

ВЕКТОРА И SFINAE

На долгом пути к `std::vector` – проектирование реалистичного
шаблонного контейнера

К. Владимиров, Intel, 2022
mail-to: konstantin.vladimirov@gmail.com

➤ Вектора

□ SFINAE

□ Вариабельные шаблоны

□ Графы

От ручного выделения к векторам

```
int *n = new int[10];
```

```
n[5] = 5;
```

```
// тут много кода
```

```
// какой сейчас размер у n?
```

```
// стереть крайний элемент?
```

```
// пуст ли теперь n?
```

```
// не забыть delete[]
```

```
vector<int> v(10);
```

```
v[5] = 5;
```

```
// тут много кода
```

```
size_t vsize = v.size();
```

```
v.pop_back();
```

```
if (v.empty()) { что-то }
```

```
// ресурсы будут освобождены
```

Два непростых вопроса

- Допустим я хочу завести в своей программе вектор из константных ссылок

```
std::vector<const int &> v;
```

- Что скажете?

- Допустим я не знаю тип переменной x, но знаю, что это контейнер

```
template<typename Cont> void foo(const Cont& x) {  
    if(x.empty()) return;  
    // do something  
}
```

- Могу ли я бы уверенным, что это будет работать для всех контейнеров?

Требования к элементам контейнеров

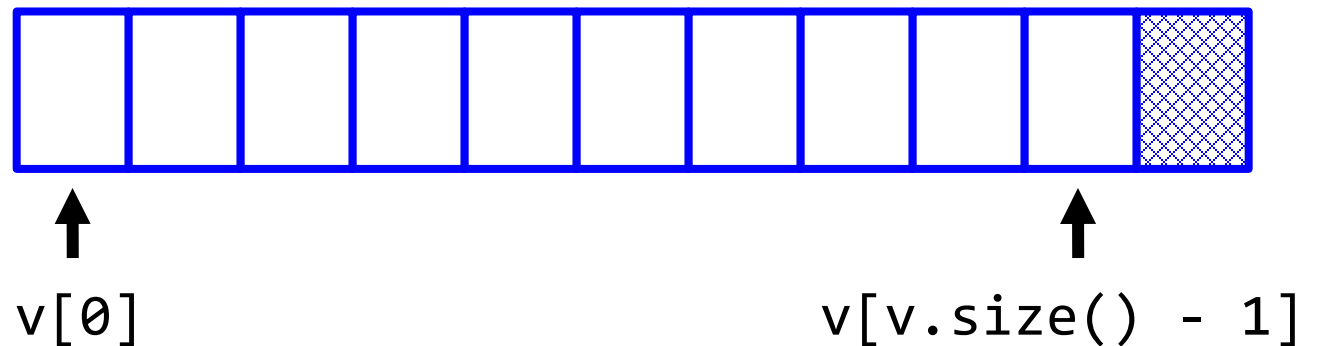
- Общие для всех контейнеров методы `[container.requirements.general]`
 - `empty` — проверка пустоты контейнера
 - `swap` — обмен контейнерных переменных содержимым
 - `size` (кроме `array`) — действительный размер контейнера
 - `clear` (кроме `array`) — очистка контейнера
 - `begin`, `end`, `cbegin`, `send` — получение итераторов (см. далее)
- Требования к элементам зависят от конкретной операции, но чаще всего
 - `DefaultConstructible` — требование к наличию конструктора по умолчанию
 - `MoveConstructible` — требование к наличию конструктора перемещения или копирования

Гарантии непрерывности памяти

```
// функция init написана в старом стиле  
template <typename T> void init (T* arr, size_t size);
```

```
// но её можно использовать с векторами
```

```
vector<T> t(n);  
T *start = &t[0];  
init_t(start, n);  
assert(t[1] == start[1]);
```



When choosing a container, remember vector is best. Leave a comment to explain if you choose from the rest

(c) Tony van Eerd

Неприятное исключение: `vector<bool>`

```
vector<bool> t(n);  
bool *start = &t[0]; // это не скомпилируется, но представим  
assert (t[1] == start[1]); // oops!
```

Важно запомнить две вещи

- `vector<bool>` не удовлетворяет соглашениям контейнера `vector`
- `vector<bool>` не содержит элементов типа `bool`
- Не используйте `vector<bool>` для обобщённого программирования

```
using vector_bool = vector<bool>;
```

```
vector_bool x(10); // условно ок, но тут лучше std::bitset
```

Задача: что можно здесь улучшить?

```
vector<int> v;  
  
for (int i = 0; i != N; ++i)  
    v.push_back(i);
```


Ответ: вектор не терпит халатности

```
vector<int> v;
```

```
v.reserve(N);
```

```
for (int i = 0; i != N; ++i)
```

```
    v.push_back(i); // теперь здесь не будет перевыделений
```

- При вставке в конец вектору могут потребоваться реаллокации памяти
- Это означает, что всегда полезно думать о памяти вектора не меньше, чем о памяти динамического массива

Ещё про size и capacity

- `size` это сколько элементов у вектора уже есть
- `capacity` это сколько элементов в нём может быть до первого перевыделения

```
vector<int> v(10000);
```

```
assert (v.size() == 10000);  
assert (v.capacity() >= 10000);
```

Размер это что-то чем можно в явном виде управлять в отличии от ёмкости

```
v.resize(100);  
assert (v.size() == 100);  
assert (v.capacity() >= 10000);
```

Амортизация

- При написании метода `push`, вам предлагалось оценить его алгоритмическую сложность
- Проблема в том, что она очевидно $O(1)$ если не надо реаллоцировать и $O(n)$ если надо
- То есть мы платим иногда. Это примерно как купить машину и платить только за бензин пока машина не износится, а потом купить новую
- В экономике распределение стоимости товара по стоимости его периода эксплуатации называется амортизацией товара
- Амортизированное $O(n)$ обозначается $O(n) +$

Амортизированная стоимость

- По определению амортизированная стоимость операции это стоимость N операций, отнесённая к N
- Для динамического массива $c_i = 1 + [realloc] * (i - 1)$
- Амортизированная стоимость одной вставки будет $\frac{\sum_i c_i}{N}$ для N вставок
- Допустим, мы, если реаллокация нужна, растим массив на 10 элементов

$$\sum_i c_i = ?$$

Амортизированная стоимость

- По определению амортизированная стоимость операции это стоимость N операций, отнесённая к N
- Для динамического массива $c_i = 1 + [\text{realloc}] * (i - 1)$
- Амортизированная стоимость одной вставки будет $\frac{\sum_i c_i}{N}$ для N вставок
- Допустим, мы, если реаллокация нужна, растим массив на 10 элементов

$$\sum_i c_i = N + \sum_{k=1}^{N/10} 10 \cdot k = O(N^2)$$

- Заметим, что это очень плохая стратегия. Амортизированная сложность push будет $\frac{O(N^2)}{N} = O(N)$ +. Можем ли мы придумать и доказать нечто лучшее?

Лучшая стратегия

- Прирост вдвое

$$\frac{\sum c_i}{N} = \frac{N + \sum_{j=1}^{\lg(N)} 2^j}{N} = \frac{O(N)}{N} = O(1) +$$

- Видно, что разница есть: при одной стратегии у нас в среднем линейное а при другой в среднем постоянное время вставки
- Увы, взять сумму $\sum_{j=1}^{\lg(N)} 2^j$ в общем уже не так просто, а при более сложных стратегиях, это становится мучительно
- Можем ли мы упростить себе жизнь?

Дополнение: метод потенциала

- Выберем функцию потенциала $\Phi(n)$ так, чтобы $\Phi(0) = 0, \Phi(n) \geq 0$
- Здесь n это номер шага
- Амортизированная стоимость это стоимость плюс изменение потенциальной функции $c_n + \Phi(n) - \Phi(n - 1)$
- Выбор потенциальной функции облегчает вычисления потому что
$$\sum_i (c_i + \Phi(i) - \Phi(i - 1)) = \Phi(n) - \Phi(0) + \sum_i c_i \geq \sum_i c_i$$
- Удачный выбор сделает выражение $\sum_i (c_i + \Phi(i) - \Phi(i - 1))$ проще, чем $\sum_i c_i$
- Обсуждение: как выбрать для массива?

Дополнение: метод потенциала

- Для массива поскольку при реаллокации вдвое $2 * s_n \geq c_n$

$$\Phi(n) = 2 * s_n - c_n$$

- Без реаллокации

$$c_i + \Phi(i) - \Phi(i-1) = 1 + (2 * s_i - C) - (2 * s_{i-1} - C) = 1 + 2(s_i - s_{i-1}) = 3$$

- С реаллокацией $\Phi(i-1) = 2k - k = k$, $\Phi(i) = 2(k+1) - 2k = 2$

$$c_i + \Phi(t_i) - \Phi(t_{i-1}) = (k+1) + 2 - k = 3$$

- В итоге в любом случае $\sum c_i \leq 3N$ и мы доказали асимптотику $O(1)$
- В качестве упражнения проанализируйте стратегию роста в $\log(N)$ раз

Обсуждение

- Выбор простого роста вдвое не всегда лучшая стратегия
- Реальная стратегия из `libstdc++` несколько сложнее и обладает рядом приятных теоретических свойств

```
const size_type __len = size() + std::max(size(), __n);
```

- Попробуйте дома проанализировать эту стратегию и обосновать почему она выбрана в качестве основной

Два механизма инициализации

- Расширенный синтаксис
- Явный конструктор из списка инициализации

```
class B {  
    int a_;  
public:  
    B (int a) : a_(a) {}  
    B (std::initializer_list<int> il);  
};
```

B b(1), c{1}; // теперь они вызывают разные конструкторы

Списочная инициализация: вектора

```
// это вектор [14, 14, 14]
```

```
vector<int> v1 (3, 14);
```

```
// а это вектор [3, 14]
```

```
vector<int> v2 {3, 14};
```

- Это связано с наличием у вектора **нескольких** конструкторов

- `v(10);` // размер 10, инициализация по умолчанию

- `v(10, 1);` // размер 10, инициализировать единицами

- `v {10, 1};` // размер = размеру списка, инициализация списком

То же для ваших контейнеров

- Хорошая новость: `initializer_list` это тоже разновидность последовательного контейнера и его можно обходить итераторами

```
template <typename T> class Tree {  
    // тут какая-то специфика дерева  
    bool add_node (T& data);  
  
public:  
    Tree(std::initializer_list<T> il) {  
        for (auto ili = il.begin(); ili != il.end(); ++ili)  
            add_node(*ili);  
    }  
};
```

- Плохая новость: вам теперь надо следить есть ли он в классе

Простое правило для {}

- Если в классе совсем нет конструкторов, это агрегат как в C

```
struct S { int x, y; }; S s = {1, 2}; // aggregate
```

- Иначе, если есть конструктор из initializer_list, возьмётся он
- Иначе, если есть любой другой конструктор, возьмётся он

```
struct S {  
    int x, y;  
    S(int n) : x(n), y(n) {}  
};
```

```
S s = {3}; // ctor
```

Первое представление об итераторах

```
vector<int> v(10);
```

```
// pi это указатель
```

```
auto pi = &v[0];
```

```
pi += 3;
```

```
assert (*pi == v[4])
```

```
// как узнать, что pi в конце?
```

```
vector<int> v(10);
```

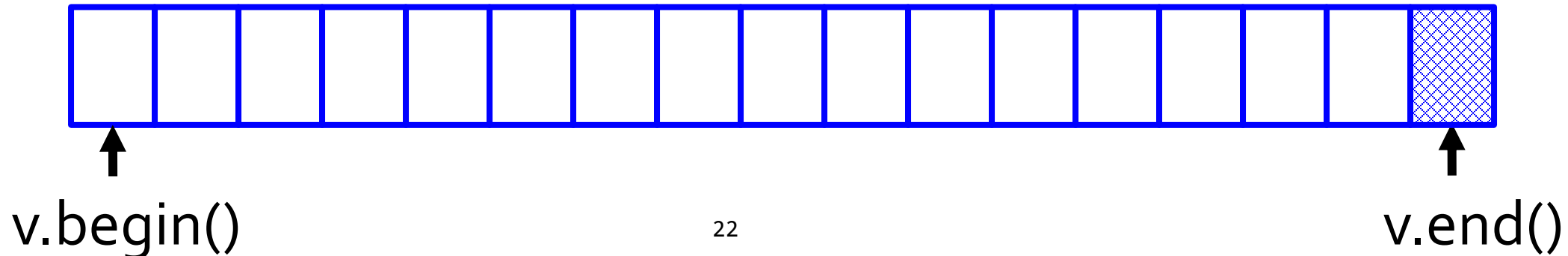
```
// vi это итератор
```

```
auto vi = v.begin();
```

```
vi += 3;
```

```
assert (*vi == v[4]);
```

```
if (vi == v.end()) { что-то }
```



Абстракция указателя

- Важно, что итераторы не являются указателями, они абстрагируют их.
- В итоге любой контейнер может быть сконструирован из любого диапазона.

```
std::list<int> l {1, 2, 3};
```

```
std::vector<int> v(l.begin(), l.end());
```

- Это потрясающе удобно чтобы перекидывать один контейнер в другой
- Как бы вы написали конструктор из пары итераторов?

Конструирование из итераторов

- Наивная попытка вызывает у нас небольшую проблему

```
template <typename T> class MyVector {  
    // ....  
public:  
    MyVector (size_t nelts, T value); // 1  
    template <typename Iter> MyVector (Iter fst, Iter lst); // 2  
    // ....  
MyVector<int> mvec (2, 2); // ошибка, выбран 2
```


□ Вектора

➤ SFINAE

□ Вариабельные шаблоны

□ Графы

Обсуждение: провал подстановки

- Что если подстановка в некотором контексте не может быть выполнена?

```
template<typename T>  
typename T::ElementT at(T const& a, int i);  
  
int *p = new int[30];  
  
auto a = at<int*>(p, i); // Substitution failure
```

- Что если вывод типов в некотором контексте провален?

```
template <typename T> T max(T a, T b);  
  
int g = max(1, 1.0); // Deduction failure
```

SFINAE

- **Substitution Failure Is Not An Error** (провал подстановки не является ошибкой)

```
template <typename T> T max(T a, T b);  
template <typename T, typename U> auto max(T a, U b);
```

```
int g = max(1, 1.0); // подстановка в 1 провалена  
                    // подстановка в 2 успешна
```

- Если в результате подстановки в **непосредственном контексте** класса (функции, алиаса, переменной) возникает **невалидная конструкция**, эта подстановка неуспешна, но не ошибочна
- В этом случае второй фазы поиска имён просто не выполняется

SFINAE и ошибки

- Не любая ошибочная конструкция это SFINAE. Важен контекст подстановки.

```
int negate (int i) { return -i; }  
  
template <typename T> T negate(const T& t) {  
    typename T::value_type n = -t();  
    // тут используем n  
}  
  
negate(2.0); // ошибка второй фазы
```

- Здесь в контексте сигнатуры и шаблонных параметров нет никакой невалидности

SFINAE и ошибки

- Не любая ошибочная конструкция это SFINAE. Важен контекст подстановки.

```
int negate (int i) { return -i; }
```

```
template <typename T> T::value_type negate(const T& t) {  
    typename T::value_type n = -t();  
    // тут используем n  
}
```

```
negate(2.0); // substitution failure
```

- Здесь в контексте сигнатуры и шаблонных параметров выводится $T \rightarrow \text{double}$ и разумеется `T::value_type` невалидно

Обсуждение

- Техника SFINAE кажется очень простой, но вообще-то её приложения многочисленны и часто очень нетривиальны
- Рассмотрим задачу: у вас есть два типа и вам нужно определить равны ли они.

```
template <typename T, typename U> int foo() {  
    // как вернуть 1 если T == U и 0 если нет?  
}
```

- Обратим внимание, что это задача отображения из типов на числа.
- Прежде чем её решать, решим обратную задачу.

Интегральные константы

- Отображение из целых чисел на типы называется интегральной константой

```
template <typename T, T v> struct integral_constant {  
    static const T value = v;  
    typedef T value_type;  
    typedef integral_constant type;  
    operator value_type() const { return value; }  
};
```

- Возможна даже арифметика

```
using ic6 = integral_constant<int, 6>;  
auto n = 7 * ic6{};
```

Истина и ложь для типов

- Самые полезные из интегральных констант – самые простые

```
using true_type = integral_constant<bool, true>;
```

```
using false_type = integral_constant<bool, false>;
```

- Всё это есть в стандарте: `std::integral_constant` и т.д.
- Попробуем написать простой определитель, чтобы проверить одинаковые ли два типа

```
template<typename T, typename U>  
struct is_same : std::false_type {};
```

- По умолчанию разные. Что дальше?

Равенство типов

- Теперь можно решить задачу определения равенства типов

```
template<typename T, typename U>  
struct is_same : std::false_type {};
```

```
template<typename T>  
struct is_same<T, T> : std::true_type {}; // для T == T
```

```
template<typename T, typename U>  
using is_same_t = typename is_same<T, U>::type;
```

- Благодаря SFINAE, будет работать

```
assert(is_same<int, int>::value && !is_same<char, int>::value);
```

Определители и модификаторы

Определитель: является ли тип ссылкой

```
template <typename T> struct is_reference : false_type {};  
template <typename T> struct is_reference<T&> : true_type {};  
template <typename T> struct is_reference<T&&> : true_type {};
```

Модификатор: убираем ссылку с типа, если ссылки не было, то оставляем тип

```
template <typename T> struct remove_reference { using type = T; };  
template <typename T> struct remove_reference<T&> { using type = T; };  
template <typename T> struct remove_reference<T&&> { using type = T; };
```

Для модификатора полезен алиас

```
template <typename T>  
using remove_reference_t = typename remove_reference<T>::type;
```

Четырнадцать категорий

- Любой тип в языке C++ попадает хотя бы под одну из перечисленных ниже категорий

```
is_void  
is_null_pointer  
is_integral, is_floating_point // для T и для cv T& транзитивно  
is_array; // только встроенные, не std::array  
is_pointer; // включая указатели на обычные функции  
is_lvalue_reference, is_rvalue_reference  
is_member_object_pointer, is_member_function_pointer  
is_enum, is_union, is_class  
is_function // обычные функции
```

- Использование довольно тривиально

```
std::cout << std::boolalpha << std::is_void<T>::value << '\n';
```

СВОЙСТВА ТИПОВ

- Также очень полезны определители свойств типов

`is_trivially_copyable` // побайтово копируемый, memcpu
`is_standard_layout` // можно адресовать поля указателем
`is_aggregate` // доступна агрегатная инициализация как в C
`is_default_constructible` // есть default ctor
`is_copy_constructible`, `is_copy_assignable`
`is_move_constructible`, `is_nothrow_move_constructible`
`is_move_assignable`
`is_base_of` // B является базой (транзитивно, включая сам тип)
`is_convertible` // есть преобразование из A к B

- И многие другие (их реально десятки)

Обсуждение: `std::copy`

- Рассмотрим наивное копирование, чем-то похожее на алгоритм `std::copy`

```
template <typename InIter, typename OutIter>
OutIter cross_copy(InIter fst, InIter lst, OutIter dst) {
    while (fst != lst) { *dst = *fst; ++fst; ++dst; }
    return dst;
}
```

- Увы, по сравнению с настоящим `std::copy` у него есть проблемы
- Можем ли мы их решить с помощью SFINAE?

Решение проблемы std::copy

- Заведём хелпер и его специализацию для true

```
template<bool Triv, typename In, typename Out> struct CpSel {  
    static Out select(In begin, In end, Out out)  
        { return CopyNormal(begin, end, out); }  
};
```

```
template<typename In, typename Out>  
struct CpSel<true, In, Out> {  
    static Out select(In begin, In end, Out out)  
        { return CopyFast(begin, end, out); } // для простых типов  
};
```

- Теперь сам алгоритм копирования будет просто решать кого он вызывает

Решение проблемы std::copy

- Также тривиально мы решаем проблему с копированием

```
template<typename In, typename Out>
Out realistic_copy(In begin, In end, Out out) {
    using in_type = pointee type (In); // как это написать?
    using out_type = pointee type (Out);

    enum { Sel = std::is_trivially_copyable<in_type>::value &&
                std::is_trivially_copyable<out_type>::value &&
                std::is_same<in_type, out_type>::value };

    return CpSel<Sel, In, Out>::select(begin, end, out);
}
```

Обсуждение

- Теперь единственным облачком на горизонте остался `emplace`

```
struct S {  
    S();  
    S(int, double, int);  
};
```

```
std::vector<S> v;
```

```
v.emplace_back(1, 1.0, 2); // создали на месте
```

- Но как это может работать для любого типа, если мы в общем случае не знаем количество аргументов конструктора?

- Вектора

- SFINAE

- Вариабельные шаблоны

- Графы

Вариабельные шаблоны

- Пример вариабельно шаблонной функции

```
template<typename ... Args> void f(Args ... args);
```

- Способы вызова:

```
f(); // ОК, пачка не содержит аргументов
```

```
f(1); // ОК, пачка содержит один аргумент: int
```

```
f(2, 1.0); // ОК, пачка состоит из: int, double
```

- Специальная конструкция `sizeof...(Args)` либо `sizeof...(args)` возвращает размер пачки в штуках

Паттерны раскрытия

- Говорят, что пачка параметров "раскрывается" в теле функции или класса

```
template<typename ... Types> void f(Types ... args);  
  
template<typename ... Types> void g(Types ... args) {  
    f(args ...);           // → f(x, y);  
    f(&args ...);          // → f(&x, &y);  
    f(h(args) ...);        // → f(h(x), h(y));  
    f(const_cast<const Types*>(&args)...);  
                           // → f(const_cast<const int*>(&x),  
                               const_cast<const double*>(&y));  
}  
  
g(1, 1.0); // → g(int x, double y);
```

Задача: раскрытие пачек

- Допустим `args` это пачка параметров `x, y, z`
- Тогда во что раскроется следующее выражение?

`f(h(args...) + h(args)...);`

- Также интересно во что раскроется следующее

`f(h(args, args...)...);`

Решение

- Допустим `args` это пачка параметров `x, y, z`
- Тогда следующее выражение имеет сложный паттерн раскрытия пачки

```
f(h(args...) + h(args)...); // → f(h(x, y, z) + h(x),  
                                     h(x, y, z) + h(y),  
                                     h(x, y, z) + h(z));
```

- Аналогично (если чувствовать технологию, эти задачи однообразны)

```
f(h(args, args...)...); // → f(h(x, x, y, z),  
                                h(y, x, y, z),  
                                h(z, x, y, z));
```

Снова прозрачная оболочка

- На лекции по rvalue refs была написана почти идеальная прозрачная оболочка для одного аргумента

```
template<typename Fun, typename Arg>  
decltype(auto) transparent(Fun fun, Arg&& arg) {  
    return fun(forward<Arg>(arg));  
}
```

- Можно ли использовать переменный шаблон и переписать её для произвольного количества аргументов?

Снова прозрачная оболочка

- На лекции по rvalue refs была написана почти идеальная прозрачная оболочка для одного аргумента

```
template<typename Fun, typename... Args>  
decltype(auto) transparent(Fun fun, Args&&... args) {  
    return fun(forward<Args>(args)...);  
}
```

- Это очень простое и чисто техническое изменение
- Следует обратить особое внимание на паттерн совместного раскрытия при пробросе

Обсуждение: пробросим функцию?

- В функции-подобном объекте оператор вызова может быть && аннотирован

```
template<typename Fun, typename... Args>  
decltype(auto) transparent(Fun&& fun, Args&&... args) {  
    return std::forward<Fun>(fun)(std::forward<Args>(args)...);  
}
```

- Теперь функции тоже не требуется быть обязательно копируемой
- Выглядит это чуть страшнее, зато теперь тут не к чему особо придраться

Контейнеры тяжёлых классов

- Мы уже говорили о хранении тяжелых классов в контейнерах

```
template <typename T> class Stack {  
    struct StackNode {  
        T elem; StackNode *next;  
        StackNode(T e, StackNode *nxt) : elem (e), next (nxt) {}  
    };  
public:  
    void push(const T& elem) { top_ = new StackNode (elem, top_); }  
    // .... и так далее ....
```

- Подумаем о следующем коде:

```
s.push(Heavy(100, 200, 300)); // всё очень плохо
```

Emplace

- Обычно метод контейнера, который размещает объект, а не пробрасывает его называют **emplace**

```
template <typename T> class Stack {  
    // детали реализации  
public:  
    void push(const T& elem) { top_ = new StackNode (top_, elem); }  
  
    template <typename U> void emplace(U&& ... args) {  
        top_ = new StackNode(top_, forward<U>(args)...);  
    }  
}
```

- В стандартной библиотеке размещение поддерживают все последовательные контейнеры

Интерлюдия: шаблонные методы

- Шаблонный метод вне класса определяется с двумя наборами параметров: своими и своего класса

```
template <typename T>
template <typename... Args>
void Stack<T>::emplace_back(Args &&... args) {
    top_ = new StackElem(top_, std::forward<Args>(args)...);
}
```

- Это не опечатка. Каждый набор идёт отдельно.
- Все наборы совокупно участвуют в template-id метода и это важно для специализации.

Специализация шаблонных методов

- При специализации шаблонных методов, важно понимать: вы должны специализировать их по всем аргументам

```
template <typename T> struct Foo {  
    template <typename U> void foo() { .... }  
};
```

```
template <>  
template <>  
void Foo<int>::foo<int>() { .... }
```

- Иначе это будет частичная специализация

Шаблонные методы против ООП

- Вы должны понимать, что любой открытый шаблонный метод в вашем классе обнуляет инкапсуляцию.

```
class Foo {  
    int donottouch_ = 42;  
  
public:  
    template <typename U> void foo() { .... }  
};  
  
struct MyTag {};  
  
template <> void Foo::foo<MyTag>() { donottouch_ = 14; }
```

Обсуждение

- Тем не менее пока что мы не очень понимаем как использовать SFINAE, пусть даже с переменными шаблонами, для решения проблемы с конструктором из пары итераторов.
- Настало время этим заняться.

void_t

- Появился в C++17 как `std::void_t` но вообще-то довольно прост
`template <typename...> using void_t = void;`
- Интуитивно `void_t <T, U, V>` означает `void` если все типы легальны и нелегален если нелегален хоть один
- Думайте о нём как о логической конъюнкции SFINAE характеристик

Задача: зависимый тип

- С ранних пор была замечена полезность техники SFINAE для трюков и хаков. Классический пример: определить наличие зависимого типа в классе.

```
struct foo { typedef float foobar; };  
struct bar { };
```

```
std::cout << std::boolalpha << ??? foo << " " << ??? bar;
```

- Это снова отображение из типов в целые и без SFINAE, задача опять выглядит не решаемой.

Решение: void_t

- Решение использует SFINAE и void_t

```
template <typename, typename = void>
struct has_typedef_foobar: std::false_type { };

template <typename T>
struct has_typedef_foobar<T,
    std::void_t<typename T::foobar>>: std::true_type{};
```

- Теперь мы можем определить вещи на этапе компиляции

```
struct foo { typedef float foobar; };
std::cout << std::boolalpha << has_typedef_foobar<foo>{};
```

Конструирование из итераторов

- Можно попытаться решить задачу с итераторами вот так

```
MyVector(size_t nelts, T value);
```

```
template <typename Iter,  
          typename = void_t<decltype(*Iter{}),  
                           decltype(++Iter{})>  
          >
```

```
MyVector(Iter fst, Iter lst);
```

- Увы это не слишком изящно. Дело в том, что инкремент требует lvalue.
- Но его-то мы как раз пока и не можем создать. Хотя иногда везет.

Абстракция значения

- В некоторых случаях (например для использования внутри `decltype`) хочется получить значение некоего типа.
- Часто для этого используется конструктор по умолчанию

```
template <typename T> struct Tricky {  
    Tricky() = delete;  
    const volatile T foo ();  
};
```

```
decltype(Tricky<int>().foo()) t; // ошибка
```

- Но что делать, если его нет? Что такое "значение вообще" для такого типа?

Абстракция значения: declval

- Интересный способ решить эти проблемы это ввести шаблон функции (который выводит типы) без тела (чтобы его нельзя было по ошибке вызвать).

```
template <typename T> add_rvalue_reference_t<T> declval();
```

- Теперь всё просто

```
template <typename T> struct Tricky {  
    Tricky() = delete;  
    const volatile T foo ();  
};
```

```
decltype(declval<Tricky<int>>().foo()) t; // ok
```

- Но **какова природа** этого значения?

Обсуждение

- Пожалуй есть всего три функции, для которых имеет смысл возвращать правую ссылку (то есть производить xvalue)
 - `std::move`
 - `std::forward`
 - `std::declval`
- Если вы хотите написать свою функцию, которая будет возвращать `&&` это значит, что
 - Вы что-то делаете не так
 - Вы хотите ещё раз написать одну из упомянутых выше функций
 - Вы пишете функцию, аннотированную как `&&`

Конструирование из итераторов

- Теперь мы видим совсем изящное решение

```
MyVector(size_t nelts, T value);
```

```
template <typename Iter,  
          typename = void_t<decltype(*std::declval<Iter&>()),  
                      decltype(++std::declval<Iter&>())>  
        >
```

```
MyVector(Iter fst, Iter lst);
```

```
....
```

```
MyVector v1(10, 3); // 1, поскольку 2 провалилось
```

```
MyVector v2(v1.begin(), v1.end()); // 2
```

Обсуждение

- Количество вещей, которые нужно знать, чтобы спроектировать свой собственный вектор впечатляет.
- Но вектор уже спроектирован за нас.
- На чём бы потренироваться?

❑ Вектора

❑ SFINAE

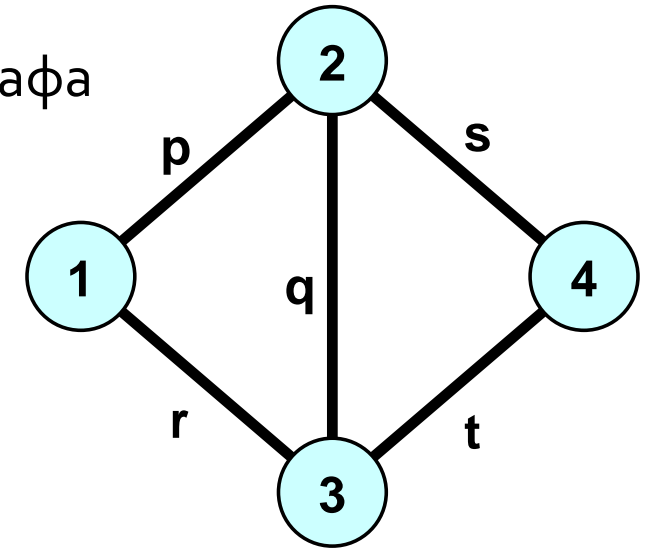
❑ Вариабельные шаблоны

➤ Графы

Представление графа

- У Кнута в ТАОСР приведено следующее представление графа

a	0	1	2	3	4	5	6	7	8	9	10	11	12	13
t	0	0	0	0	1	2	1	3	2	3	2	4	3	4
n	4	5	7	11	6	8	0	9	10	12	1	13	2	3
p	6	10	12	13	0	1	4	2	5	7	8	3	9	11



- Понятно ли почему эта таблица представляет этот граф?
- Важное свойство (заявлено в ТАОСР 7.1.2.6.S): если a это ребро от vi к vj , а b это ребро от vj к vi с тем же именем, то $a = b \wedge 1$ и $b = a \wedge 1$
- Похоже такой граф можно построить не используя ничего кроме `std::vector`

HWCG: представление графа

- Необходимо написать класс графа, представленного как в ТАОСР 7.1.2.6.S и написать для этого представления
 - списочную инициализацию из списка пар вершина-вершина
`KGraph g {{1,2},{1,3},{2,3},{2,4},{3,4}}`
 - обход в ширину
 - обход в глубину
- Допустим также, что с каждым ребром нужно связать дополнительную информацию EL, а с каждой вершиной дополнительную информацию VL.
- Можно ли сделать это вне самого графа?
- Можно ли дать возможность добавить эти данные как параметры графа?

HWCG: формальная постановка

- На стандартном вводе граф в обычном представлении $v_1 \text{ -- } v_2, w_e$:

```
1 -- 2, 4
2 -- 3, 5
3 -- 4, 6
4 -- 1, 1
```

- Необходимо считать его в эффективное представление выше.
- Далее если граф не является двудольным, вывести ошибку. Если граф является двудольным, покрасить вершины первой доли в синий цвет (цвет это атрибут вершины), второй доли в красный цвет. Цвет вершины 1 всегда синий.
- В итоге вывести на стандартный вывод все вершины и цвет каждой

```
1 b 2 r 3 b 4 r
```

Литература

- [CC11] ISO/IEC 14882 – "Information technology – Programming languages – C++", 2011
- [BS] Bjarne Stroustrup – The C++ Programming Language (4th Edition) , 2013
- [EM] Scott Meyers, "Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14"
- [SM] Scott Meyers "Type Deduction and Why You Care", CppCon, 2014
- [VJ] Davide Vandevoorde, Nicolai M. Josuttis – C++ Templates. The Complete Guide, 2nd edition, Addison-Wesley Professional, 2017

СЕКРЕТНЫЙ УРОВЕНЬ

Свертки

Свёртки

Паттерн свёртки	Результирующее выражение
<code>... op pack</code>	<code>(... (p1 op p2) op p3) ... op pN)</code>
<code>init ... op pack</code>	<code>(... (init op p1) op p2) ... op pN)</code>
<code>pack op ...</code>	<code>(p1 op (p2 op (... (pN-1 op pN) ...))</code>
<code>pack op ... fini</code>	<code>(p1 op (p2 op (... (pN op fini) ...))</code>

```
template <typename ... T>  
auto sum_all (T ... args) { return (args + ...); }
```

```
template <typename ... T>  
void print_all (T ... args) { (cout << ... << args) << endl; }
```

Проблемы print_all

```
template <typename ... T>  
void print_all (T ... args) { (cout << ... << args) << endl; }
```

- Очевидно, что print_all записанный как есть не вставляет между выводимыми числами пробельные символы

```
print_all(1, 1.5, 3); // → 11.53
```

- Как заставить его это сделать?

Решение: функтор AddSpace

- Вандерворд и Йосьюттис [3] предлагают следующее решение

```
template <typename T> class AddSpace {  
    const T& ref;  
public:  
    AddSpace(const T& r): ref(r) {}  
    ostream& operator<< (ostream& os, AddSpace s) {  
        return os << s.ref << ' ' ;  
    }  
};  
  
template <typename ... T> void print_all (T ... args) {  
    (cout << ... << AddSpace(args)) << "\n";  
}
```


Экзотические свёртки (самостоятельно)

- Представьте, что у вас есть дерево с узлом

```
template <typename T> struct Node {  
    T data;  
    Node *left;  
    Node *right;  
};
```

- Вы хотите заставить заработать код:

```
Node *top = получаем узел 1;
```

```
Node *seven = tree_get(top, right, right, left); // узел 7
```

```
Node *four = tree_get(top, left, right, left);    // узел 4
```

- Домашняя наработка: написать такую функцию (проще чем кажется)

