connected with style of function definition - both old and new syntax have the variable inderlined in index)

24 июня 2023 в 07:57

1. Пример CWEB. В этом примере представлена программа “подсчёта слов” из UNIX, переписанная на CWEB для демонстрации грамотного программирования в C. Уровень детализации в этом документе намеренно завышен, для дидактических целей; много из вещей, объясняемых здесь, не нужно объяснять в других программах.

Целью wc является подсчёт строк, слов, и/или символов в списке файлов. Числом строк в файле является число символов перевода строки, которое он содержит. Число символов — это длина файла в байтах.

“Словом” является максимальный ряд последовательных символов отличных от конца строки, пробела или табуляции, содержащий как минимум один видимый код.

Эта версия wc имеет нестандартную “тихую” опцию (-s), которая подавляет вывод кроме сумм значений по всем файлам.

2. Большинство программ CWEB делят общую структуру. Это возможно хорошая идея обозначить общую структуру явно в начале, даже хотя разные части могли быть все представлены в неназванных разделах кода если мы захотим добавлять их по частям.

Здесь, тогда, находится обзор файла wc.c который определён этой CWEB программой wc.w:

⟨ Заголовочные файлы для включения 3 ⟩
⟨ Глобальные переменные 4 ⟩
⟨ Функции 20 ⟩
⟨ Основная программа 5 ⟩

3. Мы должны включить стандартные определения ввода/вывода, т.к. мы хотим отправлять отформатированный вывод на *stdout* и *stderr*.

⟨ Заголовочные файлы для включения 3 ⟩ ≡
```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <ctype.h>
```
Этот код используется в секции 2.

4. Переменная *status* сообщит операционной системе был запуск успешным или нет, и *prog_name* используется в случае если должно быть выдано соощение об ошибке.

```
#define OK 0 /* status код для успешного запуска */
#define usage_error 1 /* status код для неверного синтаксиса */
#define cannot_open_file 2 /* status код для ошибки доступа к файлу */
```

⟨ Глобальные переменные 4 ⟩ ≡
```
int status = OK; /* статус выхода команды, вначале OK */
char *prog_name; /* кто мы */
```
Смотри также секцию 14.

Этот код используется в секции 2.

5. Сейчас мы подходим к основному описанию функции *main*.

```
⟨ Основная программа 5 ⟩ ≡
  int      /* return type */
  main(argc, argv)
       int argc;      /* число аргументов в командной строке UNIX */
       char **argv;      /* сами аргументы, массив строк */
  {
    ⟨ Переменные локальные для main 6 ⟩
    prog_name = argv[0];
    ⟨ Задать выбор опций 7 ⟩;
    ⟨ Обработать все файлы 8 ⟩;
    ⟨ Вывести суммарные значения если было несколько файлов 19 ⟩;
    return status;
  }
```

Этот код используется в секции 2.

6. Если первый аргумент начинается с '`-`', пользователь выбирает желаемые счётчики и указывает порядок в котором они должны отображаться. Каждый выбор даётся начальным символом (строки, слова, или символы). Например, '`-cl`' вызовет печать только числа символов и числа строк, в этом порядке. По умолчанию, если не задано особого аргумента, используется '`-lwc`'.

Мы не обрабатываем эту строку сейчас; мы просто запоминаем где она. Она будет использована для контроля форматирования во время вывода.

Если за '`-`' непосредственно следует '`s`', то выдаются только общие суммы.

```
⟨ Переменные локальные для main 6 ⟩ ≡
  int file_count;      /* количество файлов */
  char *which;      /* какие счётчики печатать */
  int silent = 0;      /* не ноль если была выбрана тихая опция */
```

Смотри также секции 9 и 12.

Этот код используется в секции 5.

7.

```
⟨ Задать выбор опций 7 ⟩ ≡
  which = "lwc";      /* если не задано никакой опции, выдать все три значения */
  if (argc > 1 ∧ *argv[1] ≡ '-') {
    argv[1]++;
    if (*argv[1] ≡ 's') silent = 1, argv[1]++;
    if (*argv[1]) which = argv[1];
    argc--;
    argv++;
  }
  file_count = argc − 1;
```

Этот код используется в секции 5.

8. Теперь мы просматриваем оставшиеся аргументы и пытаемся открыть файл, если возможно. Файл обрабатывается и выдается его статистика. Мы используем цикл **do** ... **while** потому что мы должны считывать стандартный ввод если не задано имя файла.

```
⟨ Обработать все файлы 8 ⟩ ≡
  argc −−;
  do {
    ⟨ Если файл дан, пробовать открыть ∗(++argv); continue если неудача 10 ⟩;
    ⟨ Инициализировать указатели и счётчики 13 ⟩;
    ⟨ Сканировать файл 15 ⟩;
    ⟨ Записать статистику для файла 17 ⟩;
    ⟨ Закрыть файл 11 ⟩;
    ⟨ Обновить большие суммы 18 ⟩;      /∗ даже если есть только один файл ∗/
  } while (−−argc > 0);
```

Этот код используется в секции 5.

9. Вот код для открытия файла. Чтобы обрабатывать ввод из *stdin* когда не задано имени файла, присвоим начальное значение 0, равное дескриптору файла для *stdin*.

```
⟨ Переменные локальные для main 6 ⟩ +≡
  int fd = 0;      /∗ файловый дескриптор, установленный в stdin ∗/
```

10.

```
#define READ_ONLY 0      /∗ считать код доступа для системного вызова open ∗/
⟨ Если файл дан, пробовать открыть ∗(++argv); continue если неудача 10 ⟩ ≡
  if (file_count > 0 ∧ (fd = open(∗(++argv), READ_ONLY)) < 0) {
    fprintf(stderr, "%s: cannot open file %s\n", prog_name, ∗argv);
    status |= cannot_open_file;
    file_count −−;
    continue;
  }
```

Этот код используется в секции 8.

11.

```
⟨ Закрыть файл 11 ⟩ ≡
  close(fd);
```

Этот код используется в секции 8.

12. Мы сделаем доморощенную буферизацию чтобы ускорить процесс: Символы будут считываться в массив *buffer* перед тем, как мы их обработаем. Чтобы сделать это мы задаём подходящие указатели и счётчики.

```
#define buf_size BUFSIZ      /∗ BUFSIZ из stdio.h выбирается для эффективности ∗/
⟨ Переменные локальные для main 6 ⟩ +≡
  char buffer[buf_size];      /∗ мы считываем ввод в этот массив ∗/
  register char ∗ptr;      /∗ первый необработанный символ в buffer ∗/
  register char ∗buf_end;      /∗ первая неиспользуемая позиция в buffer ∗/
  register char c;      /∗ текущий символ или число только что считанных символов ∗/
  ssize_t num;
  int in_word;      /∗ мы внутри слова? ∗/
  long word_count, line_count, char_count;
    /∗ число слов, строк и символов, найденных до сих пор в файле ∗/
```

13. ⟨ Инициализировать указатели и счётчики 13 ⟩ ≡

```
ptr = buf_end = buffer;
line_count = word_count = char_count = 0;
in_word = 0;
```

Этот код используется в секции 8.

14. Общие суммы должны быть установлены в нуль в начале программы. Если бы мы сделали эти переменные локальными для *main*, нам пришлось бы делать эту инициализацию явно; однако, глобальные переменные в C автоматически обнуляются. (Или скорее “статически обнуляются”.) (Понятно?)

⟨ Глобальные переменные 4 ⟩ +≡

```
long tot_word_count, tot_line_count, tot_char_count;
  /* общее количество слов, строк, и символов */
```

15. Настоящий раздел, который делает подсчёт, собственно *raison d'être* для `wc`, был вообще-то самым лёгким для написания. Мы смотрим на каждый символ и изменяем состояние если он начинает или заканчивает слово.

⟨ Сканировать файл 15 ⟩ ≡

```
while (1) {
  ⟨ Заполнить buffer если он пустой; break в конце файла 16 ⟩;
  c = *ptr++;
  if (c ≠ '␣' ∧ isprint(c)) {
    if (¬in_word) {
      word_count++;
      in_word = 1;
    }
    continue;
  }
  if (c ≡ '\n') line_count++;
  else if (c ≠ '␣' ∧ c ≠ '\t') continue;
  in_word = 0;      /* c — это перевод строки, пробел, или таб */
}
```

Этот код используется в секции 8.

16. Буферный ввод/вывод позволяет нам посчитать число символов почти задаром.

⟨ Заполнить *buffer* если он пустой; **break** в конце файла 16 ⟩ ≡

```
if (ptr ≥ buf_end) {
  ptr = buffer;
  num = read(fd, ptr, buf_size);
  if (num ≤ 0) break;
  char_count += num;
  buf_end = buffer + num;
}
```

Этот код используется в секции 15.

17. Удобно выводить статистику определив новую функцию *wc_print*; тогда одна и та же функция может быть использована для подсчёта общих сумм. Вдобавок, мы должны решить здесь знаем ли мы имя файла который мы обработали или это был лишь *stdin*.

```
⟨ Записать статистику для файла 17 ⟩ ≡
  if (¬silent) {
    wc_print(which, char_count, word_count, line_count);
    if (file_count) printf("␣%s\n", *argv);      /* не stdin */
    else printf("\n");      /* stdin */
  }
```

Этот код используется в секции 8.

18.
```
⟨ Обновить большие суммы 18 ⟩ ≡
  tot_line_count += line_count;
  tot_word_count += word_count;
  tot_char_count += char_count;
```

Этот код используется в секции 8.

19. Мы можем также немного улучшить UNIX'овый `wc`, выведя ещё и количество файлов.

```
⟨ Вывести суммарные значения если было несколько файлов 19 ⟩ ≡
  if (file_count > 1 ∨ silent) {
    wc_print(which, tot_char_count, tot_word_count, tot_line_count);
    if (¬file_count) printf("\n");
    else printf("␣total␣in␣%d␣file%s\n", file_count, file_count > 1 ? "s" : "");
  }
```

Этот код используется в секции 5.

20. Вот теперь функция которая выдаёт значения в соответствии с указанными опциями. На вызывающую процедуру возлагается обеспечение перевода строки. Если найден символ неверной опции мы информируем пользователя о правильном использовании команды. Счётчики выдаются в 8-циферных полях, чтобы выстроить их в колонки.

```
#define print_count(n) printf("%8ld", n)
⟨ Функции 20 ⟩ ≡
  void /* return type */
  wc_print(which, char_count, word_count, line_count)
      char *which; /* какие счётчики выдавать */
      long char_count, word_count, line_count; /* данные суммарные значения */
  {
    while (*which)
      switch (*which++) {
      case 'l': print_count(line_count);
        break;
      case 'w': print_count(word_count);
        break;
      case 'c': print_count(char_count);
        break;
      default:
        if ((status & usage_error) ≡ 0) {
          fprintf(stderr, "\nUsage: %s [-lwc] [filename ...]\n", prog_name);
          status |= usage_error;
        }
      }
  }
```

Этот код используется в секции 2.

21. В нашем случае тест этой программы в сравнении с обычной программы `wc` на SPARCstation показал что “официальный” `wc` был немного медленнее. Более того, хотя `wc` выдал соответствующее сообщение об ошибке для опций ‘`-abc`’, он никак не пожаловался на опции ‘`-labc`’! Осмелимся предположить что системная команда могла бы быть лучше если бы её программист использовал более грамотный подход?