

Управление памятью в языке C

Способы управления памятью в C:

- ❶ Статические переменные (`static`) — создаются и существуют на протяжении всего времени выполнения программы
- ❷ Автоматические переменные (`auto`) — создаются при входе в область видимости переменной и разрушаются при выходе
- ❸ **Динамические переменные** — требуемый объем памяти запрашивается во время работы программы; после использования — освобождается самим программистом

👉 в современных терминах это «управление памятью вручную»

Спецификатор класса памяти `static`

Указывается перед типом переменной: `static int x;`

- память выделяется на все время работы программы
- по умолчанию начальное значение ноль
- 👉 область видимости статической локальной переменной: внутри блока в котором она определена

`static` для глобальных переменных и функций

- все глобальные переменные «статические» по умолчанию
- 👉 служебное слово `static` перед глобальной переменной или перед функцией ограничивает их область видимости единицей трансляции («файлом»)
- `static` ставится в самом начале объявления:
`static int max(int x,int y) {...}`

Пример использования static переменных для инициализации

```
double log_gam(double x) {
    static int init_done = 0; // initialization block must be executed
    static double half_ln_2pi;
    static double b2k[9];
    // initialization block: calculate all 'constants'
    if( !init_done ) {
        init_done = 1; // flag that initialization is done
        half_ln_2pi = 0.5*log(2*M_PI);
        b2k[1] = 1./6.; b2k[2] = -1./30.; ...
        for(int k = 1; k < 9; k++) {b2k[k] /= (double)(2*k*(2*k-1));}
    }
    // compute the series: here we use calculated constants
    double sum = 0;
    for(int k = 1; k < 9; k++) sum += b2k[k] * ...
    return (x-0.5)*log(x) - x + half_ln_2pi + sum;
}
```

Спецификаторы auto и register

Декларация: `auto int x;`

- Служит для объявления автоматических локальных переменных
- Память выделяется во время работы программы: `run time`
- Переменные имеют неопределенное значение, до явного присваивания
- Спецификация `auto` бесполезна в языке C (и C++) так как все переменные объявленные внутри блока по умолчанию `auto`
 - ☞ в C++11 появилось `auto` для типа переменной, оно имеет другой смысл

Декларация: `register int i;`

- Имеет смысл: «прошу обеспечить доступ к локальной переменной так быстро, как только возможно»
 - ☞ компилятор волен игнорировать эту просьбу

Динамическое выделение памяти

Массив переменной длины: Variable Length Arrays (VLA)

- Массив размер которого вычисляется во время выполнения
- только для локальных массивов, память автоматически освобождается при выходе из блока
- ☞ нельзя использовать в `struct` и `union`
- не рекомендуется использовать для выделения больших объёмов памяти

Пример: open 'path/name' file using VLA

```
FILE* path_fopen(char* path, char* name, char* mode) {  
    char str[strlen(path) + strlen(name) + 2]; // str - VLA array  
    strcpy(str, path);    strcat(str, "/");    strcat(str, name);  
    return fopen(str, mode);  
}
```

☞ отсутствует в стандарте C++ и может отсутствовать в C11

Функции стандартной библиотеки C

`#include <stdlib.h>` — прототипы функций

- `malloc()`, `free()`, `calloc()` и `realloc()` — функции динамического распределения памяти, которые «всегда есть»
- выделяемая память находится в «свободной области» (`memory heap`)
- доступ к выделяемой памяти осуществляется через указатели

Основная функция выделения памяти:

```
void* malloc(size_t количество_байтов);
```

- `количество_байтов` — запрашиваемый объём памяти
- возвращает указатель `void*` на первый байт выделенной памяти или `NULL` если не может выполнить запрос

Указатель типа `void*` (обобщенный указатель)

Назначение `void*` `void_ptr`

- ✓ Хранит адрес переменной любого типа

```
int a = 1;  
void* ptr_void = &a;
```

- ✓ Легкое преобразование между `void*` и типизованными указателями

```
int* ptr_int = ptr_void;
```

- ✓ Арифметические действия с `void*` переменными запрещены

```
ptr_void++;           // should be compilation error  
ptr_int = ptr_void+1; // should be compilation error
```

👉 Компиляторы `gcc` и `clang` имеют нестандартное расширение в котором адресная арифметика с `void*` разрешена

👉 В C++ правила преобразования `void*` \rightarrow `something*` другие!

«Возврат» памяти операционной системе: `void free(void *ptr);`

- `ptr` – указатель на участок памяти, выделенный ранее с помощью `malloc()`

Пример: open ‘path/name’ file using `malloc, free`

```
FILE* PathFopen(char* path, char* name, char* mode) {  
    char* str = malloc(strlen (path) + strlen (name) + 2);  
    if(!str) {  
        printf("ERROR in %s:memory can not be allocated!\n", __func__);  
        exit(EXIT_FAILURE);  
    }  
    strcpy(str, path); strcat(str, "/"); strcat(str, name);  
    FILE* ret = fopen(str, mode);  
    free(str);    // releases the block of memory  
    return ret;  
}
```


Функции `realloc()` и `calloc()`

```
void* realloc(void* ptr, size_t количество_байтов);
```

- функция служит для изменения размера **ранее выделенной памяти** на которую указывает `ptr`
- если `ptr == NULL`, то функция работает так же как `malloc()`
- после вызова `realloc()` `ptr` указывает на «старую» память и становится «висячим» указателем (**a dangling pointer**)

```
void* calloc(size_t число_эл-ов, size_t размер_элемента);
```

Функция «обёртка» вокруг `malloc()`:

- 1 размер памяти задается двумя параметрами
- 2 все байты выделенной памяти заполняются нулями

Характерные ошибки

Использование памяти после `free()`

```
free(ptr);           // ptr now becomes a dangling pointer
*ptr = 10;           // ERROR Undefined behavior
free(ptr);           // ERROR Double-free
```

Специальное значение для указателей: `NULL`

```
#include <stdlib.h> // здесь определено NULL
int* ptr = NULL;    // показывает, что указатель не соответствует
                    // никакому реальному объекту
```

«`NULL` как предохранитель»:

```
free(ptr);
ptr = NULL;    // defensive style
free(ptr);     // it is OK now
*ptr = 10;    // an immediate crash
```

`free()` используется с «неправильным» указателем

```
char* msg = "Default message";
int tbl[100];
int* ptr = malloc(100*sizeof(int));
...
tbl[0] = *ptr++; // incrementing ptr
free(ptr);       // ERROR: Undefined behavior
...
free(msg);      // runtime error
free(tbl);     // Segmentation fault
```

Правила:

- Применяйте `free()` и `realloc` только к указателям полученным от функций `malloc()`, `calloc()`, `realloc()`
- Сохраняйте или не меняйте указатель возвращаемый функциями `malloc()`, `calloc()`, `realloc()`

☞ Проверьте, что память выделена успешно

```
char* ptr = NULL;
size_t huge = 1024*1024*1024; // 1GB
for(i = 0; i < 10; i++) {
    ptr = malloc(huge);
    /* check ptr? */
    ptr[0] = i;
}
```

Segmentation fault

Добавляем проверку:

```
ptr = malloc(huge);
if( !ptr ) {
    printf("Memory can not be allocated! i=%d\n",i);
    exit(EXIT_FAILURE);
}
```

Memory can not be allocated! i=2

Утечка памяти (memory leaks)

Пример утечки памяти

```
int* ptr = NULL;
for(int i = 0; i < 1000; i++) {
    ptr = malloc(1024*sizeof(int));
}
free(ptr);
```

	memstat -p PID
до цикла	320k
после free(ptr)	4280k

- 👉 Память, которую забывают вернуть в систему, выходит из обращения, что приводит к уменьшению ресурсов всей системы
- 👉 При завершении программы, все захваченные ресурсы возвращаются в систему

Функция `memset()`

```
#include <string.h> // заголовочный файл
```

```
void * memset(void * ptr, int c, size_t n);
```

- заполняет блок памяти размером `n` байт начиная с `ptr` символом `c`
- возвращает указатель на начало блока `ptr`
- поведение неопределено при выходе за пределы `ptr`-массива или если `ptr=0`

Функция `memset()` часто используется для обнуления массивов

```
int ibuf[10];  
memset(ibuf,0,10*sizeof(ibuf[0])); // zeroing int-array  
double buf[10];  
memset(buf,0,10*sizeof(buf[0])); // zeroing double-array
```

Библиотека файлового ввода-вывода

```
#include <stdio.h> // header for standard input/output library
```

Логика работы с файлами в С

- 1 Файл «открывается»: создаётся универсальное устройство ввода-вывода называемое *поток (stream)*
 - 2 Производится обмен информацией между программой и файлом: с точки зрения программы происходит чтение/запись данных из потока
 - 3 Файл «закрывается»:
 - 👉 при нормальном завершении (или `exit()`) происходит корректное закрытие файлов
 - 👉 при аварийном завершении (или `abort()`) информация может быть потеряна
- 👉 В языке С файл это то что описывается «моделью потока»: дисковый файл, дисплей, клавиатура, принтер ...

Потоки

- **Бинарный поток** — поток данных «как есть», без изменений
- **Текстовый поток** — поток символов собранных в «строки»: `\n` в конце строки
 - ☞ в текстовом потоке может происходить преобразование некоторых СИМВОЛОВ

При работе с файлами используют указатель на структуру типа `FILE`:

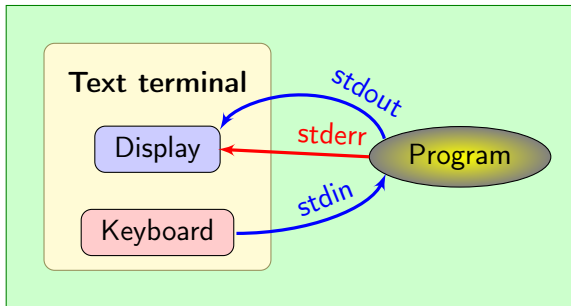
```
#include <stdio.h>
```

```
FILE * fp;
```

- структура `FILE` содержит полную информацию о потоке, но использовать нужно только указатель `FILE*`
 - ☞ Никогда ничего не меняйте в этой структуре

Стандартные потоки создаются автоматически:

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



Концепция буферизации

- **Небуферизованный поток** – запись/чтение производится символ за символом
- **Построчная буферизация** – запись/чтение в буфер, передача из буфера по приходу символа перевода строки
- **Полная буферизация** – запись/чтение в файл тогда, когда буфер полностью заполнен; размер буфера выбирается «вручную» исходя из задачи

Стандартные потоки:

- **stdin** – всегда буферизован
- **stderr** – всегда небуферизован
- **stdout** – при выводе на терминал буферизован построчно, иначе до полного заполнения буфера

Открытие файла: `fopen()`

```
FILE * fopen(const char* fname, const char* mode)
```

- открывает файл с именем `fname` и связывает с ним поток
- функция возвращает `NULL` если открыть файл не удалось
- параметр `mode` задает режим в котором файл будет открыт:

<code>mode</code>	«режим» работы с файлом
-------------------	-------------------------

r	открыть текстовый файл для чтения
----------	-----------------------------------

w	создать текстовый файл для записи или переписать если уже существует
----------	--

a	дописать в конец текстового файла
----------	-----------------------------------


+	открыть файл для обновления (чтение/запись); ставится после r,w,a
----------	--

b	создать бинарный поток; ставится после r,w,a или r+,w+,a+
----------	---

Закрытие файла: `fclose()`

```
int fclose(FILE * fp)
```

- `fclose()` закрывает поток, где `fp` – указатель полученный от `fopen()`
- все данные из буфера записываются в файл
- в случае успеха возвращает ноль
- при ошибке возвращает `EOF` (end-of-file); причину ошибки можно выяснить с помощью функции `ferror()`

 Рекомендуется закрывать ненужные потоки, так как количество одновременно открытых файлов в системе ограничено

Чтение-запись одного символа: `getc()`, `putc()`

```
int getc(FILE* fp);           // Чтение символа  
int putc(int ch, FILE* fp);  // Запись символа
```

- `fp` – указатель полученный от `fopen()`
- `ch` – записываемый символ (`putc`)
- функции возвращают **целое значение** считанного/записанного символа
- в случае ошибки возвращают `EOF`
- `getc()` так же возвращает `EOF` по достижению конца файла, более точно о причине `EOF` дает функция `feof()`

👉 `putchar(c)` то же самое, что и `putc(c, stdout)`

👉 `getchar()` то же самое, что и `getc(stdin)`

Функции семейства printf() и scanf()

① форматный ввод-вывод в файл

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

② форматный ввод-вывод в C-string

```
int sprintf(char *buf, const char *format, ...);  
int snprintf(char *buf, size_t n, const char *format, ...);  
int sscanf(const char *buf, const char *format, ...);
```

Функции возвращают:

- **семейство-printf** : число напечатанных символов (исключая `\0` для C-string) или отрицательное значение в случае ошибки
- **семейство-scanf**: число успешно «прочитанных» аргументов или `EOF`, если конец ввода случился **перед первым аргументом**

Неформатированный ввод-вывод блоков данных

Функции `fread`, `fwrite`

```
size_t fread(void *ptr, size_t size, size_t nobj, FILE *stream);  
size_t fwrite(const void *ptr, size_t size, size_t nobj, FILE *stream);
```

- `ptr` – указатель на область памяти, например массив, которая записывается `fwrite()` или считывается `fread()` из файла
- `nobj` – число элементов в буфере
- `size` – размер одного элемента в байтах

Функции возвращают:

- количество прочитанных / записанных элементов
- 📌 `size_t` – беззнаковый целый тип для задания размеров объектов
- Печать: `printf('%zu\n', sizeof(int));`

```
double d=12.23; int i=101; long l=123023L; // variables to write/read

FILE* fp = fopen("test.dat", "wb+");
if ( !fp ) {
    perror("Error open file"); // возможно: Error... Permission denied
    exit(EXIT_FAILURE);
}

// запись бинарных данных в файл
fwrite(&d, sizeof(d), 1, fp);
fwrite(&i, sizeof(i), 1, fp);
fwrite(&l, sizeof(l), 1, fp);

// чтение данных из бинарного файла
rewind(fp); // sets the file position to the beginning of the file
fread(&d, sizeof(d), 1, fp);
fread(&i, sizeof(i), 1, fp);
fread(&l, sizeof(l), 1, fp);
printf("%f %d %ld\n", d, i, l); // 12.230000 101 123023
fclose(fp);
```


Работа с ошибками ввода-вывода

```
int ferror(FILE * stream)
```

- возвращает ненулевое значение (true) если в `stream` произошла ошибка

Пример: `if(ferror(fp)) perror("Error in file");`

```
void perror(const char* str)
```

- печатает в `stderr` строку содержащую системное сообщение об ошибке произошедшей в функции `stdlib`; `str` – дополнительное сообщение

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

- возвращает ненулевое значение (true) если дошли до конца `stream`

Пример: `if(feof(fp)) {fclose(fp);}`

```
void clearerr(FILE * stream)
```

- обнуляет индикатор ошибок и индикатор `EOF` для потока `stream`

Дополнительные слайды

Спецификатор restrict для указателей

Передача указателей в функцию:

```
// Должен ли компилятор предусмотреть случай a=b=c?  
void fun(int* a, int* b, int* c) {  
    *a += *c;  
    *b += *c;  
}  
  
int x = 10;  
fun(&x, &x, &x); // x = 40
```

👉 Если всегда $a \neq b \neq c$, то можно значительно ускорить вычисления, но как сказать об этом компилятору?

```
void fun(int* restrict a, int* restrict b, int* restrict c)
```

- «гарантия», что **a, b, c** указывают на непересекающиеся блоки памяти
- 👉 компилятор может создать более эффективный код

Функции `memcpy()`, `memmove()`

```
#include <string.h> // заголовочный файл
```

```
void* memcpy(void * restrict dest, const void * restrict src, size_t n);  
void* memmove(void * dest, const void * src, size_t n);
```

- обе функции копируют `n` байт из `src` в `dest`
- `memcpy()` работает только с непересекающимися блоками памяти

```
double src[10] = {1,2,3,4,5,6,7,8,9,10};  
double dest[100];  
memcpy(dest,src,10*sizeof(src[0])); // copy src to dest  
  
memmove(&src[4],src,5*sizeof(src[0])); // overlapping memory  
for ( int i = 0; i < 10; i++ ) {  
    printf("%.1f ",src[i]); // 1.0 2.0 3.0 4.0 1.0 2.0 3.0 4.0 5.0 10.0  
}
```

Проверка на ошибки в функциях stdlib

Внешняя переменная: `volatile int errno;`

`#include <errno.h>`

- в начале работы программы ноль: `errno=0`
- номер ошибки произошедшей при вызове библиотечной функции

Функция: `char * strerror(int errno);`

`#include <string.h>`

- возвращает строку, содержащую системное сообщение об ошибке

Функция: `void perror(const char* str);`

`#include <stdio.h>`

- преобразует `errno` в строку и выводит её в `stderr`
- `str` – дополнительное сообщение

Пример `errno` для функций математической библиотеке

```
#include <math.h>
#include <errno.h>
#include <stdio.h>
#include <string.h>

int main() {
    printf("ini. errno = %d\n",errno); // ini. errno = 0
    double a = -1.2;
    double b = sqrt(a);
    if( errno != 0 ) {
        printf("b= %f errno = %d\n",b,errno); // b= -nan errno = 33
        printf("strerror msg: %s\n",strerror(errno));
        Output> strerror msg: Numerical argument out of domain
        perror("perror msg");
        Output> perror msg: Numerical argument out of domain
    }
}
```