Введение в язык программирования C.

Лекция 8
Строки. Ввод-вывод. Препроцессор

Сергей Леонидович Бабичев

Стандартный ввод-вывод.

- Кроме `printf` и `scanf` в `<stdio.h>` имеется ряд функций, вводящих со стандартного ввода и выводящими на стандартный вывод.
- `int putchar(int c);` — выводит символ с кодом `c` на стандартный вывод.
- `int puts(const char *s);` -- выводит строку `s` на стандартный вывод и добавляет переход на новую строку.
- `int getchar();` — вводит один символ со стандартного ввода. Если она возвращает число от 0 до 255, то она смогла что-то считать, и тогда её код можно присвоить в какую-то переменную. Если она вернула константу `EOF`, то стандартный ввод закончился (`EOF = End Of File`).
- Имитировать конец файла в `Linux` и `MacOS` можно нажатием `Ctrl/D`.
- Имитировать конец файла в `Windows` можно нажав `Ctrl/Z` с последующим `Enter`.

Потоки ввода/вывода

- Работа с файлами, содержащими текст, в Си мало чем отличается от работы с клавиатурой и экраном
- Когда мы пишем `printf("Hello\n");` мы какой-то текст (`"Hello\n"`) хотим куда-то вывести.
- По-умолчанию — на экран терминала.
- Экран, жёсткий диск, DVD-диск, принтер,... с точки зрения Си являются *файлами* и к ним всем применимы одни и те же операции.

- В <stdio.h> наряду с прототипами таких функций, как `printf`, `scanf`, `getchar`. и определением константы `EOF` имеется описание типа данных `FILE`.
- Этот тип данных поможет нам читать/писать информацию, расположенную в файлах.
- *Стандартный вывод*, экран терминала, — тоже файл.
- *Стандартный ввод*, клавиатура, — тоже файл.
- `FILE` — системная структура, которая требуется для работы с файлами.
- Так как структуры передаются по значению, то есть копируются, все функции работы с файлами в качестве аргументов используют указатели на объекты такого типа `FILE *`.

Структура FILE

- Три таких указателя уже имеются, они объявлены в <stdio.h> и ими можно пользоваться. Это следующие глобальные переменные:

```
FILE *stdin;
FILE *stdout;
FILE *stderr;
```

- Первая структура обеспечивает доступ к стандартному вводу. Следующие две строки эквивалентны:

```
scanf("%d", &n);
fscanf(stdin, "%d", &n);
```

- Вторая структура, обеспечивает доступ к стандартному выводу. Функция printf тоже имеет своего двойника:

```
printf("Hello, I'm robot number %d\n", robot_number);
fprintf(stdout, "Hello, I'm robot number %d\n", robot_number);
```

Структура FILE

- Третья структура обеспечивает доступ к `стандартному выводу ошибок` и тоже вначале связана с экраном.
- Каждый из *стандартных* вводов и выводов может быть *перенаправлен*, но об этом мы сейчас говорить не будем.
- Если вместо `stdin` или `stdout` мы подставим свои указатели на `FILE`, то сможем читать/писать файлы.
- Как получить `FILE *` для нашей цели?

Функция fopen

- Функция fopen принимает в своих аргументах имя файла и режимы его открывания.
- Возвращает она необходимый для операций с файлами указатель на FILE:

```
FILE *fpr = fopen("input.txt", "r"); // "r" --- readonly
FILE *fpw = fopen("output.txt", "w"); // "w" --- writeonly
FILE *fpa = fopen("append.txt", "a"); // "a" --- append and write
```

- К строке с режимами можно добавить знак +: "r+" или "w+".
- Это означает: что мы после чтения файла хотим туда что-то записать или после записи файла что-то оттуда прочитать.
- Для этого можно файл *перемотать на начало:* rewind(fp);

> Внимание: если вы использовали режим "w", то существующий до этого файл с тем же именем вначале будет обнулён.

Функция fopen

- Если файл не может быть открыт, функция `fopen` возвратит `NULL`.
- После окончания работы с файлом, открытым с помощью `fopen`, его обязательно надо *закрыть*

```
fclose(fpr);
fclose(fpw);
fclose(fpa);
```

- Попытка закрыть не открытый нами файл, такой, как `stdin` или `stdout` не приведёт ни к чему хорошему.
- Скорее всего, перестанут вводиться данные со стандартного ввода или перестанут поступать данные на экран по `printf`.
- Но может произойти и аварийное завершение программы.

Функция fclose

- Повторное закрытие файла не приветствуется.
- Закрытие файла, указатель на который равен NULL может привести к аварии.
- Типичный код при закрытии файла:

```
if (fp != NULL) {
  fclose(fp);
  fp = NULL;
}
```

- Этот код стабильно работает во всех случаях и предпочтительнее использовать именно подобный шаблон.

Позиция в файле

- Работая с файлом мы можем запомнить место, где мы находимся, номер байта.
- `long where = ftell(fp);` — `long` чаще всего 32 бита;
- `off_t where = ftell(fp);` — `off_t` 64 бита.
- Вернуться на запомненное место можно
- `fseek(fp, where, code);`
  - `code == SEEK_SET` — с начала файла.
  - `code == SEEK_CUR` — с текущей позиции файла.
  - `code == SEEK_SET` — с конца файла файла.
  - `where` может быть и отрицательным!

Препроцессор

Препроцессор

- Мы пользуемся строкой:

  `#include <stdio.h>`
- Мы не изменяем свой файл.
- Мы говорим `препроцессору`, чтобы он создал из нашего файла промежуточный, в который он добавит содержимое указанного файла.
- Это происходит до собственно этапа преобразования `Си`-кода в машинный.
- Препроцессор не знает язык `Си`, но помогает писать на нём.
- Препроцессор работает исключительно как обработчик текста.

Команды препроцессора

- Все команды препроцессора начинаются со знака #
- `#include <file>` — включить *системный* файл `file`. Искать его только в системных директориях и тех, что указаны в командной строке.
- `#include "file"` — включить *наш* файл `file`. Искать его в директории компиляции, если там его нет — в системных.
- Эти команды помогают разбить большой проект на несколько файлов.
- В `include`-файлах обычно содержатся прототипы нужных функций.

Команды препроцессора

- `#define var`
  Определить *переменную препроцессора* `var`.
- `#ifdef var`
  Если переменная препроцессора `var` определена, то включить в выходной файл все строки до
- `#endif`

Пример:

```
#define DEBUG
...
#ifdef DEBUG
int debug_func() {
    ...
}
#endif
```

Команды препроцессора

- Определить переменную препроцессора можно и из командной строки:

```
$ gcc -DDEBUG myfile.c
```

- Можно из командной строки менять содержание компилируемого файла. Пример:

```
#ifdef DEBUG
int debug_func() {
    ...
}
#endif
```

Команды препроцессора

- `#define var subst`
  Произвести текстовую замену имени `var` на текст `subst`

```
#define N 100
int a[N];
for (int i = 0; i < N; i++) {
    sum += a[i];
}
```

`#define` делает *текстовую* замену. Он понимает идентификаторы, но не понимает Си!

- `#undef var`
  Отменить определение переменной препроцессора `var`.

Команды препроцессора

- Отрицательный пример:

```
#define left -1

...
while (count != left) do_something(); // OK

struct tree {
    int left, right;                  // Fail
};
```

Команды препроцессора

- `#define min(x,y) x < y ? x : y`
  Определить *препроцессорное макро* с двумя аргументами.

  ```
  z = min(left, right);
  ```

  превратится в

  ```
  z = left < right ? left : right;   // OK
  ```

  А

  ```
  z = min(x++, y++);
  ```

  превратится в

  ```
  z = x++ < y++ ? x++ : y++;      // Very bad!
  ```

Команды препроцессора

- `#define sqr(x) x * x`

Теперь

```
x2 = sqr(x);
```

превратится в

```
x2 = x * x;                    // OK
```

А

```
y2 = sqr(y+1);
```

превратится в

```
y2 = y + 1 * y + 1;            // Very bad!
```

Использование #define

Команда препроцессора `#define` может делать чудеса в умелых руках.
Однако в неопытных она превращается в гранату.
Если вы можете обойтись без неё — постарайтесь обойтись.

Команды препроцессора

- `#if var`
  Если переменная препроцессора содержит числовое значение, в том числе и вычисляемое, его можно использовать в условной компиляции

```
#define DEBUG 10

#if DEBUG > 5
...
#endif

#if DEBUG > 10
...
#endif
```

Встроенные переменные препроцессора

- `__FILE__` — имя компилируемого файла, строка.
- `__LINE__` — номер строки компилируемого файла, число типа `int`.
- `__FUNCTION__` — имя компилируемой функции

Удачный пример использования препроцессора

```
#ifdef TRACING
    #define TRACE printf("[%s:%d]\n", __FILE__, __LINE__)
#else
    #define TRACE
#endif

int func(int arg) {
    TRACE;
    ...
    if (arg > 15) {
        TRACE;
        ...
    } else {
        TRACE;
        ...
    }
}
```