Классическое метапрограммирование

Частичная специализация шаблонов классов

Иногда хочется задать определенное поведение класса не для конкретного типа, а для целого семейства типов (например, для указателей).

```
template <class T>  // общий шаблон
struct IsPointer {
  inline static const bool value = false;
  static bool IsIntPointer() { return false; }
};

template <class T>  // частичная специализация
struct IsPointer<T*> {
  inline static const bool value = true;
  static bool IsIntPointer() { return false; }
};

template <>  // полная специализация
struct IsPointer<int*> {
  inline static const bool value = true;
  static bool IsIntPointer() { return true; }
};
```

Частичная специализация шаблонов классов

Более специализированная версия всегда побеждает менее специализированную

```
std::cout << IsPointer<int>::value << ' '                 // 0
          << IsPointer<int>::IsIntPointer() << '\n';      // 0

std::cout << IsPointer<char*>::value << ' '               // 1
          << IsPointer<char*>::IsIntPointer() << '\n';    // 0

std::cout << IsPointer<int*>::value << ' '                // 1
          << IsPointer<int*>::IsIntPointer() << '\n';     // 1
```

Частичная специализация шаблонов классов

```
template <class T, class U>
struct S {  // 1
};

template <class T>
struct S<T, int> {  // 2
};

template <class T>
struct S<float, T> {  // 3
};
```

```
S<bool, bool> a;          // 1
S<bool, int> b;           // 2
S<const float, bool> c;   // 1
S<float, int> d;          // CE
```

Частичная специализация шаблонов функций

- Ее не существует

Определители типов

Определители типов

С помощью частичной специализации можно определять простейшие свойства типов в шаблонном контексте:

```
template <class T>
struct IsConst {
  static constexpr bool value = false;  // в общем случае T не константа
};

template <class T>
struct IsConst<const T> {  // специализация для констант
  static constexpr bool value = true;
};
```

Определители типов

"Упростим" предыдущий пример

```
template <class T>
struct IsConst : std::integral_constant<bool, false> {};

template <class T>
struct IsConst<const T> : std::integral_constant<bool, true> {};
```

И еще

```
template <bool B>
using bool_constant = integral_constant<bool, B>;
using true_type = bool_constant<true>;
using false_type = bool_constant<false>;

template <class T>
struct IsConst : std::false_type {};

template <class T>
struct IsConst<const T> : std::true_type {};
```

Определители типов

```
template <class T>
struct IsPointer : std::false_type {};

template <class T>
struct IsPointer<T*> : std::true_type {};

template <class T>
struct IsPointer<T* const> : std::true_type {};

template <class T>
struct IsPointer<T* volatile> : std::true_type {};

template <class T>
struct IsPointer<T* const volatile> : std::true_type {};
```

Определители типов

```
template <class T>
struct IsLvalueReference : std::false_type {};

template <class T>
struct IsLvalueReference<T&> : std::true_type {};

template <class T>
struct IsRvalueReference : std::false_type {};

template <class T>
struct IsRvalueReference<T&&> : std::true_type {};

template <class T>
struct IsReference
  : std::bool_constant<IsLvalueReference<T>::value || IsRvalueReference<T>::value> {};
```

Определители типов

```
template <class T, class U>
struct IsSame : std::false_type {};

template <class T>
struct IsSame<T, T> : std::true_type {};
```

```
template <class T, class... Other>
struct AreSame : std::bool_constant<(IsSame<T, Other>::value && ...)> {};
```

Примеры применения

Замечание: описанные выше определители типов (std::is_const, std::is_reference, ...), а также соответствующие им шаблонные переменные (std::is_const_v, std::is_reference_v, ...) есть в заголовочном файле <type_traits>

```
template <class... Ints>
auto Sum(Ints... values) {  // сумма только int'ов
  static_assert(std::is_same_v<int, Ints> && ...);
  return (values + ...);
}
```

```
template <class Storage>
void ProcessAndDestroy(Storage storage) {
  // ...
  if (std::is_pointer_v<Storage>) {
    delete[] &storage[0];  // почему так сложно?
  }
}
```

Примеры применения: `if constexpr`

Рассмотрим следующий код

```
template <class Storage>
void ProcessAndDestroy(Storage storage) {
  // ...
  if (std::is_pointer_v<Storage>) {
    delete[] storage;
  }
}

ProcessAndDestroy(std::vector<int>{1, 2, 3, 4, 5});
ProcessAndDestroy(new int[5]{1, 2, 3, 4, 5});
```

В чем проблема?

Примеры применения: `if constexpr`

```
template <class Storage>
void ProcessAndDestroy(Storage storage) {
  // ...
  if (std::is_pointer_v<Storage>) {
    delete[] storage;
  }
}

ProcessAndDestroy(std::vector<int>{1, 2, 3, 4, 5});
ProcessAndDestroy(new int[5]{1, 2, 3, 4, 5});
```

Блок внутри `if` компилируется в любом случае!

```
main.cpp: error: type 'std::vector<int>' argument given to 'delete', expected pointer
   10 |     delete[] storage;
```

Примеры применения: `if constexpr`

`if constexpr` (C++17) - конструкция, позволяющая "отключать" ветки инстанцирования. Принимает булевское **константное выражение**.

```
template <class Storage>
void ProcessAndDestroy(Storage storage) {
  // ...
  if constexpr (std::is_pointer_v<Storage>) {
    delete[] storage;  // работает только для указателей
  }
}

ProcessAndDestroy(std::vector<int>{1, 2, 3, 4, 5});  // Ok
ProcessAndDestroy(new int[5]{1, 2, 3, 4, 5});
```

Примеры применения: `if constexpr`

```
template <class Iterator>
auto Get(Iterator iterator, size_t n) {
  using Tag = typename std::iterator_traits<Iterator>::iterator_category;
  if constexpr (std::is_base_of_v<std::random_access_iterator_tag, Tag>) {
    return iterator[n];
  } else {
    while (n--) {
      ++iterator;
    }
    return *iterator;
  }
}
```

Модификаторы типов

Модификаторы типов

```
template <class T>
struct RemoveConst {
  using type = T;
};

template <class T>
struct RemoveConst<const T> {
  using type = T;
};
```

```
template <class T>
auto CreateCopy(T* array, size_t n) {
  auto copy = new RemoveConst<T>::type[n];
  std::copy(array, array + n, copy);
  return copy;
}
```

Модификаторы типов

Упростим код

```
template <class T>
struct TypeIdentity {  // std::type_identity
  using type = T;
};

template <class T>
struct RemoveConst : TypeIdentity<T> {};
template <class T>
struct RemoveConst<const T> : TypeIdentity<T> {};

template <class T>
using RemoveConstT = typename RemoveConst<T>::type;  // шаблонный псевдоним
```

```
template <class T>
auto CreateCopy(T* array, size_t n) {
  auto copy = new RemoveConstT<T>[n];
  std::copy(array, array + n, copy);
  return copy;
}
```

Модификаторы типов

```
template <class T>
struct RemoveReference : std::type_identity<T> {};

template <class T>
struct RemoveReference<T&> : std::type_identity<T> {};

template <class T>
struct RemoveReference<T&&> : std::type_identity<T> {};

template <class T>
using RemoveReferenceT = typename RemoveReference<T>::type;
```

Модификаторы типов: `std::move` / `std::forward`

Наконец-то можем разобрать реализацию функций `std::move` и `std::forward`

```
template <class T>
??? move(??? value) {
  return ???;
}
```

- Что делает `std::move`?
- Что принимает `std::move`?
- Что возвращает `std::move`?

Модификаторы типов: `std::move` / `std::forward`

Наконец-то можем разобрать реализацию функций `std::move` и `std::forward`

```
template <class T>
??? move(??? value) {
  return ???;
}
```

- Что делает `std::move`? - преобразует к rvalue ссылке (xvalue)
- Что принимает `std::move`? - что угодно (lvalue/rvalue)
- Что возвращает `std::move`? - rvalue ссылку (xvalue)

Модификаторы типов: `std::move` / `std::forward`

Наконец-то можем разобрать реализацию функций `std::move` и `std::forward`

```
template <class T>
std::remove_reference_t<T>&& move(T&& value) {
  return static_cast<std::remove_reference_t<T>&&>(value);
}
```

- Что делает `std::move`? - преобразует к rvalue ссылке (xvalue)
- Что принимает `std::move`? - что угодно (lvalue/rvalue)
- Что возвращает `std::move`? - rvalue ссылку (xvalue)

Модификаторы типов: `std::move` / `std::forward`

Наконец-то можем разобрать реализацию функций `std::move` и `std::forward`

```
template <class T>
??? forward(??? value) {
  return ???;
}
```

- Что делает `std::forward`?
- Что принимает `std::forward`?
- Что возвращает `std::forward`?

Модификаторы типов: `std::move` / `std::forward`

Наконец-то можем разобрать реализацию функций `std::move` и `std::forward`

```
template <class T>
??? forward(??? value) {
  return ???;
}
```

- Что делает `std::forward`? - условный `std::move`
- Что принимает `std::forward`? - lvalue ссылку
- Что возвращает `std::forward`? - lvalue/rvalue ссылку в зависимости от `T`

Модификаторы типов: `std::move` / `std::forward`

Наконец-то можем разобрать реализацию функций `std::move` и `std::forward`

```
template <class T>
T&& forward(std::remove_reference_t<T>& value) {
  return static_cast<T&&>(value);
}
```

- Что делает `std::forward`? - условный `std::move`
- Что принимает `std::forward`? - lvalue ссылку
- Что возвращает `std::forward`? - lvalue/rvalue ссылку в зависимости от `T`

`decltype/declval`

`decltype`

Спецификатор `decltype` позволяет узнать тип сущности по идентификатору, а также тип и категорию значения произвольного выражения.

```
int x = 0;
const float y = 1;
const int& rx = x;

decltype(x) a = 0;   // int
decltype(y) b = 1;   // const float
decltype(rx) c = a;  // const int&
```

`decltype`

Спецификатор `decltype` позволяет узнать тип сущности по идентификатору, а также тип и категорию значения произвольного выражения.

```
template <class T>
auto ExtractNested(const std::vector<T>& v) {
  std::vector<decltype(v[0].x)> res;
  res.reserve(v.size());
  for (const auto& value : v) {
    res.push_back(value.x);
  }
  return res;
}
```

`decltype`

Если подать `decltype` *выражение* (не являющееся именем переменной или поля), то он вернет:

- просто тип, если категория выражения - *prvalue*
- тип с `&`, если категория выражения - *lvalue*
- тип с `&&`, если категория выражения - *xvalue*

```
int x = 0;

decltype(x + x);          // ???
decltype(++x);            // ???
decltype(x++);            // ???
decltype((x));            // ???
decltype(std::move(x));   // ???
```

`decltype`

Если подать `decltype` *выражение* (не являющееся именем переменной или поля), то он вернет:

- просто тип, если категория выражения - *prvalue*
- тип с `&`, если категория выражения - *lvalue*
- тип с `&&`, если категория выражения - *xvalue*

```
int x = 0;

decltype(x + x);          // int
decltype(++x);            // int&
decltype(x++);            // int
decltype((x));            // int&
decltype(std::move(x));   // int&&
```

`decltype(auto)`

`auto` выводит тип в соответствии с правилами вывода шаблонных параметров:

```
volatile int x = 0;
auto y = x;    // int
auto z = ++y;  // int
auto t = (x);  // int
```

Но можно заставить выводить тип по правилам `decltype`:

```
volatile int x = 0;
decltype(auto) y = x;    // volatile int
decltype(auto) z = ++y;  // volatile int&
decltype(auto) t = (x);  // volatile int&
```

decltype(auto)

```
template <class T>
decltype(auto) move(T&& value) {
  return static_cast<std::remove_reference_t<T>&&>(value);
}
```

```
template <class T>
decltype(auto) forward(std::remove_reference_t<T>& value) {
  return static_cast<T&&>(value);
}
```

decltype

```
template <class T, class U, class N>
auto Sum(const std::array<T, N>& lhs, const std::array<U, N>& rhs) {
  std::array<std::remove_cvref_t<decltype(lhs[0] + rhs[0])>, N> res;
  for (size_t i = 0; i < N; ++i) {
    res[i] = lhs[i] + rhs[i];
  }
  return res;
}
```

`decltype`

Добавим `noexcept` в зависимости от того, может ли сумма бросить исключение или нет:

```
template <class T, class U, class N>
auto Sum(const std::array<T, N>& lhs, const std::array<U, N>& rhs)
noexcept(noexcept(lhs[0] + rhs[0])) {

  std::array<std::remove_cvref_t<decltype(lhs[0] + rhs[0])>, N> res;
  for (size_t i = 0; i < N; ++i) {
    res[i] = lhs[i] + rhs[i];
  }
  return res;
}
```

В чем проблема?

`declval`: проблема

Допустим, хотим создать переменную, которая имела бы тот же самый тип, что и сложение двух объектов типа `T`:

```
T x = ...;
T y = ...;
decltype(x + y) z;  // пришлось создавать x и y для этого
```

```
decltype(T() + T()) z;  // откуда мы знаем, можно ли вызывать к-р по умолчанию
```

Беда

`declval`

Функция `std::declval` позволяет "во что бы то ни стало" обратиться к объекту данного типа в *невычисляемых контекстах (unevaluated context)*

Забавный факт в том, что у нее нет определения! (Оно и не нужно)

```
decltype(std::declval<T>() + std::declval<T>()) z;
```

Пояснение: `declval` *говорит "да-да я возвращаю то, что нужно". Компилятор не проверяет истинность этого, так как реально эта функция никогда не вызывается!*

`declval`

Добавим `noexcept` в зависимости от того, может ли сумма бросить исключение или нет:

```
template <class T, class U, class N>
auto Sum(const std::array<T, N>& lhs, const std::array<U, N>& rhs)
noexcept(noexcept(std::declval<T>() + std::declval<U>())) {

  std::array<std::remove_cvref_t<decltype(lhs[0] + rhs[0])>, N> res;
  for (size_t i = 0; i < N; ++i) {
    res[i] = lhs[i] + rhs[i];
  }
  return res;
}
```

`declval`

```
template <class T>
T&& declval();
```

Почему возвращает `T&&`, а не `T`?