Лекция 4. Три вида памяти: глобальные переменные, куча.

Евгений Линский

Глобальные переменные

```
1  int last_rnd = 0;
2
3  void srand() {
4    last_rnd = time(); //текущее время
5  }
6
7  int rand() {
8    last_rnd = (last_rnd * 13 +  113) % 43;
9    return last_rnd;
10 }
11
12 int main() {
13   int a[10];
14   srand();
15   for(int i = 0; i < 10; i++) a[i] = rand();
16 }
```

Статические переменные (static)

```
1  void f() {
2    static int call_count = 0; //инициализируется один раз
3    ...
4    printf("Called times: %d", call_count);
5    call_count++;
6  }
7
8  int main() {
9    f(); f(); f();
10 }
```

Не впечатлило? Смотри strtok в стандартной библиотеке =)

Глобальные переменные. Несколько файлов.

1.cpp

```
1  int last_rnd = 0;
2  void srand() {
3    last_rnd = time(); //текущее время
4  }
```

2.cpp

```
1  int last_rnd = 0;
2  int rand() {
3    last_rnd = (lst_rnd * 13 +  113) % 43;
4    return last_rnd;
5  }
```

Глобальные переменные. Несколько файлов.

1.cpp

```
1  int last_rnd = 0;
2  void srand() {
3    last_rnd = time(); //текущее время
4  }
```

2.cpp

```
1  int last_rnd = 0;
2  int rand() {
3    last_rnd = (lst_rnd * 13 +  113) % 43;
4    return last_rnd;
5  }
```

1. Переменная определена дважды (не скомпилируется)

Глобальные переменные. Несколько файлов.

1.h

```
1  extern int last_rnd;
```

1.cpp

```
1  int last_rnd = 0; //выделение памяти
2  void srand() {
3    last_rnd = time(); //текущее время
4  }
```

2.cpp

```
1  #include "1.h"
2  //extern -> выделение памяти происходит в другом месте
3  // (также знаем тип переменной)
4  int rand() {
5    last_rnd = (lst_rnd * 13 +  113) % 43;
6    return last_rnd;
7  }
```

Глобальные переменные. static.

Слово static имеет два разных смысла в зависимости от контекста.

```
1  int a = 0; // Глобальная переменная (видна во всех файлах)
2
3  static int b = 0; // Глобальная переменная
4                    // (видна только в этом файле)
5  void f() {
6      static int c = 0; // Статическая переменная
7                        // (видна только в функции)
8  }
```

К переменной a можно обращаться из других файлов при помощи extern.
Переменная b не будет видна из других файлов даже если есть extern.
Переменную c вообще не имеет смысла видеть в других файлах.

Глобальные переменные. Вычислительная сложность.

OS kernel (например, 1 Gb)
....
Static variables (например, 10 Mb)
Двоичный код программы (например, 10 Mb)

1. В заголовке двоичного исполняемого файла написано, сколько static variables ему требуется
2. Память выделяется “непрерывным куском”при загрузке программы. Освобождается — когда программа заканчивает работу.
3. Выделение происходит быстро.

Глобальные переменные. Лучше не надо.

Почему никто не любит?

1. Потенциальный конфликт имен (несколько программистов в разных файлах назвали разные переменные одинаково —> конфликт на линковке)
2. Трудно анализировать программу (сложнее следить за всеми участками кода, в которых меняется перменная)
3. Неизвестно, в каком порядке инициализируются разные файлы:

```
1 // a.cpp
2 int ten = 10; // Первый, потому что константа
3 int x = ten; // Второй или третий
4 int foo() { return x; }
5 // b.cpp
6 int y = foo(); // Второй или третий
```

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);
```

1. Временем жизни управляет программист
2. *Функция* malloc обращается к операционной системе с просьбой 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 областей
2. Найти непрерывную область нужного размера

Это гораздо дольше чем на стеке и у глобальных переменных!

Куча. Выделение массива массивов

Можно использовать как двумерный массив:

```
1  int **m = (int **)malloc(N * sizeof(int*));
2  for (int i = 0; i < N; ++i){
3    m[i] = (int *)malloc(N * sizeof(int));
4  }
5
6  m[42][42] = 42;
7
8  for (int i = 0; i < N; ++i){
9    free(m[i]);
10 }
11 free(m);
```

Как потратить 2 вызова malloc вместо N+1?

Куча. Выделение двумерного массива

Если все размерности, кроме первой, фиксированы на этапе компиляции, то можно выделить «настоящий» двумерный массив из одного блока памяти.

```
1  // Мнемоника: если разыменуем m, то будет int[10].
2  // У * приоритет ниже [], поэтому нужны скобки.
3  int (*m)[10] = (int(*)[10])malloc(N * sizeof(int[10]));
4
5  m[42][5] = 42;
6
7  free(m);
```

Зачем требование про размерности?

Куча. Выделение двумерного массива

Если все размерности, кроме первой, фиксированы на этапе компиляции, то можно выделить «настоящий» двумерный массив из одного блока памяти.

```
1  // Мнемоника: если разыменуем m, то будет int[10].
2  // У * приоритет ниже [], поэтому нужны скобки.
3  int (*m)[10] = (int(*)[10])malloc(N * sizeof(int[10]));
4
5  m[42][5] = 42;
6
7  free(m);
```

Зачем требование про размерности? Компилятор должен сгенерировать какой-то код для [], для этого надо знать размерности на этапе компиляции.

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 смежной области, 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