Лекция 3. Три вида памяти.

Евгений Линский

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)

Стек

```
int sum(int a, int b) {
  int s = a + b;
  return s;
}
int main() {
  int c = 1; int d = 2;
  int e = sum(c, d);
  return 0;
}
```

Stack, растет вниз ↓(например, 10 Mb)

Кадр main	c, d, e, RV, RA
Кадр sum	a, b, s, RV, RA

Вошли в функцию — выделили кадр (frame), вышли из функции — освободили кадр

- RV — return value (возвращаемое значение)
- RA — return address (на какой адрес вернуться в main после окончания sum)

RA. Оптимизации.

Двоичный код программы (например, 10 Mb)

Код main	адрес ...	...
	адрес 15	перейти на sum, адрес 15 запомнить в RA
	адрес ...	...
Код sum	адрес ...	...
	адрес ...	...
	адрес ...	вернуться на адрес из RA (адрес 15)

Как можно оптимизировать работу со стеком?

RA. Оптимизации.

Двоичный код программы (например, 10 Mb)

Код main	адрес ...	...
	адрес 15	перейти на sum, адрес 15 запомнить в RA
	адрес ...	...
Код sum	адрес ...	...
	адрес ...	...
	адрес ...	вернуться на адрес из RA (адрес 15)

Как можно оптимизировать работу со стеком?

1. Передача копии параметра: main –> store (сохранить на стек), sum –> load (загрузить со стека)

RA. Оптимизации.

Двоичный код программы (например, 10 Mb)

Код main	адрес ...	...
	адрес 15	перейти на sum, адрес 15 запомнить в RA
	адрес ...	...
Код sum	адрес ...	...
	адрес ...	...
	адрес ...	вернуться на адрес из RA (адрес 15)

Как можно оптимизировать работу со стеком?

1. Передача копии параметра: main –> store (сохранить на стек), sum –> load (загрузить со стека)
2. Зарезервировать несколько регистров под передачу параметров и RV и передавать через них (т.е. не выгружать в память)

Стек. Вычислительная сложность.

Stack, растет вниз ↓(например, 10 Mb)

Кадр main	c, d, e, RV, RA
Кадр sum	a, b, s, RV, RA

Вошли в функцию — выделили кадр (frame), вышли из функции — освободили кадр

- В процессоре есть регистр (например, sp), который хранит адрес “головы” стека
- Выделение кадра — уменьшение регистра на размер кадра, освобождение — увеличение (быстрые операции)
- Код, который расчитывает размер кадра и меняет sp, генерирует компилятор
- Выделение локальной переменной — это очень быстро, происходит один раз на функцию

Стек. Рекурсия.

```
int factorial(int n) {
  if (n == 1)
    return 1;
  else
    return factorial(n-1) * n;
}
```

Почему никто не любит такое?

Стек. Рекурсия.

```
int factorial(int n) {
  if (n == 1)
    return 1;
  else
    return factorial(n-1) * n;
}
```

Почему никто не любит такое?

1. Большое n, стек закончится, OS аварийно завершит программу

Стек. Рекурсия.

```
int factorial(int n) {
  if (n == 1)
    return 1;
  else
    return factorial(n-1) * n;
}
```

Почему никто не любит такое?

1. Большое n, стек закончится, OS аварийно завершит программу
2. Можно переписать циклом без потери элегантности

Стек. Рекурсия.

```
int factorial(int n) {
  if (n == 1)
    return 1;
  else
    return factorial(n-1) * n;
}
```

Почему никто не любит такое?

1. Большое n, стек закончится, OS аварийно завершит программу
2. Можно переписать циклом без потери элегантности

Иногда компилятор может сам соптимизировать в цикл (хвостовая рекурсия).

Глобальные переменные

```
int last_rnd = 0;

void srand() {
  last_rnd = time(); //текущее время
}

int rand() {
  last_rnd = (last_rnd * 13 +  113) % 43;
  return last_rnd;
}

int main() {
  int a[10];
  srand();
  for(int i = 0; i < 10; i++) a[i] = rand();
}
```

Статические переменные (static)

```
void f() {
  static int call_count = 0; //инициализируется один раз
  ...
  printf("Called times: %d", call_count);
  call_count++;
}

int main() {
  f(); f(); f();
}
```

Не впечатлило? Смотри strtok в стандартной библиотеке =)

Глобальные переменные. Несколько файлов.

1.cpp

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

2.cpp

```
int last_rnd = 0;
int rand() {
  last_rnd = (lst_rnd * 13 +  113) % 43;
  return last_rnd;
}
```

Глобальные переменные. Несколько файлов.

1.cpp

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

2.cpp

```
int last_rnd = 0;
int rand() {
  last_rnd = (lst_rnd * 13 +  113) % 43;
  return last_rnd;
}
```

1. Переменная определена дважды (не скомпилируется)

Глобальные переменные. Несколько файлов.

1.h

```
extern int last_rnd;
```

1.cpp

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

2.cpp

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

Глобальные переменные. static.

Слово static имеет два разных смысла в зависимости от контекста.

```
int a = 0; // Глобальная переменная (видна во всех файлах)

static int b = 0; // Глобальная переменная
                  // (видна только в этом файле)
void f() {
    static int c = 0; // Статическая переменная
                      // (видна только в функции)
}
```

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:

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

Куча.

```
#include <stdlib.h>

int *p = malloc(1000000 * sizeof(int));
if (p == NULL){   // NULL в C, nullptr в C++.  Старый код - 0.
  /* not enough memory */
}
// if (!p) { ...  } // Альтернативный вариант
p[0] = 1; p[13000]  = 42;
...
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 указатель 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.

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

size_t size = 0;
scanf("%zu", &size); // %zu вместо %d, потому что size_t
может != int.
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.

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

Что не так?

Куча. Ошибки.

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

Что не так?

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

Куча. Ошибки.

```
int *p = (int *)malloc(1000000 * sizeof(int));
p = (int *)malloc(1000000 * sizeof(int));
// 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.

```
int *p = (int *)malloc(1000000 * sizeof(int));
p = (int *)malloc(1000000 * sizeof(int));
// 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.

```
int *p = (int *)malloc(1000000 * sizeof(int));
p = (int *)malloc(1000000 * sizeof(int));
// 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:

```
int **m = (int **)malloc(N * sizeof(int*));
for (int i = 0; i < N; ++i){
  m[i] = (int *)malloc(N * sizeof(int));
}

m[42][42] = 42;

for (int i = 0; i < N; ++i){
  free(m[i]);
}
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.

```
// Мнемоника: если разыменуем m, то будет int[10].
// У * приоритет ниже [], поэтому нужны скобки.
int (*m)[10] = (int(*)[10])malloc(N * sizeof(int[10]));

m[42][5] = 42;

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.

```
// Мнемоника: если разыменуем m, то будет int[10].
// У * приоритет ниже [], поэтому нужны скобки.
int (*m)[10] = (int(*)[10])malloc(N * sizeof(int[10]));

m[42][5] = 42;

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