Функции

Функции позволяют отделить часто используемый код и переиспользовать его с разными значениями аргументов. С примером функции мы уже знакомы: в каждой программе вы пишете функцию `main`, которая не принимает аргументов и возвращает `int`.

Примеры функций

Напишем простейшую функцию, вычисляющую сумму двух целых чисел:

```
int Sum(int a, int b) {    // в заголовке функции указывается тип возвращаемого значения и типы аргументов
    return a + b;
}
```

Если функция ничего не должна возвращать, её можно объявить как `void`:

```
void DoSomething(double d, char c) {
    // ...
        // писать return в конце такой функции не обязательно,
     //      но если требуется завершить функцию, можно написать просто return;
}

int main() {
    int x = 17, y = 42;
    int z = Sum(x, y);
    DoSomething(3.14, '@');
}
```

Вот пример рекурсивной функции, вычисляющей факториал:

```cpp
#include <cstdint>
#include <iostream>

std::uint64_t Factorial(std::uint64_t n) {
    if (n == 0) {
        return 1;
    }
    return n * Factorial(n - 1);  // рекурсивный вызов
}

int main() {
    std::cout << Factorial(5) << "\n";  // 120
}
```

Помните, что если делать очень много рекурсивных вызовов, то рано или поздно переполнится стек — область памяти, в которой хранятся аргументы и локальные переменные текущей функции.

Аргументы функций

Параметры в функции по умолчанию передаются «по значению». Другими словами, функция работает с копиями аргументов. Чтобы лучше представить это, давайте посмотрим, что бы получилось, если бы компилятор заменил вызов функции на непосредственное исполнение кода.

Возьмём такой фрагмент кода:

```cpp
void f(int x, int y) {
    // работаем с аргументами x и y
}

int main() {
    int a, b;
    // какая-то инициализация a и b

    f(a, b);
```

```
}
```

Заменим его на такой код:

```
int main() {
    int a, b;
    // какая-то инициализация a и b

    {   // этот блок просто ограничивает время жизни
        // находящихся внутри переменных
        int x = a;
        int y = b;
        // работаем с аргументами x и y
    }
}
```

Теперь видно, что любое изменение x или y внутри функции никак не затронет a и b.

Можем ли мы изменить переданный аргумент внутри функции, чтобы это повлияло на аргументы в месте вызова? Да, для этого надо передать аргументы через ссылку или указатель. Вот классический пример функции, меняющей два аргумента местами:

```
void Swap(int& x, int& y) {   // передаём аргументы по ссылке
    int z = x;
    x = y;
    y = z;
}

int main() {
    int a = 1, b = 2;
    Swap(a, b);
    std::cout << a << " " << b << "\n";  // 2 1
}
```

Чтобы понять, как это работает, раскроем снова код функции в месте вызова:

```
int main() {
    int a = 1, b = 2;

    {
        int& x = a;
        int& y = b;
        int z = x;
        x = y;
        y = z;
    }

    std::cout << a << " " << b << "\n";  // 2 1
}
```

Видно, что x и y — это просто псевдонимы для a и b.

Заметьте, что вызов `Swap(1, 2)`, в отличие от `Swap(a, b)`, не скомпилируется, потому что обычная ссылка должна быть привязана к изменяемому объекту.

Примером функции из стандартной библиотеки, которая принимает аргумент по ссылке и изменяет его, является `std::getline`:

```
#include <iostream>
#include <string>

int main() {
    std::string line;

    // Второй аргумент передаётся по ссылке и изменяется внутри функции:
    std::getline(std::cin, line);
}
```

Иногда копирование объекта может быть очень дорогим (и ненужным). Например, копирование вектора приведёт к

копированию всех его элементов. Поэтому вот так передавать вектор в функцию неэффективно:

```
void f(std::vector<int> v) {
       // плохо: при вызове функции создаётся копия вектора
}
```

Копии можно было бы избежать, если бы вектор передавался по ссылке:

```
void f(std::vector<int>& v) {
    // Но теперь есть другие недостатки:
     // 1. В такую функцию нельзя передать константный вектор.
     // 2. Функция не защищена от случайного изменения вектора:
    v.clear();  // тут компилятор нас не схватит за руку
}
```

Поэтому самое правильное — передавать такой параметр по константной ссылке:

```
void f(const std::vector<int>& v) {
    // Такой аргумент не требует дорогого копирования,
    // его нельзя случайно изменить внутри,
    // и такую функцию можно вызывать от констант!
}
```

Давайте запомним: аргументы сложных типов (векторы, строки, любые контейнеры, большие структуры) всегда лучше передавать в функцию по константной ссылке, если функция использует их только для чтения. Из этого правила бывают исключения, но о них мы поговорим отдельно.

Впрочем, это правило не стоит распространять на обычные встроенные типы:

```
void g(const int& a, const char& c) {  // так делать не надо, это уже перебор!
```

```
    // передавайте такие параметры просто по значению,
как int или char
}
```

Возвращаемые значения функций

В отличие от аргументов, значения сложных типов можно без проблем возвращать из функций. Здесь от ненужного копирования (по крайней мере, для стандартных контейнеров) спасает copy elision.

Рассмотрим, например, функцию, которая возвращает конкатенацию всех строк из вектора:

```
#include <iostream>
#include <string>
#include <vector>

std::string Concatenate(const std::vector<std::string>&
parts) {
    std::string result;
    for (const auto& part : parts) {
        result += part;
    }
    return result;
}

int main() {
        std::vector<std::string> parts = {"abra", "ca",
"dabra"};
        std::cout << Concatenate(parts) << "\n";    //
abracadabra
}
```

Опасно возвращать из функции ссылку на локальную переменную, так как эта ссылка сразу же станет «висячей»:

```
#include <iostream>

int& Sum(int a, int b) {  // ошибка!
    int result = a + b;
    return result;
}

int main() {
     std::cout << Sum(2, 3) << "\n";   // неопределённое
поведение!
}
```

Компиляторы в таких случаях генерируют предупреждения.

Возвращать значение по ссылке можно только в случае, если оно заведомо будет доступно после завершения функции. Например, так можно вернуть глобальную переменную или аргумент, также переданный по ссылке.

Функции-компараторы

Пусть имеется структура `Date`, описывающая день, месяц и год какой-то даты. Создадим вектор дат:

```
#include <algorithm>
#include <iostream>
#include <vector>

struct Date {
    int year = 1970;
    int month = 1;
    int day = 1;
};

int main() {
    std::vector<Date> dates = {
        {2020, 3, 15},
        {2019, 1, 21},
```

```
        {2021, 1, 30}
    };

    // напечатаем содержимое:
    for (const auto& [year, month, day] : dates) {
        std::cout << year << "." << month << "." << day <<
"\n";
    }
}
```

Предположим, нам требуется отсортировать даты. Для сортировки нам поможет уже знакомая функция `std::sort`, но есть нюанс: вызов `std::sort(dates.begin(), dates.end())` не скомпилируется, так как компилятор не умеет сравнивать даты между собой. Функция `std::sort` пытается найти оператор `<` для сравнения дат, но, увы, для нашей даты такого нет. Мы можем его определить. Он выглядит как функция с особым именем `operator <`, возвращающая `true`, если первый аргумент меньше второго:

```
bool operator < (const Date& lhs, const Date& rhs) {
    if (lhs.year != rhs.year) {
        return lhs.year < rhs.year;
    }
    if (lhs.month != rhs.month) {
        return lhs.month < rhs.month;
    }
    return lhs.day < rhs.day;
}
```

Здесь `lhs` и `rhs` — сокращения от left-hand side и right-hand side. Это левый и правый аргументы оператора `<`. Этот громоздкий код можно записать лаконичнее с использованием функции `std::tie`, возвращающей кортеж из ссылок, для которого уже определено лексикографическое (покомпонентное) сравнение:

```
bool operator < (const Date& lhs, const Date& rhs) {
        return  std::tie(lhs.year,  lhs.month,  lhs.day)  <
```

```
std::tie(rhs.year, rhs.month, rhs.day);
}
```

После определения `operator <` сортировка заработает. Но что, если нам в разных случаях нужно по-разному сортировать даты — например, где-то в хронологическом порядке, а где-то — без учёта года? Можно передать в `std::sort` третьим аргументом свою функцию сравнения, которая будет использована вместо `operator <`:

```
bool CompareWithoutYear(const Date& lhs, const Date& rhs)
{
    return std::tie(lhs.month, lhs.day) < std::tie(rhs.month, rhs.day);
}

int main() {
    // ...
    std::sort(dates.begin(), dates.end(), CompareWithoutYear);
}
```

Обратите внимание, что третьим аргументом в `std::sort` мы передаём саму функцию (без круглых скобок), а не результат её вызова от каких-то аргументов.

Лямбда-функции

Иногда бывает неудобно определять отдельную именованную функцию для сравнения. Тогда можно определить анонимную лямбда-функцию прямо в месте её использования:

```
#include <algorithm>
#include <vector>

struct Date {
    int year, month, day;
};
```

```
int main() {
    std::vector<Date> dates;
      std::sort(dates.begin(), dates.end(), [](const Date&
lhs, const Date& rhs) {
                  return  std::tie(lhs.month,  lhs.day)  <
std::tie(rhs.month, rhs.day);
    });
}
```

Тип возвращаемого значения тут не указывается, компилятор умеет его угадывать по `return` (его можно указать после круглых скобок на «питоновский» манер через `->`, но не обязательно).

Разберём синтаксис лямбда-функций. Тут видны три блока.

- **Квадратные скобки отвечают за контекст. В них мы можем передать переменные, которые объявлены вне лямбда-функции через запятую, и они будут доступны в самой лямбда-функции.**
- **Круглые скобки отвечают за аргументы функции.**
- **Фигурные скобки отвечают за тело лямбда-функции.**

Когда лямбды добавлялись в стандарт C++11, разработчики очень не хотели вводить для них новое ключевое слово (как `lambda` в Python) и обошлись комбинацией скобок. Есть шутка про то, что вот такая программа является вполне корректной:

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