ФАЙЛЫ И ПРОЧЕЕ

Аргументы командной строки. Файловый ввод и вывод.
Неформатированный ввод. Вариабельные аргументы.

К. Владимиров, Syntacore, 2023
mail-to: konstantin.vladimirov@gmail.com

Поиск в нескольких файлах

- До сих пор мы читали ввод только со stdin.
- Но представим простую проблему: у нас есть несколько файлов на диске с какими-то числами. И нам нужна программа которая считала бы вхождение в них некоего числа.
- Мы хотели бы её вызывать примерно так:

```
$ ./myprog 42 1.txt 2.txt 3.txt
```

- Две основные проблемы:
- Как работать с аргументами командной строки?
- Как работать с большим количеством файлов?

Аргументы функции main

- До сих пор мы пользовались `main()` или `main(void)`.
- Но есть вторая форма основной функции

```
int main(int argc, char **argv) {
  // тело функции
}
```

- `argc` это количество аргументов.
- `argv` это массив `argc+1` строк.
- `argv[0]` это имя самой программы.

Начнём писать myprog

- Договоримся что первый аргумент это искомое число.

```
$ ./myprog 42 1.txt 2.txt 3.txt
```

- Нам надо проверить, что количество аргументов корректно.

```
int main(int argc, char **argv) {
  if (argc < 3) {
    printf("Usage: %s <n> <files>\n", argv[0]);
    return 1;
  }
```

- Далее преобразуем первый аргумент в число.

Преобразование строки в число

- Для преобразования используем функцию strtol из stdlib.

```
int main(int argc, char **argv) {
  int n; char *endptr;
  if (argc < 3) {
    printf("Usage: %s <n> <files>\n", argv[0]);
    return 1;
  }

  n = strtol(argv[1], &endptr, 10);
  if (endptr == argv[1]) {
    printf("<%s> can not be converted to int\n", argv[1]);
    return 1;
  }
```

Дальнейшие действия

```
int main(int argc, char **argv) {
  int n, i, count; char *endptr;

  // проверяем argc

  // обрабатываем argv[1], получаем n = искомое число

  for (i = 2; i < argc; ++i)
    count += find_in_file(argv[i], n);

  printf("%d\n", count);
}
```

- Осталось написать функцию обработки файла: открыть, подсчитать, закрыть.

Минимум о работе с файлами

- Файл (`FILE`) это синоним для implementation defined структуры

```
typedef struct _File_t { /* нечто */ } FILE;
```

- Даже `sizeof(FILE)` не определён. Мы всегда оперируем с `FILE*`
- Открыть файл можно с помощью функции `fopen`

```
FILE *fopen (char const * name, char const *mode);
```

- Базовые режимы открытия: "`r`", "`w`", "`a`" для текстовых файлов означают запись, перезапись и дозапись в конец
- Закрыть файл можно с помощью функции `fclose`

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

Обработка ошибок

- Посмотрим ещё раз на fopen

```
FILE *fopen (char const *name, char const *mode);
```

- Что если файла нет? Что если он есть, но доступ запрещён?
- Понятно, что вернётся NULL, но как определить что именно пошло не так?

Обработка ошибок

- Посмотрим ещё раз на fopen

```
FILE *fopen (char const *name, char const *mode);
```

- В языке C принято обрабатывать ошибки в таких функциях через глобальную переменную `errno`
- Чтобы распечатать сообщение, соответствующее коду errno, используется функция perror, позволяющая частично кастомизировать сообщение

```
f = fopen("myfile", "r");

if (f == NULL) {
  perror("Error opening file"); abort();
}
```

Холостой ход

- Напишем и протестируем функцию которая будет только открывать и закрывать файл.

```
int find_in_file(const char *name, int needle) {
  int count = 0; FILE *f;
  f = fopen(name, "r");
  if (f == NULL) {
    perror("Error opening file");
    abort();
  }
  // TODO
  fclose(f);
  return count;
}
```

Форматный ввод и вывод

- Форматированный ввод/вывод это `fprintf` и `fscanf`

```
int fprintf(FILE * stream, char const * format, ...);
int fscanf(FILE * stream, char const * format, ...);
```

- Вместо первого аргумента можно указывать stdout, stderr или stdin.

```
printf(<smth>) == fprintf(stdout, <smth>)

scanf(<smth>) == fscanf(stdin, <smth>)
```

- Все ошибки принято выводить на stderr!

```
fprintf(stderr, "Usage: %s <n> <files>\n", argv[0]);
```

Считывание чисел и их подсчёт

- Теперь мы можем дописать нашу программу.

```
int find_in_file(const char *name, int needle) {
  // fopen и проверка файлов
  for(;;) {
    int n, ret;
    ret = fscanf(f, "%d", &n);
    if (ret == EOF) break; // дочитали до конца файла
    if (ret != 1) { fclose(f); abort(); }
    if (n == needle) count += 1;
  }
  // fclose
}
```

Поиск внутри файла

- Для навигации внутри файла используются

```
long ftell(FILE *stream); // текущее полложение
int fseek (FILE *stream, long offset, int origin);
```

SEEK_SET	Начало файла
SEEK_CUR	Текущее положение в файле
SEEK_END	Конец файла

Поиск внутри файла

- Для навигации внутри файла используются

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

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

- Пример навигации и перезаписи.

```
FILE *f = fopen("example.txt", "w");
if (!f)
  perror("Error opening file");
fprintf(f, "%s", "This is an apple\n");
fseek(f, 9, SEEK_SET);
fprintf(f, "%s", " sam");
```

Обсуждение

- Сигнатура fprintf вызывает вопросы.

Перевод в строку и конкатенация

- Постановка задачи: нужно собрать строчку из строки, двух чисел и ещё одной строки.

```
// "ab", 42, 1, "cd" → "ab 42 1 cd"
void strange_concat(char *dest, char const *s1,
                    int d1, int d2, char const *s2) {
  // TODO: ваш код здесь
}
```

- Будем считать, что размер dest заведомо достаточен.
- Как бы вы это сделали?

Загадочный sprintf

- Самый простой способ: просто напечатать всё это в строку.

```
// "ab", 42, 1, "cd" → "ab 42 1 cd"
void strange_concat(char *dest, char const *s1,
                    int d1, int d2, char const *s2) {
  sprintf(dst, "%s %d %d %s", s1, d1, d2, s2);
}
```

- Функция sprintf удивительно обобщённая: она позволяет скидывать всё что угодно в строчку и часто пользоваться ей удобнее, чем специфичными.
- Хорошо. Но как написать саму функцию sprintf?
- Даже проще: что самое удивительное в функции sprintf?

Вариабельные функции

- Самое удивительное в функции sprintf то, что она берёт сколько угодно аргументов.
- Давайте сначала попробуем написать функцию, которая брала бы сколько угодно целых чисел и складывала их.

```
int x = sum_all(4, 10, 14, 24, 40); // x == 88
```

- Первый параметр это количество аргументов (иначе откуда его узнать?).
- Кажется её логика попроще.

Вариабельные функции

- Произвольное количество аргументов после троеточия.

```
int sum_all (int n, ...) {
  int res = 0;

  // здесь нужно просуммировать все аргументы

  return res;
}
```

- Здесь три точки это не сокращение на слайде, это легальный синтаксис.
- Остаётся вопрос как всё-таки получить доступ к аргументам?

Функции из stdarg

- Список аргументов создаётся через `va_list`.

```
va_list args;
```

- Аргумент с которого начинаются вариабельные отмечается через `va_start`.

```
va_start(args, n);
```

- Каждый аргумент вынимается через `va_arg` с указанием типа.

```
va_arg(args, int);
```

- В конце всё завершается через `va_end`.

```
va_end(args);
```

Пример: суммирование целых

- Собираем всё вместе: функция суммирует целые числа

```
int sum_all(int n, ...) {
  int res = 0;
  va_list args;
  va_start(args, n);
  for (int i = 0; i < n; ++i)
    res += va_arg(args, int);
  va_end(args);
  return res;
}
```

- Теперь заработает: `x = sum_all(4, 10, 14, 24, 40); // x == 88`

Именно так работают printf и scanf

- Функции printf и scanf объявлены следующим образом

```
int printf(const char *format, ...);
```

```
int scanf(const char *format, ...);
```

- Они тоже принимают произвольное число параметров и используют строку формата чтобы установить типы
- Любая ошибка в типах ведёт к непоправимым последствиям
- И конечно, именно так работает и sprintf
- Но прежде чем мы до него дойдём, ещё одно простое применение

Многоликий printf и scanf

- Основные формы:

```
int printf(const char *format, ...);
int scanf(const char *format, ...);
int fprintf(FILE *f, const char *format, ...);
int fscanf(FILE *f, const char *format, ...);
int sprintf(char *s, const char *format, ...);
int sscanf(char *s, const char *format, ...);
```

Обсуждение

- Можем ли мы имея fprintf написать printf?

```
int printf(const char *format, ...) {
  // как-то вызвать fprintf
}
```

Обсуждение

- Можем ли мы имея fprintf написать printf?

```
int printf(const char *format, ...) {
  // как-то вызвать fprintf
}
```

- Увы, в языке нет способа из функции "пробросить троеточие"
- Можно написать макрос, но мы хотим избежать макросов
- А что если передать va_list?

Волшебство vfprintf

- Теперь и printf и fprintf можно реализовать в терминах новой функции

```
int vfprintf(FILE *f, const char *format, va_list arg);

int fprintf(FILE *f, const char *format, ...) {
  va_list l; int retval;
  va_start(l, format);
  retval = vfprintf(f, format, l);
  va_end(l);
  return retval;
}

int printf(const char *format, ...) // как-то вызвать vfprintf
```

Обсуждение

- Функции, такие как vfprintf и vsprintf очень полезны при написании собственных printf-подобных функций

```
pFile = fopen (szFileName,"r");
if (pFile == NULL)
  PrintFError("Error opening '%s'", szFileName);
```

- Понятно, что здесь PrintFError должна как-то вызвать внутри perror, но как её можно реализовать?

Обсуждение

- Функции, такие как vfprintf и vsprintf очень полезны при написании собственных printf-подобных функций

```
void PrintFError(const char * format, ...) {
  char buffer[256];
  va_list args;
  va_start(args, format);
  vsprintf(buffer,format, args);
  perror(buffer);
  va_end(args);
}
```

- Эта реализация не слишком совершенна (а что если буфер переполнится?), но вполне обычна для языка C