stdio окончание. heap. ДЗ (указатели на функции, список).

Евгений Линский

stdio окончание

printf/fprintf/sprintf

```
fprintf(stdout, ...); //printf
fscanf(stdin, ....); //scanf
char s1[] = "3  4";
sscanf(s1,"%d %d", &a, &b);
char s2[256];
sprintf(s2, "%d + %d = %d", a, b, c);
```

1. Все это небыстро, т.к. внутри функции нужно разобрать форматную строку
2. Технология: функция с переменным числом параметров (см. *va_arg*)

https://en.cppreference.com/w/cpp/io/c/fprintf

Переменное число аргументов в C (printf)

va_start, va_arg, va_end — макросы.

```
1  void simple_printf(const char* fmt, ...) {
2    va_list args;
3    //записать в args адрес следующего за fmt параметра на стеке
4    va_start(args, fmt);
5    while(*fmt != '\0') {
6      if(*fmt=='d') {
7        //достать со стека переменную типа int
8        int i = va_arg(args, int)
9        // здесь должен быть код, который
10       // выводит int на экран с помощью putc
11     }
12     fmt++;
13   }
14   va_end(args);
15 }
16 //Труднообнаруживаемые ошибки
17 printf("%s", 5);
18 printf("%d %d", 4); printf("%d", 4, 5);
```

Куча

Три вида памяти в программе на C

1. **Стек** (stack)
   - локальные переменные функций, параметры функций
   - код для выделения и освобождения генерирует компилятор
   - выделяется при “входе” в функцию, освобождается при “выходе” из функции
2. **Глобальная память** (static variables)
   - глобальные переменные (вне функций), статические переменные (static)
   - код для выделения и освобождения генерирует компилятор
   - выделяется при загрузке в память, освобождается при завершении программы
   - глобальные инициализируются в *каком-то* порядке, статические — при входе в функцию
3. **Куча** (heap)
   - код для выделения и освобождения пишет программист

Карта памяти

- Расположение частей может отличаться на разных платформах.
- В общем случае адресация неважна.
- Ниже один из вариантов (“упрощенный linux”, 4 Gb)

OS kernel (например, 1 Gb) ....
Stack, растет вниз ↓(например, 10 Mb)
.... Heap, растет вверх ↑(например, ∼2,9 Gb)
Static variables (например, 10 Mb)
Двоичный код программы (например, 10 Mb)

Куча.

```
1  #include <stdlib.h>
2
3  int *p = malloc(1000000 * sizeof(int));
4  if (p == NULL){  // NULL в C, nullptr в C++. Старый код - 0.
5    /* not enough memory */
6  }
7  // if (!p) { ... } // Альтернативный вариант
8  p[0] = 1; p[13000]  = 42;
9  ...
10 free(p);
11
```

1. Временем жизни управляет программист
2. *Функция* malloc обращается к операционной системе с просьбой (sbrk) выделить место (“непрерывный кусок”) в куче и, если ОС выделяет это место, возвращает укзатель на начало области (иначе — 0).
3. *Функция* free освобождает память
4. Нет ограничений по размеру как у стека и глобальных переменных (ограничена размером свободной памяти)

Куча.

```
1  #include <stdlib.h>
2  #include <stdio.h>
3
4  size_t size = 0;
5  // %zu вместо %d, потому что size_t может != int.
6  scanf("%zu", &size);
7  int *array = malloc(size * sizeof(int));
```

1. Размер массива выясняется во время выполнения (ввел пользователь, считали из файла)
2. На стеке и у глобальных переменных размер должен быть известен во время компиляции

Куча. Ошибки.

```
1  int *p = malloc(sizeof(int));
2  free(p);
```

Что не так?

Куча. Ошибки.

```
1  int *p = malloc(sizeof(int));
2  free(p);
```

Что не так?

1. Занимает в три раза больше места чем на стеке (int*, int)

Куча. Ошибки.

```
1  int *p = (int *)malloc(1000000 * sizeof(int));
2  p = (int *)malloc(1000000 * sizeof(int));
3  // Или: p = NULL;
```

- Утечка памяти (memory leak): теперь память из первой строки невозможно освободить (мы потеряли адрес)
- Такие ошибки можно искать утилитой *valgrind*

Вопрос! В современных ОС вся память, выделенная программой, после ее завершения возвращается системе (даже если была утечка). Зачем бороться с утечками?

Куча. Ошибки.

```
1  int *p = (int *)malloc(1000000 * sizeof(int));
2  p = (int *)malloc(1000000 * sizeof(int));
3  // Или: p = NULL;
```

- Утечка памяти (memory leak): теперь память из первой строки невозможно освободить (мы потеряли адрес)
- Такие ошибки можно искать утилитой *valgrind*

Вопрос! В современных ОС вся память, выделенная программой, после ее завершения возвращается системе (даже если была утечка). Зачем бороться с утечками?

1. Сервер (работает без перезапуска). Утечка при каждом запросе пользователя.

Куча. Ошибки.

```
1  int *p = (int *)malloc(1000000 * sizeof(int));
2  p = (int *)malloc(1000000 * sizeof(int));
3  // Или: p = NULL;
```

- Утечка памяти (memory leak): теперь память из первой строки невозможно освободить (мы потеряли адрес)
- Такие ошибки можно искать утилитой *valgrind*

Вопрос! В современных ОС вся память, выделенная программой, после ее завершения возвращается системе (даже если была утечка). Зачем бороться с утечками?

1. Сервер (работает без перезапуска). Утечка при каждом запросе пользователя.
2. Сначала все замедлится (файл подкачки), потом ОС аварийно завершит процесс.

Куча. Вычислительная сложность

malloc должен:

1. Пройти по списку (одна из возможным реализаций) выделенных (точнее «выделенных, но потом освобожденных», см. дз alloc) областей
2. Найти непрерывную область нужного размера

Это гораздо дольше чем на стеке и у глобальных переменных!

Куча. Что еще бывает?

- calloc — выделяет память и инициализирует ее нулями
- realloc — изменяет размер уже существующего массива. Существует три результата работы функции:
  1. если нужное число байт не занято в данной области памяти, то увеличивает область для массива
  2. если рядом нет свободной памяти, перенесет массив в другое место
  3. если вообще нет памяти под увеличенный массив, вернет 0

ДЗ

binary-search-3: сортировка выбором

Рассмотрим на примере более простой сортировки.
selection_sort

```
void sel_sort(int *arr, int n) {
    int i, j, min_idx;

    for (i = 0; i < n-1; i++) {
        min_idx = i;
        for (j = i + 1; j < n; j++)
          if (arr[j] < arr[min_idx])
            min_idx = j;
        swap(&arr[min_idx], &arr[i]);
    }
}
```

Любой тип данных: void*.

```
struct product_s {
  char label[256];
  unsigned char weight;
  unsigned int price;
};
typedef struct product_s product_t;
```

```
void sel_sort(void *array, size_t n, ...);

product_t array1[100];
int arrray2[20];
sel_sort(array1, 100, ...);
sel_sort(array2, 20, ...)
```

- void* — не работает адресная арифметика
- В C++ неявное приведение типа в void* не вызывает ошибки (в C все можно неявно)
- В C++ требуется явное приведение типа из void* (в C все можно неявно)
- void* malloc(...)

Переставлять любые элементы: swap.

```
void sel_sort(void *array, size_t n, size_t elem_size, ...)
  char *p = array;
  //смысл:  swap(&array[i], &array[j])
  swap(p + i * elem_size, p + j * elem_size, elem_size);
}
void swap(char *p1, char *p2, size_t elem_size) {
  int i = 0; char tmp = 0;
  while( i < elem_size ) {
    tmp = *(p1 + i);
    *(p1 + i) = *(p2 + i);
    *(p2 + i) = tmp;
    i++;
  }
}
```

Сравнивать любые элементы: cmp.

Указатель — адрес в памяти:

- В памяти хранятся переменные и двоичный код (двоичные инструкции нашей программы)
- Можно хранить адрес переменной (указатель)
- Можно хранить адрес кода (указатль на функцию)

```
void dummy(int x) {
    printf( "%d\n", x );
}

int main() {
  void (*func)(int);
  func = &dummy;
  (*func)(2);
  // можно и так:
  func = dummy;
  func(2);
  return 0;
}
```

Сравнивать любые элементы: cmp.

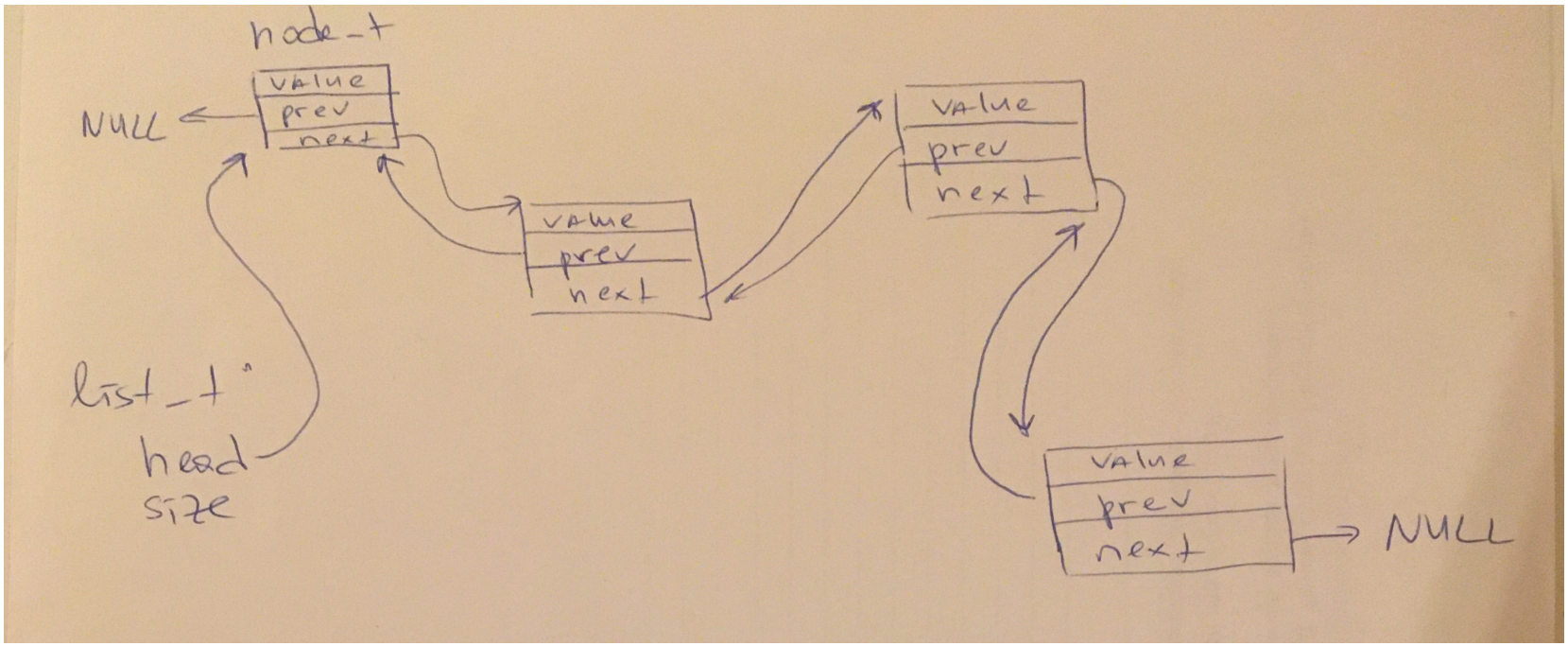
```
void sel_sort(void *array, size_t size, size_t elem_size,
              int (*cmp)(void *p1, void *p2)) {
  char *p = array;
  //смысл:  if (array[i] < array[j])
  if (cmp(p + i * elem_size, p + j * elem_size))

}
int cmp_int(void *p1, void *p2) {
  int *pi1 = p1; int *pi2 = p2;
  return (*pi1 - *pi2);
}
int cmp_product_by_weight(void *p1, void *p2) {
  product_t *pp1 = p1; product_t *pp2 = p2;
  return (*pp1->weight- *pp2->weight);
}
product_t array1[100];
int arrray2[20];
sel_sort(array1, 100, sizeof(array1[0]),
  cmp_product_by_weight);
sel_sort(array2, 20, sizeof(array2[0]), cmp_int);
```

alloc: связный список

- размер может увеличиваться при добавлении элементов
  - каждый элемент выделяется с помощью malloc
  - элементы расположены на произвольных (в векторе — на последовательных) адресах
  - каждый элемент хранит адреса следующего и предыдущего (необязательно)
- из-за того, что адреса элементов непоследовательные, не требуется смещать элементы при этих операциях вставки и удаления

list


node_t
value
prev
next
NULL
value
prev
next
value
prev
next
value
prev
next
NULL
list_t*
head
size

node_t

```c
struct node_s; // forward declaration

typedef struct node_s {
  int value;
  struct node_s *prev; // адрес предыдущего (необязательно)
  struct node_s *next; // адрес следующего
} node_t;

node_t *head = NULL; // адрес нулевого элемента
```

print

```
void print(node_t *head) {
  node_t *cur = head;
  while (cur != NULL) {
    printf("%d ", cur->value);
    cur = cur->next;
  }
  printf("\n");
}
```

list iterate