Искусство не писать код

Углянский Иван Евгеньевич
Excelsior@Huawei, НГУ
ivan.ugliansky@gmail.com
@dbg_nsk

Соловьёв Владимир Валерьевич
Huawei, НГУ, СУНЦ
vladimir.conwor@gmail.com
vk.com/conwor

В итоге у нас будут стандартные каталоги обобщённых компонентов с чётко определенными интерфейсами и чётко определенными сложностными характеристиками. Программисты больше не будут программировать на микро-уровне. Вам **никогда** не придётся писать бинарный поиск снова.

Александр Степанов, автор STL, 1995-ый год

```
std::not1(boost::make_adaptable<bool,char_t>(boost::bind(&std::
ctype<char_t>::is, &ct, std::ctype_base::space, _1)));
```

Пример из документации библиотеки Boost, 2016-ый год

Принципы написания обобщенного кода в C++ слишком сильно отличаются от того, как пишется остальной код.

Бьёрн Страуструп, автор языка C++, 2017-ый год

Функция max для типа int

```
int max(int x, int y) {
    return (x > y) ? x : y;
}
```

Функция max для типа int

```
int max(int x, int y) {
    return (x > y) ? x : y;
}
```

А если написать max для типа float, в чём будут отличия?

Функция max для типа int

```
int max(int x, int y) {
    return (x > y) ? x : y;
}
```

А если написать max для типа float, в чём будут отличия?

```
float max(float x, float y) {
    return (x > y) ? x : y;
}
```

Функция max для типа int

```
int max(int x, int y) {
    return (x > y) ? x : y;
}
```

А если написать max для типа float, в чём будут отличия?

```
float max(float x, float y) {
    return (x > y) ? x : y;
}
```

Функция max для другого типа

```
int max(int x, int y) {
    return (x > y) ? x : y;
}



float max(float x, float y) {
    return (x > y) ? x : y;
}
```

Функция max для другого типа

```
int max(int x, int y) {
    return (x > y) ? x : y;
}
```

```
float max(float x, float y) {
    return (x > y) ? x : y;
}
```

```
Matrix max(Matrix x, Matrix y) {
    return (x > y) ? x : y;
}
```

Функция max для другого типа

```
int max(int x, int y) {
    return (x > y) ? x : y;
}
```

```
float max(float x, float y) {
    return (x > y) ? x : y;
}
```

```
Matrix max(Matrix x, Matrix y) {
    return (x > y) ? x : y;
}
```

Что для этого должно быть
в классе Matrix?

Функция max для другого типа

```
int max(int x, int y) {
    return (x > y) ? x : y;
}
```

```
float max(float x, float y) {
    return (x > y) ? x : y;
}
```

```
Matrix max(Matrix x, Matrix y) {
    return (x > y) ? x : y;
}
```

Что для этого должно быть в классе Matrix?

1. Оператор >
2. (Адекватный) конструктор копирования!

Функция max для другого типа

```
int max(int x, int y) {
    return (x > y) ? x : y;
}
```

```
float max(float x, float y) {
    return (x > y) ? x : y;
}
```

```
Matrix max(Matrix x, Matrix y) {
    return (x > y) ? x : y;
}
```

В функции max опять отличие только в типах.

Что делать с дупликацией кода?

Функция max для произвольного типа T

Хочется как-то так:

```
T max(T x, T y) {
   return (x > y) ? x : y;
}
```

Функция max для произвольного* типа T

Хочется как-то так:

```
T max(T x, T y) {
   return (x > y) ? x : y;
}
```

Такой код накладывает ограничения на T:

1. Должен быть переопределен оператор >,
2. Корректный конструктор копирования

Функция сортировки массива целых чисел

```
void sort(int* arr, int len) {
    for (int i = 0; i < len-1; i++) {
        for (int j = 0; j < len - i - 1; j++) {
            if (arr[j] > arr[j+1]) {
                int tmp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = tmp;
            }
        }
    }
}
```

Функция сортировки массива целых чисел

```
void sort(int* arr, int len) {
    for (int i = 0; i < len-1; i++) {
        for (int j = 0; j < len - i - 1; j++) {
            if (arr[j] > arr[j+1]) {
                int tmp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = tmp;
            }
        }
    }
}
```

Что здесь от типа int?

Функция сортировки массива

```
void sort(T* arr, int len) {
    for (int i = 0; i < len-1; i++) {
        for (int j = 0; j < len - i - 1; j++) {
            if (arr[j] > arr[j+1]) {
                T tmp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = tmp;
            }
        }
    }
}
```

Что здесь от типа int?

Что нужно от T, чтобы обобщить алгоритм?

Функция сортировки массива

```
void sort(T* arr, int len) {
    for (int i = 0; i < len-1; i++) {
        for (int j = 0; j < len - i - 1; j++) {
            if (arr[j] > arr[j+1]) {
                T tmp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = tmp;
            }
        }
    }
}
```

Что здесь от типа int?

Что нужно от T, чтобы обобщить алгоритм?

Оператор >
Оператор =
Конструктор копий

Функция сортировки массива

```
void sort(T* arr, int len) {
    for (int i = 0; i < len-1; i++) {
        for (int j = 0; j < len - i - 1; j++) {
            if (arr[j] > arr[j+1]) {
                T tmp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = tmp;
            }
        }
    }
}
```

Что здесь от типа int?

Что нужно от T, чтобы обобщить алгоритм?

Оператор >
Оператор =
Конструктор копий

Функция, суммирующая массив целых чисел

```
int sum(int* arr, int len) {
    int sum = 0;
    for (int i = 0; i < len-1; i++) {
        sum += arr[i];
    }
    return sum;
}
```

Функция, суммирующая массив целых чисел

```
int sum(int* arr, int len) {
    int sum = 0;
    for (int i = 0; i < len-1; i++) {
        sum += arr[i];
    }
    return sum;
}
```

Что здесь от типа int?

Функция, суммирующая массив целых чисел

```
int sum(int* arr, int len) {
    int sum = 0;
    for (int i = 0; i < len-1; i++) {
        sum += arr[i];
    }
    return sum;
}
```

Что здесь от типа int?

Что нужно от T, чтобы обобщить алгоритм?

Функция, суммирующая массив целых чисел

```
int sum(int* arr, int len) {
    int sum = 0;
    for (int i = 0; i < len-1; i++) {
        sum += arr[i];
    }
    return sum;
}
```

Что здесь от типа int?

Что нужно от T, чтобы обобщить алгоритм?

оператор +=

конструктор копирования

Функция, суммирующая массив целых чисел

```
int sum(int* arr, int len) {
    int sum = 0;
    for (int i = 0; i < len-1; i++) {
        sum += arr[i];
    }
    return sum;
}
```

Что здесь от типа int?

Что нужно от T, чтобы обобщить алгоритм?

оператор +=
конструктор копирования
оператор =(int val)

Функция, суммирующая массив целых чисел

```
int sum(int* arr, int len) {
    int sum = 0;
    for (int i = 0; i < len-1; i++) {
        sum += arr[i];
    }
    return sum;
}
```

Что здесь от типа int?

Что нужно от T, чтобы обобщить алгоритм?

оператор +=
конструктор копирования
оператор =(int val)

неудобно получается!

Функция, суммирующая массив целых чисел

```
int sum(int* arr, int len) {
    int sum = arr[0];
    for (int i = 1; i < len-1; i++) {
        sum += arr[i];
    }
    return sum;
}
```

Что здесь от типа int?

Что нужно от T, чтобы обобщить алгоритм?

оператор +=
конструктор копирования
~~оператор =(int val)~~

~~неудобно получается!~~

Функция, суммирующая массив

```
int sum(T* arr, int len) {
    T sum = arr[0];
    for (int i = 1; i < len-1; i++) {
        sum += arr[i];
    }
    return sum;
}
```

Что здесь от типа int?

Что нужно от T, чтобы обобщить алгоритм?

оператор +=

конструктор копирования

~~оператор =(int val)~~

~~неудобно получается!~~

Обобщённое программирование

Обобщённое программирование

Концепция - список методов с описанием сигнатур и семантики

Например, концепция T:

1) `bool operator > (const T& that)`
2) `T(const T& that)`

Обобщённое программирование

Концепция - список методов с описанием сигнатур и семантики

Например, концепция T:

1) `bool operator > (const T& that)`

   возвращает true, если this больше that

2) `T(const T& that)`

   создаёт независимую копию that

Обобщённое программирование

Концепция - список методов с описанием сигнатур и семантики

Концепции похожи на абстрактные классы, но не требуют явного выражения в системе типов

Обобщённое программирование

Концепция - список методов с описанием сигнатур и семантики

Концепции похожи на абстрактные классы, но не требуют явного выражения в системе типов (вместо этого вы просто пишите обобщенный код)

Обобщённое программирование

Концепция - список методов с описанием сигнатур и семантики

Концепции похожи на абстрактные классы, но не требуют явного выражения в системе типов (вместо этого вы просто пишите обобщенный код)

ОП не является частью ООП

Обобщённое программирование

Алгоритм может полагаться не на конкретный тип данных, а на концепцию

Если всё, что он использует от этого типа - это элементы концепции

Например, функция max полагается на концепцию T

Обобщённое программирование

Тип данных может удовлетворять концепции,

Если в нём реализованы все элементы этой концепции в соответствии с сигнатурой и семантикой

Все примитивные типы удовлетворяют концепции T из функции max

Класс будет удовлетворять концепции T, если объявит соответствующие операторы

Обобщённое программирование

Алгоритмы должны быть применимы ко всем типам данных, удовлетворяющим их концепциям

Чем больше подходящих типов, тем лучше ⇒ стоит использовать минималистичные (и простейшие по семантике) концепции

Шаблоны C++

Инструмент реализации ОП

```
template <typename T>
T max(T x, T y) {
    return (x > y) ? x : y;
}
```

Шаблоны C++

Инструмент реализации ОП

```
template <typename T>
T max(T x, T y) {
    return (x > y) ? x : y;
}
```

T - формальный тип (концепция)

Шаблоны C++

Инструмент реализации ОП

```
template <typename T>
T max(T x, T y) {
    return (x > y) ? x : y;
}
```

T - формальный тип (концепция)

```
int a, b, c;
...
a = max<int>(b, c);
```

int - фактический тип

Шаблоны C++

Инструмент реализации ОП

```
template <typename T>
T max(T x, T y) {
    return (x > y) ? x : y;
}
```

T - формальный тип (концепция)

```
float a, b, c;
...
a = max<float>(b, c);
```

float - фактический тип

Шаблоны C++

Инструмент реализации ОП

```
template <typename T>
T max(T x, T y) {
    return (x > y) ? x : y;
}
```

```
Rational a, b, c;
...
a = max<Rational>(b, c);
```

T - формальный тип (концепция)

Такое скомпилируется?

Шаблоны C++

Инструмент реализации ОП

```
template <typename T>
T max(T x, T y) {
    return (x > y) ? x : y;
}
```

```
Rational a, b, c;
...
a = max<Rational>(b, c);
```

T - формальный тип (концепция)

Такое скомпилируется?

Зависит от Rational!

Шаблоны C++

Инструмент реализации ОП

```
template <typename T>
T max(T x, T y) {
    return (x > y) ? x : y;
}
```

```
Rational a, b, c;
...
a = max<Rational>(b, c);
```

```
struct Rational {
    int numerator;
    unsigned int denominator;
};
```

Такое скомпилируется?

Шаблоны C++

Инструмент реализации ОП

```
template <typename T>
T max(T x, T y) {
    return (x > y) ? x : y;
}
```

```
Rational a, b, c;
...
a = max<Rational>(b, c);
```

```
struct Rational {
    int numerator;
    unsigned int denominator;
};
```

Такое скомпилируется?

Нет, такой тип не удовлетворяет концепции, т.к. оператор > не переопределен

Шаблоны C++

Инструмент реализации ОП

```
template <typename T>
T max(T x, T y) {
    return (x > y) ? x : y;
}
```

```
Rational a, b, c;
...
a = max<Rational>(b, c);
```

```
struct Rational {
    int numerator;
    unsigned int denominator;
    bool operator>
        (const Rational& t) { ... }
};
```

Скомпилируется, теперь Rational удовлетворяет концепции.

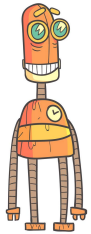
Шаблоны C++

Как это работает?

```
template <typename T>
T max(T x, T y) {
    return (x > y) ? x : y;
}
```

```
int a, b = 3, c= 10;
a = max<int>(b, c);

Rational k, l, m;
k = max<Rational>(l, m);
```

```
template <typename T>
T max(T x, T y) {
    return (x > y) ? x : y;
}
```

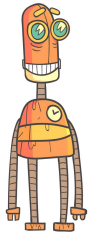


Компилятор находит множество возможных фактических типов:

T = {int, Rational}

```
int a, b = 3, c= 10;
a = max<int>(b, c);

Rational k, l, m;
k = max<Rational>(l, m);
```

```
template <typename T>
T max(T x, T y) {
    return (x > y) ? x : y;
}
```



Компилятор находит множество возможных фактических типов:

T = {int, Rational}

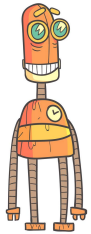
генерирует версии методов max,

```
int max_1(int x, int y) {
    return (x > y) ? x : y;
}
```

```
Rational max_2(Rational x,
               Rational y) {
    return (x > y) ? x : y;
}
```

```
int a, b = 3, c= 10;
a = max<int>(b, c);

Rational k, l, m;
k = max<Rational>(l, m);
```

```
template <typename T>
T max(T x, T y) {
    return (x > y) ? x : y;
}
```



Компилятор находит множество возможных фактических типов:

T = {int, Rational}

генерирует версии методов max,

заменяет вызовы на новые

```
int max_1(int x, int y) {
    return (x > y) ? x : y;
}
```

```
Rational max_2(Rational x,
               Rational y) {
    return (x > y) ? x : y;
}
```

```
int a, b = 3, c= 10;
a = max_1(b, c);

Rational k, l, m;
k = max_2(l, m);
```

Шаблоны C++: реализация

Плюсы:

Шаблоны C++: реализация

Плюсы:

1. БЕСПЛАТНО (во время исполнения, т.е. работать программа будет так же быстро, как без шаблонов)

Шаблоны C++: реализация

Плюсы:

1. БЕСПЛАТНО (во время исполнения, т.е. работать программа будет так же быстро, как без шаблонов)

2. Безопасно по типам (компилятор проверит)

Шаблоны C++: реализация

Плюсы:

1. БЕСПЛАТНО (во время исполнения, т.е. работать программа будет так же быстро, как без шаблонов)
2. Безопасно по типам (компилятор проверит)

Минусы:

1. Платит компилятор: временем компиляции и размером кода

Шаблоны C++: реализация

Плюсы:

1. БЕСПЛАТНО (во время исполнения, т.е. работать программа будет так же быстро, как без шаблонов)
2. Безопасно по типам (компилятор проверит)

Минусы:

1. Платит компилятор: временем компиляции и размером кода
2. Компилятор не проверит семантику

Шаблоны C++: реализация

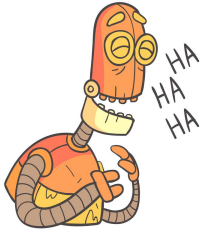
Минусы:

3. Отладка шаблонного кода - это боль

Шаблоны C++: реализация

Минусы:

```
const string&)>&>; T2 = jinja2::EmptyValue]'
/home/travis/build/flexferrum/Jinja2Cpp/thirdparty/nonstd/variant-light/include/nonstd/variant.hpp:1597:17:   required from 'static R nonstd::variants::detail::VisitorApplicatorImpl<R,
VT>::apply(const Visitor&, const T&) [with Visitor = nonstd::variants::detail::TypedVisitorUnwrapper<2ul, jinja2::Value, jinja2::detail::UCInvoker<const
UserCallableTest_SimpleUserCallableWithParams2_Test::TestBody()::<lambda(const string&, const string&)>&>, jinja2::EmptyValue>; T = bool; R = jinja2::Value; VT = bool]'
/home/travis/build/flexferrum/Jinja2Cpp/thirdparty/nonstd/variant-light/include/nonstd/variant.hpp:1807:59:   required from 'static R
nonstd::variants::detail::VisitorApplicator<R>::apply_visitor(const Visitor&, const V1&) [with long unsigned int Idx = 1ul; Visitor = nonstd::variants::detail::TypedVisitorUnwrapper<2ul,
jinja2::Value, jinja2::detail::UCInvoker<const UserCallableTest_SimpleUserCallableWithParams2_Test::TestBody()::<lambda(const string&, const string&)>&>, jinja2::EmptyValue>; V1 =
nonstd::variants::variant<jinja2::EmptyValue, bool, std::basic_string<char>, std::basic_string<wchar_t>, long int, double, nonstd::vptr::value_ptr<std::vector<jinja2::Value>,
nonstd::vptr::detail::default_clone<std::vector<jinja2::Value> >, std::default_delete<std::vector<jinja2::Value> > >, nonstd::vptr::value_ptr<std::unordered_map<std::basic_string<char>,
jinja2::Value>, nonstd::vptr::detail::default_clone<std::unordered_map<std::basic_string<char>, jinja2::Value> >, std::default_delete<std::unordered_map<std::basic_string<char>, jinja2::Value> > >,
jinja2::GenericList, jinja2::GenericMap, nonstd::vptr::value_ptr<jinja2::UserCallable, nonstd::vptr::detail::default_clone<jinja2::UserCallable>, std::default_delete<jinja2::UserCallable> > >; R =
jinja2::Value]'
/home/travis/build/flexferrum/Jinja2Cpp/thirdparty/nonstd/variant-light/include/nonstd/variant.hpp:1778:44:   required from 'static R nonstd::variants::detail::VisitorApplicator<R>::apply(const
Visitor&, const V1&) [with Visitor = nonstd::variants::detail::TypedVisitorUnwrapper<2ul, jinja2::Value, jinja2::detail::UCInvoker<const
UserCallableTest_SimpleUserCallableWithParams2_Test::TestBody()::<lambda(const string&, const string&)>&>, jinja2::EmptyValue>; V1 = nonstd::variants::variant<jinja2::EmptyValue, bool,
std::basic_string<char>, std::basic_string<wchar_t>, long int, double, nonstd::vptr::value_ptr<std::vector<jinja2::Value>, nonstd::vptr::detail::default_clone<std::vector<jinja2::Value> >,
std::default_delete<std::vector<jinja2::Value> > >, nonstd::vptr::value_ptr<std::unordered_map<std::basic_string<char>, jinja2::Value>,
nonstd::vptr::detail::default_clone<std::unordered_map<std::basic_string<char>, jinja2::Value> >, std::default_delete<std::unordered_map<std::basic_string<char>, jinja2::Value> > >,
jinja2::GenericList, jinja2::GenericMap, nonstd::vptr::value_ptr<jinja2::UserCallable, nonstd::vptr::detail::default_clone<jinja2::UserCallable>, std::default_delete<jinja2::UserCallable> > >; R =
jinja2::Value]'
/home/travis/build/flexferrum/Jinja2Cpp/thirdparty/nonstd/variant-light/include/nonstd/variant.hpp:1735:43:   [ skipping 5 instantiation contexts, use -ftemplate-backtrace-limit=0 to disable ]
/home/travis/build/flexferrum/Jinja2Cpp/thirdparty/nonstd/variant-light/include/nonstd/variant.hpp:1871:45:   required from 'typename nonstd::variants::detail::VisitorImpl<sizeof... (V), Visitor, V
...>::result_type nonstd::variants::visit(const Visitor&, const V& ...) [with Visitor = jinja2::detail::UCInvoker<const UserCallableTest_SimpleUserCallableWithParams2_Test::TestBody()::<lambda(const
string&, const string&)>&>; V = {nonstd::variants::variant<jinja2::EmptyValue, bool, std::basic_string<char, std::char_traits<char>, std::allocator<char> >, std::basic_string<wchar_t,
std::char_traits<wchar_t>, std::allocator<wchar_t> >, long int, double, nonstd::vptr::value_ptr<std::vector<jinja2::Value, std::allocator<jinja2::Value> >,
nonstd::vptr::detail::default_clone<std::vector<jinja2::Value, std::allocator<jinja2::Value> > >, std::default_delete<std::vector<jinja2::Value, std::allocator<jinja2::Value> > > >,
nonstd::vptr::value_ptr<std::unordered_map<std::basic_string<char, std::char_traits<char>, std::allocator<char> >, jinja2::Value, std::hash<std::basic_string<char, std::char_traits<char>,
std::allocator<char> > >, std::equal_to<std::basic_string<char, std::char_traits<char>, std::allocator<char> > >, std::allocator<std::pair<const std::basic_string<char, std::char_traits<char>,
std::allocator<char> >, jinja2::Value> > >, nonstd::vptr::detail::default_clone<std::unordered_map<std::basic_string<char, std::char_traits<char>, std::allocator<char> >, jinja2::Value,
std::hash<std::basic_string<char, std::char_traits<char>, std::allocator<char> > >, std::equal_to<std::basic_string<char, std::char_traits<char>, std::allocator<char> > >,
std::allocator<std::pair<const std::basic_string<char, std::char_traits<char>, std::allocator<char> >, jinja2::Value> > > >, std::default_delete<std::unordered_map<std::basic_string<char,
std::char_traits<char>, std::allocator<char> >, jinja2::Value, std::hash<std::basic_string<char, std::char_traits<char>, std::allocator<char> > >, std::equal_to<std::basic_string<char,
std::char_traits<char>, std::allocator<char> > >, std::allocator<std::pair<const std::basic_string<char, std::char_traits<char>, std::allocator<char> >, jinja2::Value> > > > >, jinja2::GenericList,
jinja2::GenericMap, nonstd::vptr::value_ptr<jinja2::UserCallable, nonstd::vptr::detail::default_clone<jinja2::UserCallable>, std::default_delete<jinja2::UserCallable> >,
nonstd::variants::detail::TX<nonstd::variants::detail::S11>, nonstd::variants::detail::TX<nonstd::variants::detail::S12>, nonstd::variants::detail::TX<nonstd::variants::detail::S13>,
nonstd::variants::detail::TX<nonstd::variants::detail::S14>, nonstd::variants::detail::TX<nonstd::variants::detail::S15> >, nonstd::variants::variant<jinja2::EmptyValue, bool,
std::basic_string<char, std::char_traits<char>, std::allocator<char> >, std::basic_string<wchar_t, std::char_traits<wchar_t>, std::allocator<wchar_t> >, long int, double,
nonstd::vptr::value_ptr<std::vector<jinja2::Value, std::allocator<jinja2::Value> >, nonstd::vptr::detail::default_clone<std::vector<jinja2::Value, std::allocator<jinja2::Value> > >,
std::default_delete<std::vector<jinja2::Value, std::allocator<jinja2::Value> > > >, nonstd::vptr::value_ptr<std::unordered_map<std::basic_string<char, std::char_traits<char>, std::allocator<char> >,
jinja2::Value, std::hash<std::basic_string<char, std::char_traits<char>, std::allocator<char> > >, std::equal_to<std::basic_string<char, std::char_traits<char>, std::allocator<char> > >,
std::allocator<std::pair<const std::basic_string<char, std::char_traits<char>, std::allocator<char> >, jinja2::Value> > >, nonstd::vptr::detail::default_clone<std::unordered_map<std::basic_string<char,
std::char_traits<char>, std::allocator<char> >, jinja2::Value, std::hash<std::basic_string<char, std::char_traits<char>, std::allocator<char> > >, std::equal_to<std::basic_string<char,
std::char_traits<char>, std::allocator<char> > >, std::allocator<std::pair<const std::basic_string<char, std::char_traits<char>, std::allocator<char> >, jinja2::Value> > > >, std::default_delete<std::unordered_map<std::basic_string<char,
std::char_traits<char>, std::allocator<char> >, jinja2::Value, std::hash<std::basic_string<char, std::char_traits<char>, std::allocator<char> > >, std::equal_to<std::basic_string<char,
std::char_traits<char>, std::allocator<char> > >, std::allocator<std::pair<const std::basic_string<char, std::char_traits<char>, std::allocator<char> >, jinja2::Value> > > > >, jinja2::GenericList, jinja2::GenericMap,
nonstd::vptr::value_ptr<jinja2::UserCallable, nonstd::vptr::detail::default_clone<jinja2::UserCallable>, std::default_delete<jinja2::UserCallable> >,
nonstd::variants::detail::TX<nonstd::variants::detail::S11>, nonstd::variants::detail::TX<nonstd::variants::detail::S12>, nonstd::variants::detail::TX<nonstd::variants::detail::S13>,
```



Выведение фактического типа

```
template <typename T>
T max(T x, T y) { ...}

int a, b;
float f, g;

max<int>(a, b);
max<float>(f, g);
```

Выведение фактического типа

```
template <typename T>
T max(T x, T y) { ...}

int a, b;
float f, g;

max<int>(a, b);
max<float>(f, g);
```

Компилятор может и сам догадаться о фактическом типе, раз вы передаёте два значения одинакового типа

Выведение фактического типа

```
template <typename T>
T max(T x, T y) { ...}

int a, b;
float f, g;

max(a, b);
max(f, g);
```

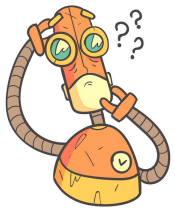
Компилятор может и сам догадаться о фактическом типе, раз вы передаёте два значения одинакового типа

Выведение фактического типа

```
template <typename T>
T max(T x, T y) { ...}

int a, b;
float f, g;

max(a, b);
max(f, g);

max(a, f);
```

Компилятор может и сам догадаться о фактическом типе, раз вы передаёте два значения одинакового типа

А вот здесь компилятор вас уже не понимает!



Выведение фактического типа

```
template <typename T>
T max(T x, T y) { ...}

int a, b;
float f, g;

max(a, b);
max(f, g);

max<float>(a, f);
// max<float>((float) a, f);
```

Компилятор может и сам догадаться о фактическом типе, раз вы передаёте два значения одинакового типа

А вот здесь компилятор вас уже не понимает!

Выведение фактического типа

```
template <typename T>
T max(T x, T y) { ...}

int a, b;
float f, g;

max(a, b);
max(f, g);

max<int>(a, f);
// max<float>(a, (int) f);
```

Компилятор может и сам догадаться о фактическом типе, раз вы передаёте два значения одинакового типа

А вот здесь компилятор вас уже не понимает!

Специализация шаблона

```
template <typename T>
T max(T x, T y) {
    return (x > y) ? x : y;
}

template <>
int max(int x, int y) {
    cout << "Specialized!\n";
    return (x > y) ? x : y;
}
```

Специализация шаблона

```
template <typename T>
T max(T x, T y) {
    return (x > y) ? x : y;
}

template <>
int max(int x, int y) {
    cout << "Specialized!\n";
    return (x > y) ? x : y;
}
```

```
cout << max(3.14, 5.27);

    5.27


cout << max(5, 9);

    Specialized!
    9
```

Шаблоны от значений

Шаблоны от значений

```
template <int N>
void foo() {
    for (int i = 0; i < N; i++) {
        cout << "Hi!\n";
    }
}

void main() {
    foo<5>();
}
```

Шаблоны от значений

```
template <int N>
void foo() {
    for (int i = 0; i < N; i++) {
        cout << "Hi!\n";
    }
}

void main() {
    foo<5>();
}
```

```
Hi!
Hi!
Hi!
Hi!
Hi!
```

Шаблоны от значений (специализация)

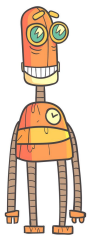
```
template <int N>
void foo() {
    for (int i = 0; i < N; i++) {
        cout << "Hi!\n";
    }
}

template <>
void foo<0> {
    cout << "No greetings for zeroes";
}
```

Шаблоны от значений (специализация)

```
template <int N>
void foo() {
    for (int i = 0; i < N; i++) {
        cout << "Hi!\n";
    }
}

template <>
void foo<0> {
    cout << "No greetings for zeroes";
}
```

Умеем генерировать функции для N и для конкретного числа во время компиляции...

Что бы нам такого с этим сделать?

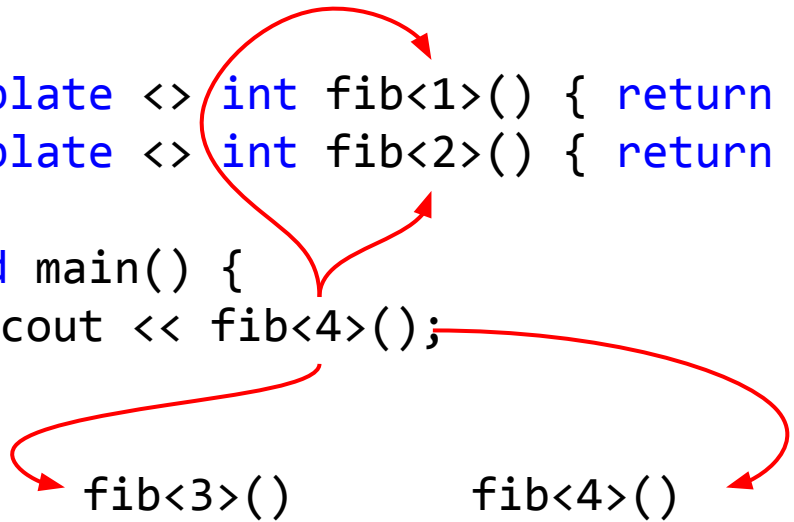


Вычисления времени компиляции

```
template <int N>
int fib() {
    return fib<N-1>() + fib<N-2>();
}

template <> int fib<1>() { return 1; }
template <> int fib<2>() { return 1; }

void main() {
    cout << fib<4>();
}
```

Вычисления времени компиляции

```
template <int N>
int fib() {
    return fib<N-1>() + fib<N-2>();
}

template <> int fib<1>() { return 1; }
template <> int fib<2>() { return 1; }

void main() {
    cout << fib<4>();
}
```



Вычисления времени компиляции

```
template <int N>
int fib() {
    return fib<N-1>() + fib<N-2>();
}

template <> int fib<1>() { return 1; }
template <> int fib<2>() { return 1; }

void main() {
    cout << fib<4>();
}
```

Ответ - 3

Размер .exe - 58Кб

Вычисления времени компиляции

```
template <int N>
int fib() {
    return fib<N-1>() + fib<N-2>();
}

template <> int fib<1>() { return 1; }
template <> int fib<2>() { return 1; }

void main() {
    cout << fib<17>();
}
```

Ответ - 1597

Размер .exe - 61Кб

Вычисления времени компиляции

```
template <int N>
int fib() {
    return fib<N-1>() + fib<N-2>();
}

template <> int fib<1>() { return 1; }
template <> int fib<2>() { return 1; }

void main() {
    cout << fib<1000>();
}
```

Ответ - ???

Размер .exe - **204**Кб

Вычисления времени компиляции

```
template <int N>
int fib() {
    return fib<N-1>() + fib<N-2>();
}

template <> int fib<1>() { return 1; }
template <> int fib<2>() { return 1; }

void main() {
    cout << fib<1000>();
}
```

Ответ - ???

Размер .exe - **204**Кб

Зачем это нужно?

1) Параметр передаётся по-другому, но процесс вычислений не меняется

2) Исполняемый файл пухнет от версий функции

Заставим компилятор сделать inline

1) Добавим спецификатор inline к методам (это не обязывает, но рекомендует компилятору сделать inline)

2) Включим агрессивно оптимизирующий режим компилятора (в MSVC - Release конфигурацию)

Вычисления времени компиляции

```
template <int N>
inline int fib() {
    return fib<N-1>() + fib<N-2>();
}

template <> inline int fib<1>() { return 1; }
template <> inline int fib<2>() { return 1; }

void main() {
    cout << fib<4>();
}
```

Вычисления времени компиляции

```
inline int fib_1() { return 1; }

inline int fib_2() { return 1; }

inline int fib_3() { return fib_2() + fib_1(); }

inline int fib_4() { return fib_3() + fib_2(); }

void main() {
    cout << fib_4();
}
```

Вычисления времени компиляции

```
inline int fib_1() { return 1; }

inline int fib_2() { return 1; }

inline int fib_3() { return fib_2() + fib_1(); }

inline int fib_4() { return fib_3() + fib_2(); }

void main() {
    cout << fib_4();
}
```

Вычисления времени компиляции

```
inline int fib_2() { return 1; }

inline int fib_3() { return fib_2() + 1; }

inline int fib_4() { return fib_3() + fib_2(); }

void main() {
    cout << fib_4();
}
```

Вычисления времени компиляции

```
inline int fib_3() { return 1 + 1; }

inline int fib_4() { return fib_3() + 1; }

void main() {
    cout << fib_4();
}
```

INLINE!

Вычисления времени компиляции

```
inline int fib_4() { return 1 + 1 + 1; }

void main() {
    cout << fib_4();
}
```

INLINE!

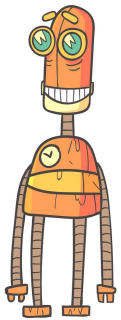
Вычисления времени компиляции

```
void main() {
    cout << 1 + 1 + 1;
}
```

INLINE!

Вычисления времени компиляции

```
void main() {
    cout << 3;
}
```



Исполнятся будет уже вот такой код!

Остальное вычислено во время компиляции.

Вычисления времени компиляции

```
template <int N>
inline int fib() {
    return fib<N-1>() + fib<N-2>();
}

template <> inline int fib<1>() { return 1; }
template <> inline int fib<2>() { return 1; }

void main() {
    cout << fib<4>();
}
```

Ответ - 3

Размер .exe - 10Кб

Вычисления времени компиляции

```
template <int N>
inline int fib() {
    return fib<N-1>() + fib<N-2>();
}

template <> inline int fib<1>() { return 1; }
template <> inline int fib<2>() { return 1; }

void main() {
    cout << fib<17>();
}
```

Ответ - 1579

Размер .exe - **10Кб**

Но время компиляции уже чувствуется

Вычисления времени компиляции

```
template <int N>
inline int fib() {
    return fib<N-1>() + fib<N-2>();
}

template <> inline int fib<1>() { return 1; }
template <> inline int fib<2>() { return 1; }

void main() {
    cout << fib<20>();
}
```

Ответ - 6765

Размер .exe - **10Кб**

Но время компиляции очень чувствуется!

Вычисления времени компиляции

```
template <int N>
inline int fib() {
    return fib<N-1>() + fib<N-2>();
}

template <> inline int fib<1>() { return 1; }
template <> inline int fib<2>() { return 1; }

void main() {
    cout << fib<1000>();
}
```

Ответ - ???

Размер .exe - ???

Не дождался результата компиляции

Вычисления времени компиляции

С помощью шаблонов от значений можно заставить компилятор в процессе своей работы вычислять значения функций.

Вычисления времени компиляции

С помощью шаблонов от значений можно заставить компилятор в процессе своей работы вычислять значения функций.

Реализация шаблонов в любом компиляторе языка C++ образует **Тьюринг-полный** язык программирования

> То есть способна выполнить произвольную вычислимую функцию

Вычисления времени компиляции

С помощью шаблонов от значений можно заставить компилятор в процессе своей работы вычислять значения функций.

Реализация шаблонов в любом компиляторе языка C++ образует **Тьюринг-полный** язык программирования

То есть способна выполнить произвольную вычислимую функцию

Шаблоны классов

Шаблоны классов

Коллекции - списки, расширяемые массивы, стеки, множества, хэш-таблицы и прочие - практически никак не зависят от хранимого в них типа данных

Шаблоны классов

```
class List {
    Elem* head;
public:
    List(): head(NULL) {}
    ~List();

    void addHead(int x) {
        Elem* e = new Elem();
        e->data = x;
        e->next = head;
        head = e;
    }
};
```

```
struct Elem {
    int data;
    Elem* next;
};
```

Шаблоны классов

```
class List {
    Elem* head;
public:
    List(): head(NULL) {}
    ~List();

    void addHead(int x) {
        Elem* e = new Elem();
        e->data = x;
        e->next = head;
        head = e;
    }
};
```

```
struct Elem {
    int data;
    Elem* next;
};
```

Что нужно от типа, если хотим обобщить int?

Шаблоны классов

```
class List {
    Elem* head;
public:
    List(): head(NULL) {}
    ~List();

    void addHead(int x) {
        Elem* e = new Elem();
        e->data = x;
        e->next = head;
        head = e;
    }
};
```

```
struct Elem {
    int data;
    Elem* next;
};
```

Что нужно от типа, если хотим обобщить int?

1. Конструктор копирования

Шаблоны классов

```
class List {
    Elem* head;
public:
    List(): head(NULL) {}
    ~List();

    void addHead(int x) {
        Elem* e = new Elem();
        e->data = x;
        e->next = head;
        head = e;
    }
};
```

```
struct Elem {
    int data;
    Elem* next;
};
```

Что нужно от типа, если хотим обобщить int?

1. Конструктор копирования
2. Оператор присваивания

Шаблоны классов

```
class List {
    Elem* head;
public:
    List(): head(NULL) {}
    ~List();

    void addHead(int x) {
        Elem* e = new Elem();
        e->data = x;
        e->next = head;
        head = e;
    }
};
```

```
struct Elem {
    int data;
    Elem* next;
};
```

Что нужно от типа, если хотим обобщить int?

1. Конструктор копирования
2. Оператор присваивания
3. Конструктор без параметров

Шаблоны классов

```
class List {
    Elem* head;
public:
    List(): head(NULL) {}
    ~List();

    void addHead(int x) {
        Elem* e = new Elem();
        e->data = x;
        e->next = head;
        head = e;
    }
};
```

Многие типы подходят!

```
struct Elem {
    int data;
    Elem* next;
};
```

Что нужно от типа, если хотим обобщить `int`?

1. Конструктор копирования
2. Оператор присваивания
3. Конструктор без параметров

Шаблоны классов

```
class List {
    Elem<int>* head;
public:
    List(): head(NULL) {}
    ~List();

    void addHead(int x) {
        Elem<int>* e = new Elem<int>();
        e->data = x;
        e->next = head;
        head = e;
    }
};
```

```
template <typename T>
struct Elem {
    T data;
    Elem<T>* next;
};
```

Шаблоны классов

```
template <typename T>
class List {
    Elem<T>* head;
public:
    List(): head(NULL) {}
    ~List();

    void addHead(T x) {
        Elem<T>* e = new Elem<T>();
        e->data = x;
        e->next = head;
        head = e;
    }
};
```

```
template <typename T>
struct Elem {
    T data;
    Elem<T>* next;
};
```

Шаблоны классов

```
template <typename T>
class List {
    Elem<T>* head;
public:
    List(): head(NULL) {}
    ~List();

    void addHead(T x) {
        Elem<T>* e = new Elem<T>();
        e->data = x;
        e->next = head;
        head = e;
    }
};
```

```
template <typename T>
struct Elem {
    T data;
    Elem<T>* next;
};
```

Свой формальный тип можно указывать, как фактический для других шаблонов

Шаблоны классов

```
template <typename T>
class List {
    Elem<T>* head;
public:
    List(): head(NULL) {}
    ~List();

    void addHead(T x) {
        Elem<T>* e = new Elem<T>();
        e->data = x;
        e->next = head;
        head = e;
    }
};
```

```
template <typename T>
struct Elem {
    T data;
    Elem<T>* next;
};
```

```
List<int> il;
il.add(42);

List<float> fl;
fl.add(4.2);
```

Шаблоны классов

```
template <typename T>
class List {
    Elem<T>* head;
public:
    List(): head(NULL) {}
    ~List();

    void addHead(T x) {
        Elem<T>* e = new Elem<T>();
        e->data = x;
        e->next = head;
        head = e;
    }
};
```

```
template <typename T>
struct Elem {
    T data;
    Elem<T>* next;
};
```

Свой формальный тип можно указывать, как фактический для других шаблонов

Методы шаблонного класса

Являются шаблонными функциями

Генерируются для конкретных фактических типов по необходимости

Если тип не соответствует концепции, это может обнаружиться не сразу

Методы шаблонного класса

```
template <typename T>
class Foo {
    T x;
public:
    Foo(T x): x(x) {}
    bool operator > (const Foo& that) const {
        return this->x > that.x;
    }
};
```

Методы шаблонного класса

```
template <typename T>
class Foo {
    T x;
public:
    Foo(T x): x(x) {}
    bool operator > (const Foo& that) const {
        return this->x > that.x;
    }
};
```

```
class Bar {
};


Bar b1;
Foo<Bar> f1(b1);

Bar b2;
Foo<Bar> f2(b2);
```

Методы шаблонного класса

```
template <typename T>
class Foo {
    T x;
public:
    Foo(T x): x(x) {}
    bool operator > (const Foo& that) const {
        return this->x > that.x;
    }
};
```

```
class Bar {
};


Bar b1;
Foo<Bar> f1(b1);

Bar b2;
Foo<Bar> f2(b2);
```

Компилируется!

Методы шаблонного класса

```
template <typename T>
class Foo {
    T x;
public:
    Foo(T x): x(x) {}
    bool operator > (const Foo& that) const {
        return this->x > that.x;
    }
};
```

```
class Bar {
};


Bar b1;
Foo<Bar> f1(b1);

Bar b2;
Foo<Bar> f2(b2);

bool t = f1 > f2;
```

Методы шаблонного класса

```
template <typename T>
class Foo {
    T x;
public:
    Foo(T x): x(x) {}
    bool operator > (const Foo& that) const {
        return this->x > that.x;
    }
};
```

```
class Bar {
};


Bar b1;
Foo<Bar> f1(b1);

Bar b2;
Foo<Bar> f2(b2);

bool t = f1 > f2;
```

Не компилируется!

Вычисления времени компиляции strike back!

Вычисления времени компиляции strike back!

```
template<int X, int Y>
struct Pow {

};
```

Вычисления времени компиляции strike back!

```
template<int X, int Y>
struct Pow {
    static const int result = X * Pow<X, Y - 1>::result;
};
```

Вычисления времени компиляции strike back!

```
template<int X, int Y>
struct Pow {
    static const int result = X * Pow<X, Y - 1>::result;
};

template<int X>
struct Pow<X, 1> {
    static const int result = X;
};
```

Специализируем!

Вычисления времени компиляции strike back!

```
template<int X, int Y>
struct Pow {
    static const int result = X * Pow<X, Y - 1>::result;
};

template<int X>
struct Pow<X, 1> {
    static const int result = X;
};

const int z = Pow<5, 3>::result;
```

Специализируем!

Вычислит 125

Вычисления времени компиляции strike back!

```
template<int X, int Y>
struct Pow {
    static const int result = X * Pow<X, Y - 1>::result;
};

template<int X>
struct Pow<X, 1> {
    static const int result = X;
};

const int z = Pow<5, 3>::result;
```

Специализируем!

Вычислит 125

Во время компиляции!

Curiously Recurring Template Pattern (CRTP)

Curiously Recurring Template Pattern (CRTP)

Задача: посчитать все созданные и все живые на данный момент экземпляры класса.

Curiously Recurring Template Pattern (CRTP)

Задача: посчитать все созданные и все живые на данный момент экземпляры класса.

Повторить для любого класса.

```
template <typename T>
struct Counter {




};
```

```
template <typename T>
struct Counter {
    static int objects_created;
    static int objects_alive;

};
```

```
template <typename T>
struct Counter {
    static int objects_created;
    static int objects_alive;

    Counter() {
        ++objects_created;
        ++objects_alive;
    }

    Counter(const Counter&) {
        ++objects_created;
        ++objects_alive;
    }




};
```

```
template <typename T>
struct Counter {
    static int objects_created;
    static int objects_alive;

    Counter() {
        ++objects_created;
        ++objects_alive;
    }

    Counter(const Counter&) {
        ++objects_created;
        ++objects_alive;
    }
protected:
    ~Counter() {
        --objects_alive;
    }
};
```

```
template <typename T>
struct Counter {
    static int objects_created;
    static int objects_alive;

    Counter() {
        ++objects_created;
        ++objects_alive;
    }

    Counter(const Counter&) {
        ++objects_created;
        ++objects_alive;
    }
protected:
    ~Counter() {
        --objects_alive;
    }
};
```

```
template <typename T> int Counter<T>::
                          objects_created(0);
template <typename T> int Counter<T>::
                            objects_alive(0);
```

```cpp
template <typename T>
struct Counter {
    static int objects_created;
    static int objects_alive;

    Counter() {
        ++objects_created;
        ++objects_alive;
    }

    Counter(const Counter&) {
        ++objects_created;
        ++objects_alive;
    }
protected:
    ~Counter() {
        --objects_alive;
    }
};
```

```cpp
class X : Counter<X> {
    // ...
};

class Y : Counter<Y> {
    // ...
};
```

```
template <typename T>
struct Counter {
    static int objects_created;
    static int objects_alive;

    Counter() {
        ++objects_created;
        ++objects_alive;
    }

    Counter(const Counter&) {
        ++objects_created;
        ++objects_alive;
    }
protected:
    ~Counter() {
        --objects_alive;
    }
};
```

```
class X : Counter<X> {
    X(const X& other):Counter<X>(other) { }
    // ...
};


class Y : Counter<Y> {
    Y(const X& other):Counter<Y>(other) { }
    // ...
};
```

```cpp
template <typename T>
struct Counter {
    static int objects_created;
    static int objects_alive;

    Counter() {
        ++objects_created;
        ++objects_alive;
    }

    Counter(const Counter&) {
        ++objects_created;
        ++objects_alive;
    }
protected:
    ~Counter() {
        --objects_alive;
    }
};
```

```cpp
class X : Counter<X> {
    // ...
};

class Y : Counter<Y> {
    // ...
};
```

```
template <typename T>
struct Counter {
    static int objects_created;
    static int objects_alive;

    Counter() {
        ++objects_created;
        ++objects_alive;
    }

    Counter(const Counter&) {
        ++objects_created;
        ++objects_alive;
    }
protected:
    ~Counter() {
        --objects_alive;
    }
};
```

```
class X : Counter<X> {
    // ...
};

class Y : Counter<Y> {
    // ...
};
```

Классы наследуют Counter,
параметризованный этими же классами

```
template <typename T>
struct Counter {
    static int objects_created;
    static int objects_alive;

    Counter() {
        ++objects_created;
        ++objects_alive;
    }

    Counter(const Counter&) {
        ++objects_created;
        ++objects_alive;
    }
protected:
    ~Counter() {
        --objects_alive;
    }
};
```

```
class X : Counter<X> {
    // ...
};

class Y : Counter<Y> {
    // ...
};
```

Классы наследуют Counter, параметризованный этими же классами

Для каждого класса сгенерируется свой Counter!

```
class X : Counter<X> {
    // ...
};

class Y : Counter<Y> {
    // ...
};


X a, b, c, d;
{
    Y e, f, j, k;
    cout << "alive X = " << Counter<X>::objects_alive << endl;     // 4
    cout << "alive Y = " << Counter<Y>::objects_alive << endl;     // 4
}
cout << "alive X = " << Counter<X>::objects_alive << endl;         // 4
cout << "alive Y = " << Counter<Y>::objects_alive << endl;         // 0
```

Curiously Recurring Template Pattern (CRTP)

CRTP на самом деле гораздо круче:

Curiously Recurring Template Pattern (CRTP)

CRTP на самом деле гораздо круче:

1. Позволяет явно выражать интерфейсы (с использованием static_cast)
2. "Подмешивать" в классы новую функциональность (MixIn)
3. Подменять базовые классы в иерархии в разных случаях

Шаблоны C++

1) Отличный инструмент для создания обобщённых алгоритмов и структур данных

Шаблоны C++

1) Отличный инструмент для создания обобщённых алгоритмов и структур данных (но и не только!)

Шаблоны C++

1) Отличный инструмент для создания обобщённых алгоритмов и структур данных (но и не только!)

2) Позволяют писать вычислимые в compile-time программы в функциональной парадигме!

Шаблоны C++

1) Отличный инструмент для создания обобщённых алгоритмов и структур данных (но и не только!)

2) Позволяют писать вычислимые в compile-time программы в функциональной парадигме!

3) Отличный инструмент для создания катастрофически запутанных систем

Шаблоны C++

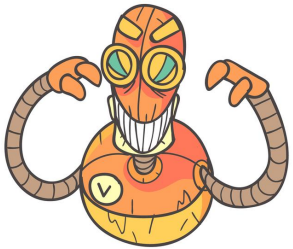
1) Отличный инструмент для создания обобщённых алгоритмов и структур данных (но и не только!)

2) Позволяют писать вычислимые в compile-time программы в функциональной парадигме!

3) Отличный инструмент для создания катастрофически запутанных систем

Убедитесь, что вынесли с этой лекции

1. Понятие концепции, связь с системой типов
2. Идея обобщённого программирования
3. Синтаксис шаблонов C++
4. Выведение типов, специализация
5. Шаблоны от значений, вычисления времени компиляции
6. Шаблоны классов



Проверочные вопросы

1) Опишите концепцию, подходящую для бинарного поиска
2) Как ограничить типы данных, применимые к шаблону?
3) Сколько реализаций шаблонной функции появится в бинарном файле?
4) Сколько реализаций шаблонной функции конкретного фактического типа появится в бинарном файле?

* - заставьте компилятор посчитать функцию Аккермана от (4, 4)

* - реализуйте шаблонный класс Tensor<typename T, int size, int rank>

**- докажите, что шаблоны образуют Тьюринг полный ЯП

Про задачи

В сложных вариантах нужно реализовать хеш-таблицы.

Источники информации:

1. Семинары
2. Кормен:
   http://staff.ustc.edu.cn/~csli/graduate/algorithms/book6/chap12.htm
3. Видео:
   https://www.coursera.org/lecture/algorithms-graphs-data-structures/hash-tables-implementation-details-part-i-Ob0K7

Q & A

Особенности компиляции

1) Описание шаблона (включая реализацию) не является кодом. Его можно рассматривать, как макрос - инструкцию для компилятора

2) В каждой единице компиляции для каждого конкретного фактического типа компилятор создаёт реализацию шаблона - конкретный код

3) Впоследствии линкер объединит эти реализации в бинарном файле

Особенности компиляции

lib.h

```
template <typename T>
T foo() {
        // impl
}
```

Особенности компиляции

lib.h

```
template <typename T>
T foo() {
        // impl
}
```

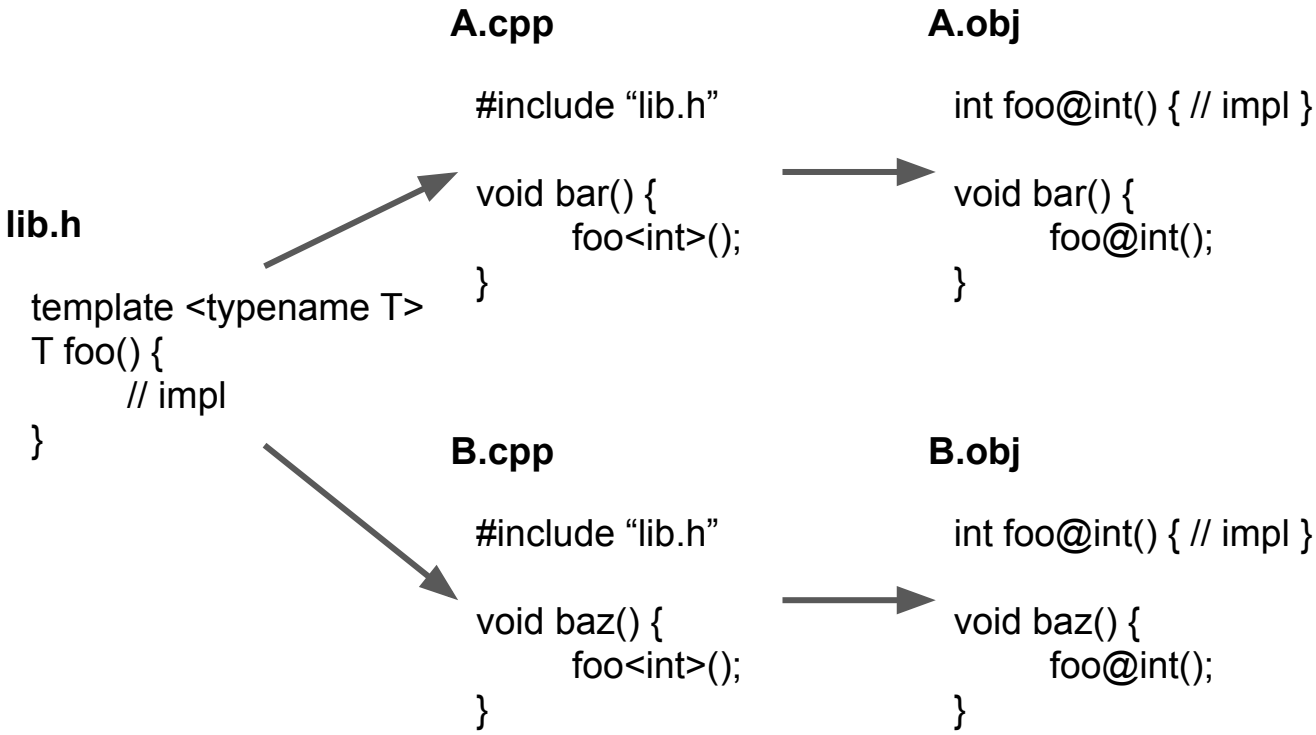
A.cpp

```
#include “lib.h”

void bar() {
        foo<int>();
}
```

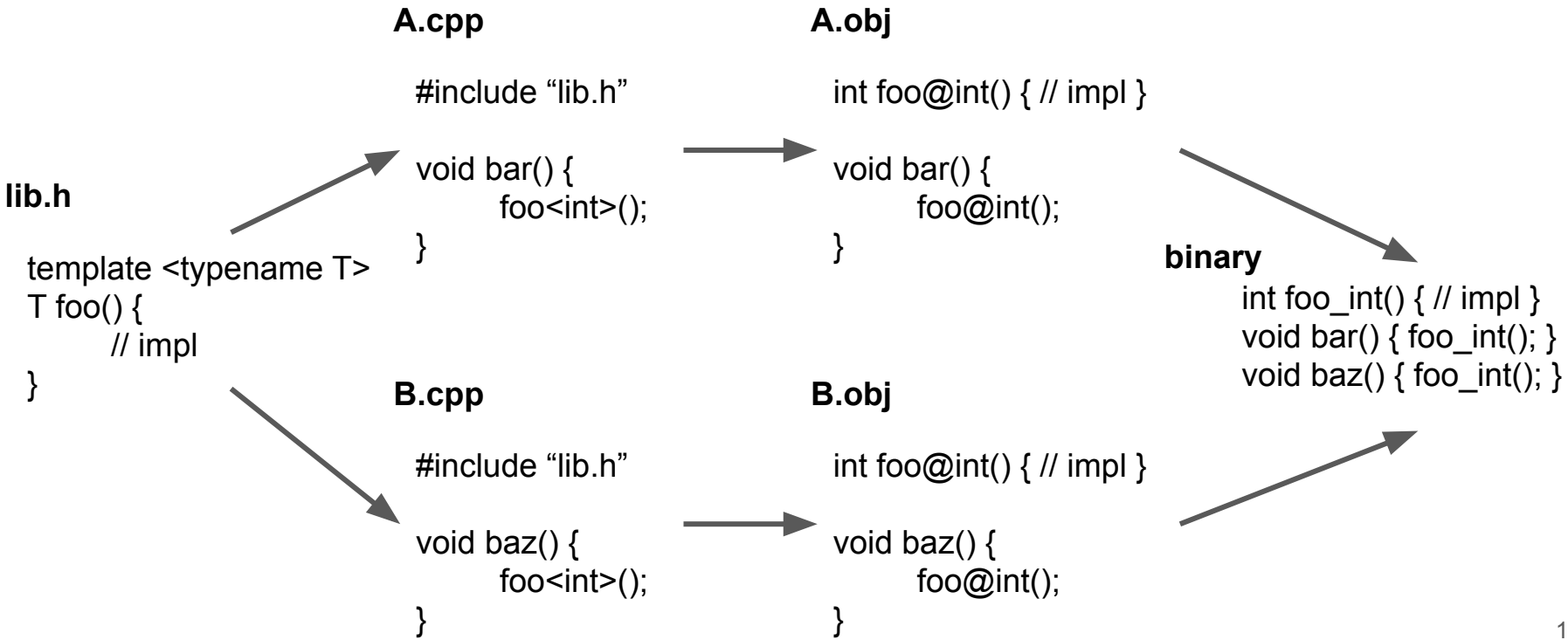
B.cpp

```
#include “lib.h”

void baz() {
        foo<int>();
}
```

Особенности компиляции



Особенности компиляции



Особенности компиляции

1) Разделение шаблона на объявление в заголовочном файле и реализацию в независимой единице компиляции не работает

   Для этого было ключевое слово export, но его убрали из стандарта, начиная с C++11

2) Таким разделением можно ограничить набор типов, поддерживаемых шаблоном

Особенности компиляции

lib.h

```
template <typename T>
T foo();
```

lib.cpp

```
#include <lib.h>

template <typename T>
T foo() {
        // impl
}

void compiler_anchor() {
        foo<int>();
        foo<float>();
}
```

main.cpp

```
#include <lib.h>

void main() {
        foo<int>();
        foo<float>();
}
```

Особенности компиляции

lib.h

```
template <typename T>
T foo();
```

lib.cpp

```
#include <lib.h>

template <typename T>
T foo() {
        // impl
}

void compiler_anchor() {
        foo<int>();
        foo<float>();
}
```

main.cpp

```
#include <lib.h>

void main() {
        foo<int>();
        foo<float>();
        foo<char>();
}
```

Особенности компиляции

main.cpp

```
#include <lib.h>

void main() {
        foo<int>();
        foo<float>();
        foo<char>();
}
```

foo<int> и foo<float> найдутся в lib.obj
foo<char> нигде не создано компилятором

Особенности компиляции

main.cpp

```
#include <lib.h>

template <>
char foo() {
        // impl
}

void main() {
        foo<int>();
        foo<float>();
        foo<char>();
}
```

foo<int> и foo<float> найдутся в lib.obj
foo<char> нигде не создано компилятором

Можно определить специализированную версию!

Особенности компиляции

main.cpp

```
#include <lib.h>

template <typename T>
T foo() {
        // impl
}

void main() {
        foo<int>();
        foo<float>();
        foo<char>();
}
```

foo<int> и foo<float> найдутся в lib.obj
foo<char> нигде не создано компилятором

Можно определить специализированную версию!

Или даже переопределить общую!

Особенности компиляции

main.cpp

```
#include <lib.h>

template <typename T>
T foo() {
        cout << “Hi!”;
        // impl
}

void main() {
        foo<int>();
        foo<float>();
        foo<char>();
}
```

foo<int> и foo<float> найдутся в lib.obj
foo<char> нигде не создано компилятором

Можно определить специализированную версию!

Или даже переопределить общую!

Причём с любой реализацией!

Особенности компиляции

lib.cpp

```
#include <lib.h>
template <typename T>
T foo() {
        cout << “Hi from lib!”
}
void compiler_anchor() {
        foo<int>();
}
```

lib.h

```
template <typename T>
T foo();
```

main.cpp

```
#include <lib.h>
template <typename T>
T foo() {
        cout << “Hi from main!”
}
void main() {
        foo<int>();
        foo<float>();
}
```

Особенности компиляции

lib.cpp

```
#include <lib.h>
template <typename T>
T foo() {
        cout << “Hi from lib!”
}
void compiler_anchor() {
        foo<int>();
}
```

lib.h

```
template <typename T>
T foo();
```

main.cpp

```
#include <lib.h>
template <typename T>
T foo() {
        cout << “Hi from main!”
}
void main() {
        foo<int>();
        foo<float>();
}
```

Результат зависит от порядка компиляции lib.cpp и main.cpp