Лекция XII

22 апреля 2017

Дополнительный способ передачи начальной конфигурации в программу

Функция main

В седьмой лекции (20 - 24 слайды) приводился пример про *конфигурационные файлы*. Идея была проста: каждая прикладная программа зависит от некоторого набора начальных данных. В случае вычислительных задач такими данными могут выступать: точность вычислений, какие-нибудь подгоночные параметры, начальные и/или граничные условия, названия временных/выходных файлов. И каждый раз при запуске программы вводить их через консольный (или оконный) интерфейсы рано или поздно надоедает. Для упрощения жизни и используют начальные конфигурации, которое могут быть переданы через файлы.
Но не только через них.

Функция main

Другой полезный механизм передачи параметров в программу относится к способу, которым происходит запуск прикладных программ в операционной системе. Когда вы нажимаете любой значёк в настольных системах или кликаете по иконкам в смартфонах/планшетах, ОС в этот момент получает название файла для запуска **текстовом виде**. Например, команда на запуск браузера Chrome в ОС Windows может выглядить так:

```
C:\Program Files (x86)\Chrome\chrome.exe
```

, где **chrome.exe** - сам запускаемый файл, все символы до него - полный путь к нему. Команду можно набрать в *командной строке*(cmd.exe) и будет выполнен запуск браузера точно также, как при клике на иконке.

Функция main

Современный браузер - это очень сложная программа, работа которой зависит в том числе и от оборудования на компьютере (тип процессора, стоит/отсутствует внешняя видеокарта, тип её). В этом случае при установке некоторой программы разумно эти параметры определить и сформировать строку запуска более многословно. Как пример:

```
chrome.exe --type=gpu-broker --enable-pinch
           --ppapi-flash-plugin=path_to_flash
```

Все символы после **chrome.exe** называются **аргументами командной строки**. Каждый аргумент отделяется от других **пробелом**. Это и есть второй способ передачи конфигурационных параметров в программу. Аргументы могут быть переданы в **любую** программу, при этом программа на любом языке программирования оставляет за собой право проигнорировать или принять их.

Функция main

Языки C/C++ тут не исключение, а приём параметров осуществляется, как можно было заметить из названия последних слайдов, с помощью функции **main**. Вспомним, что **main** - это точка входа в любую исполняемую программу, самая первая функция, строки которой выполняются. Её имя определено стандартом языка и не может быть изменено. Минимальный вариант этой функции есть:

```
1 int main()
2 {
3   return 0;
4 }
```

Функция main

Кроме варианта без параметров, **main** можно определять как функцию, принимающую **два** аргумента: первый является целым числом, которое равно **общему количеству переданных аргументов в командной строке**; а второй - массивом строк *C-style*, где каждый элемент массива содержит **значение каждого аргумента в строковом виде**. Названия параметров **main** может быть любым, но по негласным стандартам используются только **argc** (arguments count) и **argv** (arguments values), соответственно.

```
1 int main(int argc, char *argv[])
2 {
3   return 0;
4 }
```

Здесь параметр функции **argv** - это массив указателей на **char**, а **argc** - количество элементов массива.

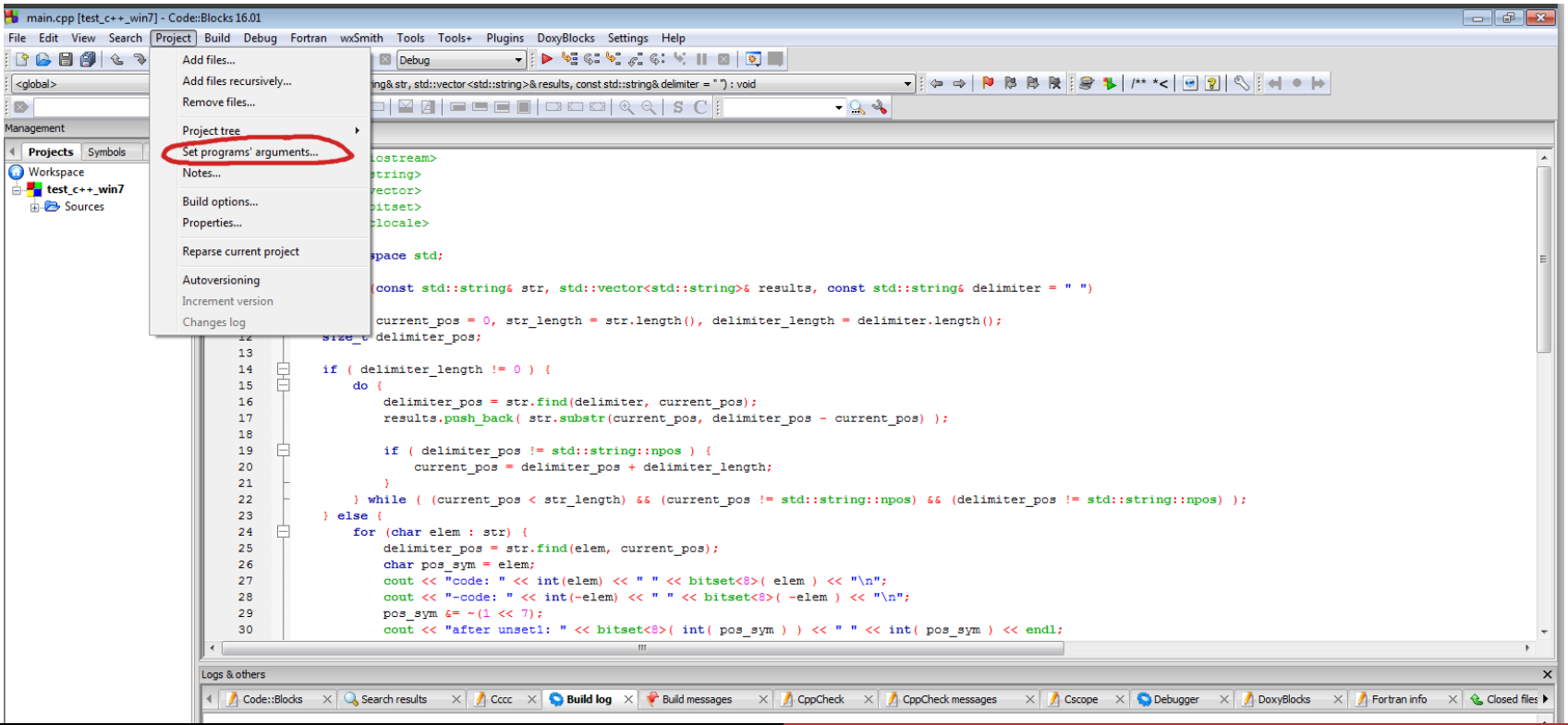
Функция main

Если вспомнить пятый слайд, то там было название программы и три аргумента командной строки. Первая особенность их передачи заключается в том, что **полный путь у исполняемому файлу** тоже попадает в функцию **main**. Таким образом, **argc** всегда положителен и равен единице при отсутствиии других аргументов. Элементы массива **argv** имеют индексы от **argv[0]** до **argv[argc - 1]**. Пример:

```
1  #include <iostream>
2  #include <clocale>
3
4  int main(int argc, char *argv[])
5  {
6    std::setlocale(LC_ALL, "RUS");
7    for (int i = 0; i < argc; i++) {
8      std::cout << "Аргумент номер " << i + 1
9           << " содержит значение" << argv[i] << '\n';
10   }
11   return 0;
12 }
```

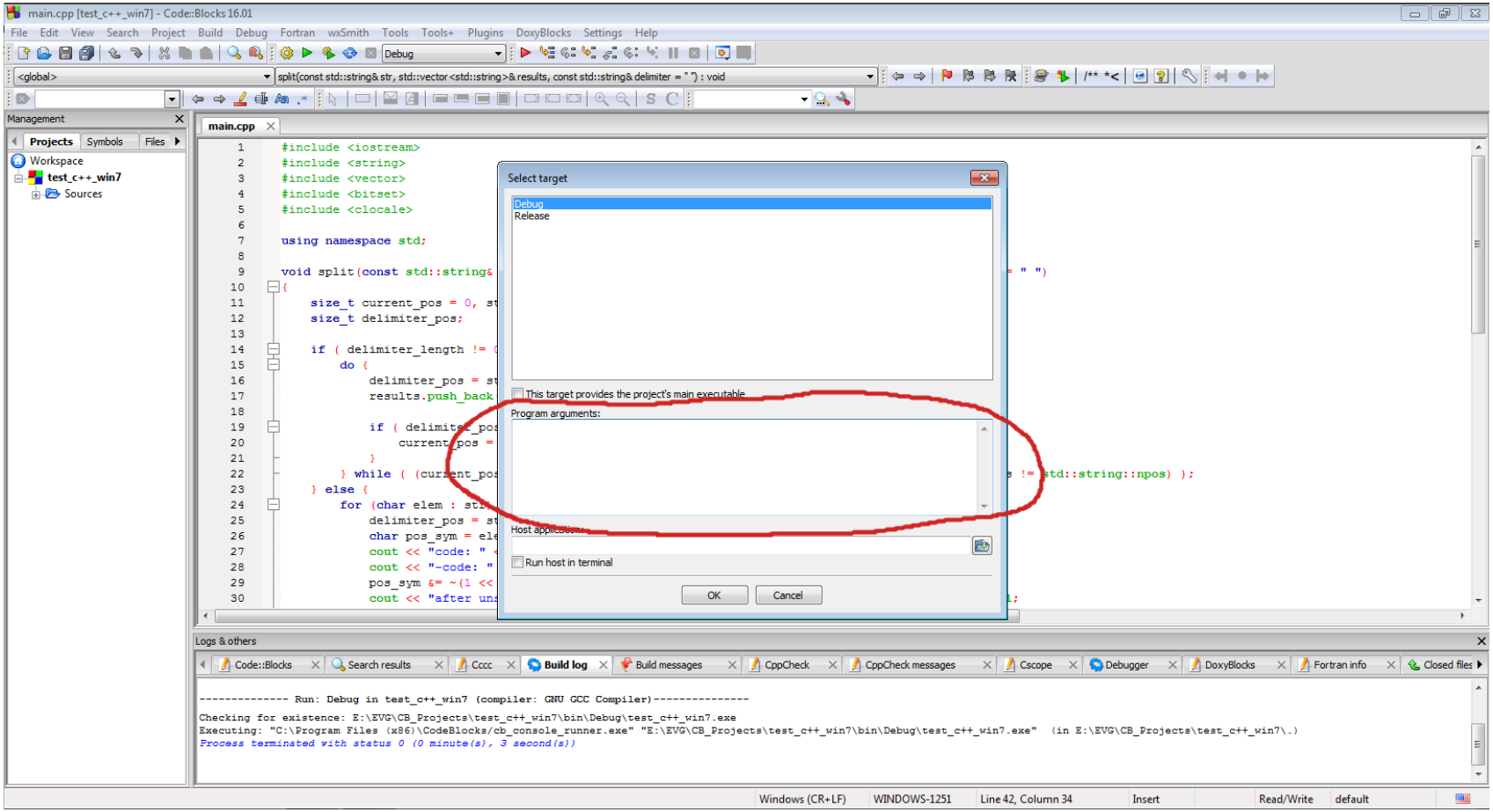
Функция main

Пример предыдущего слайда просто печатает все аргументы из командной строки, переданные в программу. Для того, чтобы увидеть работающий пример желательно эти аргументы задать. Это делается двумя способами - либо идём в *командную строку*, там ищем директорию с созданным исполняемым файлом и вызываем его руками. Либо пользуемся возможностями IDE. В CodeBlocks для программы аргументы в текстовом виде можно ввести через меню **"Project"** -> **"Set programs' aruments"**



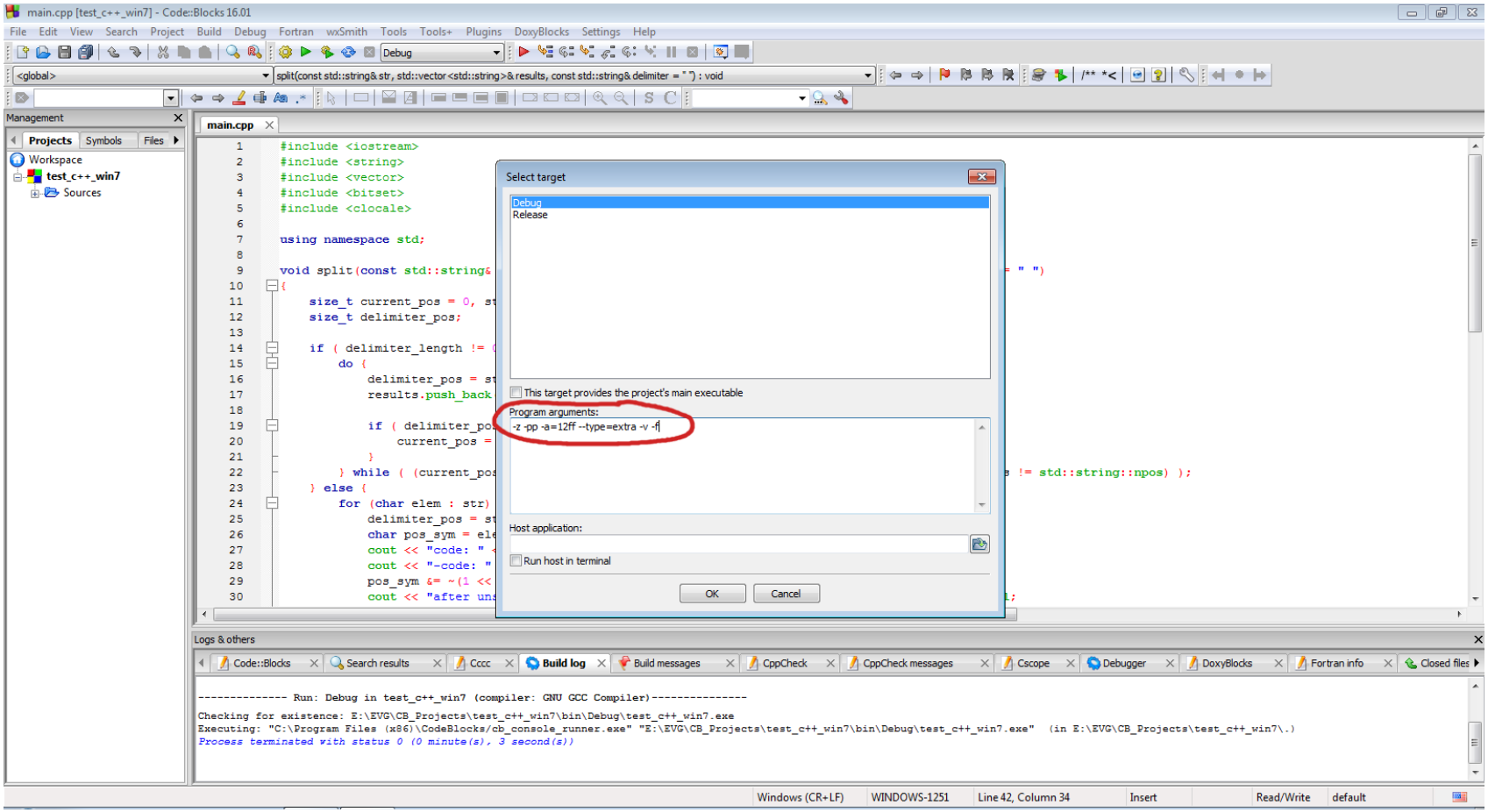
Функция main

Далее найти текстовое поле **Program arguments**



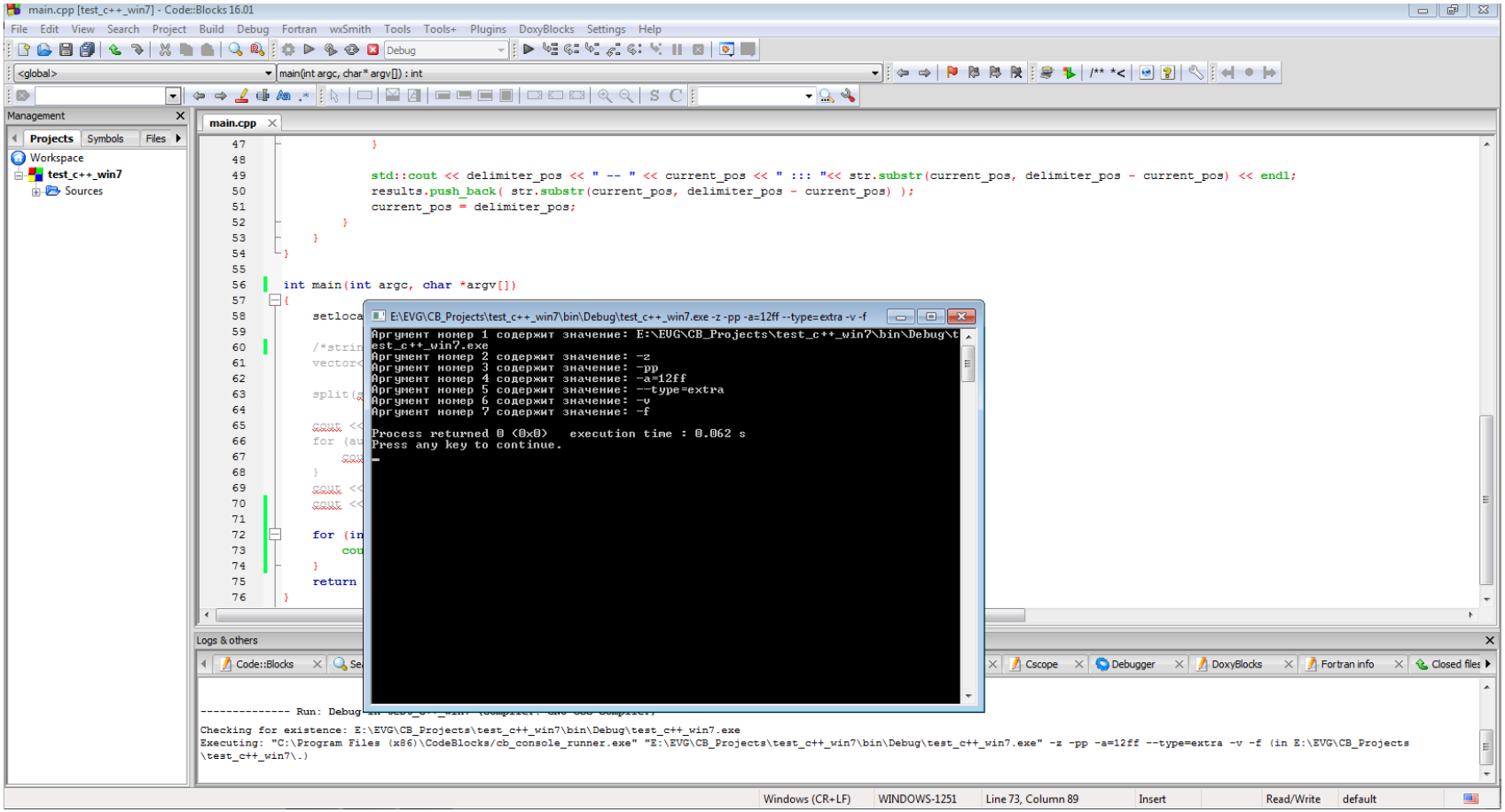
Функция main

Ввести что-нибудь туда



Функция main

И проверить работоспособность



Функция main

Аналогичным образом можно задавать аргументы командной строки в других средах для разработоки (QtCreator, Visual Studio, ...).

Далее решим следующую задачу: написать программу для численного интегрирования одномерной фукнции на заданном отрезке $[a; b]$ и с заданной точностью eps. Возможный ввод функций ограничим пятью штуками: $\sin, \cos, x, x^2, x^3$. Синус и косинус берутся из стандартной библиотеки **cmath**, степенные - сами определим. Вызов программы возможен такой строкой:

```
proga-name.exe --left=4 --right=12
       --accuracy=0.0000001 --fn=x
```

Функция main

Программа на одном слайде никак не поместится, идём по частям. Для начала - нужные библиотеки и вспомогательное перечисление

```
1  #include <iostream> // cout, cin
2  #include <clocale>  // setlocale
3  #include <cmath>    // sin, cos
4  #include <cstdlib>  // atoi, atof
5  #include <cstring>  // strncmp
6
7  // Перечисления для проверки успешности разбора
8  // аргументов командной строки
9  enum class ParseStatus
10 {
11   OK, ERROR
12 };
```

Функция main

Теперь - вспомогательные функции

```
13
14 // Вспомогательные функции: x, x^2, x^3
15 double fn_x(double x)
16 {
17   return x;
18 }
19
20 double fn_x2(double x)
21 {
22   return x * x;
23 }
24
25 double fn_x3(double x)
26 {
27   return x * x * x;
28 }
```

Функция разбора аргументов командной строки. Часть 1

```
29
30 ParseStatus parse_cmd_args(int argc, char *args[],
31       // параметры для сохранения a, b, eps
32       double& left_limit, double& right_limit, double& eps,
33       // параметр для сохранения указателя на нужную функцию
34       double (*& p_fun)(double), std::string& fn_name)
35 {
36   /*
37     argc — имеет тоже значение, что и для main.
38     args — аналогичен argv, изменено только название,
39     для избежания путаницы.
40
41     Если количество аргументов меньше 5 (включая название
42     программы)— вернуть константу перечисления
43     ParseStatus, свидетельствующую об ошибке разбора.
44   */
45   if (argc < 5) {
46     return ParseStatus::ERROR;
47   }
```

Функция разбора аргументов командной строки. Часть 2

```
48
49    /*
50      Задаём строгий порядок следования аргументов:
51      первым идёт левый предел интегрирования.
52      Проверяем нужный набор символов (strncmp)
53      на соответствие ожидаемому названию аргумента,
54      если всё хорошо — из части строки argv[1],
55      справа от знака "=" извлекаем число с помощью
56      функции *atof*
57    */
58    if ( strncmp(args[1], "--left=", 7) == 0) {
59      left_limit = std::atof(args[1] + 7);
60    } else {
61      return ParseStatus::ERROR;
62    }
63    // Аналогично для правого предела
64    if ( strncmp(args[2], "--right=", 8) == 0) {
65      right_limit = std::atof(args[2] + 8);
66    } else {
67      return ParseStatus::ERROR;
68    }
```

Функция разбора аргументов командной строки. Часть 3

```
69
70    // Третьим параметром требуем точность
71    if ( strncmp(args[3], "--accuracy=", 11) == 0) {
72      eps = std::atof(args[3] + 11);
73    } else {
74      /*
75        Выше было мало места, чтобы написать,
76        но если хотя бы один параметр не совпал
77        по порядку или по имени — сразу же
78        возращаем индикатор неудавшегося разбора
79      */
80      return ParseStatus::ERROR;
81    }
```

Функция разбора аргументов командной строки. Часть 4

```
82
83    if ( strncmp(args[4], "--fn=", 5) == 0) {
84      /*
85        В параметр fn_name запишется название функции.
86        Оно будет использовано при выводе рзультата
87        вычисления интеграла.
88      */
89      fn_name = std::string(args[4] + 5);
90
91      /*
92        За неименеем механизма более продвинутого
93        выбора, просто сравниваем переданное название
94        с совсеми допустимыми функциями и
95        устанавливаем значения указателя-на-функцию
96        p_fun. Если хоть одно условие *if* совпало -
97        можно выходить из функции с помощью
98        *return*
99      */
100     if (fn_name == "sin") {
101       p_fun = std::sin;
102       return ParseStatus::OK;
103     }
```

Функция разбора аргументов командной строки. Часть 4

```
104
105     if (fn_name == "cos") {
106       p_fun = std::cos;
107       return ParseStatus::OK;
108     }
109
110     if (fn_name == "x") {
111       p_fun = fn_x;
112       return ParseStatus::OK;
113     }
114
115     if (fn_name == "x2") {
116       p_fun = fn_x2;
117       return ParseStatus::OK;
118     }
119
120     if (fn_name == "x3") {
121       p_fun = fn_x2;
122       return ParseStatus::OK;
123     }
124
125     return ParseStatus::ERROR;
126   } else {
127     return ParseStatus::ERROR;
128   }
129 }
```

Функция интегрирования (используется метод прямоугольников)

```
130
131 double integrate_fn(double a, double b, double eps,
132                       double (*fn)(double))
133 {
134     double diff = 1.0 + std::abs(eps), h, first_sum, second_sum;
135     unsigned long splits = 1000;
136
137     if (eps < 0) { eps = std::abs(eps); }
138
139     first_sum = second_sum = 0.0;
140     h = (b - a) / splits;
141     for (unsigned i = 0; i < splits - 1; ++i) {
142         first_sum += h * fn(a + (i + 1) * h);
143     }
144
145     while (diff > eps) {
146         splits *= 2;
147         h = (b - a) / splits;
148         for (unsigned i = 1; i <= splits; ++i) {
149             second_sum += h * fn(a + (i + 1) * h);
150         }
151
152         diff = std::abs(second_sum - first_sum);
153         first_sum = second_sum;
154         second_sum = 0;
155     }
156
157     return first_sum;
158 }
```

Всё готово, функция **main**

```
159
160 int main(int argc, char *argv[])
161 {
162   std::setlocale(LC_ALL, "RUS");
163
164   double left_bound, right_bound;
165   double eps, (*p_fun)(double);
166   std::string fun_name;
167   ParseStatus st = parse_cmd_args(argc, argv, left_bound,
168                       right_bound, eps, p_fun, fun_name);
169
170   if (st == ParseStatus::OK) {
171     double result = integrate_fn(left_bound, right_bound, eps, p_fun);
172     std::cout << "Интеграл функции " << fun_name << " на отрезке от "
173           << left_bound << " до " << right_bound
174           << " с точностью " << eps << " равен " << result;
175   } else {
176     std::cout << "Использование: "
177       << argv[0] << " --left=<число> --right=<число> --accuracy=<число> --fn=<↩
            название_функции>\n\n"
178       << "  --left  - левая граница интегрирования\n"
179       << "  --right - правая граница интегрирования\n"
180       << "  --accuracy - точность интегрирования\n"
181       << "  <название_функции> может быть одним из: sin, cos, x, x2, x3\n\n";
182   }
183 }
```

Функция main

Вводя в IDE аргументы программы вида:

```
--left=4 --right=12 --accuracy=0.0005 --fn=x2
```

а после - запуская вообще без них: наглядно увидите разницу в текстовом выводе

Дополнительная порция фактов про классы в C++

this в методах

В предыдущих лекциях были приведены несколько примеров построения собственных классов. Для повторения: основная цель объявляения классов - спрятать некоторые поля-данные от прямого доступа и предоставить открытые методы для работы с ними. Обыкновенный метод класса отличается от обычной функции как раз тем, что у него есть доступ к полям конкретной **переменной класса**.

this в методах

Для примера, рассмотрим тестовый класс

```
1  class TestThis
2  {
3  public:
4    TestThis(unsigned integer, char symbol) : num{integer}, sym{⟵
         symbol}
5    { arr = new double[num]; }
6
7    ~TestThis()
8    { delete[] arr; }
9
10   void method1()
11   {
12     cout << "num: " << num << ", sym: " << sym << endl;
13   }
14
15 private:
16   unsigned  num;
17   char sym;
18   double *arr
19 };
```

this в методах

Достаточно рядовой пример, два простых поля, одно - динамический массив (**arr**). Метод **method1** печатает два поля, можно проверить вывод следующим кодом:

```
1 TestThis tt1{10, '#'}, tt2{15, '&'};
2
3 tt1.method1();
4 tt2.method1();
```

Однако у методов класса есть ещё одна особенность: внутри них доступен **специальный указатель** по имени **this**. Это ключевое слово языка C++, оно не может быть использовано в качестве переменной. Данный указатель ведёт себя в точности также, как указатель на структуру, только по отношению ко всем полям-данным класса. Применяя к нему операцию разыменования, можно получать значения полей. Говорят, что **this** указывает на **внутреннее состояние** конкретного объекта класса. Рассмотрим далее на примере.

this в методах

Явное использование this

```
1  class TestThis
2  {
3  public:
4    TestThis(unsigned integer, char symbol) :
5                              num{integer}, sym{symbol}
6    { this->arr = new double[num]; }
7
8    ~TestThis()
9    { delete[] this->arr; }
10
11   void method1()
12   {
13     cout << "Адрес оъекта: " << this << endl;
14     cout << "num: " << this->num << ", sym: "
15                    << this->sym << endl;
16   }
17
18 private:
19   unsigned  num;
20   char sym;
21   double *arr
22 };
```

this в методах

По примеру выше: само поведение метода не изменилось, просто обращение к каждому полю стали происходить по полному названию. Плюс начали печатать адрес каждого объекта. Напомним, что поля каждого объекта класса расположены непрерывно в памяти. Как следствие, **this** указывает на начало блока для каждой переменной класса.

```
1 TestThis tt1{10, '#'}, tt2{15, '&'};
2
3 tt1.method1();
4 tt2.method1();
```

Для обычных методов **this** не очень то и полезен. Но есть одна ситуация - копирование объектов с динамическими блоками памяти. Как выше, есть поле **arr** - которое представляет собой динамический массив (память выделяется в конструкторе).

this в методах

В восьмой лекции был пример создания собственного двумерного динамического массива. В ней на 36 слайде определялся конструктор копирования. Его идея была в том, чтобы исключить существование двух разных **объектов класса**, ссылающихся на один динамический блок памяти. Повторим конструктор копирования и для тестового класса

```
1  class TestThis
2  {
3  public:
4    // ...
5    TestThis(const TestThis& other) :
6                              num{other.num}, sym{other.sym}
7    {
8      arr = new double[num];
9      for (size_t i = 0; i < num; ++i) {
10       arr[i] = other.arr[i];
11     }
12   }
13 private:
14   // ...
15 };
```

this в методах

Всё хорошо: при создании нового объекта из существующего динамически выделенная память никогда не пересечётся

```
1 TestThis tt3{14, '*'};
2 TestThis tt4 = tt3; // используем конструктор копирования
```

Однако, что будет, если попробывать присвоить значение одного уже созданного объекта другому?

```
3
4 TestThis tt5{4, '@'};
5 tt5 = tt4;
6 // Снова проблема: два объекта указывают на один блок ↩
      динамической памяти
```

Здесь не вызывается конструктор копирования, поскольку и **tt4** и **tt5** - уже созданные объекты. Пятая строка полностью корректна с точки зрения языка: C++ предоставляет **оператор присваивания по умолчанию** для каждого класса. Вот только всё, что он умеет - это скопировать значения полей от одного объекта к другому.

Использвание **оператора присваивания по умолчанию** для классов, динамиески выделяющих ресурсы, приводит к двум проблемам:

1. Утечка памяти (поля объекта поменяли значения, но старый блок памяти никто не удалил)
2. Вызов деструктора становится небезопасным (скорее всего, программа с предыдущего слайда просто падает при её запуске)

Для устранения этих недостатков следует делать **перегрузку** оператора присваивания для подобных классов. Раз следует - то сейчас сделаем

Перегрузка оператора присваивания

```
1  class TestThis
2  {
3  public:
4    // ...
5    TestThis& operator=(const TestThis& other)
6    {
7      double *old = arr;
8
9      num = other.num;
10     sym = other.sym;
11
12     arr = new double[num];
13     for (size_t i = 0; i < num; ++i) {
14       arr[i] = other.arr[i];
15     }
16
17     delete[] old;
18     return *this; // Возвращаем ссылку на тот объект, что стоял↩
           слева от знака "="
19   }
20 private:
21   // ...
22 };
```

this в методах

Оператор присваивания сдедали по образу и подобию конкструктора копирования. Добавилась только одна операция - освобождение динамической памяти, которая хранилась в объекте до того, как было вызвано присваивание. Теперь то никаких утечек памяти и опасностей при вызове деструкторов разных объектов. Осталось расмотреть одну ситуацию, чтобы понять, чем же **this** может быть тут полезен.

```
1 TestThis tt3{14, '*'};
2 TestThis tt4 = tt3;
3 TestThis tt5{4, '@'};
4 tt5 = tt4; // нет проблем, всё работает
5
6 tt5 = tt5; // а здесь?
```

На самом деле, наша перегрузка оператора присваивания безопасна и в случаи 6-ой строки. Но самоприсваивание делается страшно неэффективно: новый буфер того же размера, копирование элементов, удалени старого буфера. Надо исправлять

this в методах

Здесь то this и пригодится

```
1  class TestThis
2  {
3  public:
4    // ...
5    TestThis& operator=(const TestThis& other)
6    {
7      // При самокопировании — ничего не делаем
8      if (this == &other) { return *this; }
9
10     delete[] arr;
11     num = other.num;
12     sym = other.sym;
13
14     arr = new double[num];
15     for (size_t i = 0; i < num; ++i) {
16       arr[i] = other.arr[i];
17     }
18
19     return *this;
20   }
21 private:
22   // ...
23 };
```

this в методах

Вот теперь всё в ниже лежащих строках работает оптимально.

```
1 TestThis tt3{14, '*'};
2 TestThis tt4 = tt3;
3 TestThis tt5{4, '@'};
4
5 tt5 = tt4;
6 tt5 = tt5;
```

static поля и методы

Все поля и методы классов, которые приводились как примеры в данной и предыдущих лекциях, имели одно важное свойство: **они принадлежали объектам класса**. Как следствие, при объявлении 10 переменных класса - происходит выделение 10 блоков памяти под все поля, объявленные в нём.

Но классы позволяют дополнительно объявлять поля, которые инициализируются **только раз** и могут быть использованы либо любыми другими методами, либо кодом вне класса (при нужной области видимости). Такие поля получили название **статических** и для их объявления используется ключевое слово **static**.

Аналогично классы в C++ позволяют определять **статические методы**. Их особенности будут рассмотрены немного позднее.

static поля и методы I

Статические поля. Пример

```
1  class TestStatic
2  {
3  public:
4    static int open_num;
5
6    // статические поля доступны внутри любого метода класса
7    void do_it()
8    { hidden_num = 2.5 * open_num; }
9
10 private:
11   static double hidden_num;
12 };
```

В приведённом примере два поля объявлены с ключевым словом **static**. Технически это означает, что под поля **open_num** и **hidden_num** память будет выделена в момент начала выполнения программы. Это ключевой момент: под обычные, нестатические поля память выделяется только в момент объявления переменной класса. А здесь - пораньше.

static поля и методы II

Далее **open_num** и **hidden_num** ведут себя как обычные переменные: память под них выделена, но никаких чисел не присвоено, соответственно внутри какие-то "мусорные"значения. Обратите внимание, что **open_num** - **открытое поле** класса, а **hidden_num** - **закрытое**. Это влияет на то, как до них можно добраться в программе. Открытое статическое поле может быть получено с помощью имени класса:

```
12
13 TestStatic::open_num = 12;
14 int res = TestStatic::open_num * 14;
15 cout << TestStatic::open_num << '\n';
```

static поля и методы III

Открытые статические поля от какой-нибудь глобальной переменной имеют только одно отличие: требуется добавлять название класса и два двоеточия. А так - переменная как переменная, ничего особенного. Обратиться напрямую к полю **hidden_num** возможности нет, оно закрыто. Нужно создавать хотя бы один объект класса.

```
16
17 TestStatic ts1;
18 ts1.do_it(); // теперь hidden_num получила осмысленное зачение
```

А что, если захочется задать некоторые начальные значения статическим полям до их использования в программе? При необходимости C++ позволяет сделать это как для открытых, так и для закрытых полей. На примере:

static поля и методы IV

```
1  class TestStatic
2  {
3  public:
4    static int open_num;
5
6    void do_it() { hidden_num = 2.5 * open_num; }
7  private:
8    static double hidden_num;
9  };
10
11 // Синтаксис: <тип_поля> <имя_класса>::<имя_поля> = <значение>
12 int TestStatic::open_num = 15; // *static* можно не указывать
13 double TestStatic::hidden_num = 5.5;
```

На задание начальных значений для статических полей накладываются два ограничения:

- Присвоение должно быть выполнено вне исходного класса и вне любых функций (в т.ч. вне **main**)
- Присвоение может находиться только в ***.cpp** файле (*чтобы это сейчас не значило*)

static поля и методы

Статические поля могут быть **константными**. Рассмотрим на примере с комментариями:

```
1  class TestStatic2
2  {
3  public:
4    /*
5      Объявлены два неизменяемых статических поля. Первое —
6      целого типа, второе — действительного. И здесь
7      отличие — константным статическим полям целых типов
8      данных (int, unsigned, long, size_t и т.д.) можно
9      присваивать значения прямо в месте объявления.
10   */
11   static const int CONST_INT_NUM = 145;
12   /*
13     А для константных статических полей других типов
14     данных — извините, присвоение значения всегда
15     должно быть вне класса и любых других функций.
16   */
17   static const double CONST_REAL_NUM;
18 };
19
20 // Например, здесь. *const* — обязательно к указанию
21 const double TestStatic2::CONST_REAL_NUM = 5.5;
```

static поля и методы

Очевидна некоторая несогласованность при объявлении константных статических полей. Для её преодоления можно воспользоваться ключевым словом **constexpr**, появившемся в языке начиная со стандарта C++11. Если **const** объявляет переменную константой времени выполнения (после запуска программы - значение переменной не поменяется), то **constexpr** обозначает константу **времени компиляции**.

```
1 class TestStatic2
2 {
3 public:
4   // Присвоение начальных значений в одном месте
5   static constexpr int CONST_INT_NUM = 145;
6   static constexpr double CONST_REAL_NUM = 5.5;
7   static constexpr char CONST_STR[] = "Какой-то токен";
8 };
```

Предупреждение

Не все составные типы данных (суть - классы) подходят для того, чтобы быть константными переменными/полями

static поля и методы

Статические методы класса имеют следующие особенности:

- статический метод **не может быть вызван** для конкретного объекта класса
- статический метод не имеет *внутреннего состояния*, то есть отсутствует указатель **this** и прямое обращение к **нестатическим полям** класса
- статический метод имеет прямой доступ к статическим полям (внезапно, да)
- статический метод имеет возможность обращаться к закрытым полям/методам **объекта своего класса**, если он был передан в метод как аргумент
- статические методы могут быть как открытыми, так и закрытыми
- синтаксис вызова статического метода аналогичен обращению к открытому статическому полю

static поля и методы

Пример на статические методы (тестовый, разбираемся самостоятельно)

```
1  class TestStatic3
2  {
3  public:
4    static void meth1(int val)
5    { st_field = val; }
6
7    static int meth2()
8    { return st_field; }
9
10   static void meth3(TestStatic3 obj)
11   { cout << obj.field; }
12
13 private:
14   int field;
15   static int st_field;
16 };
17 // ....
18 TestStatic3::meth1(105);
19 cout << TestStatic3::meth2();
20 TestStatic3 ts31;
21 TestStatic3::meth3( ts31 );
```

static поля и методы

Что ещё интересного: подсчёт количества объектов класса

```
1  #include <iostream>
2  #include <clocale>
3
4  class TestStatic4
5  {
6  public:
7    TestStatic4() { ++obj_count; }
8    ~TestStatic4() { --obj_count; }
9
10   static unsigned get_objects_count()
11   { return obj_count; }
12
13 private:
14   static unsigned obj_count;
15 };
16
17 unsigned TestStatic4::obj_count = 0;
18
19 int main()
20 {
21   std::setlocale(LC_ALL, "RUS");
22   TestStatic4 ts41, ts42, ts43, ts44;
23   cout << "Сейчас существуют " << TestStatic4::get_objects_count()
24        << " объектов класса TestStatic4\n";
25 }
```

Конечно, подсчёт имеет больше смысла, когда объекты создаются динамически. О других применениях статических методов - спрашивайте индивидуально.

Динамическое создание объектов класса

Пока все примеры с классами (да и со структурами ранее) примечательны тем, что всегда использовались обычные переменные. Никаких динамически выделенных массивов объектов, оператор **new** совсем забыт. Будем исправлять, начнём со структур (в смысле языка C)

```
1  // Простая структура
2  struct MaterialPoint
3  {
4    int x, y, z;
5    double mass;
6  };
7
8  // Выделение память под одну переменную структуры
9  MaterialPoint *p_mp1 = new MaterialPoint;
10 // Обращение к полям через указатель на структуру
11 p_mp1->y = p_mp1->y = p_mp1->y = 10;
12 // Можно и так, но дольше набирать
13 (*p_mp1).mass = 5.5;
14
15 // Возрат памяти в ОС
16 delete p_mp;
```

Динамическое создание объектов класса

Обратить внимание стоит только на то, как через указатель получать поля с помощью -> (впрочем, уже было). Оператор **new** для структуры действует просто выделяет блок памяти подо все её поля. И в выделенный объект становится возможным запись нужных значений. Динамический массив структур создаётся аналогично простым типам данных

```
1  // Простая структура
2  struct MaterialPoint;
3
4  // Выделение память под массив структур
5  MaterialPoint *mp_arr = new MaterialPoint[10];
6
7  // Какое-то использование созданного массива
8  for (size_t i = 0; i < 10; ++i) {
9    mp_arr[i].x = mp_arr[i].y =  mp_arr[i].z = i * i;
10 }
11
12 // Возрат используемой памяти в ОС
13 delete[] mp_arr;
```

Динамическое создание объектов класса

Освежив информацию про структуры, переходим к классам. А для них оператор **new** всегда делает два действия:

1. Выделяет память под все поля класса
2. Вызывает конструктор

Второй пункт очень важен - при динамическом создании **одного объекта** класса **new** обязан вызвать какой-нибудь конструктор. Вариантов два: либо вызывается конструктор без параметров, либо мы явно задаём нужный.

```
1  class TestNew1
2  {
3  private:
4    int val;
5  };
6
7  // Тут new выделает память под поле val
8  // и вызывает конструктор по умолчанию,
9  // который ничего не делает
10 TestNew1 *p_tn1 = new TestNew1;
11 // Создать массив объектов — нет проблем
12 TestNew1 *tn_arr1 = new TestNew1[88];
```

Динамическое создание объектов класса

Добавим один конструктор с одним параметром.

```
1  class TestNew1
2  {
3  public:
4    TestNew1(int value) : val{value}
5    {}
6
7  private:
8    int val;
9  };
10
11 /*
12   Теперь конструктор без параметров недоступен,
13   так что при вызове new следует явно передать
14   параметр в конструктор. В данном случае —
15   число 15. А вот создать массив объектов — уже проблема.
16   Для простого динамического массива объектов нет никакого
17   способа передать аргумент в конструктор каждого объекта.
18 */
19 TestNew1 *p_tn2 = new TestNew1(15);
20 // Ниже — будет ошибка компиляции
21 // TestNew1 *tn_arr2 = new TestNew1[12];
```

Динамическое создание объектов класса

Если есть несколько конструкторов - можно выбирать

```
1  class TestNew1
2  {
3  public:
4    TestNew1(int value) : val{value}
5    {}
6
7    TestNew1() : val{15}
8    {}
9
10 private:
11   int val;
12 };
13
14 /*
15   В первом случае явно вызываем конструктор с одни параметром,
16   во втором — конструктор без параметров
17 */
18 TestNew1 *p_tn3 = new TestNew1(5),
19          *p_tn4 = new TestNew1;
```

Динамическое создание объектов класса

И финальный шаг - оператор **delete** для указателя-на-объект **всегда** вызывает **деструктор класса**.

```
1  class TestNew1
2  {
3  public:
4    TestNew1(int value) : val{value}
5    {}
6
7    TestNew1() : val{15}
8    {}
9
10   ~TestNew1()
11   { std::cout << "Деструктор был вызван\n"; }
12
13 private:
14   int val;
15 };
16
17 TestNew1 *p_tn5 = new TestNew1(55);
18 // В момент действия *delete* будет напечатана строка из ↩
       деструктора
19 delete p_tn5;
```

Динамическое создание объектов класса

Резюме по работе с динамическими объектами:

- при использовании оператора **new** для класса **всегда** происходит вызов конструктора;
- если конструктор был перегружен - при создании динамического объекта можно выбирать любую версию;
- если нужен динамический массив объектов класса - проще и логичнее использовать класс **vector** из стандартной библиотеки;
- при использовании оператора **new** для объекта класса всегда происходит вызов деструктора.

```
1 // Пример с вектором
2 class TestNew1
3 { // ... };
4
5 vector<TestNew1> tn_vec;
6 // Добавляем в массив объект, созданный конструктором без параметров
7 tn_vec.push( TestNew1() );
8 // Добавляем в массив объект, созданный конструктором с параметром
9 tn_vec.push( TestNew1(8) );
```

const-методы

Методы класса не всегда меняют поля объекта. В разных случаях они могут служить для вычисления какой-либо величины, основываясь на полях класса; на формирование некоторых строк для вывода информации, просто для возращения значения некоторого поля. Такие методы по отношению к объекту, для которого они вызываются, являются неизменяемыми (постоянными) в том смысле, что не меняют его внутреннее состояние. Это не строгое, скорее логическое определение. Современные компиляторы C++ позволяет отметить такие методы с помощью ключевого слова **const**, после чего компилятор начнёт следить, чтобы при изменении кода метода никакие поля объекта не менялись. **const** помещается после **списка аргументов** метода. Получается бесплатный надзор за желанием гарантировать отсутствие изменений полей в некотором методе.

const-методы

Делается это следующим образом

```
1  class TestConstMeth
2  {
3  public:
4    void set_f1(int value)
5    { f1 = value; }
6
7    void set_f2(int value)
8    { f2 = value; }
9
10   std::string as_string() const
11   {
12     std::string str = "f1 = ";
13     str += to_string(f1);
14     str += "; f2 = ";
15     str += to_string(f2);
16     return str;
17   }
18
19 private:
20   int f1, f2;
21 };
```

Технически, внутри метода **as_string** считается, что поля **f1** и **f2** - постоянны. Если в нём попробывать изменить значения одного из полей (например - поставить «++**f1**» вместо «**f1**») - будет ошибка компиляции.