

**Курс «Алгоритмы и алгоритмические языки»
1 семестр 2017/2018**

Лекция 12

Динамическое распределение памяти

◆ Функция

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

выделяет область памяти размером `size` байтов и возвращает указатель на выделенную область памяти.

Если память не выделена (например, в системе не осталось свободной памяти требуемого размера), возвращаемый указатель имеет значение `NULL`.

◆ Поскольку результат операции `sizeof` имеет тип `size_t` и равен длине операнда в байтах, в качестве `size` можно использовать результат операции `sizeof`.

◆ Тривиальные примеры:

(1) Выделение непрерывного участка памяти объемом 1000 байтов:

```
char *p;  
p = (char *) malloc (1000 * sizeof (char));
```

(2) Выделение памяти для 50 целых:

```
int *p;  
/* явное приведение типа необязательно */  
p = malloc (50 * sizeof (int));
```

Динамическое распределение памяти

◆ Функция

```
void free (void *p) ;
```

возвращает системе выделенный ранее участок памяти с указателем `p`.

◆ **Важное замечание:** Аргументом функции `free ()` обязательно должен быть указатель `p` на участок памяти, выделенной ранее функцией `malloc ()`.

Вызов функции `free ()` с неправильным указателем не определен и может привести к разрушению системы распределения памяти.

Вызов функции `free ()` с указателем `NULL` не приводит ни к каким действиям (C99).

Обращение к освобожденному указателю не определено.

◆ Функции `malloc ()` и `free ()` входят в состав библиотеки `stdlib.h`.

Динамическое распределение памяти

◇ *Пример.* Динамическое выделение памяти для строки:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main (void) {
    char *s;
    int t;

    s = (char *) malloc (80 * sizeof (char));
    if (!s) {
        fprintf (stderr, "требуемая память не выделена.\n");
        return 1; /* исключительная ситуация */
    }
    fgets (s, 80, stdin); s[strlen (s) - 1] = '\\0';
    /* посимвольный вывод перевернутой строки на экран */
    for (t = strlen(s) - 1; t >= 0; t--)
        putchar (s[t]);
    free (s);
    return 0;
}
```

Динамическое распределение памяти

- ◇ *Пример.* Динамическое выделение памяти для двумерного целочисленного массива (матрицы):

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

long pwr (int a, int b) {
    long t = 1;
    for (; b; b--)
        t *= a;
    return t;
}
```

Динамическое распределение памяти

◇ *Пример.* Динамическое выделение памяти для двухмерного целочисленного массива (матрицы):

```
int main (void) {
    long *p[6]; int i, j;
    for (i = 0; i < 6; i++)
        if (!(p[i] = malloc (4 * sizeof (long)))) {
            printf ("требуемая память не выделена. \n");
            exit(1);
        }
    for (i = 1; i < 7; i++)
        for (j = 1; j < 5; j++)
            p[i - 1][j - 1] = pwr (i, j);
    for (i = 1; i < 7; i++) {
        for (j = 1; j < 5; j++)
            printf ("%10ld ", p[i - 1][j - 1]);
        printf ("\n");
    }
    for (i = 0; i < 6; i++)
        free (p[i]);
    return 0;
}
```

VLA-массивы

- ◇ В Си-89 размер массива обязан являться константой. Это неудобно при передаче массивов (многомерных) в функции:

```
/* можно передать int a[5]; int a[42]; ... */
```

```
int asum1d (int a[], int n) {
```

```
    int s = 0;
```

```
    for (int i = 0; i < n; i++)
```

```
        s += a[i];
```

```
    return s;
```

```
}
```

```
/* можно передать только int a[???][5] */
```

```
int asum2d (int a[][5], int n) {
```

```
    int s = 0;
```

```
    for (int i = 0; i < n; i++)
```

```
        for (int j = 0; j < 5; j++)
```

```
            s += a[i][j];
```

```
    return s;
```

```
}
```

VLA-массивы

- ◇ В Си-99 размер массива автоматического класса памяти может задаваться во время выполнения программы (Си-11 сделал VLA **необязательными**, проверка через макрос `__STDC_NO_VLA__`):

```
int foo (int n) {
    int a[n];
    <... Можно обрабатывать a[i]...>
}

/* можно передать int a[???][???] */
int asum2d (int m, int n, int a[m][n]) {
    int s = 0;
    for (int i = 0; i < m; i++)
        for (int j = 0; j < n; j++)
            s += a[i][j];
    return s;
}
```

- ◇ Объявление функции `asum2d`:

```
int asum2d (int m, int n, int a[m][n]);
int asum2d (int, int, int [*][*]);
```

VLA-массивы и динамическое выделение памяти

- ◆ Функция `asum2d` может использоваться с VLA-массивами, но они всегда выделяются в автоматической памяти:

```
int foo (int m, int n) {  
    int a[m][n]; int s;  
    <... Считаем a[i][j]...>  
    s = asum2d (m, n, a);  
}
```

- ◆ Можно выделить VLA-массив в динамической памяти:

```
int main (void) {  
    int m, n;  
    scanf ("%d%d", &m, &n);  
    int (*pa)[n];  
    pa = (int (*)[n]) malloc (m * n * sizeof (int));  
    <... Считаем pa[i][j]...>  
    s = asum2d (m, n, pa);  
    free (pa);  
}
```

Динамическое распределение памяти

- ◇ Состав функций динамического распределения памяти библиотеки `stdlib` (заголовочный файл `<stdlib.h>`)

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

```
void free (void *p);
```

```
void *realloc (void *p, size_t size);
```

```
void *calloc(size_t num, size_t size);
```

- ◇ Функция

```
void *calloc (size_t num, size_t size)
```

работает аналогично функции `malloc (size1)`,

где `size1 = num * size` (т.е. выделяет память для размещения массива из `num` объектов размера `size`).

Выделенная память инициализируется нулевыми значениями

Динамическое распределение памяти



Функция

```
void *realloc (void *p, size_t size)
```

согласно стандарту Си99 сначала выполняет `free (p)`,

а потом `p = malloc (size)`, возвращая новое значение

указателя `p`. При этом значения первых `size` байтов новой и старой областей совпадают.

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

```
#include <stdlib.h>
```

```
int main(void) {
```

```
    int *p = (int*)malloc (sizeof(int));
```

```
    int *q = (int*)realloc (p, sizeof(int));
```

```
    *p = 1;
```

```
    *q = 2;
```

```
    if (p == q)
```

```
        printf ("%d %d\n", *p, *q);
```

```
    return 0;
```

```
}
```

```
$ clang -O2 realloc.c && ./a.out
```

```
1 2
```

Динамическое распределение памяти



Функция

```
void *realloc (void *p, size_t size)
```

согласно стандарту Си99 сначала выполняет `free (p)`,

а потом `p = malloc (size)`, возвращая новое значение

указателя `p`. При этом значения первых `size` байтов новой и старой областей совпадают.

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

```
#include <stdlib.h>
```

```
int main(void) {
```

```
    int *p = (int*)malloc (sizeof(int));
```

```
    int *q = (int*)realloc (p, sizeof(int));
```

```
    *p = 1;
```

```
    *q = 2;
```

```
    if (p == q)
```

```
        printf ("%d %d\n", *p, *q);
```

```
    return 0;
```

```
}
```

```
$ clang -O2 realloc.c && ./a.out
```

```
1 2
```

Массив переменного размера в структуре (C99)

- ◆ Flexible array member – последнее поле структуры:

```
struct polygon {  
    int np; /* число вершин */  
    struct point points[];  
}
```

- ◆ Варьирование размера переменного массива:

```
int np; struct polygon *pp;  
scanf ("%d", &np);  
pp = malloc (sizeof (struct polygon)  
            + np * sizeof (struct point));  
pp->np = np;  
for (int i = 0; i < np; i++)  
    scanf ("%d%d", &pp->points[i].x,  
          &pp->points[i].y);
```

Отладка программ

- ◇ Все программы содержат ошибки, отладка – это процесс поиска и удаления некоторых ошибок
- ◇ Существуют другие методы обнаружения ошибок (тестирование, верификация, статические и динамические анализаторы кода), но их применение не гарантирует отсутствия ошибок
- ◇ Для отладки используют инструменты, позволяющие получить информацию о поведении программы на некоторых входных данных, не изменяя ее поведения
- ◇ Простейший метод: *отладочная печать*

```
int *a; int n;
n = read_array (a);
debug_array (a, n);

static void debug_array (int *a, int n) {
    fprintf (stderr, "Array (%d)", n);
    for (int i = 0; i < n; i++)
        fprintf (stderr, "%d ", a[i]);
    fprintf (stderr, "\n");
}
```

- ◇ Отладочная печать может контролироваться макросом (**NDEBUG**) 14

Отладка программ: отладчики

- ◇ Отладчик – основной инструмент отладки программы
- ◇ Отладчик позволяет
 - ◆ запустить программу для заданных входных данных
 - ◆ останавливать выполнение по достижении заданных точек программы безусловно или при выполнении некоторого условия на значения переменных
 - ◆ останавливать выполнение, когда некоторая переменная изменяет свое значение
 - ◆ выполнить текущую строку исходного кода программы и снова остановить выполнение
 - ◆ посмотреть/изменить значения переменных, памяти
 - ◆ посмотреть текущий стек вызовов
- ◇ Необходимое условие для отладки на уровне исходного кода: наличие в исполняемом файле программы *отладочной информации* (связи между командами процессора и строками исходного кода программы, связь между адресами и переменными и т.д.)

Отладка программ: отладчик `gdb`

- ◆ Компиляция с отладочной информацией: `gcc -g`
- ◆ Некоторые команды `gdb`
 - ◆ `gdb <file> --args <args>` – загрузить программу с заданными параметрами командной строки
 - ◆ `run/continue` – запустить/продолжить выполнение
 - ◆ `break <function name/file:line number>` – завести безусловную *точку останова*
 - ◆ `cond <bp#> condition` – задать условие остановки выполнения для некоторой точки останова
 - ◆ `watch <variable/address>` – задать *точку наблюдения* (остановка выполнения при изменении значения переменной или памяти по адресу)
 - ◆ `next/step` – выполнить текущую строку исходного кода программы без захода/с заходом в вызываемые функции
 - ◆ `print <var>/set <var> = expression` – посмотреть /изменить текущие значения переменных, памяти
 - ◆ `bt` – посмотреть текущий стек вызовов
- ◆ Среда Code::Blocks поддерживает `gdb` в своем интерфейсе

Отладка программ: примеры команд gdb

◆ Установка точек останова

◆ МОЖНО ИСПОЛЬЗОВАТЬ '.' ВМЕСТО '->'

```
b fancy_abort
```

```
b 7199
```

```
b sel-sched.c:7199
```

```
cond 2 insn.u.fld.rt_int == 112
```

```
cond 3 x_rtl->emit.x_cur_insn_uid == 1396
```

◆ Просмотр и изменение значений переменных

```
p orig_ops.u.expr.history_of_changes.base
```

```
p bb->index
```

```
set sched_verbose=5
```

```
call debug_vinsn (0x4744540)
```

◆ Установка точек наблюдения

```
wa can_issue_more
```

```
wa ((basic_block) 0x7ffff58b5680)->preds.base.prefix.num
```

Поиск ошибок работы с памятью

- ❖ Частые ошибки работы с динамической памятью тяжело отлаживать (даже в небольших программах)
- ❖ Ошибки доступа за границы буфера
Ошибки использования неинициализированного или уже освобожденного указателя
Недостаточный размер буфера
- ❖ Разработан ряд инструментов анализа, которые облегчают жизнь программисту
- ❖ valgrind: динамический двоичный транслятор (<http://valgrind.org>)
- ❖ sanitizers: компиляторная инструментация от Google <https://github.com/google/sanitizers/wiki>
- ❖ Disclaimer: Linux-only tools



Поиск ошибок работы с памятью

◇ valgrind: динамический двоичный транслятор
(плюс набор инструментов, ваш – memcheck)

```
#include <stdlib.h>
void f(void) {
    int* x = malloc(10 * sizeof(int));
    x[10] = 0;          // problem 1: heap block overrun
}
int main(void) {
    f();
    return 0;
}
```

```
$ gcc -Og -g -o me && valgrind ./me
```

```
==27164== Memcheck, a memory error detector
```

```
==27164== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
```

```
==27164== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
```

```
==27164== Command: ./me
```

```
==27164==
```

```
==27164== Invalid write of size 4
```

```
==27164==    at 0x400554: f (me.c:4)
```

```
==27164==    by 0x400568: main (me.c:7)
```

```
==27164== Address 0x51da068 is 0 bytes after a block of size 40 alloc'd
```

```
==27164==    at 0x4C2C12F: malloc (in /usr/lib64/valgrind/vgpreload_memcheck-amd64-linux.so)
```

```
==27164==    by 0x400553: f (me.c:3)
```

```
==27164==    by 0x400568: main (me.c:7)
```

Поиск ошибок работы с памятью

◆ sanitizers: встроенная в gcc/clang инструментация
(нас интересует address sanitizer)

```
#include <stdlib.h>
void f(void) {
    int* x = malloc(10 * sizeof(int));
    x[10] = 0;          // problem 1: heap block overrun
}
int main(void) {
    f();
    return 0;
}
```

```
$ gcc -Og -g -fsanitize-address -o mesa && ./mesa
```

```
==27179==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x60400000dff8 at pc
0x0000004007c1 bp 0x7fffc32a1420 sp 0x7fffc32a1418
```

```
WRITE of size 4 at 0x60400000dff8 thread T0
```

```
#0 0x4007c0 in f /home/bonzo/tmp/me.c:4
```

```
#1 0x4007d5 in main /home/bonzo/tmp/me.c:7
```

```
#2 0x7fba219d870f in __libc_start_main (/lib64/libc.so.6+0x2070f)
```

```
#3 0x4006b8 in _start (/home/bonzo/tmp/mesa+0x4006b8)
```

```
0x60400000dff8 is located 0 bytes to the right of 40-byte region
[0x60400000dfd0,0x60400000dff8)
```

```
allocated by thread T0 here:
```

```
#0 0x7fba21df074a in malloc (/usr/lib64/libasan.so.2+0x9674a)
```

```
#1 0x400793 in f /home/bonzo/tmp/me.c:3
```

```
SUMMARY: AddressSanitizer: heap-buffer-overflow /home/bonzo/tmp/me.c:4 f
```