7.10.2024

Классы хранения, связывание и управление памятью. Файловый ввод-вывод. Структуры – начальные сведения.

Филиппов Михаил Витальевич

m.filippov@g.nsu.ru

89232283872

Императивное программирование, 2024-2025

Новосибирский государственный университет

*НАСТОЯЩАЯ НАУКА

Давайте познакомимся

Филиппов Михаил Витальевич

- Окончил магистратуру ФФ НГУ
- Окончил аспирантуру ИТ СО РАН
- Являюсь м.н.с. ИТ СО РАН
- 7+ лет опыт в программировании C/C++

Адженда

Классы хранения, связывание и управление памятью

40 минут

Файловый ввод-вывод

30 минут

Структуры – начальные сведения

20 мин

Адженда

Классы хранения, связывание и управление памятью

40 минут

Файловый ввод-вывод

30 минут

Структуры – начальные сведения

20 мин

Классы хранения

- Для хранения данных в памяти язык С предлагает пять разных моделей, или классов хранения.
- ***Аппаратный аспект:*** любое сохраненное значение находится в физической памяти

Объект – термин в С, предназначенный для описания участка памяти хранения данных программы. Объект может хранить одно или большее количество значений. В определенный момент объект может пока не содержать сохраненного значения, но он будет иметь правильный размер для помещения подходящего значения. В общем случае выражение, которое обозначает объект, называется l-значением.

- ***Программный аспект:*** программе нужен какой-нибудь способ доступа к объекту.

Пример, объявление переменной: int entity = 3;

Другие примеры, взгляните на следующие объявления:

```
int *pt = &entity;
int ranks[10];
```

Область видимости

Область видимости описывает участок или участки программы, где можно обращаться к идентификатору.

Переменная в C имеет одну из следующих областей видимости: в пределах блока, в пределах функции, в пределах прототипа функции и в пределах файла.

Блок — это часть кода, содержащаяся между открывающей фигурной скобкой и соответствующей ей закрывающей скобкой

Пример: cleo, patric внутри фунции, q – внутренний блок

```
double blocky(double cleo) {
    double patrick = 0.0;
    int i ;
    for (i =0; i < 10; i++) {
        double q = cleo * i; // начало области видимости для q
        …
        patrick *= q; // конец области видимости для q
    }
    return patrick;
}
```

C99 расширил концепцию блока путем включения в нее кода, убавляющего циклами for, while, do while или оператором if.

Область видимости

Область видимости в пределах функции применяется только к меткам, применяемым с операторами **goto**. Это означает, что, если метка впервые появляется во внутреннем блоке функции, ее область видимости простирается на всю функцию

Область видимости в пределах ***прототипа функции*** применяется к именам переменных, используемым в прототипах функций, как в следующем случае:

int mighty (int mouse, double large);

Область видимости в пределах прототипа функции распространяется от места определения переменной до конца объявления прототипа.

Переменная, определение которой находится за рамками любой функции, имеет область видимости в пределах файла.

Продолжительность хранения

Объект в C имеет одну из следующих четырех продолжительностей хранения:

- статическую
- потоковую
- автоматическую
- выделенную

Если объект имеет статическую продолжительность хранения, он существует на протяжении времени выполнения программы.

Объект с потоковой продолжительностью хранения существует с момента его объявления и до завершения потока. (_Thread_local)

Переменные с областью видимости в пределах блока обычно имеют автоматическую продолжительность хранения.

Продолжительность хранения

```
void more(int number) {
   int index;

   static int ct = 0;

   …

   return 0;

}
```

Здесь переменная ct хранится в статической памяти; она существует с момента загрузки программы в память и вплоть до завершения выполнения программы.

Пять классов хранения

Класс хранения	Продолжительность хранения	Область видимости	Связывание	Объявление
Автоматический	Автоматическая	В пределах блока	Нет	В блоке
Регистровый	Автоматическая	В пределах блока	Нет	В блоке с указанием ключевого слова register
Статический с внешним связыванием	Статическая	В пределах файла	Внешнее	За рамками всех функций
Статический с внутренним связыванием	Статическая	В пределах файла	Внутреннее	За рамками всех функций с указанием ключевого слова static
Статический без связывания	Статическая	В пределах файла	Нет	В блоке с указанием ключевого слова static

Автоматические переменные

По умолчанию любая переменная, объявленная в блоке или в заголовке функции, относится к автоматическому классу хранения. Однако вы можете совершенно ясно сформулировать свои намерения, явным образом указав ключевое слово auto:

```
int main(void) {
    auto int plox;
```

Пример:

```
int loop(int n) {
   int m; // m находится в области видимости
   scanf("%d", &m);
   {
        int i;
        for (i = m; i < n; i++) // и m, и i находятся в области видимости
            puts("i is local to a sub-block\n") ;
        }
    return m; // m в области видимости, i исчезла
}
```

Автоматические переменные

```c
#include <stdio.h>
int main()
{
    int x = 30; // исходная переменная x
    printf("x во внешнем блоке: %d по адресу %p\n", x, &x);
    {
        int x = 77; // новая переменная x, скрывающая первую x
        printf("x во внутреннем блоке: %d по адресу %p\n", x, &x);
    }
    printf("x во внешнем блоке: %d по адресу %p\n", x, &x);
    while (x++ < 33) // исходная переменная x
    {
        int x = 100; // новая переменная x, скрывающая первую x
        x++;
        printf("x в цикле while: %d по адресу %p\n", x, &x);
    }
    printf("x во внешнем блоке: %d по адресу %p\n", x, &x);
    return 0;
}
```

```
x во внешнем блоке: 30 по адресу 0x7ffd90e4ba70
x во внутреннем блоке: 77 по адресу 0x7ffd90e4ba74
x во внешнем блоке: 30 по адресу 0x7ffd90e4ba70
x в цикле while: 101 по адресу 0x7ffd90e4ba74
x в цикле while: 101 по адресу 0x7ffd90e4ba74
x в цикле while: 101 по адресу 0x7ffd90e4ba74
x во внешнем блоке: 34 по адресу 0x7ffd90e4ba70
```

Автоматические переменные

```c
int main()
{
    int n = 8;
    printf(" Первоначально n = %d по адресу %p\n", n, &n);
    for (int n = 1; n < 3; n++)
        printf(" цикл 1: n = %d по адресу %p\n", n, &n);
    printf(" После цикла 1 n = %d по адресу %p\n", n, &n);
    for (int n = 1; n < 3; n++)
    {
        printf(" индекс цикла 2 n = %d по адресу %p\n", n, &n);
        int n = 6;
        printf(" цикл 2: n = %d по адресу %p\n", n, &n);
        n++;
    }
    printf(" После цикла 2 n = %d по адресу %p\n", n, &n);
    return 0;
}
```

```
//  Первоначально n = 8 по адресу
0x7ffd31b90e4c
//  цикл 1: n = 1 по адресу
0x7ffd31b90e54
//  цикл 1: n = 2 по адресу
0x7ffd31b90e54
//  После цикла 1 n = 8 по адресу
0x7ffd31b90e4c
//  индекс цикла 2 n = 1 по
адресу 0x7ffd31b90e50
//  цикл 2: n = 6 по адресу
0x7ffd31b90e54
//  индекс цикла 2 n = 2 по
адресу 0x7ffd31b90e50
//  цикл 2: n = 6 по адресу
0x7ffd31b90e54
//  После цикла 2 n = 8 по адресу
0x7ffd31b90e4c
```

Регистровые переменные

Переменные обычно хранятся в памяти компьютера. При благоприятном стечении обстоятельств регистровые переменные хранятся в регистрах центрального процессора, или, в общем случае, в самой быстрой имеющейся памяти, что обеспечивает доступ и манипулирование ими с меньшими затратами времени, чем для рядовых переменных.

Поскольку регистровая переменная может находиться в регистре, а не в памяти, получить адрес такой переменной не удастся.

Пример:

```
int main(void) {
    register int quick;

void macho(register int n)
```

Компилятор должен сопоставить ваши требования с количеством доступных регистров или объема быстродействующей памяти, или же он может просто проигнорировать запрос, и ваше пожелание не будет удовлетворено.

Статические переменные с областью видимости в пределах блока

```c
#include <stdio.h>
void trystat(void);
int main(void)
{
    int count;
    for (count = 1; count <= 3; count++)
    {
        printf("Начинается итерация %d:\n", count);
        trystat();
    }

    return 0;
}
void trystat(void)
{
    int fade = 1;
    static int stay = 1;
    printf("fade = %d и stay = %d\n", fade++, stay++);
}
```

int wontwork(static int flu); // не разрешено

```
Начинается итерация 1:
fade = 1 и stay = 1
Начинается итерация 2:
fade = 1 и stay = 2
Начинается итерация 3:
fade = 1 и stay = 3
```

Статические переменные с внутренним связыванием

Переменные с этим классом хранения имеют статическую продолжительность хранения, область видимости в пределах файла и внутреннее связывание.

```
static int svil = 1; // статическая переменная, внутреннее связывание
int main(void)
{


int traveler = 1; // внешнее связывание
static int stayhome = 1; // внутреннее связывание
int main()
{
    extern int traveler; // использование глобальной переменной traveler
    extern int stayhome; // использование глобальной переменной stayhome
```

Спецификаторы классов хранения

В языке С имеется шесть ключевых слов, которые сгруппированы вместе как спецификаторы классов хранения: auto, register, static, extern, _Thread_local и typedef.

- Спецификатор auto указывает переменную с автоматической продолжительностью хранения.
- Спецификатор register также может использоваться только с переменными, имеющими область видимости в пределах блока.
- Спецификатор static создает объект со статической продолжительностью хранения, который появляется после загрузки программы в память и исчезает при завершении программы.
- Спецификатор extern указывает, что вы объявляете переменную, которая была определена в каком-то другом месте.

Функция генерации случайных чисел и статическая переменная

В действительности функция rand() является “генератором псевдослучайных чисел”, т.е. фактическая последовательность чисел предсказуема, но числа достаточно равномерно распределены по диапазону возможных значений.

```c
#include <stdio.h>
static unsigned long int next = 1; /* начальное число
*/
int rand0(void)
{
    /* магическая формула генерации псевдослучайных
чисел */
    next = next * 1103515245 + 12345;
    return (unsigned int)(next / 65536) % 32768;
}
int main(void)
{
    int count;
    for (count = 0; count < 5; count++)
        printf("%d\n", rand0());
    return 0;
}
```

```
16838
5758
10113
17515
31051
```

Функция генерации случайных чисел и статическая переменная

```c
void srandl(unsigned int seed)
{
    next = seed;
}
int main(void)
{
    int count;
    unsigned seed;
    printf("Введите желаемое начальное число.\n");
    while (scanf("%u", &seed) == 1)
    {
        srandl(seed); /* переустановка начального числа */
        for (count = 0; count < 5; count++)
            printf("%d\n", rand1());
        printf("Введите следующее начальное число (q для завершения):\n");
    }
    printf("Программа завершена.\n");
    return 0;
}
```

```
Введите желаемое начальное число.
1
16838
5758
10113
17515
31051
Введите следующее начальное число (q для завершения):
13
22290
13853
11622
27833
1243
```

Выделенная память: malloc() и free()

Все программы должны резервировать пространство памяти, достаточное для хранения данных, с которыми они работают.

Примеры:

```
float x;
char place [] = "Поющие в терновнике";
int plates[100];
```

Язык С выходит за эти рамки. Основным инструментом является функция malloc(), которая принимает один аргумент: нужное количество байтов памяти. Затем malloc() ищет подходящий блок свободной памяти. Память будет анонимной, т.е. функция malloc() выделяет блок памяти, но не назначает ему имя. Тем не менее, она возвращает адрес первого байта в этом блоке.

- malloc() традиционно была объявлена с типом указателя на char. В стандарте ANSI С используется новый тип: указатель на void. Этот тип задумывался как обобщенный указатель

Пример:

```
double * ptd;
ptd = (double *) malloc(30 * sizeof(double));
```

Выделенная память: malloc() и free()

Теперь у вас есть три способа создания массива.

- Объявить массив, используя константные выражения для размерностей, и применять для доступа к элементам имя массива. Такой массив может быть создан с использованием либо статической, либо автоматической памяти.
- Объявить массив переменной длины, применяя переменные выражения для размерностей, и использовать для доступа к элементам имя массива. (Вспомните, что эта возможность предусмотрена стандартом C99.) Такой вариант доступен только для автоматической памяти.
- Объявить указатель, вызвать malloc(), присвоить возвращаемое значение указателю и применять для доступа к элементам указатель. Этот указатель может быть либо статическим, либо автоматическим.

Примеры:

double item[n]; /* до C99: не разрешено, если n является переменной */

ptd = (double *) malloc(n * sizeof (double)) ; /* нормально */

Выделенная память: malloc() и free()

Функция free() принимает в качестве аргумента адрес, возвращенный ранее функцией malloc(), и освобождает память, которая была выделена. Таким образом, продолжительность существования выделенной памяти рассчитывается с момента, когда была вызвана функция malloc() для выделения памяти, и до момента, когда вызывается функция free() с целью освобождения памяти для ее повторного использования.

Функции malloc() и free() имеют прототипы в заголовочном файле stdlib.h.

Если выделить нужную память не удалось, для прекращения работы программы вызывается функция exit() , прототип которой содержится в stdlib.h. Значение EXIT_FAILURE определено в этом же заголовочном файле. Стандарт предоставляет два возвращаемых значения, которые гарантированно распознают все операционные системы: EXIT_SUCCESS (эквивалентно значению 0) для указания на нормальное завершение программы и EXIT_FAILURE для указания на аварийное завершение.

Выделенная память: malloc() и free()

```
// Введите максимальное количество
элементов типа double.
// 5
// Введите значения (q для выхода):
// 29 30 35 25 40 80
// Введено 5 элементов:
//   29.00   30.00   35.00   25.00   40.00
// Программа завершена.
```

```c
#include <stdio.h>
#include <stdlib.h> /* для malloc(), free() */
int main(void)
{
    double *ptd;
    int max = 0;
    int number;
    int i = 0;
    puts("Введите максимальное количество элементов типа double.");
    if (scanf("%d", &max) != 1)
    {
        puts("Количество введено некорректно -- программа завершена.");
        exit(EXIT_FAILURE);
    }
    ptd = (double *)malloc(max * sizeof(double));
    if (ptd == NULL)
    {
        puts("Не удалось выделить память. Программа завершена.");
        exit(EXIT_FAILURE);
    }
    /* ptd теперь указывает на массив из max элементов */
    puts("Введите значения (q для выхода):");
    while (i < max && scanf("%lf", &ptd[i]) == 1)
        ++i;
    printf("Введено %d элементов:\n", number = i);
    for (i = 0; i < number; i++)
    {
        printf("%7.2f ", ptd[i]);
        if (i % 7 == 6)
            putchar('\n');
    }
    if (i % 7 != 0)
        putchar('\n');
    puts("Программа завершена.");
    free(ptd);
    return 0;
}
```

Важность функции free()

Объем статической памяти фиксируется во время компиляции;
он не изменяется на протяжении выполнения программы.

```
…
int main() {
   double glad[2000];
   int i;
   …
   for (i = 0 ; i < 1000; i++)
        gobble(glad, 2000);
}
void gobble(double ar[], int n) {
   double * temp = (double*)malloc(n * sizeof(double));
   ...              /* free(temp); // забыли воспользоваться free() */
}
```

Цикл повторяется 1000 раз, и ко
времени его завершения из
пула памяти будет изъято
16 000 000 байтов.

Функция calloc()

Еще один способ выделения памяти предусматривает применение функции calloc(). Ниже показан типичный случай ее использования:

```
long *newmem;
newmem = (long*) calloc(100, sizeof (long));
```

Эта новая функция принимает два аргумента, которые оба должны быть целыми числами без знака (типа size_t в ANSI). Первый аргумент задает желаемое количество ячеек памяти. Второй аргумент задает размер каждой ячейки в байтах.

Применение sizeof(long) вместо 4 делает код более переносимым. Он будет работать в системах, где тип long имеет размер, отличающийся от 4.

Функция calloc() обладает еще одним свойством: она устанавливает в 0 все биты в блоке.

Функция free() может также использоваться для освобождения памяти, выделенной с помощью calloc().

Классы хранения и динамическое распределение памяти

Функциональность массивов переменной длины и malloc() пересекается. Например, оба средства могут применяться для создания массива, размер которого определяется во время выполнения. Одно из отличий между ними заключается в том, что массив переменной длины является автоматической памятью. Таким образом, вы не должны переживать о вызове free().

```
int n = 5;
int m = 6;
int ar2[n][m]; // массив переменной длины n x m
int (*p2) [6]; // работает до выхода стандарта C99
int (*p3) [m]; // требуется поддержка массивов переменной длины
p2 = (int (*)[6]) malloc(n * 6 * sizeof(int)); // массив n * 6
p3 = (int (*)[m]) malloc(n * m * sizeof(int)); // массив n * m
// предыдущее выражение также требует поддержки массивов переменной длины
ar2[1] [2] = p2[1][2] =12;
```

Классы хранения и динамическое распределение памяти

```c
// where.c-- где что находится в памяти ?
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int static_store = 30;
const char *pcg = "Строковый литерал";
int main()
{
    int auto_store = 40;
    char auto_string[] = "Автоматический массив char";
    int *pi;
    char *pcl;
    pi = (int *)malloc(sizeof(int));
    *pi = 35;
    pcl = (char *)malloc(strlen("Динамическая строка") + 1);
    strcpy(pcl, "Динамическая строка");
    printf("static_store: %d по адресу %p\n", static_store, &static_store);
    printf("auto_store: %d по адресу %p\n", auto_store, &auto_store);
    printf("*pi: %d по адресу %p\n", *pi, pi);
    printf("%s по адресу %p\n", pcg, pcg);
    printf("%s по адресу %p\n", auto_string, auto_string);
    printf("%s по адресу %p\n", pcl, pcl);
    printf("%s по адресу %p\n", "Строка в кавычках", "Строка в кавычках");
    free(pi);
    free(pcl);
    return 0;
}
```

```
// static_store: 30 по адресу
0x5578fcae4010
// auto_store: 40 по адресу
0x7fff60091a9c
// *pi: 35 по адресу 0x5578fe9b32a0
// Строковый литерал по адресу
0x5578fcae2008
// Автоматический массив char по
адресу 0x7fff60091ab0
// Динамическая строка по адресу
0x5578fe9b32c0
// Строка в кавычках по адресу
0x5578fcae20b8
```

Квалификаторы типов ANSI C: const

Ключевое слово const в объявлении создает переменную, значение которой не может быть изменено посредством присваивания либо инкрементирования/декрементирования. В случае компилятора, совместимого со стандартом ANSI, код

```
const int nochange; /* указывает, что m является константой */
nochange = 12; /* не разрешено */
const int nochange = 12; /* все в порядке */
```

Ключевое слово const можно применять, например, для создания массива данных, которые в программе не могут изменяться:

```
const int days1 [12] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

Использование const с объявлениями указателей и параметров

Применять ключевое слово const при объявлении простой переменной и массива довольно просто.
const float * pf; /* pf указывает на константное значение float */
float * const pt; /* pt является указателем const */
const float * const ptr;
float const * pfc; // то же, что и const float * pfc;
Как отражает комментарий, помещение const после имени типа и перед символом * означает, что указатель не может использоваться для изменения значения, на которое он ссылается. Выражаясь кратко, ключевое слово const, находящееся где угодно слева от символа *, делает константными данные, а справа — делает константным сам указатель.
Распространенным использованием этого нового ключевого слова является объявление указателей, которые служат формальными параметрами функций.

Однако следующий прототип предотвращает такое изменение:
void display(const int array[], int limit); // данные, на которые указывает array, не могут быть изменены.
Например, объявление функции strcat(), принятое в ANSI C, имеет такой вид:
char *strcat(char * restrict sl, const char * restrict s2);

Квалификаторы типов ANSI C: volatile

Квалификатор volatile сообщает компилятору, что переменная может иметь значение, которое изменяется действиями, внешним и по отношению к программе.
Синтаксис этого квалификатора подобен синтаксису const:

```
volatile int locl;  /* locl является изменчивой ячейкой */
volatile int * ploc; /* ploc указывает на изменчивую ячейку */
```

Эти операторы объявляют locl как значение volatile и ploc как указатель на значение volatile.
Концепция квалификатора volatile довольно интересна, и вы наверняка хотите узнать, почему комитет ANSI счел необходимым сделать volatile ключевым словом. Причина в том, что оно облегчает проведение оптимизации компилятором. Предположим, например, что есть такой код:

```
val1 = x;
/* код, в котором x не используется */
val2 = x;
```

Квалификатор типа restrict

Ключевое слово restrict расширяет вычислительную поддержку, выдавая компилятору разрешение на оптимизацию определенных разновидностей кода.

```
int ar[10];
int * restrict restar = (int *) malloc(10 * sizeof(int)) ;
int * par = ar;
```

Здесь указатель restar является единственным первичным средством доступа в память, выделенную malloc(). Следовательно, он может быть квалифицирован с помощью ключевого слова restrict. Однако указатель par не является ни первичным, ни единственным средством доступа к данным в массиве ar, поэтому он не может быть квалифицирован как restrict.

Пример:

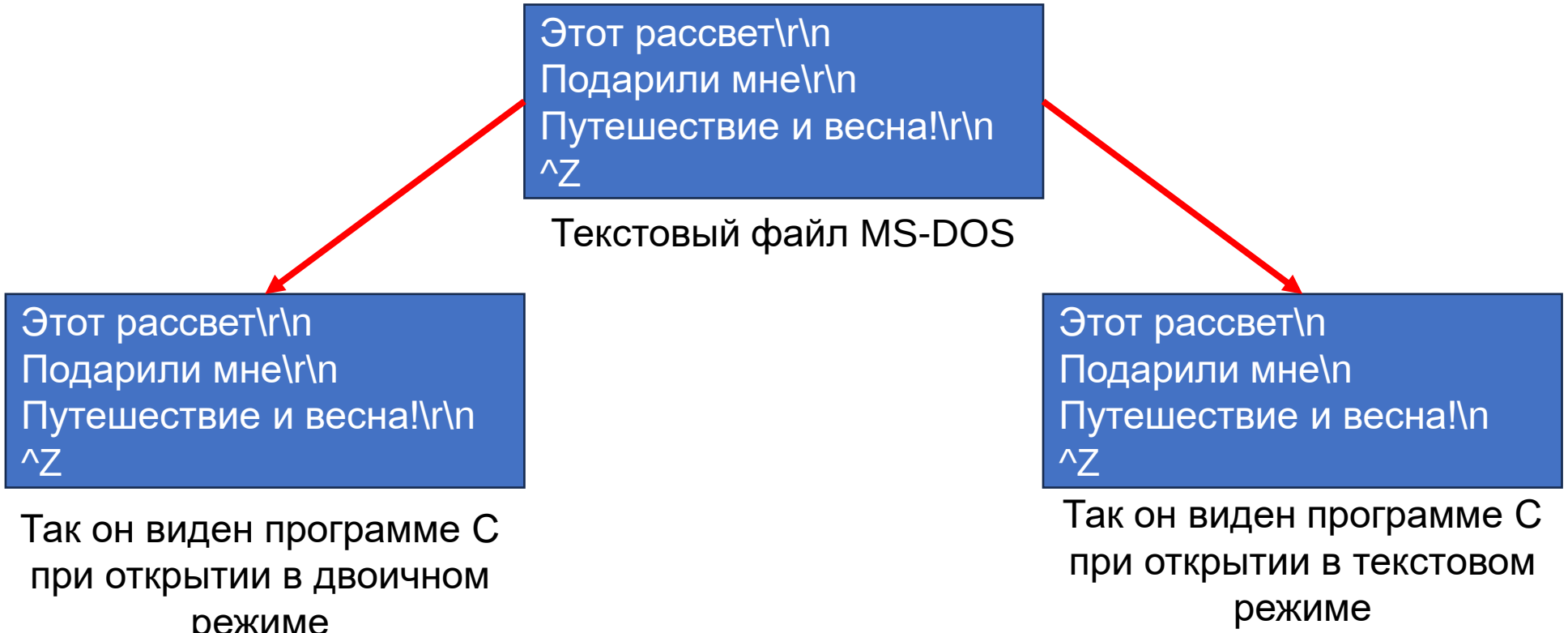
```
for (n = 0; n < 10; n++) {
    par[n] += 5;
    restar[n] += 5;
    ar[n] *= 2;
    par[n] += 3;
    restar[n] + = 3;
}
```

Адженда

Классы хранения, связывание и управление памятью	**Файловый ввод-вывод**	Структуры – начальные сведения
40 минут	30 минут	20 мин

Текстовый режим и двоичный режим

Файл — это именованный раздел хранилища, обычно расположенный на жестком диске или на получающем все большее распространение в последнее время твердотельном диске.



Стандартные файлы

Программы на С автоматически ***открывают три файла***, которые называются *стандартным вводом*, *стандартным выводом* и *стандартным выводом, ошибок*.

Естественно, стандартный ввод обеспечивает ввод данных в программу. Это файл, который читается с помощью функций getchar() и scanf(). Стандартный вывод-место, куда направляется обычный вывод программы. Он используется функциями putchar(), puts() и printf().

Перенаправление приводит к тому, что другие файлы опознаются как стандартный ввод и стандартный вывод. Назначение файла стандартного вывода ошибок заключается в том, чтобы предоставить логически обособленное место для отправки сообщений об ошибках.

Например, если вместо экрана вы перенаправите вывод в файл, то сообщения, отправляемые в стандартный вывод ошибок, по-прежнему будут попадать на экран. Это удобно, т.к. если бы сообщения об ошибках направлялись также в файл, то вы бы их не увидели до тех пор, пока не просмотрели файл.

Стандартный ввод-вывод

```c
#include <stdio.h>
#include <stdlib.h> // прототип exit()
int main()
{
    int ch;   // место для хранения каждого символа по мере чтения
    FILE *fp; // "указатель файла"
    unsigned long count = 0;
    if ((fp = fopen("a.txt", "r")) == NULL)
    {
        printf("Не удается открыть %s\n", "a.txt");
        exit(EXIT_FAILURE);
    }
    while ((ch = getc(fp)) != EOF)
    {
        putc(ch, stdout); // то же, что и putchar (ch) ;
        count++;
    }
    fclose(fp);
    printf("Файл %s содержит %lu символов\n", "a.txt", count);
    return 0;
}
//1 2 3 4 5Файл a.txt содержит 9 символов
```

Функция fopen()

Строка режима	Описание
"r"	Открыть текстовый файл для чтения
"w"	Открыть текстовый файл для записи с усечением существующего файла до нулевой длины или созданием файла, если он не существует
"a"	Открыть текстовый файл для записи с добавлением данных в конец существующего файла и ли созданием файла, если он не существует
"r+"	Открыть текстовый файл для обновления (т.е. для чтения и записи)
"w+"	Открыть текстовый файл для обновления (чтения и записи), предваритель но выполнив усечение файла до нулевой длины, если он существует, или создав файл, если его нет
"a+"	Открыть текстовый файл для обновления (чтения и записи) с добавлени ем данных в конец существующего файла или созданием файла, если он не существует; читать можно весь файл, но записывать допускается только в конец файла
"rb'\ "wb", "ab”, ab+",1"a+b", "wb+", "w+b", "ab+", "a+b”	Подобны предыдущим режимам, за исключением того, что вместо текстового режима они используют двоичный режим
"wx ", "wbx", "w+x", "wb+x" или"w+bx"	(C11) Подобны режимам без буквы x, за исключением того, что они отказываются работать, если файл существует, и открывают файл в монопольном режиме, если это возможно

Функции getc() и putc()

Функции getc() и putc() работают очень похоже на getchar() и putchar(). Отличие заключается в том, что этим новым функциям потребуется указать, с каким файлом работать. Таким образом, приведенный ниже многократно использованный нами оператор означает “получить символ из стандартного ввода":

```
ch = getchar();
```

Тем не менее, следующий оператор означает “получить символ из файла, идентифицируемого fp ”:

```
ch = getc(fp);
```

Аналогично, показанный далее оператор означает “поместить символ ch в файл, идентифицируемый указателем fpout на FILE”:

```
putc(ch, fpout);
```

Конец файла

Программа, читающая данные из файла, должна останавливаться, когда она достигает конца файла. Как можно сообщить программе о том, что встретился конец файла?
Функция getc() возвращает специальное значение EOF, если она пытается прочитать символ и обнаруживает, что достигнут конец файла.

Пример:

```
int ch;
FILE * fp;
fp = fopen("wacky.txt", "r");
while ((ch = getc(fp)) ! = EOF)
{
    putchar(ch); // обработать ввод
}
```

Указанные меры предосторожности касаются и других функций ввода. Они также возвращают сигнал об ошибке (EOF или пустой указатель), столкнувшись с концом файла.

Функция fclose()

Функция fclose(fp) закрывает файл, идентифицируемый fp, при необходимости сбрасывая буферы. В более ответственной программе вы должны удостовериться, что файл закрыт успешно.

Функция fclose() возвращает значение 0, сели файл был закрыт успешно, и EOF, если нет:

```
if (fclose(fp) != 0)
    printf("Ошибка при закрытии файла %s\n", argv[l]);
```

Функция fclose() может завершиться неудачно, если, например, жесткий диск заполнен, съемное устройство хранения извлечено или произошла ошибка ввода-вывода.

Указатели на стандартные файлы

Стандартный файл	Указатель файла	Обычное устройство
Стандартный ввод	stdin	Клавиатура
Стандартный вывод	stdout	Экран
Стандартный вывод ошибок	stderr	Экран

Все они имеют тип указателя на FILE, поэтому могут использоваться в качестве аргументов для стандартных функций ввода-вывода подобно fp в приведенном ранее примере.

Стандартный ввод-вывод

```c
#include <stdio.h>
#include <stdlib.h> // для exit ()
#include <string.h>
int main()
{
    FILE *in, *out; // объявление двух указателей на FILE
    int ch, count = 0;
    char name[100]; // хранилище для имени выходного файла
    if ((in = fopen("a.txt", "r")) == NULL)
    {
        fprintf(stderr, "Не удается открыть файл \"%s\"\n", "a.txt");
        exit(EXIT_FAILURE);
    } // настройка вывода
    if ((out = fopen("b.txt", "w")) == NULL)
    { // открытие файла для записи
        fprintf(stderr, "Не удается создать выходной файл.\n");
        exit(3);
    } // копирование данных
    while ((ch = getc(in)) != EOF)
        if (count++ % 3 == 0)
            putc(ch, out);                    // выводить каждый третий символ
    if (fclose(in) != 0 || fclose(out) != 0) // очистка
        fprintf(stderr, "Ошибка при закрытии файлов.\n");
    return 0;
}
```

Функции fprintf() и fscanf()

```c
#define MAX 41
int main(void)
{
    FILE *fp;
    char words[MAX];
    if ((fp = fopen("a.txt", "a+")) == NULL)
    {
        fprintf(stdout, "Не удается открыть файл \"a.txt\".\n");
        exit(EXIT_FAILURE);
    }
    puts("Введите слова для добавления в файл; для завершения");
    puts("введите символ # в начале строки.");
    while ((fscanf(stdin, "%40s", words) == 1) && (words[0] != '#'))
        fprintf(fp, "%s\n", words);
    puts("Содержимое файла:");
    rewind(fp); /* возврат в начало файла */
    while (fscanf(fp, "%s", words) == 1)
        puts(words);
    puts("Готово!");
    if (fclose(fp) != 0)
        fprintf(stderr, "Ошибка при закрытии файла\n");
    return 0;
}
```

Функции fgets() и fputs()

fgets()

- Первым аргументом является адрес (типа char *), где должны сохраняться введенные данные.
- Второй аргумент - целое число, представляющее максимальный размер входной строки.
- Заключительный аргумент — это указатель файла, который идентифицирует файл, подлежащий чтению.

Вызов функции выглядит следующим образом:

fgets(buf, STLEN, fp);

Здесь buf – это имя массива char, MAX – максимальный размер строки, а fp — указатель на FILE.

- fgets() добавляет завершающий нулевой символ
- Функция fgets() возвращает значение NULL, когда сталкивается с EOF.
- Функция fputs() принимает два аргумента: ***адрес строки*** и ***указатель файла.***

Вызов fputs() выглядит следующим образом:

fputs(buf, fp);

Функции fseek() и ftell()

```c
#define CNTL_Z '\032' /* маркер конца файла в текстовых файлах DOS */
#define SLEN 81
int main(void)
{
    char file[SLEN], ch;
    FILE *fp;
    long count, last;
    puts("Введите имя файла для обработки:");
    scanf("%80s", file);
    if ((fp = fopen(file, "rb")) == NULL)
    { /* режим только для чтения */
        printf("reverse не удается открыть %s\n", file);
        exit(EXIT_FAILURE);
    }
    fseek(fp, 0L, SEEK_END); /* перейти в конец файла */
    last = ftell(fp);
    for (count = 1L; count <= last; count++)
    {
        fseek(fp, -count, SEEK_END); /* двигаться в обратном направлении */
        ch = getc(fp);
        if (ch != CNTL_Z && ch != '\r') /* файлы MS-DOS */
            putchar(ch);
    }
    putchar('\n');
    fclose(fp);
    return 0;
```

```
// Введите имя файла для обработки:
// a.txt

// !niatirB taerG fo latipac a si nodnoL
```

Работа функции fseek()

Режим	Откуда измеряется смещение
SEEK_SET	От начала файла
SEEK_CUR	От текущей позиции
SEEK_END	От конца файла

Ниже приведены примеры вызова функции (fp — указатель файла):
fseek(fp, 0L, SEEK_SET); // перейти в начало файла
fseek (fp, 10L, SEEK_SET); // перейти на 10 байтов от начала файла
fseek (fp, 2L, SEEK_CUR); // перейти вперед на 2 байта от текущей позиции
fseek (fp, 0L, SEEK_END); // перейти в конец файла
fseek(fp, -10L, SEEK_END); // перейти назад на 10 байтов от конца файла

Значение, возвращаемое fseek(), равно 0, если все в порядке, и -1, если возникла ошибка вроде попытки выхода за границы файла.

Работа функций ftell()

Поскольку изначально функция ftell() была реализована в Unix, она указывает позицию в файле, возвращая количество байтов от начала файла, причем первый байт получает номер 0, второй — номер1 и т.д.

В ANSI C такое определение применяется к файлам, открытым в двоичном режиме, но не обязательно к файлам, открытым в текстовом режиме.

Пример:
last = ftell(fp); // присваивает last количество байтов от начала до конца файла.

Сравнение двоичного и текстового режимов

Многие редакторы MS-DOS помечают конец текстового файла посредством символа <Ctrl+Z>. Когда такой файл открывается в текстовом режиме, в C этот символ распознается как признак конца файла. Тем не менее, когда тот же самый файл открывается в двоичном режиме, <Ctrl+Z> является обычным символом в файле, а действительный признак конца файла появляется позже. Он может находиться сразу после <Ctrl+Z> или же файл может быть дополнен нулевыми символами, чтобы сделать размер файла кратным, скажем, 256. Нулевые символы в среде MS-DOS не отображаются, поэтому мы предусмотрели код, предотвращающий вывод символа <Ctrl+Z>.

О другом отличии мы упоминали ранее: символ новой строки текстового файла в MS-DOS представлен с помощью комбинации \r\n. Программа на C, открывающая тот же самый файл в текстовом режиме, “видит” символы \r\n как просто \n, но в случае применения двоичного режима программа видит оба символа, т.е. \r и \n.

Переносимость

В идеальном случае функции fseek() и ftell() должны соответствовать модели Unix. Тем не менее, отличия в реальных системах иногда делают это невозможным. По этой причине стандарт ANSI предоставляет для этих функций пониженные ожидания.

- В двоичном режиме реализации не обязаны поддерживать режим SEEK_END. Поэтому переносимость кода из примера не гарантируется. Более переносимый подход предусматривает чтение всего файла байт за байтом, пока не встретится конец. Но последовательное чтение файла для нахождения конца медленнее, чем просто переход в его конец. Директивы условной компиляции препроцессора C предлагают более систематизированный способ для поддержки выбора альтернативного кода.
- В текстовом режиме будут гарантированно работать только следующие вызовы fseek():

Вызов функции	Результат
fseek(file, 0L, SEEK_SET)	Перейти в начало файла
fseek(file, 0L, SEEK_CUR)	Оставаться в текущей позиции
fseek (file, 0L, SEEK_END)	Перейти в конец файла
fseek(file, ftell_pos, SEEK_SET)	Перейти в позицию ftell_pos от начала файла; ftellpos — это значение, возвращаемое функцией ftell ()

Функции fgetpos() и fsetpos()

Одна потенциальная проблема с функциями fseek() и ftell() заключается в том, что они ограничивают размеры файлов значениями, которые могут быть представлены типом long.

В ANSI C появились две новых функции позиционирования, которые с проектированы на работу с файлами крупных размеров. Вместо применения для представления позиции значения long, они используют новый тип под названием fpos_t (от file position type — тип для позиции в файле). Тип fpos_t не является фундаментальным, а определяется в терминах других типов.

В ANSI C определено, как применять тип fpos_t. Функция fgetpos() имеет следующий прототип:

int fgetpos(FILE * restrict stream, fpos_t * restrict pos);

Вызов fgetpos() помещает текущее значение тина fpos_t в ячейку, указанную pos; это значение описывает позицию в файле. Функция возвращает ноль в случае успеха и ненулевое значение при отказе.

Прототип функции fsetpos() выглядит так:

int fsetpos(FILE *stream, const fpos_t *pos);

Вызов fsetpos() приводит к использованию значения типа fpos_t из ячейки, заданной с помощью pos, для установки указателя файла в позицию, которую отражает это значение. Функция возвращает ноль в случае успеха и ненулевое значение при отказе.

"За кулисами" стандартного ввода-вывода

Обычно первым шагом в применении стандартного ввода-вывода является вызов функции fopen() для открытия файла.

- Функция fopen() не только открывает файл, но и настраивает буфер (или два буфера для режимов чтения-записи) и устанавливает структуру данных, содержащую сведения о файле и о буфере. Кроме того, fopen() возвращает указатель на эту структуру, так что другие функции знают, где ее искать.
- Эта структура данных обычно включает индикатор позиции в файле, предназначенный для определения текущей позиции в потоке. Она также содержит индикаторы для ошибок и конца файла, указатель на начало буфера, идентификатор файла и счетчик количества байтов, действительно скопированных в буфер.
- После инициализации структуры данных и буфера функция ввода читает запрошенные данные из буфера. В результате индикатор позиции устанавливается так, чтобы указывать на символ, следующий за последним прочитанным символом.
- После того, как функция прочитает последний символ финальной порции данных, она устанавливает индикатор конца файла в истинное значение. Следующий вызов любой функции ввода возвратит EOF.

Функция int ungetc(int c, FILE *fp)

Функция ungetc() заталкивает символ, указанный в c, обратно во входной поток.

Оператор	Входная очередь
Начальное состояние	с л а в н ы е _ д е л а
ch = getchar();	_ л а в н ы е _ д е л а
ungetc(ch, stdin);	с л а в н ы е _ д е л а

Функция int fflush()

Прототип fflush() выглядит так:
int fflush(FILE *fp);

- Вызов функции fflush() приводит к тому, что любые незаписанные данные в буфере вывода отправляются в выходной файл, идентифицируемый с помощью fp.
- Этот процесс называется сбросом буфера.
- Если fp — нулевой указатель, то сбрасываются все буферы вывода.
- Результат использования функции fflush() на входном потоке не определен.
- Ее можно применять с потоком обновления (для любого режим, а чтения-записи), при условии, что самая последняя операция, и с пользующая поток, не была операцией ввода.

Функция int setvbuf()

Прототип setvbuf() имеет следующий вид:
int setvbuf(FILE * restrict fp, char * restrict buf, int mode, size_t size);

Функция setvbuf() устанавливает альтернативный буфер, предназначенный для применения стандартными функциями ввода-вывода.
Она вызывается после того, как файл был открыт, и перед выполнением любой другой операции на потоке данных.
Указатель fp идентифицирует поток, а buf указывает на используемое хранилище. Значение buf, не равное NULL, говорит о том, что буфер вы создаете самостоятельно.
Аргумент size сообщает setvbuf() размер этого массива.
Для mode доступны следующие варианты:

- _IOFBF означает полиную буферизацию
- _IOLBF — построчную буферизацию
- _IONBF — отсутствие буферизации.

Предположим, что у вас есть программа, которая работает с сохраненными объектами данных, имеющими размер, скажем, по 3000 байтов каждый. Вы могли бы с помощью setvbuf () создать буфер, размер которого кратен размеру объекта данных.

Двоичный ввод-вывод: fread() и fwrite()

int num = 12345;

Сохраняет 12345 как двоичное число в num

00110000	00111001

fprintf(fp,"'%d", num);

Записывает в файл двоичные коды символов ’1’, '2', '3', '4', '5’

00110001	00111010	00110011	00110100	00110101

fwrite (&num, sizeof(int), 1, fp);

Записывает в файл двоичный код значения 12345

00110000	00111001

Функция size_t fwrite()

Ниже показан прототип функции fwrite ():
size_t fwrite(const void * restrict ptr, size_t size, size_t nmemb, FILE * restrict fp);

Функция fwrite () записывает двоичные данные в файл. Тип size_t определен в терминах стандартных типов C.
Указатель ptr — это адрес порции данных, предназначенной для записи.
Аргумент size представляет размер в байтах порции данных, подлежащих записи, а nmemb — количество таких порций.
Как обычно, fp идентифицирует файл, в который должна производиться запись.
Например, чтобы сохранить объект данных (такой как массив) размером 256 байтов, можно поступить так:

```
char buffer[256];
fwrite (buffer, 256, 1, fp) ;
```

Этот вызов fwrite() записывает одну порцию данных размером 256 байтов из буфера в файл. Чтобы сохранить, скажем, массив из 10 элементов double, понадобятся следующие операторы:

```
double earnings[10];
fwrite(earnings, sizeof (double), 10, fp) ;
```

Функция size_t fread()

Прототип функции fread() имеет следующий вид:
size_t fread(void * restrict ptr, size_t size, size_t nmemb, FILE *restrict fp);

Функция fread() принимает такой же набор аргументов, как и fwrite(). На этот раз ptr представляет собой адрес области памяти, куда помещаются данные, прочитанные из файла, а fp идентифицирует читаемый файл. Эту функцию следует использовать для чтения данных, которые были записаны в файл с помощью fwrite().
Например, вот как восстановить массив из 10 элементов double, сохраненный в предыдущем примере:

```
double earnings[10];
fread(earnings, sizeof (double), 10, fp) ;
```

Этот вызов копирует 10 значений размера double в массив earnings.
Функция fread() возвращает количество успешно прочитанных элементов. Обычно оно равно nmemb, однако может быть меньше, если произошла ошибка записи или был достигнут конец файла.

Функции int feof (FILE *fp) и int ferror (FILE * fp)

Когда стандартные функции ввода возвращают EOF, это обычно означает, что достигнут конец файла. Тем не менее, возврат EOF может также указывать на возникновение ошибки чтения.

Функции feof() и ferror() позволяют проводить различие между этими двумя возможностями.
Функция feof() возвращает ненулевое значение, если при последнем вызове функции ввода был обнаружен конец файла, и ноль в противном случае.
Функция ferror() возвращает ненулевое значение, если произошла ошибка чтения или записи, и ноль в противном случае.

Пример использования fread() и fwrite()

Мы примем первый подход, который предполагает выполнение перечисленных ниже действий.

- Запрос имени файла назначения и его открытие.
- Применение цикла для запроса исходных файлов.
- Поочередное открытие каждого исходного файла в режим е чтения и добавление его содержимого в конец файла назначения.

Следующий этап детализации связан с открытием файла назначения. Мы будем использовать следующие шаги.

- Открытие файла назначения в режиме добавления.
- Если сделать это не удается, то завершение работы.
- Установка буфера размером 4096 байтов для этого файла.
- Если сделать это не удается, то завершение работы.

Аналогично, мы можем уточнить часть программы, отвечающую за копирование, для чего выполнить с каждым файл о м такие действия.

- Если это файл назначения, то пропустить его и перейти к следующему файлу.
- Если файл не может быть открыт в режим е чтения, то пропустить его и перейти к следующему файлу.
- Добавить содержимое файла в файл назначения.

Пример использования fread() и fwrite() – демо ПК

```c
/* append.c -- добавление содержимого файлов в файл назначения */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define BUFSIZE 4096
#define SLEN 81
void append(FILE *source, FILE *dest);
char *s_gets(char *st, int n);
int main(void)
{
    FILE *fa, *fs;       // fa для файла назначения, fs для исходного файла
    int files = 0;       // количество добавляемых файлов
    char file_app[SLEN]; // имя файла назначения
    char file_src[SLEN]; // имя исходного файла
    int ch;
    puts("Введите имя файла назначения:");
    s_gets(file_app, SLEN);
    if ((fa = fopen(file_app, "a+")) == NULL)
    {
        fprintf(stderr, "Не удается открыть %s\n", file_app);
        exit(EXIT_FAILURE);
    }
    if (setvbuf(fa, NULL, _IOFBF, BUFSIZE) != 0)
    {
        fputs("Не удается создать выходной буфер\n", stderr);
        exit(EXIT_FAILURE);
    }

    puts("Введите имя первого исходного файла (или пустую строку для завершения): ");
    while (s_gets(file_src, SLEN) && file_src[0] != '\0')
    {
        if (strcmp(file_src, file_app) == 0)
            fputs("Добавить файл в конец самого себя невозможно\n", stderr);
        else if ((fs = fopen(file_src, "r")) == NULL)
            fprintf(stderr, "Не удается открыть %s\n", file_src);
        else
        {
            if (setvbuf(fs, NULL, _IOFBF, BUFSIZE) != 0)
            {
                fputs("Не удается создать входной буфер\n", stderr);
                continue;
            }
            append(fs, fa);
            if (ferror(fs) != 0)
                fprintf(stderr, "Ошибка при чтении файла %s.\n",
                        file_src);
            if (ferror(fa) != 0)
                fprintf(stderr, "Ошибка при записи файла %s.\n",
                        file_app);
            fclose(fs);
            files++;
            printf("Содержимое файла %s добавлено.\n", file_src);
            puts("Введите имя следующего файла (или пустую строку для завершения):");
        }
    }
    printf("Добавление завершено. Количество добавленных файлов: %d.\n", files);
    rewind(fa);
    printf("Содержимое %s:\n", file_app);
    while ((ch = getc(fa)) != EOF)
        putchar(ch);
    puts("Отображение завершено.");
    fclose(fa);
    return 0;
}
void append(FILE *source, FILE *dest)
{
    size_t bytes;
    static char temp[BUFSIZE]; // выделить память один раз
    while ((bytes = fread(temp, sizeof(char), BUFSIZE, source)) > 0)
        fwrite(temp, sizeof(char), bytes, dest);
}
char *s_gets(char *st, int n)
{
    char *ret_val;
    char *find;
    ret_val = fgets(st, n, stdin);
    if (ret_val)
    {
        find = strchr(st, '\n'); // поиск символа новой строки
        if (find)                // если адрес не является NULL,
            *find = '\0';        // поместить туда нулевой символ
        else
            while (getchar() != '\n')
                continue;
    }
    return ret_val;
}
```

Пример использования fread() и fwrite() – демо ПК

```c
/* randbin.c – произвольный доступ, двоичный ввод-вывод */
#include <stdio.h>
#include <stdlib.h>
#define ARSIZE 1000
int main()
{
    double numbers[ARSIZE];
    double value;
    const char *file = "numbers.dat";
    int i;
    long pos;
    FILE *iofile; // создание набора значений double
    for (i = 0; i < ARSIZE; i++)
        numbers[i] = 100.0 * i + 1.0 / (i + 1); // попытка открыть файл
    if ((iofile = fopen(file, "wb")) == NULL)
    {
        fprintf(stderr, "Не удается открыть файл %s для вывода.\n", file);
        exit(EXIT_FAILURE);
    }
    // запись в файл массива в двоичном формате
    fwrite(numbers, sizeof(double), ARSIZE, iofile);
    fclose(iofile);
    if ((iofile = fopen(file, "rb")) == NULL)
    {
        fprintf(stderr,
                "Не удается открыть файл %s для произвольного доступа.\n", file);
        exit(EXIT_FAILURE);
    }
    // чтение избранных элементов из файла
    printf("Введите индекс в диапазоне 0-%d.\n", ARSIZE - 1);
    while (scanf("%d", &i) == 1 && i >= 0 && i < ARSIZE)
    {
        pos = (long)i * sizeof(double); // вычисление смещения
        fseek(iofile, pos, SEEK_SET);   // переход в нужную позицию
        fread(&value, sizeof(double), 1, iofile);
        printf("По этому индексу находится значение %f.\n", value);
        printf("Введите следующий индекс (или значение за пределами диапазона для"
               " завершения): \n ");
    }
    // завершение
    fclose(iofile);
    puts("Программа завершена.");
    return 0;
}
// Введите индекс в диапазоне 0-999.
// 500
// По этому индексу находится значение 50000.001996.
// Введите следующий индекс (или значение за пределами диапазона для завершения):
//  900
// По этому индексу находится значение 90000.001110.
// Введите следующий индекс (или значение за пределами диапазона для завершения):
//  0
// По этому индексу находится значение 1.000000.
// Введите следующий индекс (или значение за пределами диапазона для завершения):
//  -1
// Программа завершена.
```

Адженда

Классы хранения, связывание и управление памятью

40 минут

Файловый ввод-вывод

30 минут

Структуры – начальные сведения

20 мин

Демо ПК: создание каталога книг

```c
/* book.c – каталог для одной книги */
#include <stdio.h>
#include <string.h>
char *s_gets(char *st, int n);
#define MAXTITL 41 /* максимальная длина названия + 1 */
#define MAXAUTL 31 /* максимальная длина имени автора + 1 */
struct book
{ /* шаблон структуры: дескриптором является book */
    char title[MAXTITL];
    char author[MAXAUTL];
    float value;
}; /* конец шаблона структуры */
int main(void)
{
    struct book library; /* объявление library в качестве переменной типа book */
    printf("Введите название книги.\n");
    s_gets(library.title, MAXTITL); /* доступ к разделу названия книги */
    printf("Теперь введите ФИО автора.\n");
    s_gets(library.author, MAXAUTL);
    printf("Теперь введите цену.\n");
    scanf("%f", &library.value);
    printf("%s авторства %s: $%.2f\n", library.title,
           library.author, library.value);
    printf("%s: \"%s\" ($%.2f)\n", library.author,
           library.title, library.value);
    printf("Готово\n");
    return 0;
}
char *s_gets(char *st, int n)
{
    char *ret_val;
    char *find;
    ret_val = fgets(st, n, stdin);
    if (ret_val)
    {
        find = strchr(st, '\n'); // поиск новой строки
        if (find)
            // если адрес не равен NULL,
            *find = '\0'; // поместить туда нулевой символ
        else
            while (getchar() != '\n')
                continue; // отбросить остаток строки
    }
    return ret_val;
}
// Введите название книги.
// Chicken of the Andes
// Теперь введите ФИО автора.
// Disrna Lapoult
// Теперь введите цену.
// 29.99
// Chicken of the Andes авторства Disrna Lapoult: $29.99
// Disrna Lapoult: "Chicken of the Andes" ($29.99)
// Готово
```

Объявление структуры

Объявление структуры выглядит следующим образом:

```
struct book {
    char title[MAXTITL];
    char author[MAXAUTL];
    float value;
};
```

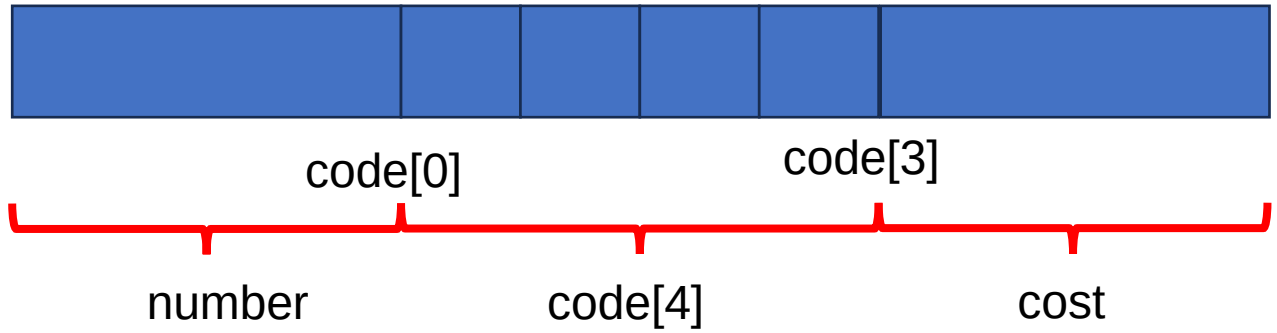
Это объявление описывает структуру, образованную из двух символьных массивов и одной переменной типа float. Оно не создает реальный объект данных, но определяет, из чего состоит такой объект.

Объявление переменной типа struct book:
struct book library;
Оно объявляет library как переменную типа структуры, которая использует схему структуры book.
Точка с запятой после закрывающей фигурной скобки завершает определение шаблона структуры.

Определение переменной типа структуры

Понятие структура применяется в двух смыслах. Одним из них является “схема структуры". Схема структуры сообщает компилятору, как представлять данные, но она не приводит к выделению пространства в памяти для этих данных. Следующий шаг заключается в создании переменной типа структуры, и в этом состоит второй смысл понятия.

```
struct stuff {
    int number;
    char code[4];
    float cost;
};
```




```
struct stuff p;
```

=

```
struct stuff {
    int number;
    char code[4];
    float cost;
} p;
```

Инициализация структуры

```
struct book library = {
    "The Pious Pirate and the Devious Damsel",
    "Renee Vivotte",
    1.95
 };
```

Как видите, используется список разделенных запятыми инициализаторов, заключенный в фигурные скобки. Тип каждого инициализатора должен соответствовать типу члена структуры, который он инициализирует.

При инициализации переменной со статической продолжительностью хранения должны использоваться константные значения. Это применимо также к структурам. Если вы инициализируете структуру со статической продолжительностью хранения, то значения в списке инициализаторов должны быть константными выражениями. Если продолжительность хранения является автоматической, значения в списке инициализаторов не обязательно должны быть константами.

Доступ к членам структуры

Как получить доступ к индивидуальным членам структуры?
Для этого служит операция членства в структуре — точка (.).

Например, library.value — это компонент value структуры library. Конструкцию library.value можно использовать подобно любой другой переменной типа float. Аналогично, library, title можно применять в точности как массив типа char.
По этой причине в приведенной выше программе используются такие выражения, как

```
s_gets(library.title, MAXTITL);
```

и

```
scanf("%f", &library.value);
```

Инициализаторы для структур

Стандарты C99 и C11 предоставляют назначенные инициализаторы для структур. Синтаксис похож на синтаксис назначенных инициализаторов для массивов. Однако назначенные инициализаторы для структур при идентификации конкретных членов используют операцию точки и имена членов, а не квадратные скобки и индексы. Например, чтобы инициализировать только член value структуры book, можно поступить так:

```
struct book surprise = {.value = 10.99};
```

Назначенные инициализаторы можно указывать в любом порядке:

```
 struct book gift = {.value = 25.99,
                     .author = "James Broadfool",
                     .title = "Rue for the Toad"};
```

Как и в случае массивов, обычный инициализатор, следующий за назначенным, присваивает значение члену, который следует за членом, указанным в назначенном инициализаторе. Кроме того, член получает значение, которое было предоставлено последним. Например:

```
 struct book gift= {.value = 18.90,
                    .author = "Philionna Pestle",
                    0.25};
```

Резюме

- Мы поговорили о типах переменных, их областях видимости
- Рассмотрели классы хранения
- Рассмотрели, что такое динамическая память
- Познакомились с ключевыми словами volatile и restrict
- Поработали с огромным количеством файловых функций
- Немного затронули структуры