Функторы

Функторы

Функторы (или функциональные объекты) - объекты, для которых определена операция функционального вызова (функции, указатели на функцию, объекты с перегруженным `operator()`).

Функторы : примеры

- Передача пользовательского делитера в умный указатель:

```
std::unique_ptr<FILE, int(*)(FILE*)> file_ptr(std::fopen("filename", "wr"),
                                              std::fclose);
```

- Передача компаратора в `std::sort`

```
// хотим отсортировать строки по длине
struct SizeCompare {
  bool operator()(const std::string& lhs, const std::string& rhs) const {
    return lhs.size() < rhs.size();
  }
};

std::sort(v.begin(), v.end(), SizeCompare{});
```

А что лучше, структура с `operator()` или функция? [demo: sort]

Функторы : стандартная библиотека

В стандартной библиотеке определено небольшое количество функторов, которые могут быть использованы как параметры в других алгоритмах.

https://en.cppreference.com/w/cpp/utility/functional (operator function objects)

Наиболее часто используемые из них: `std::less` (<), `std::greater` (>), `std::equal_to` (==), `std::plus` (+), `std::minus` (-).

Функторы : беда

Вернемся к примеру с компаратором.

```
// хотим отсортировать строки по длине
struct SizeCompare {
  bool operator()(const std::string& lhs, const std::string& rhs) const {
    return lhs.size() < rhs.size();
  }
};

std::sort(v.begin(), v.end(), SizeCompare{});
```

Не кажется ли вам, что здесь написано много лишнего (boilerplate) кода, который не несет большой смысловой нагрузки?

Единственная полезная строка - это строка со сравнением.

Кроме того, пришлось создать структуру, которая, скорее всего, будет использована в коде ровно один раз.

Лямбда выражения (C++11)

Анекдот дня!

Говорю мужу.....

напиши программу на C++

Ответ убил.....

```
int main(){[](){}();}
```

Лямбда выражения : пример

Передача компаратора в `std::sort`

```
// хотим отсортировать строки по длине
std::sort(v.begin(), v.end(), [](const std::string& lhs, const std::string& rhs) {
  return lhs.size() < rhs.size();
});
```

Лямбда выражения : синтаксис

- В начале пишется *список захвата* - `[...]`. Смысл обсудим позже, пока его можно оставить пустым.
- Затем в `(...)` аргументы функции (произвольное количество). Если аргументов нет, то можно опустить.
- В конце в `{...}` идет тело функции.

```
auto f = [](int x) { std::cout << x << ' '; };
auto g = [](int x, double y) { std::cout << x << ' ' << y << ' '; return x; };
auto h = [] { std::cout <<  "empty";};

f(g(1, 2.0));
h();
```

```
1 2 1 empty
```

Лямбда выражения : возвращаемый тип

Заметьте, что до этого мы не прописывали тип возвращаемого значения.

Дело в том, что в случае одного (или нуля) `return` выражения этот тип можно легко вывести (по правилам вывода шаблонного параметра).

```
[] {};  // void

[](const std::string& lhs, const std::string& rhs) {  // bool
  return lhs < rhs;
};
```

Лямбда выражения : возвращаемый тип

Если `return` выражений несколько, и они не противоречат друг другу, то тип тоже выводится без проблем:

```
[](int x) { if (x > 0) return; };  // void

[](const std::string& lhs, const std::string& rhs) {  // bool
  if (lhs < rhs) return true;
  return false;
};
```

Лямбда выражения : возвращаемый тип

Если `return` выражений несколько, и типы возвращаемых значений различны, то возникает ошибка компиляции:

```
[](int x) { if (x > 0) return; return 0; };  // void или int?

[](const std::string& lhs, const std::string& rhs) {  // bool или int?
  if (lhs < rhs) return true;
  return 0;
};
```

```
error: inconsistent types 'void' and 'int' deduced for lambda return type
error: inconsistent types 'bool' and 'int' deduced for lambda return type
```

Лямбда выражения : возвращаемый тип

В случае конфликта, его можно указать явно после списка аргументов:

```
[](const std::string& lhs, const std::string& rhs) -> bool {  // bool
  if (lhs < rhs) return true;
  return 0;
};
```

Кстати, этот же синтаксис можно использовать и с обычными функциями:

```
auto main() -> int {
  return 0;
}
```

Лямбда выражения : аргументы

Единственное, что достойно упоминания: шаблонные лямбда выражения.

Если хочется, чтобы выражение работало с произвольными типами (или лень писать полный тип аргументов), можно на месте типа арумента писать `auto` (с любыми квалификаторами).

```
std::sort(v.begin(), v.end(), [](const auto& lhs, const auto& rhs) {
  return lhs.size() < rhs.size();
});
```

Лямбда выражение при этом аналогично объявлению:

```
template <class T, class U>  // T и U не обязаны совпадать
auto lambda(const T&, const U&);
```

Лямбда выражения : список захвата

Внутри лямбда выражений можно свободно использовать глобальные переменные, переменные из глобальных пространств имен, а также статические переменные функции.

Однако локальные переменные внутри лямбда выражения не видны:

```
int global_x = 0;

void Function() {
  static int static_x = 1;
  int local_x = 2;
  [](){
    std::cout << global_x << '\n';  // Ok
    std::cout << static_x << '\n';  // Ok
    std::cout << local_x  << '\n';  // CE
  }();  // <-- тут же вызываем лямбда функцию
}
```

```
error: 'local_x' is not captured
```

Лямбда выражения : список захвата

Для использования локальных переменных функций их нужно захватить.

Для захвата по значению нужно просто написать имя переменной в списке захвата `[...]`. Если хочется захватить несколько переменных, их нужно перечислить через запятую.

После этого внутри лямбда выражения будет сожержаться *неизменяемая* копия захваченного объекта. Почему так, станет ясно позднее.

```
void Function() {
  int x = 0;
  std::vector<int> v{1, 2, 4};
  [x, v]() { std::cout << x << ' ' << v[2]; }();  // сразу вызываем
}
```

Лямбда выражения : список захвата

(Внимание, плохой стиль!) Для захвата всех локальных переменных по значению, можно написать `[=]`.

```
void Function() {
  int x = 0;
  std::vector<int> v{1, 2, 4};
  [=]() { std::cout << x << ' ' << v[2]; }();  // сразу вызываем
}
```

Чтобы появилась возможность изменения захваченных копий, можно написать `mutable` перед телом лямбды:

```
void Function() {
  int x = 0;
  std::vector<int> v{1, 2, 4}; // <-- этот вектор не изменится!
  [=]() mutable { v[0] = x; std::cout << v[0] << v[1] << v[2]; }();
}
```

Лямбда выражения : список захвата

Для захвата локальных переменных по ссылке необходимо перед именем переменной написать `&`. Переменные, на которые ссылается лямбда *можно* изменять. Почему так, станет ясно позднее.

```
void Function() {
  int x = 0;
  std::vector<int> v{1, 2, 4};  // <-- этот вектор изменится!
  [&v, x]() { v[0] = x; }();  // сразу вызываем
}
```

(Внимание, плохой стиль!) Для захвата всех локальных переменных по ссылке, можно написать `[&]`.

```
void Function() {
  int x = 0;
  std::vector<int> v{1, 2, 4}; // <-- этот вектор изменится!
  [&]() { v[0] = x; std::cout << v[0] << v[1] << v[2]; }();
}
```

Лямбда выражения : список захвата

(Внимание, плохой стиль!) Можно захватить все переменные по значению и некоторые по ссылке, и наоборот:

```
void Function() {
  int x = 0;
  std::vector<int> v{1, 2, 4};  // <-- этот вектор изменится!
  [=, &v]() { v[0] = x; }();  // захватываем все по значению, кроме v
}
```

Захват с переименованием: можно задать имя, по которому можно обращаться к захваченной переменной:

```
void Function() {
  int x = 0;
  std::vector<int> v{1, 2, 4};
  [value = x, &buf = v, &const_buf = std::as_const(v)]() {
    buf[0] = value;
    const_buf[0] = value;  // CE, const_buf - константная ссылка на v
  }();
}
```

Лямбда выражения : список захвата

Поля класса внутри методов не могут быть захвачены по имени:

```
struct S {
  int x;

  void f() {
    [x]() { std::cout << x; }();
  }
};
```

```
error: capture of non-variable 'S::x'
```

Лямбда выражения : список захвата

(Внимание, плохой стиль!) Для использования полей класса необходимо (внезапно) захватить `this`:

```
struct S {
  int x;

  void f() {
    [this]() { std::cout << x; }();
  }
};
```

Но лучше использовать захват с переименованием:

```
struct S {
  int x;

  void f() {
    [x = this->x]() { std::cout << x; }();
  }
};
```

Лямбда выражения : как это работает

Что происходит при использовании лямбда выражений?

```
std::sort(v.begin(), v.end(), [](const std::string& lhs, const std::string& rhs) {
  return lhs.size() < rhs.size();
});
```

https://cppinsights.io/s/04efe87c

TL;DR : компилятор за вас генерирует этот код:

```
struct __UnknownClosureType__ {
  bool operator()(const std::string& lhs, const std::string& rhs) const {
    return lhs.size() < rhs.size();
  }
};

std::sort(v.begin(), v.end(), __UnknownClosureType__{});
```