Лекция VII

6 декабря 2017

Потоковый ввод/вывод: файлы

Функции, для проверки состояния потока ввода-вывода

- Достигнут ли конец файла?

```
int feof(FILE *stream);
```

- Произошли ли какие-либо ошибки при операциях ввода-вывода?

```
int ferror(FILE *stream);
```

Обе функции возращают **нуль**, если ответ на соответствующий вопрос отрицательный, и **не нулевое целое число** - если утвердительный.

Потоковый ввод/вывод: файлы

Задача: есть текстовый файл со случайными действительными числами. Нужно посчитать статистику, например - среднее и медиану

```
3.43434 53.5 43.546 4 6.776 0.86
1.546E2 8.986E-5 5.55 3.01
2.32 6.777
```

Для начала - выведем все числа из файла на экран.
На словах алгоритм действий таков: загружаем последовательно по одному действительному числу и печатаем с помощью **printf**

Потоковый ввод/вывод: файлы

С учётом знаний об **feof** - реализация в лоб выглядит как

```
1  FILE *f_stat = fopen("source.dat", "r");
2
3  if (f_stat != NULL) {
4    double cur_num;
5
6    while ( !feof(f_stat) ) {
7      fscanf(f_stat, "%lf", &cur_num);
8      printf("Текущее число: %f\n", cur_num);
9    }
10 }
```

А вывод будет таков:

```
Текущее число: 3.434340
Текущее число: 53.500000
Текущее число: 43.546000
...
Текущее число: 2.320000
Текущее число: 6.777000
Текущее число: 6.777000
```

Потоковый ввод/вывод: файлы

А вывод будет таков:

```
Текущее число: 3.434340
Текущее число: 53.500000
Текущее число: 43.546000
...
Текущее число: 2.320000
Текущее число: 6.777000
Текущее число: 6.777000
```

Притом, что исходный файл:

```
3.43434 53.5 43.546 4 6.276 0.86
1.546E2 8.986E-5 5.55 3.01
2.32 6.777
```

Версия без изъянов: нужно быть уверенным в том, что **fscanf** завершилась без ошибок

```
1  FILE *f_stat = fopen("source.dat", "r");
2
3  if (f_stat != NULL) {
4    double cur_num;
5
6    while ( !feof(f_stat) ) {
7      if ( fscanf(f_stat, "%lf", &cur_num) == 1 ) {
8        printf("Текущее число: %f\n", cur_num);
9      }
10   }
11 }
```

Будет введено ровно то количество чисел, что записаны в файле.

Потоковый ввод/вывод: файлы

Исходная задача: среднее + медиана

```
1  double *nums = NULL, sum = 0.0;
2  size_t nums_count = 0;
3
4  FILE *f_stat = fopen("source.dat", "r");
5  if (f_stat != NULL) {
6    double cur_num; size_t cur_index = 0;
7
8    while ( !feof(f_stat) ) {
9      if ( fscanf(f_stat, "%lf", &cur_num) == 1 ) {
10       nums_count = cur_index + 1;
11       nums = (double *) realloc(nums,
12             nums_count * sizeof(double));
13
14       if (nums == NULL) { break; }
15       nums[cur_index++] = cur_num;
16       sum += cur_num;
17     }
18   }
19   fclose(f_stat);
20 } // Продолжение на следующем слайде...
```

Потоковый ввод/вывод: файлы

Исходная задача: среднее + медиана

```
21 // Начало на предыдущем слайде...
22
23 if (nums != NULL) {
24   double average = sum / nums_count;
25   // compare_reals — функция сравнения из 5-ой ↩
       лекции
26   qsort(nums, nums_count, sizeof(double), ↩
       compare_reals);
27   double median = nums[nums_count/2];
28
29   printf("Среднее: %f\nМедиана: %f\n",
30                              average, median);
31   free(nums);
32 }
```

Потоковый ввод/вывод: файлы

Исходная задача: среднее + медиана

```
21 // Начало на предыдущем слайде...
22
23 if (nums != NULL) {
24   double average = sum / nums_count;
25   // compare_reals — функция сравнения из 5-ой ↩
       лекции
26   qsort(nums, nums_count, sizeof(double), ↩
       compare_reals);
27   double median = nums[nums_count/2];
28
29   printf("Среднее: %f\nМедиана: %f\n",
30                              average, median);
31   free(nums);
32 }
```

Для исходного файла с числами, вывод:

```
Среднее: 23.656119
Медиана: 5.550000
```

Потоковый ввод/вывод: файлы

Задача: дан текстовый файл, запомнить все числа из него

```
В работе [60] проведено численное исследование
равновесной динамики неупорядоченной модели
Изинга с некоррелированными дефектами
для спиновых концентраций p = +0.8, +0.85 и
+0.65. При использовании конечномерного
скейлингового анализа для решеток с
12 < L < 64 получено значение критического
показателя z = 2.35(2) для системы с
концентрацией спинов p = 1 - 0.2. Спадание
корреяционной функции происходило с
показателем -1.578.
```

Потоковый ввод/вывод: файлы

Задача: дан текстовый файл, запомнить все числа из него

В дополнении к функции **ferror**, понадобятся ещё две:

- Вернуть уже прочитанный символ в поток:

```
int ungetc(int character, FILE *stream);
```

- Убрать индикатор конца файла и индикатор ошибки операций ввода/вывода:

```
void clearerr(FILE *stream );
```

- Проверить, является ли переданный символ - цифрой:

```
#include <ctype.h>

int isdigit(int character);
```

и посимвольное чтение файла, для начала.

Потоковый ввод/вывод: файлы

Задача: дан текстовый файл, запомнить все числа из него

```
1  FILE *source = fopen("text.txt", "r");
2
3  if (source == NULL) {
4    perror("Ошибка открытия файла");
5    exit(1);
6  }
7
8  double *reals = NULL;
9  size_t reals_cnt = 0, cur_index = 0;
10 char symb;
11
12 while ( (symb = fgetc(source)) != EOF ) {
13   if (isdigit(symb) || symb == '+'
14                     || symb == '-') {
15     ungetc(symb, source);
16     // Продолжение на следущем слайде
```

Потоковый ввод/вывод: файлы

Задача: дан текстовый файл, запомнить все числа из него

```
21      // Начало на предыдущем слайде
22      double tmp;
23      if (fscanf(source, "%lf", &tmp) == 1) {
24        reals_cnt = cur_index + 1;
25        reals = (double *) realloc(reals,
26                  reals_cnt * sizeof(double));
27
28        if (reals == NULL) { break; }
29
30        reals[cur_index++] = tmp;
31      } else {
32        clearerr(source);
33        fgetc(source);
34      }
35    } // if (isdigit(...) ... )
36  } // while
37
38  // Продолжение далее
```

Потоковый ввод/вывод: файлы

Задача: дан текстовый файл, запомнить все числа из него

```
39 // Начало на предыдущем слайде
40
41 fclose(source);
42
43 if (reals != NULL) {
44   printf("Найденные числа: ");
45   for (size_t i = 0; i < reals_cnt; ++i) {
46     printf("%.5f ", reals[i]);
47   }
48 }
```

И вывод:

```
Найденные числа: 60.000 0.800 0.850 0.650 12.000
64.000 2.350 2.000 1.000 0.200 -1.578
```

Потоковый ввод/вывод: файлы

Неформатированный ввод/вывод - предназначен для записи/чтения строго определённого количества байт. Стандартная библиотека предоставляет следующие функции:

- Записать байты в файл

```
size_t fwrite(const void *ptr, size_t elem_sz,
              size_t elems_count, FILE *stream);
```

- Прочитать байты из файла:

```
size_t fread(void *ptr, size_t elem_sz,
             size_t elems_count, FILE *stream);
```

, где **ptr** - указатель на начало блока памяти (либо куда записываются байты, либо откуда берутся для вывода в файл); **elem_sz** - размер (в байтах) одного элемента для ввода/вывода, **elems_count** - общее количество элементов, **stream** - поток ввода/вывода.
Возращаемое значение: число, которое равно **elems_count** в случае успеха и не равно, иначе.

Неформатированный ввод/вывод

Задача: одной программой записать в файл заданное количество массивов целых чисел из 10 элементов. Второй программой - определить количество записанных массивов (сколько штук) и загрузить один из них по выбору

Потоковый ввод/вывод: файлы

Неформатированный ввод/вывод: запись 10-элементных массивов в файл. Здесь то двоичный режим и пригодится.

```
1  const size_t SZ = 10;
2  FILE *out_stream = fopen("arrays.bin", "wb");
3
4  if (out_stream != NULL) {
5    int arr[SZ]; size_t how_many;
6
7    printf("Введите количество массивов: ");
8    scanf("%lu", &how_many);
9
10   for (size_t att = 1; att <= how_many; ++att) {
11     for (size_t i = 0; i < SZ; ++i) {arr[i] = rand();}
12
13     size_t recorded = fwrite(arr, sizeof(int), SZ, ↩
           out_stream);
14     if (recorded != SZ) { break; }
15   }
16   fclose(out_stream);
17 }
```

Потоковый ввод/вывод: файлы

Неформатированный ввод/вывод: чтение 10-элементных массивов из файла.
Каждый экземпляр структуры **FILE** имеет поле, сохраняющее **позицию** в файле: на каком байте от начала файла находится связанный с ним поток (смещение происходит в результате операций ввода/вывода).

- Узнать текущую позицию потока

  ```
  long ftell(FILE *stream);
  ```

  В случае ошибки - функция вернёт значение **-1L**.
- Изменить позицию потока

  ```
  int fseek(FILE *stream, long offset, int from);
  ```

  offset - отступ от некоторой позиции в файле, заданной аргументом **from**. В качестве которого используются три константы: **SEEK_SET** (начало файла), **SEEK_CUR** (текущая позиция), **SEEK_END** (конец файла).

Неформатированный ввод/вывод: чтение 10-элементных массивов из файла.

Что нужно для второй программы?

1. Узнать количество массивов в файле
2. Запросить номер загружаемого массива
3. Считать нужный массив из файла

Неформатированный ввод/вывод: чтение 10-элементных массивов из файла.

```
1  const size_t SZ = 10, ARR_BYTES = sizeof(int) * SZ;
2  FILE *in_stream = fopen("arrays.bin", "rb");
3
4  if (in_stream != NULL) {
5    fseek(in_stream, 0, SEEK_END); // Шаг (1) начат
6    long how_many = ftell(in_stream) / ARR_BYTES;
7    if (how_many < 1) {
8      perror("Нет массивов в файле"); exit(1);
9    }
10   fseek(in_stream, 0, SEEK_SET); // Шаг (1) выполнен
11
12   int arr_num = 0;  // Шаг (2) начат
13   do {
14     printf("Введите номер (всего - %ld): ", how_many);
15     scanf("%d", &arr_num);
16   } while (arr_num < 1 || arr_num > how_many);
17                     // Шаг (2) выполнен
18   // Продолжение — ниже
```

Неформатированный ввод/вывод: чтение 10-элементных массивов из файла.

```
19    // Начало — выше
20
21    arr_num--; // Для вычисления смещения. Шаг (3) начат
22    int arr[SZ];
23    fseek(in_stream, arr_num * ARR_BYTES, SEEK_SET);
24    size_t read = fread(arr, sizeof(int), SZ, in_stream);
25    fclose(in_stream); // Шаг (3) выполнен
26
27    if (read != SZ) {
28      perror("Количество элементов меньше 10");
29      exit(1);
30    }
31
32    printf("Прочитанный массив:\n  ");
33    for (size_t i = 0; i < SZ; ++i) {
34      printf("%d ", arr[i]);
35    }
36  }
```

Потоковый ввод/вывод: файлы

Ещё примеры на неформатированный ввод/вывод и **ftell/fseek**

https://github.com/posgen/OmsuMaterials/blob/master/2course/Programming/examples/8_file_operations_example/rewrite_example.c

https://github.com/posgen/OmsuMaterials/blob/master/2course/Programming/examples/8_file_operations_example/save_and_get_structs.c

Дополнительные сведения про функции

Указатели на функции

В C существуют **указатели на функцию** - указатели специального типа, позволяющие использовать функции языка как переменные. Их основные характеристики:

- позволяют передавать функции как аргументы в другие функции;
- позволяют объявлять массивы функций, одинаковых по типу возвращаемого значения и со совпадающим списком аргументов;
- позволяют делать отложенный вызов функций;
- не требуют разыменования;
- не требуют явного присвоения адреса существующей функции.

Общий синтаксис:

```
<тип_возвращаемого_значения>
        (*<название_указателя>)(<типы аргументов>);
```

Указатели на функции

Задача: вычислять производную функции одного аргумента в заданной точке

```
1  double fun(double x)
2  {
3    return x * sin(x) - exp(x) * cos(2 * x);
4  }
5
6  double derivative(double pt)
7  {
8    double dh = 5E-4;
9    double diff = fun(pt - dh) - fun(pt + dh);
10   return diff / (2.0 * dh);
11 }
12 ...
13 derivative(4.5);
14 derivative(11);
```

Что не так: писать вычисление производной для каждой нужной функции - утомительно.

Указатели на функции

Задача: вычислять производную функции одного аргумента в заданной точке Модернизируем функцию **derivative**

```
1  double derivative(double pt,
2                    double (*pfn)(double))
3  {
4    double dh = 5E-4;
5    double diff = pfn(pt - dh) - pfn(pt + dh);
6    return diff / (2.0 * dh);
7  }
8  ...
9  derivative(4.5, fun);
10 derivative(11 * M_PI, sin);
11
12 double (*fn_ptr)(double);
13 fn_ptr = exp;
14 derivative(4, fn_ptr);
```

Функции без аргументов

Вопрос: сколько аргументов можно передать в следующую функцию?

```
1 double magic_number()
2 {
3   return -3.2828282828;
4 }
```

Функции без аргументов

Вопрос: сколько аргументов можно передать в следующую функцию?

```
1 double magic_number()
2 {
3   return -3.2828282828;
4 }
```

Да сколько угодно!

```
5 ...
6 magic_number("a string", 4, 5.678, '$');
```

Функции без аргументов

Если функция не принимает аргументов, то вместо их списка должно стоять ключевое слово **void**. Это правило верно в том числе и для функции **main**.

```
1  double magic_number(void)
2  {
3    return -3.2828282828;
4  }
5
6  ...
7  // Так нельзя теперь:
8  // magic_number(4, 5.664, '#');
9
10 // Так — всё правильно:
11 double val = magic_number();
```

Работа со временем, или контролируй каждую секунду.

Работа со временем

Для базовых операций используется следующая библиотека:

```
1 #include <time.h>
```

Работа со временем

Первая операция: подсчёт времени выполнения участка кода. Для этого <**time.h**> предоставляет:

- тип данных **clock_t** представляющий собой **целочисленный** тип данных (зависящий от ОС);
- функцию **clock()**

```
1 clock_t clock();
```

, которая не принимает никаких аргументов и возращает **количество тактов**, прошедших с начала выполнения программы;

- ОС-зависимую константу **CLOCKS_PER_SEC** - хранящую количество тактов, выполняющихся за секунду (говоря языком физики, размерность этой константы есть - число_тактов / секунду).

Работа со временем

Первая операция: подсчёт времени выполнения участка кода.

```
1  unsigned count_of_primes(unsigned n)
2  {
3    unsigned count = 0;
4    for (unsigned i = 2; i <= n; ++i) {
5      for (unsigned j = sqrt(i); j > 1; --j) {
6        if ( (i % j) == 0 ) { break; }
7      }
8      ++count;
9    }
10   return count;
11 }
12
13 printf("Вычисляем...\n");
14 clock_t cl_val = clock();
15 printf("Количество простых чисел "
16         "меньше 100000 равно %u\n", count_of_primes(100000));
17
18 cl_val = clock() - cl_val;
19 double spens_secs = double(cl_val) / CLOCKS_PER_SEC;
20
21 printf("Вычисление произведено за %lu "
22        " тактов (%f секунд)", cl_val, spens_secs);
```

Работа со временем I

Вторая операция: отображение времени в понятном виде. Для этого <**time.h**> предоставляет:

- тип данных **time_t** представляющий собой **целочисленный** тип данных (зависящий от ОС);
- структуру **struct tm**

```
1  struct tm {
2    int tm_sec,   // секунды,      0–59
3    int tm_min,   // минуты,       0–59
4    int tm_hour,  // часы,         0–23
5    int tm_mday,  // день месяца,  1–31
6    int tm_mon,   // месяц,        0–11
7    int tm_year,  // год,       от 1900
8    int tm_wday,  // день недели   0–6
9    int tm_yday,  // день с начала года 0–365
10   int tm_isdst, // спец. флаг
11 };
```

Работа со временем II

, которая используется в функциях для представления времени (не должна использоваться напрямую);

- функцию **time**

```
1 time_t time(time_t *timer);
```

, интересную тем, что она и возвращает, и присваивает указателю **timer** одно и тоже значение: число секунд, прошедших с некоторого рубежа (на данный момент - "00:00:00 1 января 1970");

- две функции - **gmtime** и **localtime**

```
1 struct tm* gmtime(const time_t *timer);
2
3 struct tm* localtime(const time_t *timer);
```

Работа со временем III

, преобразующих время **timer** в объект структуры **struct tm** и возращающих указатель-на-структуру. Различаются тем, что первая их них считает текущим часовым поясом UTC, а вторая - берёт локальный часовой пояс согласно настройкам ОС;

- функцию **mktime**

```
1 time_t mktime(tm *tm_ptr);
```

, осуществляющею обратное преобразование: из объекта структуры **struct tm** в число секунд с укащанного выше рубежа;

- функцию **difftime**

```
1 double difftime(time_t end, time_t start);
```

, которая возращает число секунд между моментами **end** и **start**;

Работа со временем

Отдельно стоит рассмотреть функцию **strftime**. Она позволяет представить время в человеко-читаемом формате.

```
1 size_t strftime(char *str_to_save, size_t max_size,
2                 const char *format, const struct tm *tm_ptr);
```

, где

- **str_to_save** - строка, в которую будет записан результат;
- **max_size** - максимальное количество символов, которое попадёт в **str_to_save**;
- **format** - форматная строка, задающая правила преобразования времени в строковое представление через спецификаторы;
- **tm_ptr** - указатель-на-объект структуры **struct tm**, представляющий собой момент времени для отображения.

Работа со временем

Функция **strftime**. Специальные **спецификаторы** для преобразования времени (указываются в строке **format**)

	Что означает?	Пример значений
%a	Аббревиатура дня недели	Sun, Mon, Thu *
%A	Полное название дня недели	Sunday, Monday *
%b	Аббревиатура месяца	Aug, Jun, Jul *
%B	Название месяца	June, Jule, March *
%d	Номер дня в месяце	01-31
%H	Час в 24-формате	00-23
%I	Час в 12-формате	01-12
%M	Минута	00-59
%S	Секунда	00-59
%p	Знак 12- или 24-часового формата	AM, PM
%m	Номер месяца	01-12
%W	Номер недели (с понедельника)	00-53
%y	Год, две последние цифры	00-99
%Y	Год	2017
%z	Смещение часовой зоны	+0600
%Z	Аббревиатура временной зоны	UTC *

* в третьем столбке означает зависимость от настроек локали в ОС

Функции из <ctime>

Вторая операция: отображение времени в понятном виде.

Примеры

```
1  time_t now = time( nullptr );
2  tm *tm_ptr = localtime( &now );
3  char buf[70];
4
5  strftime(buf, 70, "%I:%M:%S %p", tm_ptr);
6  printf("12-часовое представление: %s\n", buf);
7
8  strftime(buf, 70, "%H:%M:%S", tm_ptr);
9  printf("24-часовое представление: %s\n", buf);
10
11 strftime(buf, 70, "%d/%m/%y", tm_ptr);
12 printf("ДД/ММ/ГГ: %s\n", buf);
13
14 strftime(buf, 70, "%d/%m/%Y", tm_ptr);
15 printf("ДД/ММ/ГГГГ: %s\n", buf);
16
17 strftime(buf, 70, "Дата: %d/%m/%Y, %a %b", tm_ptr);
18 printf("%s\n, "buf);
```

Препроцессор в C

Сборка программ

Схематично, создание исполняемого или библиотечного файла состоит из трёх шагов, выполняемых компилятором:

1. **Препроцессинг**: обработка исходного текста программы с получением раскрытием специальных "команд"
2. **Компиляция**: преобразование расширенного исходного файла(-ов) в объектный(-ые), содержащий представление на языке *ассемблера* (создание объектного файла)
3. **Связывание** (linking): преобразование объектного файла программы в двоичный файл (исполняемый или библиотечный) для данной операционной системы

Директивы препроцессора

Директивы, использующиеся для замены одного текста другим (определение макросов):

```
(1) #define <идентификатор>
(2) #define <идентификатор> [текст_для_замены]
(3) #define <идентификатор>(<параметры>) <текст>
(4) #undef  <идентификатор>
```

1. Определяет **идентификатор** для пустого макроса
2. Определяет макрос замены **идентификатора** на **текст_для_замены**
3. **Идентификатор** может получать параметры и использовать в подстовляемом тексте. Синтаксис параметров аналогичен функциям, за исключением отсутствия каких-либо упоминаний об типах
4. Отменяет любой ранее опредлённый **идентификатор**

Директивы препроцессора

Примеры **макросов**

```
1  #define ROWS 10
2  #define COLS 15
3
4  #define AUTHOR "Это я"
5
6  #define MAX(x, y) (x > y) ? x : y
7  ...
8
9  double matrix[ROWS][COLS];
10 /* После работы препроцессора, в исходном
11    файле появляется строка:
12      double matrix[10][15];
13 */
14
15 printf(AUTHOR);
16 // printf("Это я");
17
18 int val = MAX(15, -8);
19 // int val = (15 < -8) ? 15 : -8;
```

Примеры **макросов**

```
1  #define FUNCTION(name, a) int fun_##name() { return a;}
2
3  FUNCTION(first, 12)
4  FUNCTION(second, 2)
5  FUNCTION(third, 23)
6
7  #undef FUNCTION
8  #define FUNCTION 34
9  #define OUTPUT(a) printf(#a "\n");
10
11
12 printf("first: %d\n", fun_first());
13 printf("first: %d\n", fun_second());
14 printf("first: %d\n", fun_third());
15
16 printf("Значение FUNCTION: %d\n", FUNCTION);
17
18 OUTPUT(Русский текст без кавычек и переносов!);
```

Директивы препроцессора

Условные директивы, использующиемся для задание логики при препроцессинге:

```
(1) #if <выражение>
(2) #ifdef <выражение>
(3) #ifndef <выражение>
(4) #elif <выражение>
(5) #else
(6) #endif
```

Директивы препроцессора

Пример использования **условных директив**

```
1 #if defined(WINDOWS_H)
2   #error Не буду компилироваться в ОС Windows
3 #endif
```

Директивы препроцессора

Пример использования **условных директив**

```
1  #define MACROS1 2
2
3  #ifdef MACROS1
4    printf("1: определён\n");
5  #else
6    printf("1: не определён\n");
7  #endif
8
9  #ifndef MACROS1
10   printf("2: не определён\n");
11 #elif MACROS1 == 2
12   printf("2: определён\n");
13 #else
14   printf("2: не определён\n");
15 #endif
16
17 #if !defined(DCBA) && (MACROS1 < 2*4-3)
18   printf("3: выражение истинно\n");
19 #endif
```

Директивы препроцессора

Встроенные макросы

```
1 printf(__DATE__ " " __TIME__ "\n");
2 printf(__FILE__ "\n");
```

Директивы препроцессора

Директивы, использующиеся для включения других исходных файлов

```
(1) #include <file_name>
(2) #include "file_name"
```

Вообще говоря - две эквивалентные формы включения стандартных или внешних **библиотек** (файлов, которые предоставляют некоторый набор констант, переменных, функций, структур и т.п. для решения каких-либо задач). Разница только в том, что форма **(2)** сначало ищет указанный файл **filename** в той же директории, что и файл, который хотим скомпилировать. Если не найден - делается попытка поиска в *стандартных путях поиска*. Форма **(1)** - производит поиск только в стандартных путях.
Стандартные пути поиска библиотек зависят от способа, как компилятор языка был установлен в ОС, а также могут быть добавлены с помощью дополнительных опций компилятора.

Про один способ передачи кофигурационных параметров в программу

Функция main и её аргументы

Полезным механизмом передачи параметров в программу является способ, 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 и её аргументы

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 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 и её аргументы

Язык 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 современный вариант этой функции есть:

```
1 int main(void)
2 {}
```

Функция main и её аргументы

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, где каждый элемент массива содержит **значение переданного аргумента в строковом виде**. Названия параметров **main** может быть любым, но по негласным стандартам используются два: **argc** (arguments count) и **argv** (arguments values), соответственно.

```
1 int main(int argc, char *argv[])
2 {}
```

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

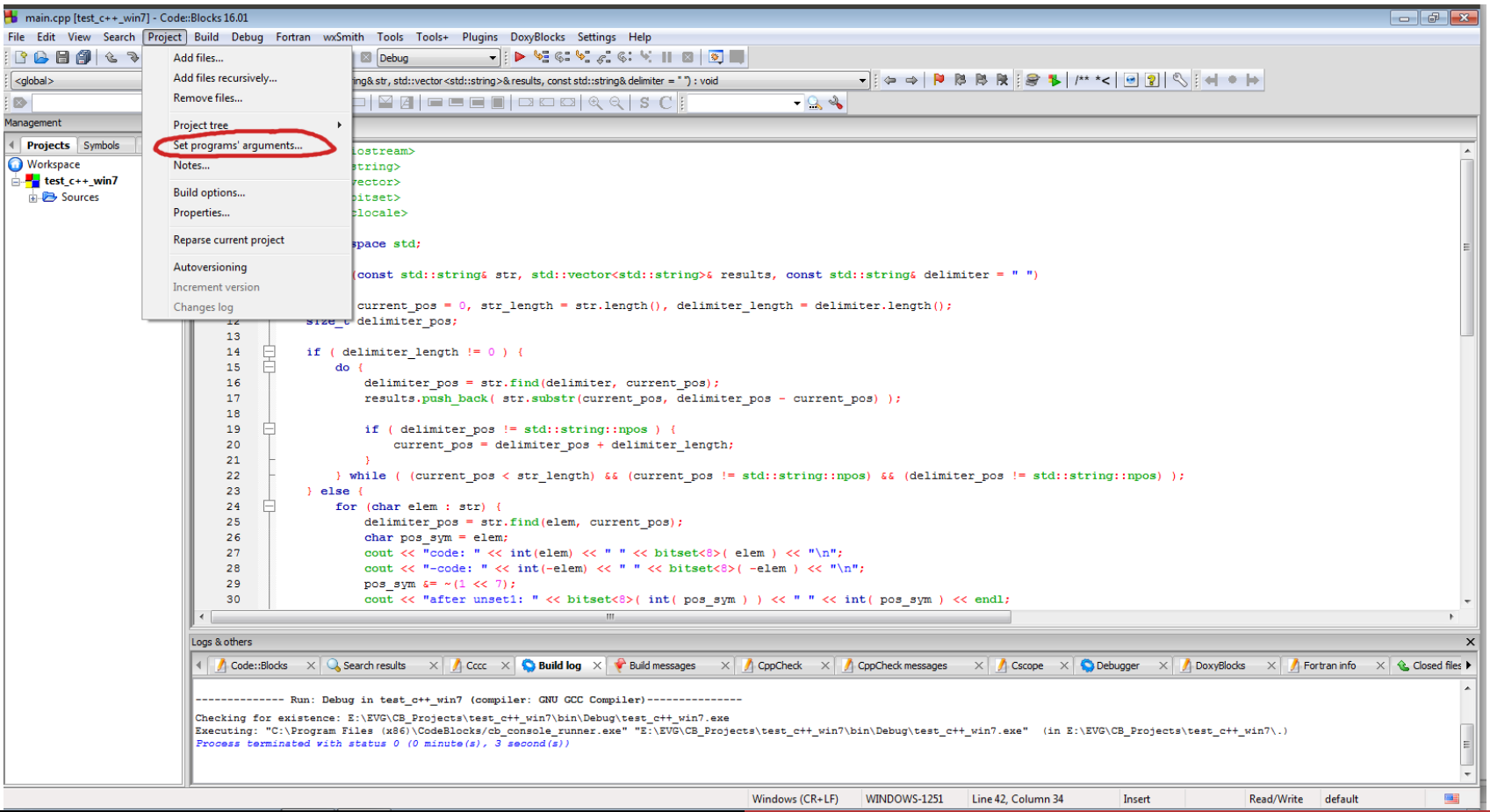
Функция main и её аргументы

На пятидесятом слайде было название программы и три аргумента командной строки. Первая особенность их передачи заключается в том, что **полный путь к 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 <stdio.h>
2
3 int main(int argc, char *argv[])
4 {
5   for (int i = 0; i < argc; i++) {
6     printf("Аргумент номер %d"
7            " содержит значение %s\n", i + 1, argv[i]);
8   }
9 }
```

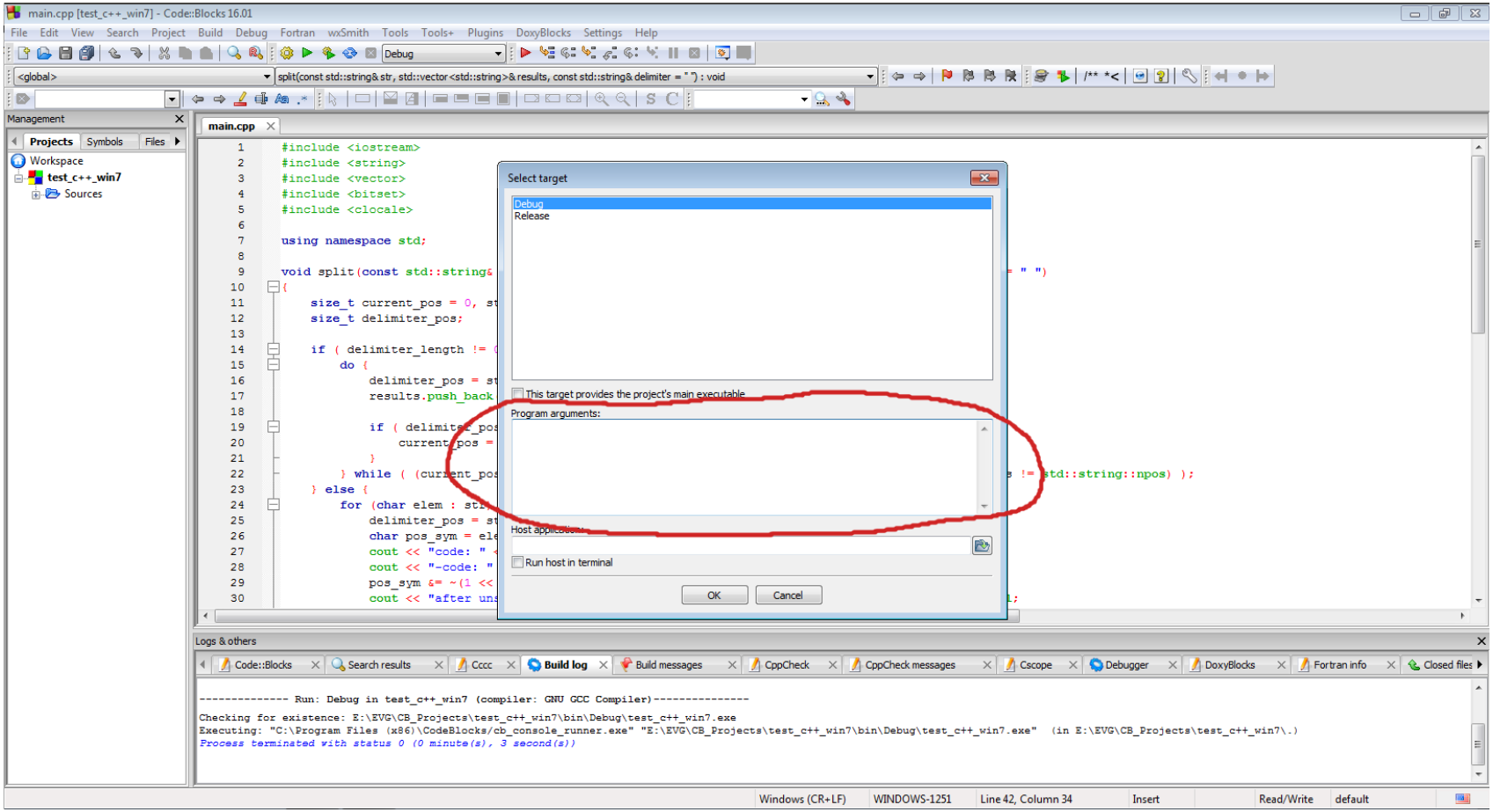
Функция main

Проверить пример выше можно 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 program's aruments"**

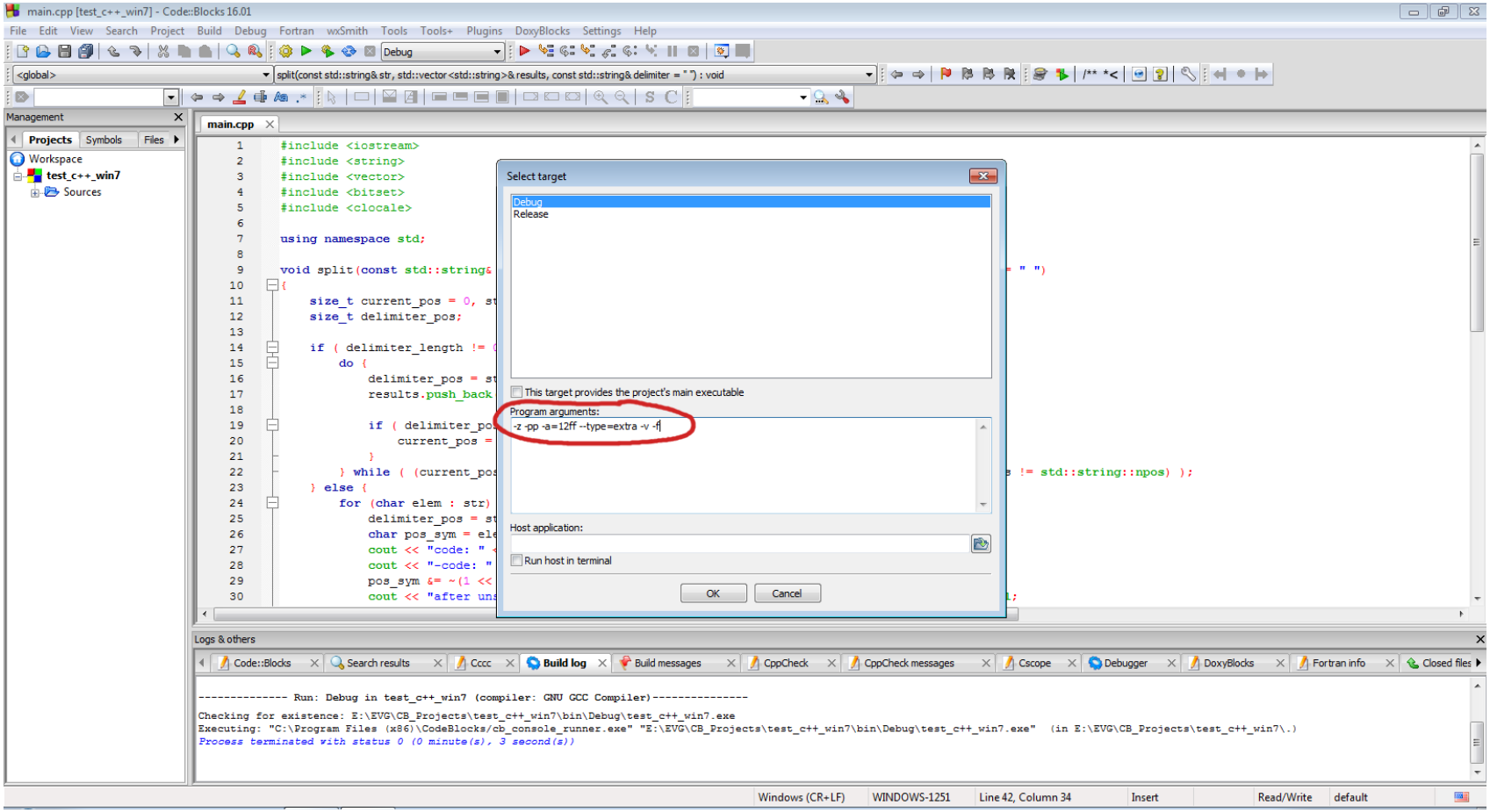


Функция main

Далее найти текстовое поле **Program arguments**

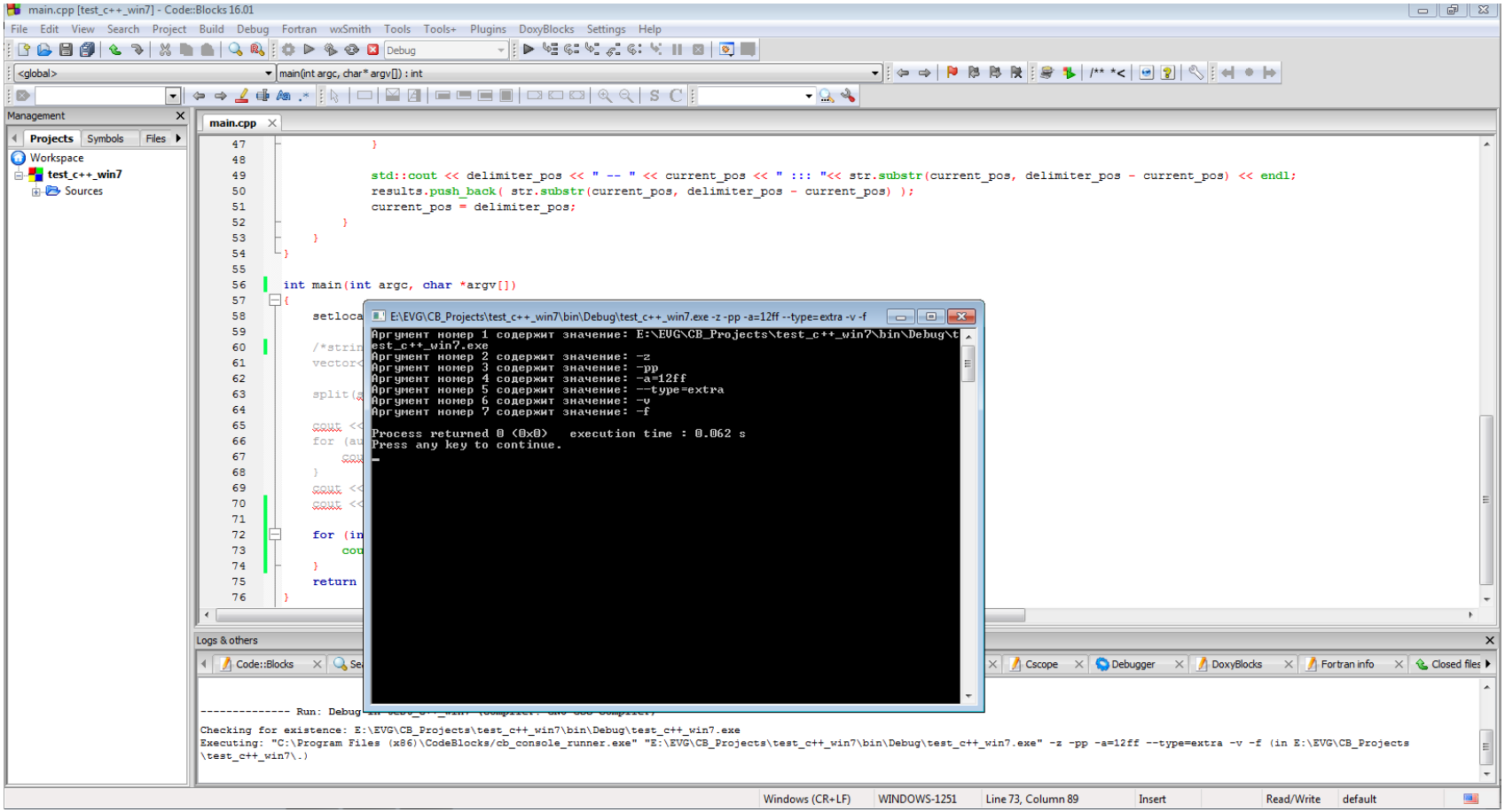


Ввести что-нибудь туда



Функция main

И проверить работоспособность



Функция main

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

На следующих слайдах **приведён код** для решения такой задачи: 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 <math.h>     // sin, cos
2  #include <stdio.h>    // printf, scanf
3  #include <stdlib.h>   // atoi, atof
4  #include <string.h>   // strncmp, strlen
5
6  // Перечисления для проверки успешности разбора
7  // аргументов командной строки
8  typedef enum
9  {
10   PARSE_OK, PARSE_ERROR
11 } ParseStatus;
```

Функция main

Теперь - вспомогательные функции

```
12
13 // Вспомогательные функции: x, x^2, x^3
14 double fn_x(double x)
15 {
16   return x;
17 }
18
19 double fn_x2(double x)
20 {
21   return x * x;
22 }
23
24 double fn_x3(double x)
25 {
26   return x * x * x;
27 }
```

Функция разбора аргументов командной строки. Часть 1

```
28 // Функция разбора вернёт указатель
29 // на интегрируемую функцию
30 double (*)(double) 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 названия функции и статуса ↩
             разбора
34       char **fn_name, ParseStatus *st)
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   *st = PARSE_ERROR;
46   if (argc < 5) {
47     return NULL;
48   }
```

Функция разбора аргументов командной строки. Часть 2

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

Функция разбора аргументов командной строки. Часть 3

```
67
68    // Третьим параметром требуем точность
69    if ( strncmp(args[3], "--accuracy=", 11) == 0) {
70      eps = atof(args[3] + 11);
71    } else {
72      /*
73        Выше было мало места, чтобы написать,
74        но если хотя бы один параметр не совпал
75        по порядку или по имени — сразу же
76        возращаем нулевой указатель
77      */
78      return NULL;
79    }
```

Функция разбора аргументов командной строки. Часть 4

```
80
81    if ( strncmp(args[4], "--fn=", 5) == 0) {
82      /* В параметр fn_name запишется название функции.
83         Оно будет использовано при выводе рзультата
84         вычисления интеграла. */
85      *fn_name = (char*) calloc(strlen(args[4]) * sizeof(char));
86      strcat(*fn_name, args[4] + 5);
87
88      /* За неименеем механизма более продвинутого
89         выбора, просто сравниваем переданное название
90         с совсеми допустимыми функциями и
91         устанавливаем значения указателя-на-функцию
92         p_fun. Если хоть одно условие *if* совпало -
93         можно выходить из функции с помощью *return* */
94      if (fn_name == "sin") {
95        *st = PARSE_OK;
96        return sin;
97      }
98
99      if (fn_name == "cos") {
100       *st = PARSE_OK;
101       return cos;
102     }
```

Функция разбора аргументов командной строки. Часть 5

```
103
104     if (fn_name == "x") {
105       *st = PARSE_OK;
106       return fn_x;
107     }
108
109     if (fn_name == "x2") {
110       *st = PARSE_OK;
111       return fn_x2;
112     }
113
114     if (fn_name == "x3") {
115       *st = PARSE_OK;
116       return fn_x3;
117     }
118
119     return NULL;
120   } else {
121     return NULL;
122   }
123 }
```

Функция интегрирования (метод прямоугольников)

```
124
125 double integrate_fn(double a, double b, double eps,
126                       double (*fn)(double))
127 {
128   unsigned long splits = 1000;
129   double diff = 1.0 + std::abs(eps), h = (b - a) / splits;
130   double first_sum = 0.0, second_sum = 0.0;
131   if (eps < 0) { eps = abs(eps); }
132
133   for (unsigned i = 0; i < splits - 1; ++i) {
134     first_sum += h * fn(a + (i + 1) * h); }
135
136   while (diff > eps) {
137     splits *= 2; h = (b - a) / splits;
138     for (unsigned i = 1; i <= splits; ++i) {
139       second_sum += h * fn(a + (i + 1) * h); }
140
141     diff = abs(second_sum - first_sum);
142     first_sum = second_sum; second_sum = 0;
143   }
144
145   return first_sum;
146 }
```

Функция main

Всё готово, функция **main**

```
147
148 int main(int argc, char *argv[])
149 {
150   double left_bound, right_bound, eps, (*p_fun)(double);
151   ParseStatus st; char *f_name;
152   p_fun = parse_cmd_args(argc, argv, left_bound,
153                          right_bound, eps, f_name, st);
154
155   if (st == PARSE_OK) {
156     double result = integrate_fn(left_bound, right_bound, eps,↩
          p_fun);
157     printf("Интеграл функции %s на отрезке от "
158            "%f до %f с точностью %f равен %f",
159            f_name, left_bound, right_bound, eps, result);
160   } else {
161     printf("Запуск: %s --left=<число> --right=<число>"
162      " --accuracy=<число> --fn=<имя_функции>\n\n"
163      "  --left  - левая граница интегрирования\n",
164      "  --right - правая граница интегрирования\n",
165      "  --accuracy - точность интегрирования\n",
166      "  <имя_функции>: sin, cos, x, x2 или x3\n", argv[0]);
167   }
168 }
```

Функция main

Вводя в IDE аргументы командной строки вида:

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

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