3, 3>;
using t_5 = append_t<t_4, 4>;

using expected = list<int, 3, 1, 0, 2, 4>;

static_assert(std::is_same<t_5, expected>::value);
```

Пример теста

```
using arg = list<int, 0, 9, 1, 8, 2, 7, 3, 6, 4, 5>;

using result = span<
    arg,
    aux::bind_1st<int, 7>::is_not_equal_to>;

using expected_left = list<int, 0, 9, 1, 8, 2>;
using expected_right = list<int, 7, 3, 6, 4, 5>;

static_assert(std::is_same<result::left, expected_left>::value);
static_assert(std::is_same<result::right, expected_right>::value);
```

Пример теста

```
using arg = list<int, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9>;

using result = partition<
    arg,
    aux::bind_1st<int, 2>::is_divisor_of>;

using expected_left = list<int, 0, 2, 4, 6, 8>;
using expected_right = list<int, 1, 3, 5, 7, 9>;

static_assert(std::is_same<result::left, expected_left>::value);
static_assert(std::is_same<result::right, expected_right>::value);
```