Лекция V

19 ноября 2016

Указания компилятору

Переменная:

```
[указания компилятору] <тип_данных> <имя_переменной
```

Функция:

```
[указания_компилятору] <тип_данных>
                       <имя_функции>
                       ( <список_аргументов> )
{
  [инструкции;]
  return <значение>;
}
```

Указания компилятору. Storage class specifiers

Класс памяти (storage class) - характеристика, определяющая правила хранения и доступности идентификатора (имени переменной или функции).
Класс памяти определяется двумя независимыми свойствами: **время жизни (storage duration)** и **связывание имени (linkage)**.

Указания компилятору. Storage duration

Время жизни определяет момент, когда под идентификатор выделяется память для хранения, и момент, когда выделенная память возвращается в ОС. Имеются 3 основных типа:

- **автоматическое** время жизни (или время жизни по умолчанию): память под переменную/функцию выделяется при входе в *блок кода*, и высвобождается при выходе из него
- **статическое** время жизни: память под идентификатор выделяется в начале работы программы и высвобождается при завершении
- **динамическое** время жизни: память выделяется/высвобождается вручную по запросу (операторы **new, new[], delete, delete[]**

Указания компилятору. Linkage

Связывание (компоновка) имени определяет области видимости, в которых идентификатор будет доступен. Опять 3 основных типа:

- **отсутствие** связывания: идентификаторы доступны в той области видимости, в которой определены и во всех *вложенных областях*
- **внутреннее (internal)** связывание:идентификатор доступен в любой области видимости конкретного файла с исходным кодом (имеются ввиду один файл *.cpp)
- **внешнее (external)** связывание: идентификатор доступен в областях видимости вне файла с исходным кодом (в других *.cpp файлах, подключаемых во время компоновки исполняемого файла)

Указания компилятору. Linkage

```
1  #include <iostream>
2
3  int left_term = 5; // Глобальная переменная
4
5  int magic_sum(int num1, int num2)
6  {
7    int right_term; // Локальная переменная
8    if ( (num1 / num2) > 0 )
9      right_term = num1;
10   else
11     right_term = -num2;
12   // Здесь есть полный доступ к left_term
13   return left_term + right_term;
14 }
15
16 int main()
17 {
18   std::cout << magic_sum(8, 88) << '\n';
19   return 0
20 }
```

Указания компилятору. Storage class specifiers

1. **Автоматический** класс памяти: идентификаторы имеют *автоматическое* время жизни и связывание *отсутствует*. Применяется только для идентификаторов, расположенных **в блоках кода**. Ключевое слово для указания компилятору отсутствует в C++.

```
1  double reverse_sum(int arr[], size_t arr_size)
2  {
3    double result = 0;
4
5    for (size_t i = 0; i < arr_size; ++i) {
6      result += 1.0 / arr[i];
7    }
8
9    //std::cout << i;
10
11   return double;
12 }
```

Указания компилятору. Storage class specifiers

2. **Статический** класс памяти: идентификаторы имеют *статическое* время жизни. Связывание *отсутствует* для идентификаторов, расположенных **в блоках кода** и является *внутренним* для глобальных идентификаторов. Ключевое слово для указания компилятору: **static**.

```
1 static int global_num = 10;
2
3 void do_action(double param1, double param2)
4 {
5   static double result = 5;
6   result += param1 * param2;
7
8   std::cout << result;
9 }
```

2. **Статический** класс памяти.

Подсчёт количества вызовов функции

```
1  int get_max(int par1, int par2)
2  {
3    static unsigned calls_count = 0;
4    ++calls_count;
5    std::cout << "\n Количество вызовов ↩
       get_max: " <<  calls_count << '\n';
6
7    return (par1 > par2) ? par1 : par2;
8  }
9
10 ...
11 get_max(5, 6); // calls_count равен 1
12 get_max(88, 2); // 2
13 get_max(105, -444); // 3
```

Указания компилятору. Storage class specifiers

3. **Внешний** класс памяти: идентификаторы имеют *статическое* время жизни и *внутренним* связывание. Ключевое слово для указания компилятору: **extern**.

```
1  int global_num = 10;
2
3  void do_action(double param1, double param2)
4  {
5    extern int global_num;
6    double result = global_num / 5.5;
7    result += param1 * param2;
8
9    std::cout << result;
10 }
```

Указания компилятору. Storage class specifiers

К определению функций также может быть добавлено ключевое слово **static** или **extern**.

```
1 static int find_max(int param1, int param2)
2 {
3    ...
4 }
5
6 extern int find_min(int param1, int param2)
7 {
8    ...
9 }
```

Указания компилятору. Storage class specifiers

Глобальные идентификаторы могут получать спецификатор класса памяти **неявно** по следующим правилам:

- глобальные неконстантные переменные и функции по умолчанию являются **extern**;
- глобальные константы по умолчанию являются **static**

```
1 unsigned globaL_score = 50; // extern
2 const unsigned SIZE = 8; // static
3
4 extern const unsigned LMAX = 145;
5
6 void print_a() // extern функция
7 {
8   std::cout << 'a';
9 }
```

Функции. Объявление и определение

```
1  // Объявление функции
2  long long factorial(unsigned );
3
4  // Определение функции
5  long long factorial(unsigned n)
6  {
7    if ( n < 2) {
8      return 1;
9    }
10
11   long long result = 1;
12   for (unsigned i = 2; i <= n; ++i) {
13     result *= i;
14   }
15
16   return result;
17 }
```

Функции. Объявление и определение

Зачем нужно:

```
1  // Объявление функции
2  long long factorial(unsigned );
3
4  void solve_problem(int a, int b, int c)
5  {
6    ...
7    factorial(a - b + c/2);
8    ...
9  }
10
11 long long factorial(unsigned n) { ... }
```

Функции. Передача указателя

Передача указателя в функцию осуществляется **по значению**, аналогино переменным.

Также указатель может быть передан **по ссылке**.

```
1  void init_array(int *p_arr, size_t count)
2  {
3    p_arr = new int[count];
4    for (size_t i = 0; i < count; ++i) {
5      p_arr[i] = 2 + i*i;
6    }
7  }
8  int *my_int_arr = nullptr;
9  init_array(my_int_arr, 15);
10
11 if (my_int_arr == nullptr) {
12   std::cout << "my_int_arr не ↩
       инициализирован\n";
13 }
```

Функции. Передача указателя

Исправление: использование многомерных указателей

```
1  void init_array(int **p_arr, size_t count)
2  {
3    (*p_arr) = new int[count];
4    for (size_t i = 0; i < count; ++i) {
5      (*p_arr)[i] = 2 + i*i;
6    }
7  }
8
9  ...
10 int *my_int_arr = nullptr;
11 // Для создания 2D указателя из 1D
12 // просто берём его адрес
13 init_array(&my_int_arr, 15);
```

Функции. Передача указателя

Исправление: использование ссылки на указатель

```
1  void init_array(int* &p_arr, size_t count)
2  {
3    p_arr = new int[count];
4    for (size_t i = 0; i < count; ++i) {
5      p_arr[i] = 2 + i*i;
6    }
7  }
8
9  ...
10 int *my_int_arr = nullptr;
11 // Даже адрес указателя не требуется
12 // при передачи по ссылке
13 init_array(my_int_arr, 15);
```

Рекурсивные функции

В информатике под *рекурсией* понимается метод, в котором решение исходной задачи зависит от набора решений частных случаев этой же задачи. В языках C/C++ для реализации подобных методов была предоставлена возможность внутри функции делать вызов самой себя. Такие функции называются **рекурсивными**.

```
1 int get_loop(int num)
2 {
3   static int calls_count = 0;
4   std::cout << "Номер вызова: " << ++↩
        calls_count << '\n';
5
6   return get_loop(num);
7 }
```

Рекурсивные функции

Различают *прямую* и *косвенную* рекурсивную функцию.

- **прямая рекурсивная функция** - вызывает саму себя непосредственно в своём теле
- **косвенная рекурсивная функция** - вызывает некоторую другую функцию, в теле которой происходит обратный вызов исходной функции

```
1  int stat_loop(int);
2
3  int get_loop(int num)
4  {
5    return stat_loop(num);
6  }
7
8  int stat_loop(int n)
9  {
10   return get_loop(n - 5);
11 }
```

Рекурсивные функции

Для выполнения полезных действий рекурсивная функция должны выполняться два необходимых условия:

1. Тело функции должно обязательно содержать **условия остановки рекурсивных вызовов**.
2. Параметры функции при рекурсивном вызове **должны меняться, а не оставаться постоянными**.

Рекурсивные функции

```
1  int get_loop(int num)
2  {
3    static int calls_count = 0;
4    std::cout << "Номер вызова: " << ++↩
       calls_count << '\n';
5
6    if (num < 5) {
7      return 15;
8    }
9
10   return get_loop(--num);
11 }
12 ...
13 get_loop(225);
```

Рекурсивные функции

Факториал числа

$$n! = 2 * 3 * 4 ... * n$$

```
1  unsigned long long factorial(unsigned n)
2  {
3    if (n < 2) {
4      return 1;
5    }
6
7    return n * factorial(n - 1);
8  }
9  ...
10 factorial(16);
```

Числа Фибоначчи

$$F_1 = 1, F_2 = 1, F_n(n > 2) = F_{n-1} + F_{n-2}$$

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, ...

Рекурсивные функции

Числа Фибоначчи

$$F_0 = 0, F_1 = 1, F_2 = 1, F_n(n \geq 2) = F_{n-1} + F_{n-2}$$

```
1  unsigned long long fib_num(unsigned n)
2  {
3    // условие выхода из рекурсии
4    if (n < 2) {
5      return n;
6    }
7
8    return fib_num(n - 1) + fib_num(n - 2);
9  }
10
11 ...
12 fib_recursive(8);
```

Пространства имён

Пространства имён (namespaces) - способ группировки переменных и функций под общим префиксом, который служит для предотвращения конфликтов в названиях. По сути, каждое пространство имён ограничивает область видимости вложенных в него идентификаторов, но никак не влияет на их время жизни (storage duration).

Синтаксис объявления нового пространства имён:

```
namespace <имя_пространства_имён>
{
  // Объявление переменных, констант, функций, ...
}
```

Пространства имён

```
1  namespace my_lib
2  {
3    const int LSIZE = 50;
4    double rate;
5
6    long long factorial(unsigned n)
7    { ... }
8  }
9
10 ...
11 std::cout << my_lib::LSIZE;
12 my_lib::rate = 55;
13 long long ten_fact = my_lib::factorial(10);
```

Пространства имён

Для "импорта"идентификаторов из пространства имён используется ключевое слово **using**

```
1  namespace my_lib
2  {
3    const int LSIZE = 50;
4    double rate;
5
6    long long factorial(unsigned n)
7    { ... }
8  }
9
10 void f() {
11   using namespace my_lib;
12   std::cout << LSIZE;
13   rate = 55;
14   long long ten_fact = factorial(10);
15 }
```

Пространства имён

using может быть применён в любой области видимости (глобальная, функция, цикл, условие if)

```
1  namespace my_lib
2  {
3    const int LSIZE = 50;
4    double rate;
5
6    long long factorial(unsigned n)
7    { ... }
8  }
9
10 using my_lib::LSIZE;
11
12 ...
13
14 std::cout << "Линейный размер: " << LSIZE;
15 my_lib::rate = 8.5;
```

Пространства имён

При импорте из разных пространств имён, компилятор до последнего будет стараться выбрать правильную версию функций с одинаковым именем

```
1  namespace A {
2    void f(char sym) { ... }
3    void g(double num) { ... }
4  }
5
6  namespace B {
7    void f(int i) { ... }
8    void g(double num) { ... }
9  }
10
11 using namespace A;
12 using namespace B;
13
14 f('a'); // Всё хорошо
15 f(567); // Всё хорошо
16 g(56.88); // Ошибка, выбор не очевиден
```

Пространства имён

Пространства имён могут быть вложенными

```
1  namespace my_lib
2  {
3    const int LSIZE = 50;
4    double rate;
5
6    namespace my_math
7    {
8      long long factorial(unsigned n)
9      { ... }
10   }
11 }
12
13 ...
14
15 std::cout << "факториал шести: " << my_lib::↩
     my_math::factorial(6);
```

Пространства имён

Пространства имён являются расширяемыми - переменные и функции одного пространства имён могут быть определены в любом количестве исходных файлов

```
1 // Так делать не стоит,
2 // однако возможно.
3 namespace std
4 {
5   double positive_rate = 5.5;
6 }
7
8
9 std::cout << std::positive_rate;
```

Пространства имён

Пространства имён могут быть безымянными

```
1 namespace
2 {
3   double positive_rate = 5.5;
4   void test_fun(double a, double b);
5 }
```

что эквивалентно коду:

```
1 static double positive_rate = 5.5;
2 static void test_fun(double a, double b);
```

Безымянные пространства имён ограничивают доступ к иденктификаторам только для текущего файла с исходным кодом.

Пространства имён

Пространства имён могут иметь псевдонимы

```
1 #include <iostream>
2
3 namespace std_io = std;
4
5 using std_io::cout;
6 cout << "Строка. Просто строка. И число: ";
7 int num
8 std_io::cin >> num;
```