1 Занятие №7

1.1 Операторы передачи управления

1.1.1 Оператор `switch`

Оператор выбора **`switch`** используется для передачи управления в зависимости от значения некоторого целого выражения. Оператор записывается следующим образом:

`switch (<выражение>) <оператор>`

Где `<оператор>` почти всегда составной оператор, но это не обязательно.

Внутри оператора могут располагаться метки **`case`**, записываемые следующим образом

`case <константное целое выражение>:`

Оператор вычисляет выражение в заголовке оператора **`switch`**, и затем переходит на метку **`case`** с равным значением. В операторе не может содержаться две метки **`case`** с равным значением. Если соответствующая метка **`case`** не найдена, производится переход на метку **`default:`**, если она существует, и выход из оператора в противном случае.

Метки **`case`** и **`default`** могут содержаться во вложенных операторах (например, **`while`** или **`for`**), но не во вложенных операторах **`switch`**.

Если внутри оператора **`switch`** встречается оператор **`break;`**, управление передаётся за оператор выбора. Оператор **`break;`** должен использоваться для выхода из оператора выбора после вычисления альтернативы, потому что в противном случае управление «провалится» на следующую метку **`case`**. Например,

```
switch (light) {
  case red:
    printf("Stop\n");
    break;
  case green:
    printf("Go\n");
    break;
  case yellow:
    printf("Be␣ready\n");
  default:
    printf("Unknown␣light\n");
}
```

1.1.2 Оператор `goto`

Оператор `goto <метка>;` передаёт управление на заданную метку. Метка должна находится в той же функции, помеченный оператор записывается `<метка>:`. Где `<метка>` — это произвольный идентификатор. Метки никак специально в функции не объявляются, и переходы на метки вперёд допустимы. Метки находятся в своём пространстве имён, отличном от идентификаторов и тегов, это значит, что имена меток могут совпадать с именами других объектов. Метки всегда локальны в пределах одной функции. Две метки с одним именем внутри одной функции не допускаются. Оператор перехода может входить в блок в обход инициализации, например:

```
if (x) goto label;
/* ... */
while (y) {
  int z = 1;

  /* ... */
label:
  /* ... */
}
```

тогда при переходе на метку `label` под переменную будет выделена память, но она *не будет инициализирована*.

1.2 Инициализация составных типов

Массивы, структуры и перечисления точно также, как и простые типы могут быть проинициализированы в точке определения. Массивы типов **`char`**, **`signed char`**, **`unsigned char`** могут быть проинициализированы строкой. Для одномерного массива произвольного типа инициализация выглядит следующим образом:

```
<тип> <имя>[<размер>] = { <зн. 1>, <зн. 2>, ... <зн. K> };
```

Все инициализирующие значения должны иметь тип, совпадающий с типом массива. Их не должно быть больше, чем размер массива, если их меньше, чем размер массива, оставшиеся элементы инициализируются по умолчанию. Например,

```
char str[20] = { 'a', ' ', 's', 't', 'r', 'i', 'n', 'g', '\0' };
```

Если массив содержит инициализацию, тогда размер массива может быть опущен. Он будет вычислен по количеству инициализирующих элементов.

Структура инициализируется похожим образом.

```
<тип> <имя> = { <зн. 1>, <зн. 2>, ..., <зн. K>};
```

В этом случае полям структуры последовательно присваиваются указанные значения. У объединений может быть проинициализирован только первый элемент.

Если массив сам имеет тип массива (многомерный массив), или тип структуры, или структура имеет поля типа массива или структуры, инициализаторы могут быть вложенными. Например

```
struct circle { double x, y, r; };
struct circle cc[2] = {{1.0, 2.0}, {1.3,2.0,0.4}};
```

Если фигурные скобки, отделяющие внутренние инициализаторы, опускаются, инициализация идёт подряд, переходя при необходимости границы типов. Например, если в указанном выше примере опустить внутренние фигурные скобки, `cc[1]` пример значение `{1.0,2.0,1.3}`, а `cc[2]` — `{2.0,0.4,0.0}` (если переменная `cc` — глобальная).

1.3 Передача параметров в программу

При запуске программы операционная система передаёт ей параметры, которые были указаны в командной строке. Если программа желает работать с переданными ей параметрами, заголовок функции `main` должен выглядеть следующим образом:

```
int main(int argc, char *argv[]);
```

Здесь первый параметр `argc` задаёт количество аргументов командной строки, а параметр `argv` — это массив указателей на строки, хранящие значения соответствующих аргументов. По соглашению `argv[0]` содержит само имя программы, а `argv[argc]` содержит 0 (`NULL`). Следовательно, если программе не было передано никаких аргументов, `argc` равно 1. Разбиение командной строки на аргументы запускаемой программы производит командный процессор, обычно аргументы разделяются друг от друга пробелами. Например, если программа была запущена командой

```
./prog1 f1 f2 ../f3
```

`argc` пример значение 4, `argv[0]` указывает на строку `"./prog1"`, `argv[1]` — на строку `"f1"` и т. д.

Значение, возвращаемое функцией `main` (или аргумент функции `exit`), — это «код возврата» программы. Он используется чтобы проинформировать вызвавшую программу о статусе завершения работы программы. Код возврата 0 означает, что работа программы завершилась нормально, ненулевые коды возврата означают, что при выполнении программы возникли какие-то проблемы.

Пример программы, которая печатает свои аргументы.

```
#include <stdio.h>
int main(int argc, char *argv[])
{
        int i;

        for (i = 0; i < argc; i++) {
                printf("argv[%d] = %s\n", i, argv[i]);
        }
        return 0;
}
```

1.4 Функции работы с файлами

Мы ещё не рассмотрели две функции для посимвольной работы со стандартным потоком. Функция

```
int getchar(void);
```

вводит один символ из стандартного входного потока. Если символ введён успешно, возвращается код символа в диапазоне 0–255 (положительное число). В случае ошибки или конца файла возвращается значение `EOF` (оно обычно равно -1). Функция

```
int putchar(int c);
```

выводит один символ на стандартный поток вывода.

Обратите внимание, что функция `getchar` может возвращать 257 различных значений, поэтому *переменной типа* **`char`** *для хранения возвращаемого значения недостаточно!* Следующий пример копирует стандартный ввод на стандартный вывод.

```
#include <stdio.h>
int main()
{
        int c;
        while ((c = getchar()) != EOF) putchar(c);
        return 0;
}
```

Стандартная библиотека содержит средства для работы с произвольными файлами. Для хранения информации об открытом файле используется структура `FILE`. Функции работы с файлами принимают или возвращают указатель на эту структуру. Функция

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

открывает файл с именем `name`. Строка `mode` содержит флаги, с которыми открывается файл.

`"r"`	открыть для чтения. Текущая позиция в файле устанавливается на начало файла.
`"w"`	открыть для записи. Если файл не существовал, он создаётся, если существовал, он очищается. Текущая позиция в файле устанавливается на начало файла.
`"a"`	открыть для добавления. Если файл не существовал, он создаётся. Текущая позиция в файле устанавливается на конец файла.
`"r+"`	открыть для чтения и записи. Текущая позиция в файле устанавливается на начало файла.
`"w+"`	открыть для чтения и записи. Если файл не существовал, он создаётся, если существовал, он очищается. Текущая позиция в файле устанавливается на начало файла.
`"a+"`	открыть для чтения и записи. Если файл не существовал, он создаётся. Текущая позиция в файле устанавливается на конец файла.

Если файл был открыт успешно, возвращается указатель на структуру, хранящую состояние открытого файла. Если при открытии произошла ошибка, возвращается `NULL`.

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

Закрывает поток. Если закрытие прошло успешно, возвращается 0, иначе возвращается `EOF`.

При запуске программы открываются стандартные потоки `stdin`, `stdout`, `stderr`. Например, функции `putchar`, `puts` работают с потоком `stdin`, а функции `getchar`, `gets` работают с потоком `stdout`. Вы можете использовать эти имена для указания имён стандартных потоков в функциях, перечисленных ниже.

```
int getc(FILE *stream);
int putc(int c, FILE *stream);
int fscanf(FILE *stream, char *format, ...);
int fprintf(FILE *stream, char *format, ...);
```

Соответствуют функциям `getchar`, `putchar`, `scanf`, `printf`.

Функция

```
int fputs(char *str, FILE *stream);
```

записывает строку символов `str` в файл. Символ-терминатор строки отбрасывается. *В отличие от `puts` эта функция не дописывает `'\n'` в выходной файл.*

Функция

```
char *fgets(char *str, int size, FILE *stream);
```

Считывает самое большее `size - 1` символ из входного файла. Чтение останавливается по достижению конца файла или по достижению символа `'\n'`. Если `'\n'` считан, он добавляется в строку. После прочитанных символов добавляется символ-терминатор `'\0'`. При успешном завершении функция возвращает `str`, а при неудаче (конец файла или ошибка ввода) возвращается `NULL`. *Используйте эту функцию для чтения строк из входного файла, поскольку эта функция ограничивает максимальную длину считываемой строки.*

Следующий пример копирует содержимое заданного файла на стандартный вывод.

```
#include <stdio.h>
int main(int argc, char *argv[])
{
  FILE *f;
  char  buf[256];

  if (argc != 2) {
    fprintf(stderr, "Wrong number of arguments\n");
    return 1;
  }
  if (!(f = fopen(argv[1], "r"))) {
    fprintf(stderr, "Cannot open file %s\n", argv[1]);
    return 1;
  }
  while (fgets(buf, sizeof(buf), f))
    fputs(buf, stdout);
  fclose(f);
  return 0;
}
```

Функция

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

Заталкивает один символ обратно во входной поток, так что следующая функция `getc` считает именно этот символ. Таким образом затолкнуть назад можно не более одного символа.

Функция

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

возвращает ненулевое значение, если в структуре `FILE` установлен флаг ошибки или конца файла. Этот флаг устанавливается *по результату работы функций чтения или записи*. До первого чтения флаг сброшен, даже если файл пуст. В следующем примере последняя строка будет напечатана дважды.

```
#include <stdio.h>
int main(int argc, char *argv[])
{
        char  buf[64];
        while (!feof(stdin)) {
                fgets(buf, sizeof(buf), stdin);
                fputs(buf, stdout);
```

```
        }
        return 0;
}
```