

Управление памятью в языке C

Способы управления памятью:

- 1 Статические переменные (`static`) — создаются и существуют на протяжении всего времени выполнения программы
- 2 Автоматические переменные (`auto`) — создаются при входе в область видимости переменной и разрушаются при выходе
- 3 Динамические переменные — требуемый объем памяти запрашивается во время работы программы;
после использования память освобождается самим программистом — в современных терминах это «управление памятью вручную»
- 4 В C++ имеются «умные указатели» (`smart pointers`), реально работают начиная с C++11

Спецификатор класса памяти `static`

1 Для локальных переменных


- `static` ставится в самом начале объявления:
`static int var; // по умолчанию начальное значение ноль`
- память выделяется **на все время работы программы**
- область видимости статической локальной переменной: внутри блока в котором она определена

2 Для глобальных переменных и функций

- все глобальные переменные статические по умолчанию
- служебное слово `static` перед глобальной переменной или функцией ограничивает их область видимости единицей трансляции («файлом»)

Пример использования static переменных для инициализации

```
double log_gamma_function(double x) {
    static int init_done = 0; // initialization block must be executed
    static double half_ln_2pi;
    static double b2k[9];
    if( !init_done ) { // initialization block: calculate 'constants'
        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;
}
```

 начиная с C23 и в C++11 можно использовать `constexpr` для вычисления констант на стадии компиляции

Константные выражения constexpr в C23 и C++

1 ВЫЧИСЛЕНИЕ КОНСТАНТ НА СТАДИИ КОМПИЛЯЦИИ (C23 и C++11)

```
constexpr double half_ln_2pi = 0.5*log(2*M_PI); // no guarante: gcc OK
printf("C23: half_ln_2pi= %f\n",half_ln_2pi); // 0.918939
constexpr double bernoulli2k[] = {1., 1./6, -1./30, 1./42 };
printf("C23: B_2= %f\n",bernoulli2k[1]); // 0.166667

// constexpr int i2pi = (int)2*M_PI; // C23 ERROR: is not an integer; OK in C++

// C23 no constexpr pointers except NULL (C++ it's OK)
static int ib = 23;
// constexpr int* pb = &ib; // C23 ERROR: is not null

// constexpr can be used for struct and union
struct bernoulli { double k0, k2, k4, k6; };
constexpr struct bernoulli B = {1., 1./6, -1./30, 1./42 };
constexpr double B_4 = B.k4; // only '.' operator may be used
printf("C23: B_4= %f\n",B_4); // -0.033333
```

2 constexpr функции (только в C++)

```
constexpr unsigned long Factorial(unsigned int n) {  
    unsigned long result = 1UL;  
    for ( ; n>1; --n ) { result *=n; }  
    return result;  
}  
  
constexpr int F7 = Factorial(7);           // compile time  
cout << "F7= " << F7 << endl;             // F7= 5040  
volatile int b = 8;                        // disallow optimization  
cout << b << "!=" << Factorial(b) << endl; // 8!=40320 - run time
```

- в C23 constexpr функции не поддерживаются
- в C++ constexpr функция может вызываться как обычная функция
- constexpr подразумевает const
- вычисления в constexpr должны быть «достаточно простыми»

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 языках C и C++ так как все переменные объявленные внутри блока по умолчанию **auto**

👉 в C++11 и C23 появилось **auto** для определения типа переменной (type inference), например: `auto N = 10;`

Декларация: `register int i`

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

👉 в C++ объявлено устаревшим, а в C++17 удалено полностью

Динамическое выделение памяти

Массив переменной длины: Variable Length Arrays (VLA)

- локальный массив, размер которого вычисляется во время выполнения
- память автоматически освобождается при выходе из блока
- не рекомендуется использовать для выделения больших объёмов памяти

🚫 **нельзя использовать в `struct` и `union`**

Пример: open ‘path/file_name’ 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);  
}
```

🚫 может отсутствовать в C11 и выше (макрос `__STDC_NO_VLA__`)

🚫 совсем нет в стандарте C++ (`gcc`, `clang` поддерживают)

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*` или 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
```

👉 Компиляторы `gcc` и `clang` имеют нестандартное расширение в котором `sizeof(void)=1` и поэтому адресная арифметика с `void*` разрешена

👉 В C++ правила преобразования `void* -> some_ptr*` более строгие

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

- `ptr` – указатель, **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

❶ 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;    // указатель ptr ни на что не указывает
```

👉 В C23 и C++11 имеется специальная константа «нулевой указатель» `nullptr`, эта константа имеет специальный тип `nullptr_t`

«`NULL` как предохранитель»:

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

Нулевой указатель `nullptr` в C++ и C23

```
C:          int* ptr = NULL;    // #define NULL ((void*)0)
                                   // but may be #define NULL 0
C++:        int* ptr = 0;      // instead of NULL

C23,C++11:  int* ptr = nullptr; // with special type:
                                   // nullptr_t, std::nullptr_t
```

🔔 `nullptr` неявно преобразуется в любой тип указателя, но не в `int`

Зачем `nullptr` нужен в C++?

```
void func(int n)
void func(char* s)  // two overloaded func() in C++
...
func(0);            // guess which function gets called? ... func(int)
func(nullptr);      // no doubt: C++11
func((char*) 0);    // no doubt: C++98
```

🔔 В C23 `nullptr` для лучшей совместимости с C++

2 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()`

3 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);
```

	<code>memstat -p PID</code>
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 начиная с `ptr` символом `c`, первые `n` байт
- возвращает указатель на начало блока `ptr`
- поведение неопределенно при выходе за пределы `ptr`-массива или если `ptr=0`

Пример: обнуление 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
```

Спецификатор restrict для указателей

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

☞ Если всегда $a \neq b \neq c$, то можно 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
- ☞ в стандарте C++ **restrict** отсутствует

Функции `memcpy()`, `memmove()`

```
#include <string.h> // заголовочный файл, <cstring> в C++
```

```
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 с непересекающимися блоками памяти (и в C++)

```
double src[] = {1,2,3,4,5,6,7,8,9};  
double dest[100];  
memcpy(dest,src,9*sizeof(src[0])); // copy src to dest  
memmove(&src[3],src,6*sizeof(src[0])); // overlapping memory  
for ( int i = 0; i < 9; i++ ) {  
    printf("%.1f ",src[i]); // 1.0 2.0 3.0 1.0 2.0 3.0 4.0 5.0 6.0  
}  
printf("\n");
```

Библиотека файлового ввода-вывода

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

Логика работы с файлами в C

- 1 Файл «открывается»: создаётся универсальное устройство ввода-вывода называемое *поток (stream)*
 - 2 Производится обмен информацией между программой и файлом: чтение/запись данных из потока, в поток
 - 3 Файл «закрывается»:
 - ✓ при нормальном завершении (или `exit()`) происходит корректное закрытие файлов
 - ✓ при аварийном завершении (или `abort()`) информация может быть потеряна
-  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`):
может происходить преобразование некоторых символов

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

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

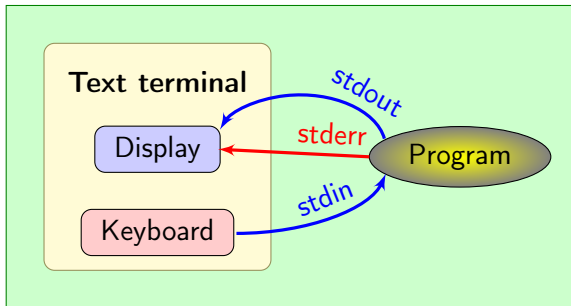
```
FILE * fp;
```

- 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+
----------	---

Заккрытие 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**

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

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, при ошибке это число меньше чем `nobj`

```
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);
```

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
    }
}
```