# Размещение в памяти

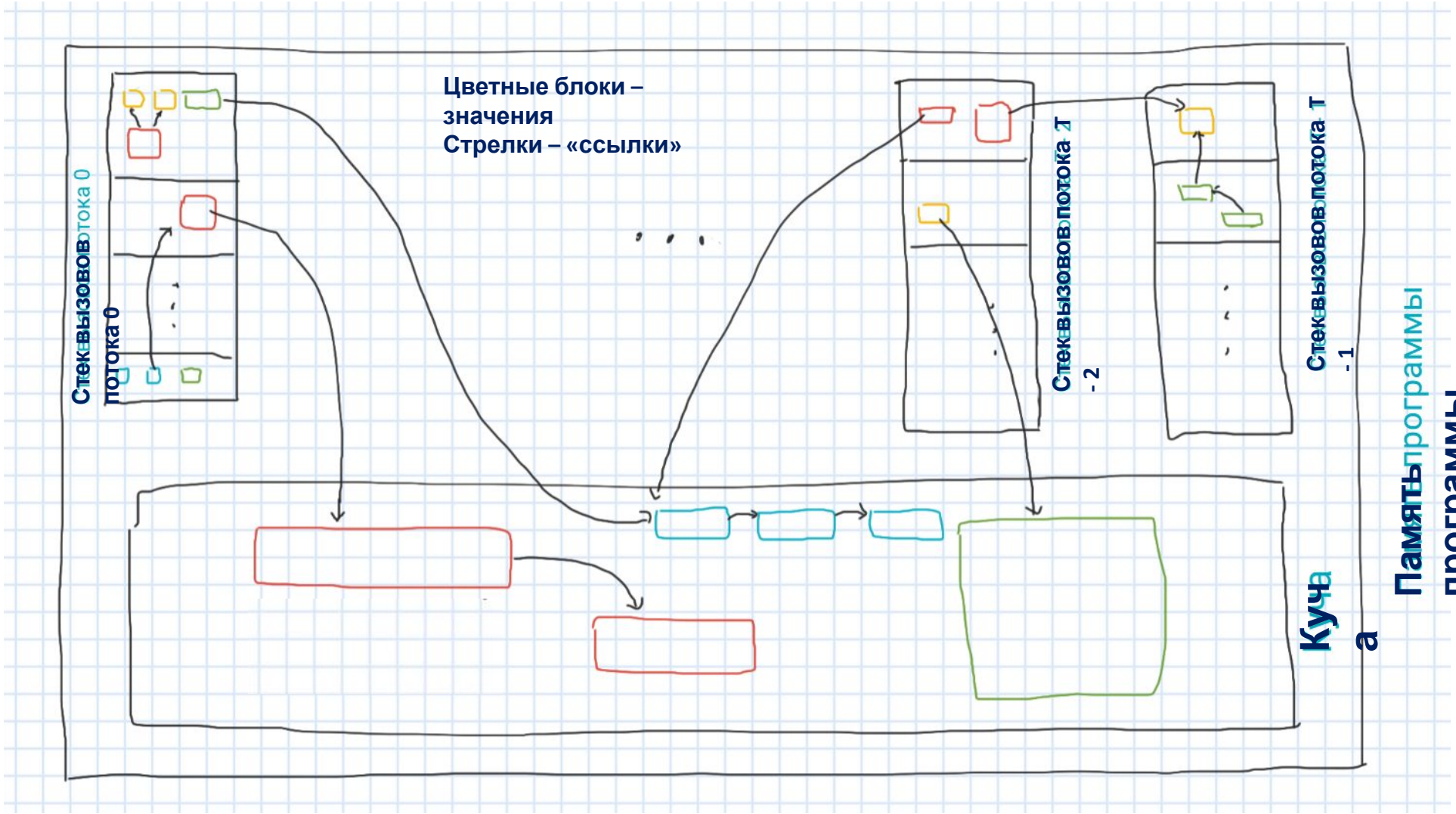
лекция 6½

# План лекции

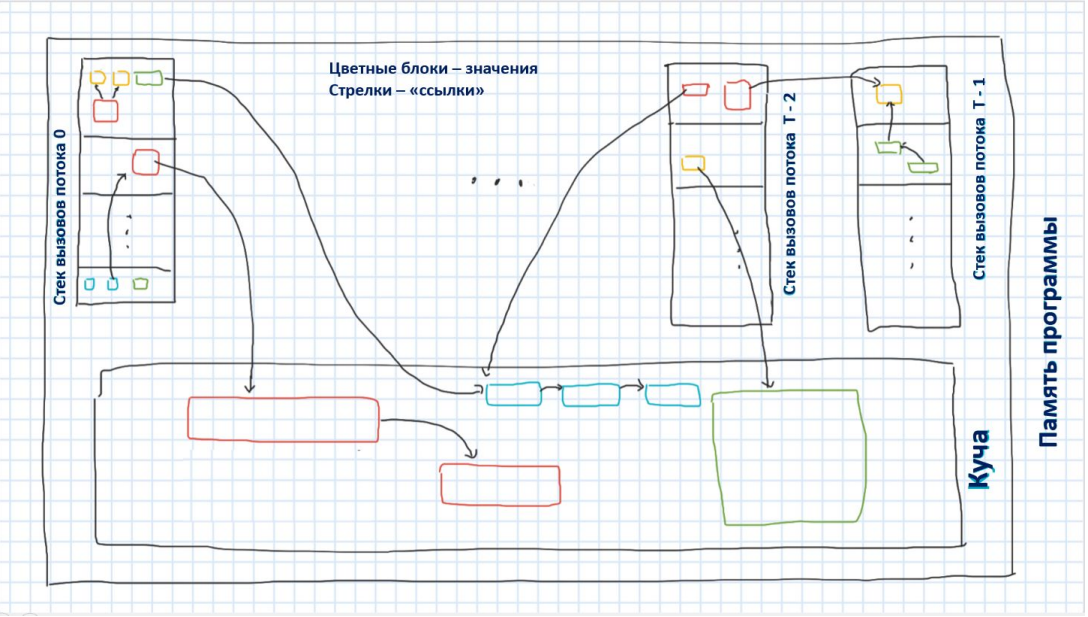
- Про модель памяти программы
- Размещение в стековом кадре
  - Выравнивание
  - Связь выравниваний производного типа и его элементов
  - Выравнивающие байты
- Динамическое распределение памяти
  - Стандартные функции языка Си malloc, free и др.
    - Doug Lea’s malloc
  - Накладные расходы, фрагментация
  - Виды ошибок и address sanitizer

# Модель памяти программы

# Модель памяти программы

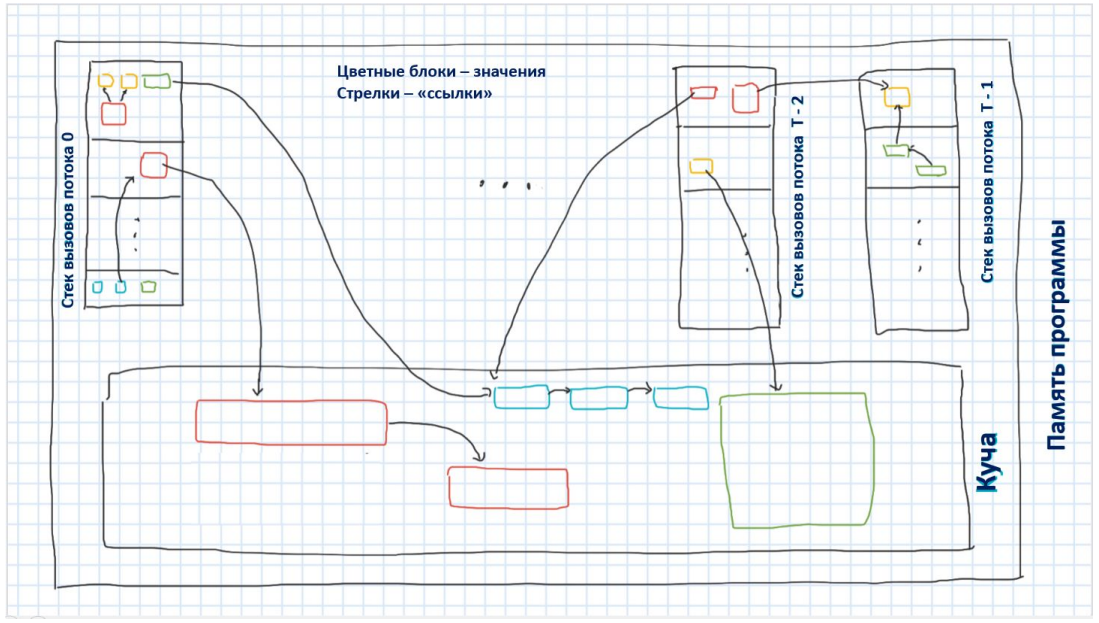


# Модель памяти программы



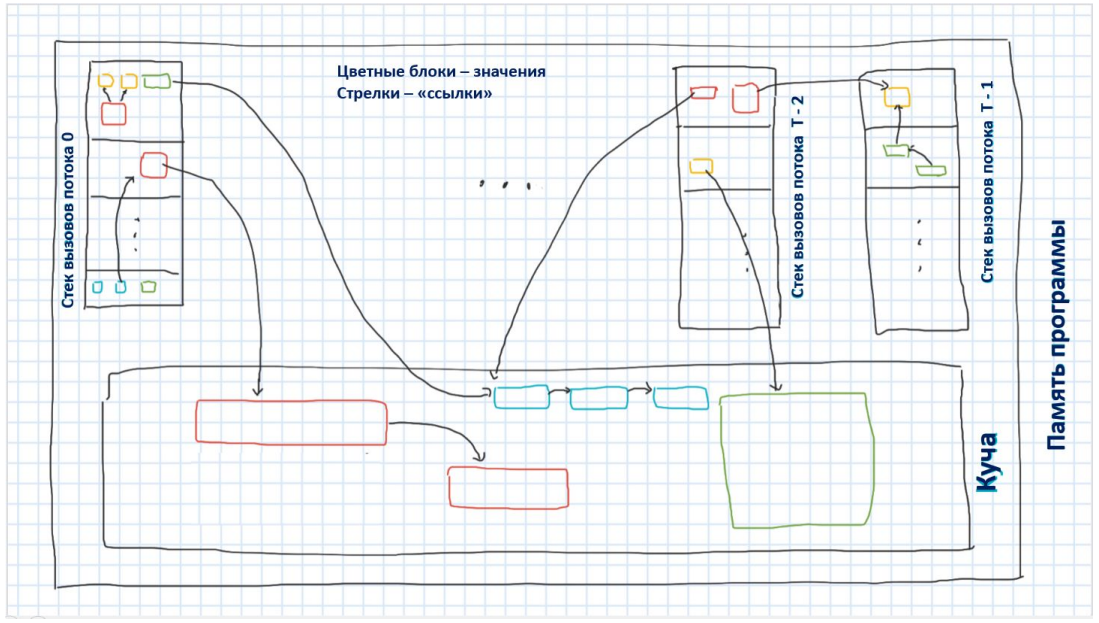

- Языки без указателей
- Языки с указателями

# Модель памяти программы




- Языки без указателей
  - Java, Python, C#, Haskel, Ocaml, etc.
- Языки с указателями

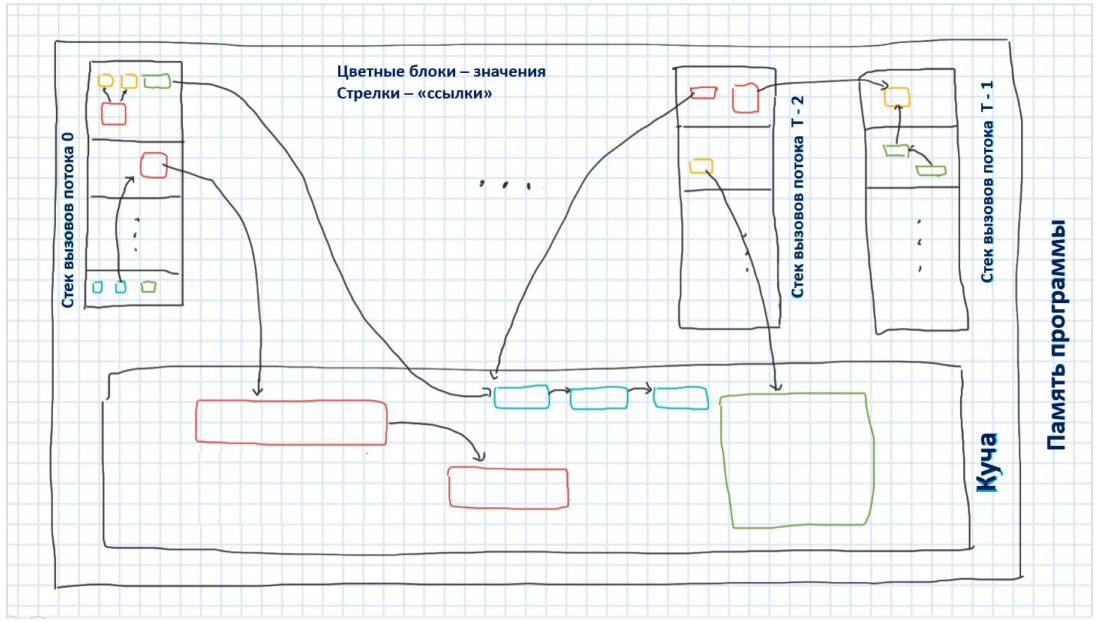
# Модель памяти программы




- Языки без указателей
  - Java, Python, C#, Haskel, Ocaml, etc.
  - Работа с памятью 100% автоматическая
    - Сборка мусора, безопасность – бесплатно

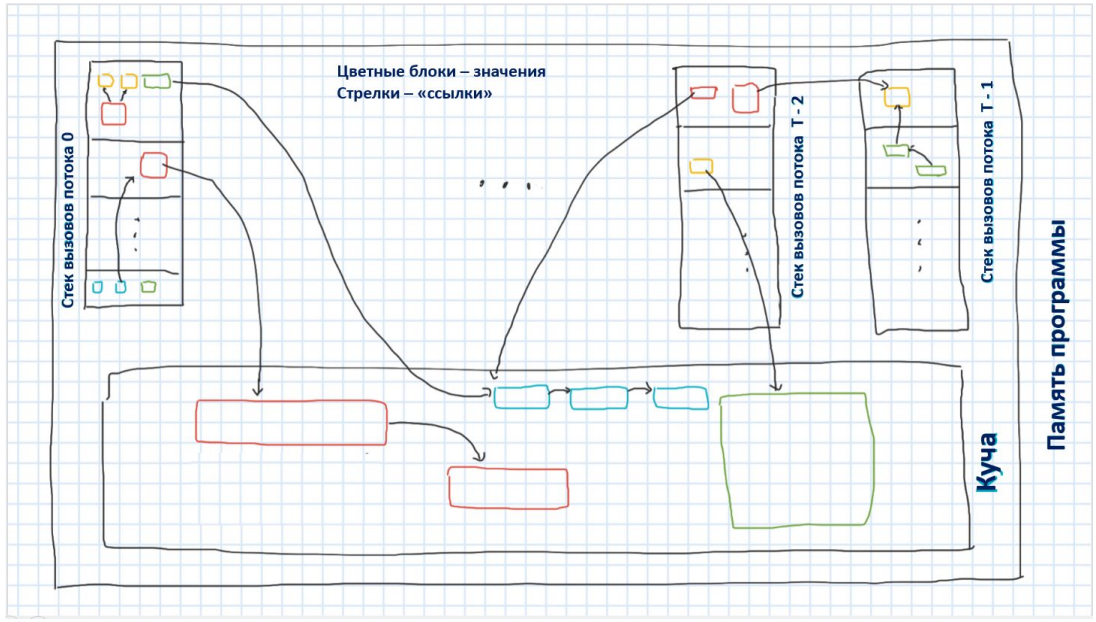
- Языки с указателями

# Модель памяти программы



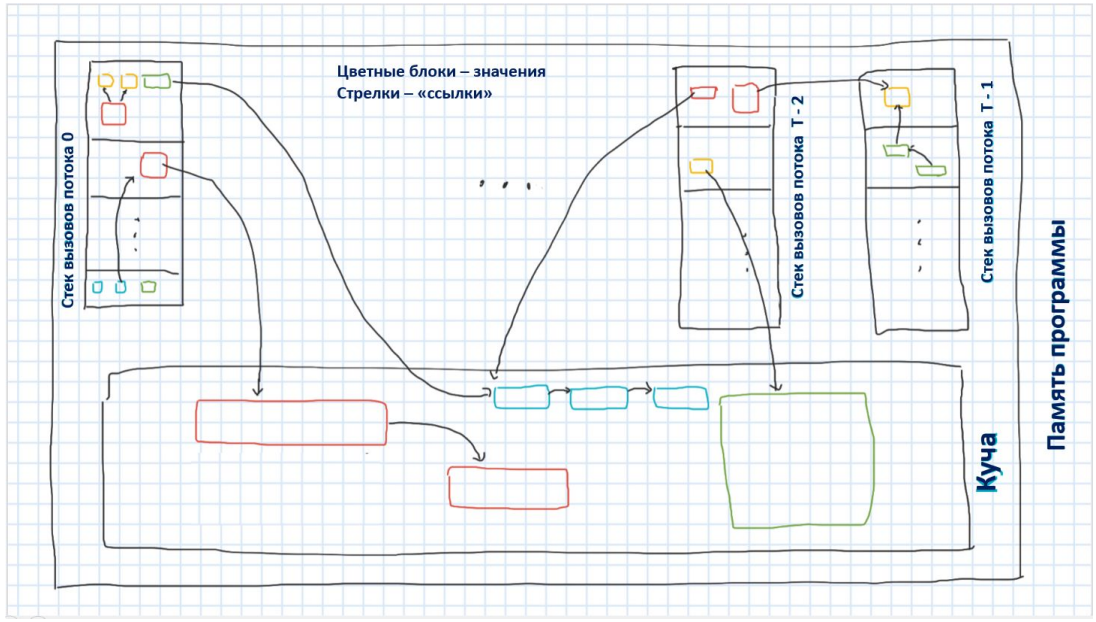

- Языки без указателей
  - Java, Python, C#, Haskel, Ocaml, etc.
  - Работа с памятью 100% автоматическая
    - Сборка мусора, безопасность – бесплатно
  - Скорость работы ▼
  - Расход памяти ▲
- Языки с указателями

# Модель памяти программы



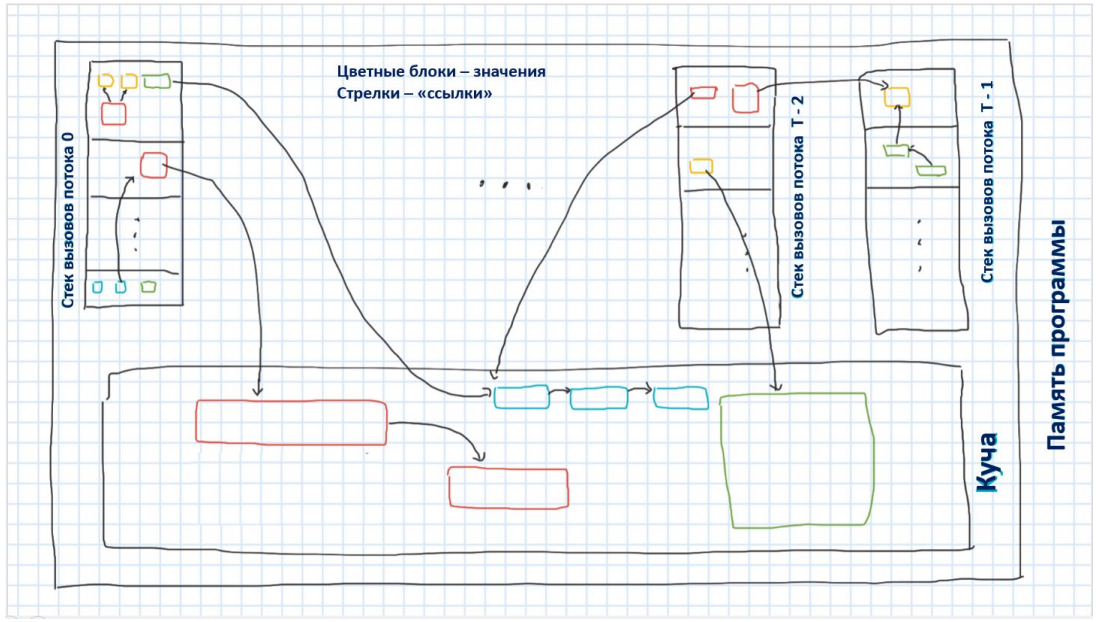

- Языки без указателей
  - Java, Python, C#, Haskel, Ocaml, etc.
  - Работа с памятью 100% автоматическая
    - Сборка мусора, безопасность – бесплатно
  - Скорость работы ▼
  - Расход памяти ▲
- Языки с указателями
  - Pascal, C, C++, golang, etc.

# Модель памяти программы




- Языки без указателей
  - Java, Python, C#, Haskel, Ocaml, etc.
  - Работа с памятью 100% автоматическая
    - Сборка мусора, безопасность – бесплатно
  - Скорость работы ▼
  - Расход памяти ▲
- Языки с указателями
  - Pascal, C, C++, golang, etc.
  - Работа с памятью полуавтоматическая
    - Сами уничтожаем ненужные значения и правильно работаем с указателями

# Модель памяти программы




- Языки без указателей
  - Java, Python, C#, Haskel, Ocaml, etc.
  - Работа с памятью 100% автоматическая
    - Сборка мусора, безопасность – бесплатно
  - Скорость работы ▼
  - Расход памяти ▲

- Языки с указателями
  - Pascal, C, C++, golang, etc.
  - Работа с памятью полуавтоматическая
    - Сами уничтожаем ненужные значения и правильно работаем с указателями
  - Скорость работы ▲
  - Расход памяти ▼

# Размещение данных в стековом кадре

# Размещение данных в стековом кадре

- Компилятор размещает значения переменных в стековом кадре в соответствии со стандартом языка Си

# Размещение данных в стековом кадре

- Компилятор размещает значения переменных в стековом кадре в соответствии со стандартом языка Си
  - Назначает переменным адреса для хранения

# Размещение данных в стековом кадре

- Компилятор размещает значения переменных в стековом кадре в соответствии со стандартом языка Си
  - Назначает переменным адреса для хранения

- Переменные располагаются в стековом кадре в порядке описания

# Размещение данных в стековом кадре

- Компилятор размещает значения переменных в стековом кадре в соответствии со стандартом языка Си
  - Назначает переменным адреса для хранения

- Переменные располагаются в стековом кадре в порядке описания
  - Если описаны без static/extern
  - Возможно присутствие неиспользуемых байтов между последовательно описанными переменными

# Выравнивание

# Выравнивание

- Значения типа T должны храниться по адресам, кратным alignof(T) – выравниванию типа T
  - Оператор alignof появился в C99

# Выравнивание

- Значения типа T должны храниться по адресам, кратным alignof(T) – выравниванию типа T
  - Оператор alignof появился в C99

- У всех популярных компиляторов alignof(T) -- это небольшая степень 2
  - Зависящая от T

# Выравнивание

- Значения типа T должны храниться по адресам, кратным alignof(T) – выравниванию типа T
  - Оператор alignof появился в C99
- У всех популярных компиляторов alignof(T) -- это небольшая степень 2
  - Зависящая от T
- Доступ к значению типа T, хранящемуся по адресу, некратному alignof(T), – это undefined behavior

# Выравнивание

- Значения типа T должны храниться по адресам, кратным alignof(T) – выравниванию типа T
  - Оператор alignof появился в C99
- У всех популярных компиляторов alignof(T) -- это небольшая степень 2
  - Зависящая от T
- Доступ к значению типа T, хранящемуся по адресу, некратному alignof(T), – это undefined behavior

```
char array[4] = {0};
// undefined behavior,
// if alignof(char) %
//         alignof(int) != 0
if (*(int*)array == 0) {
    // ...
}
```

# Выравнивание простых типов и указателей

# Выравнивание простых типов и указателей

- Зависит от используемого компилятора (implementation defined)
  - До C99 alignof(T) можно узнать в документации по компилятору

# Выравнивание простых типов и указателей

- Зависит от используемого компилятора (implementation defined)
  - До C99 alignof(T) можно узнать в документации по компилятору

- alignof(T) определяется требованиями, которые предъявляют к адресам инструкции процессора для чтения и записи в память данных размера sizeof(T)

# Как компилятор выравнивает производный тип

# Как компилятор выравнивает производный тип

- Пусть T – производный тип
- Пусть $T_1$, …, $T_n$ – типы элементов T

# Как компилятор выравнивает производный тип

- Пусть T – производный тип
- Пусть $T_1$, …, $T_n$ – типы элементов T

- Необходимо, чтобы alignof(T) было кратно наибольшему общему кратному всех alignof($T_i$)

# Как компилятор выравнивает производный тип

- Пусть T – производный тип
- Пусть $T_1$, …, $T_n$ – типы элементов T

- Необходимо, чтобы alignof(T) было кратно наибольшему общему кратному всех alignof($T_i$)

$$\text{alignof}(T) \,\%\, \text{НОК}(\text{alignof}(T_1), \ldots, \text{alignof}(T_n)) == 0$$

# Как компилятор выравнивает производный тип

- Пусть T – производный тип
- Пусть $T_1$, …, $T_n$ – типы элементов T

- Необходимо, чтобы alignof(T) было кратно наибольшему общему кратному всех alignof($T_i$)

$$\text{alignof}(T) \,\%\, \text{НОК}(\text{alignof}(T_1), \ldots, \text{alignof}(T_n)) == 0$$

  - Иначе некоторые элементы T могут быть выровнены неправильно

# Как компилятор выравнивает производный тип

- Пусть T – производный тип
- Пусть $T_1$, …, $T_n$ – типы элементов T

- Необходимо, чтобы alignof(T) было кратно наибольшему общему кратному всех alignof($T_i$)

$$\text{alignof}(T) \,\%\, \text{НОК}(\text{alignof}(T_1), \ldots, \text{alignof}(T_n)) == 0$$

  - Иначе некоторые элементы T могут быть выровнены неправильно
  - Все популярные компиляторы используют max вместо НОК, т.к. у них все alignof($T_i$) – это степени 2

# Пример про выравнивание массива

# Пример про выравнивание массива

- T = `int (*)[4]` – массив из 4 `int`
- $T_1 = T_2 = T_3 = T_4$ = `int` – типы элементов T

# Пример про выравнивание массива

- T = `int (*)[4]` – массив из 4 `int`
- $T_1 = T_2 = T_3 = T_4$ = `int` – типы элементов T

- Пусть `alignof(int) == 4`, но `alignof(T) == 1` или `2`
  - т.е. нарушена кратность НОК(выравнивания элементов)

# Пример про выравнивание массива

- T = `int (*)[4]` – массив из 4 `int`
- $T_1 = T_2 = T_3 = T_4$ = `int` – типы элементов T

- Пусть `alignof(int) == 4`, но `alignof(T) == 1` или `2`
  - т.е. нарушена кратность НОК(выравнивания элементов)

- Тогда разрешалось бы разместить массив `int a[4]` так, что

```
(size_t)&a % 4 == 2
```

- И доступ к элементам `a[0]` и `a[2]` приводил бы к undefined behavior

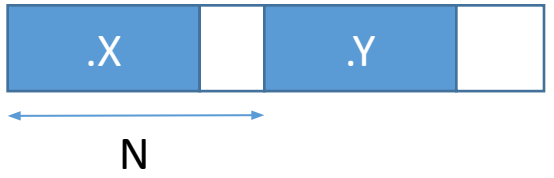
# Пример с выравниванием struct

# Пример с выравниванием struct

- T = `struct XY { int X; double Y; }`
- $T_1$ = `int`, $T_2$ = `double`

# Пример с выравниванием struct

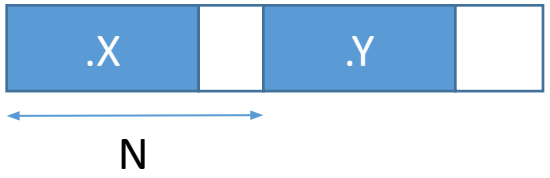
- T = `struct XY { int X; double Y; }`
- $T_1$ = `int`, $T_2$ = `double`



# Пример с выравниванием struct

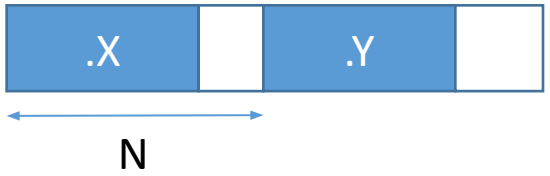
- T = `struct XY { int X; double Y; }`
- $T_1$ = `int`, $T_2$ = `double`

- Пусть `alignof(int)` ≤ `alignof(double)` == `8`, но `alignof`(T) == `1`, `2` или `4`
  - т.е. нарушена кратность НОК(выравнивания элементов)



# Пример с выравниванием struct

- T = `struct XY { int X; double Y; }`
- $T_1$ = `int`, $T_2$ = `double`

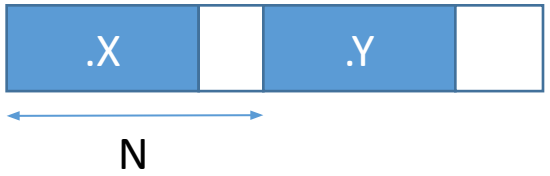



- Пусть `alignof(int)` ≤ `alignof(double)` == 8, но `alignof`(T) == 1, 2 или 4
  - т.е. нарушена кратность НОК(выравнивания элементов)

- Пусть a и b – переменные типа `struct XY`

# Пример с выравниванием struct

- T = `struct XY { int X; double Y; }`
- $T_1$ = `int`, $T_2$ = `double`

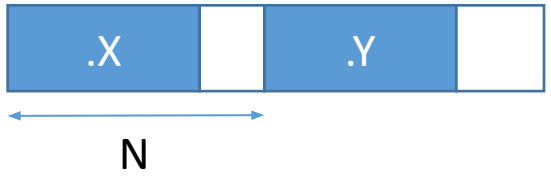



- Пусть `alignof(int)` ≤ `alignof(double)` == `8`, но `alignof`(T) == `1`, `2` или `4`
  - т.е. нарушена кратность НОК(выравнивания элементов)

- Пусть a и b – переменные типа `struct XY`
- При `alignof`(T) < `8` возможно `(size_t)&a % 8 == A > 0, (size_t)&b % 8 == 0`

# Пример с выравниванием struct

- T = `struct XY { int X; double Y; }`
- $T_1$ = `int`, $T_2$ = `double`



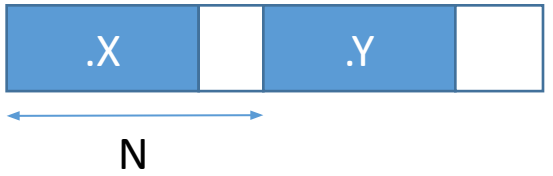

- Пусть `alignof(int)` ≤ `alignof(double)` == `8`, но `alignof`(T) == `1`, `2` или `4`
  - т.е. нарушена кратность НОК(выравнивания элементов)

- Пусть a и b – переменные типа `struct XY`
- При `alignof`(T) < `8` возможно `(size_t)&a % 8 == A > 0, (size_t)&b % 8 == 0`

- При доступе к `a.Y` должно быть `0 == (size_t)&a.Y % 8 == ((size_t)&a + N) % 8 == (A + N) % 8`
  - Иначе – undefined behavior

# Пример с выравниванием struct

- T = `struct XY { int X; double Y; }`
- $T_1$ = `int`, $T_2$ = `double`



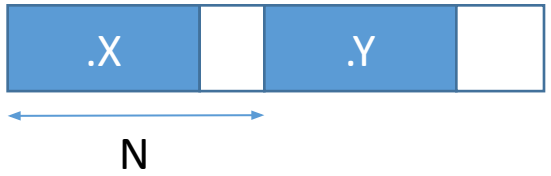

- Пусть `alignof(int)` ≤ `alignof(double)` == `8`, но `alignof`(T) == `1`, `2` или `4`
  - т.е. нарушена кратность НОК(выравнивания элементов)

- Пусть a и b – переменные типа `struct XY`
- При `alignof`(T) < `8` возможно `(size_t)&a % 8 == A > 0, (size_t)&b % 8 == 0`

- При доступе к `a.Y` должно быть `0 == (size_t)&a.Y % 8 == ((size_t)&a + N) % 8 == (A + N) % 8`
  - Иначе – undefined behavior
- При доступе к `b.Y` должно быть `0 == (size_t)&b.Y % 8 == ((size_t)&b + N) % 8 == N % 8`
  - Иначе – undefined behavior

# Пример с выравниванием struct

- T = `struct XY { int X; double Y; }`
- $T_1$ = `int`, $T_2$ = `double`




- Пусть `alignof(int)` ≤ `alignof(double)` == `8`, но `alignof`(T) == `1`, `2` или `4`
  - т.е. нарушена кратность НОК(выравнивания элементов)

- Пусть a и b – переменные типа `struct XY`
- При `alignof`(T) < `8` возможно `(size_t)&a % 8 == A > 0, (size_t)&b % 8 == 0`

- При доступе к `a.Y` должно быть `0 == (size_t)&a.Y % 8 == ((size_t)&a + N) % 8 == (A + N) % 8`
  - Иначе – undefined behavior
- При доступе к `b.Y` должно быть `0 == (size_t)&b.Y % 8 == ((size_t)&b + N) % 8 == N % 8`
  - Иначе – undefined behavior

- Требование `N % 8 == 0` противоречит `(A + N) % 8 == 0`, т.к. `A > 0`

# Выравнивающие байты в конце struct/union

# Выравнивающие байты в конце struct/union

- Для правильного выравнивания элементов массива T требуется, чтобы sizeof(T) был кратен alignof(T)

# Выравнивающие байты в конце struct/union

- Для правильного выравнивания элементов массива T требуется, чтобы sizeof(T) был кратен alignof(T)

- Поэтому компилятор может добавлять выравнивающие байты в конце структур и объединений

# Выравнивающие байты в конце struct/union

- Для правильного выравнивания элементов массива T требуется, чтобы sizeof(T) был кратен alignof(T)
- Поэтому компилятор может добавлять выравнивающие байты в конце структур и объединений

```
struct XY {
    double X;
    char Y;
};
// В зависимости от
// alignof(double),
//
sizeof(struct XY) == 16
// или 12
```

# Выравнивающие байты внутри struct

# Выравнивающие байты внутри struct

- Компилятор может добавлять выравнивающие байты между элементами структуры для правильного выравнивания ее элементов
  - см. N в примере про кратность выравниванию элементу структуры

# Выравнивающие байты внутри struct

- Компилятор может добавлять выравнивающие байты между элементами структуры для правильного выравнивания ее элементов
  - см. N в примере про кратность выравниванию элементу структуры

```
struct YX {
    char Y;
    double X;
};
// В зависимости от
// alignof(double),
// sizeof(struct YX) == 16
// или 12
```

# Динамическое распределение памяти

# Динамическое распределение памяти

- Программа в процессе работы сама резервирует и освобождает блоки памяти для хранения необходимых ей данных – использует динамическое распределение памяти

# Динамическое распределение памяти

- Программа в процессе работы сама резервирует и освобождает блоки памяти для хранения необходимых ей данных – использует динамическое распределение памяти

- Для резервирования и освобождения блока памяти используются стандартные функции языка Си

# Динамическое распределение памяти

- Программа в процессе работы сама резервирует и освобождает блоки памяти для хранения необходимых ей данных – использует динамическое распределение памяти

- Для резервирования и освобождения блока памяти используются стандартные функции языка Си

- Блоки памяти резервируются в специальной области памяти «куче» (heap)

# Динамическое распределение памяти

- Программа в процессе работы сама резервирует и освобождает блоки памяти для хранения необходимых ей данных – использует динамическое распределение памяти
- Для резервирования и освобождения блока памяти используются стандартные функции языка Си
- Блоки памяти резервируются в специальной области памяти «куче» (heap)
- Динамическое распределение памяти используется, если во время компиляции неизвестна «разумная» верхняя граница на максимальный размер обрабатываемых данных

# Стандартные функции malloc, free и др.

# Стандартные функции malloc, free и др.

- void* malloc(size_t size)
- void* realloc(void* ptr , size_t size)
- void* calloc(size_t count, size_t size)
- void free(void* ptr)

# Стандартные функции malloc, free и др.

- void* malloc(size_t size)
  - резервирует непрерывный блок из size байтов и возвращает указатель на него
- void* calloc(size_t count, size_t size)
- void* realloc(void* ptr , size_t size)
- void free(void* ptr)

# Стандартные функции malloc, free и др.

- void* malloc(size_t size)
  - резервирует непрерывный блок из size байтов и возвращает указатель на него
  - возвращает NULL, если резервирование невозможно
- void* calloc(size_t count, size_t size)
- void* realloc(void* ptr , size_t size)
- void free(void* ptr)

# Стандартные функции malloc, free и др.

- void* malloc(size_t size)
  - резервирует непрерывный блок из size байтов и возвращает указатель на него
  - возвращает NULL, если резервирование невозможно
- void* calloc(size_t count, size_t size)
  - резервирует непрерывный блок из count · size байтов, заполняет нулями и возвращает указатель на него
- void* realloc(void* ptr , size_t size)
- void free(void* ptr)

# Стандартные функции malloc, free и др.

- void* malloc(size_t size)
  - резервирует непрерывный блок из size байтов и возвращает указатель на него
  - возвращает NULL, если резервирование невозможно
- void* calloc(size_t count, size_t size)
  - резервирует непрерывный блок из count · size байтов, заполняет нулями и возвращает указатель на него
  - возвращает NULL, если резервирование невозможно
- void* realloc(void* ptr , size_t size)
- void free(void* ptr)

# Стандартные функции malloc, free и др.

- void* malloc(size_t size)
  - резервирует непрерывный блок из size байтов и возвращает указатель на него
  - возвращает NULL, если резервирование невозможно
- void* calloc(size_t count, size_t size)
  - резервирует непрерывный блок из count · size байтов, заполняет нулями и возвращает указатель на него
  - возвращает NULL, если резервирование невозможно
- void* realloc(void* ptr , size_t size)
  - резервирует непрерывный блок из size байтов и возвращает указатель на него
- void free(void* ptr)

# Стандартные функции malloc, free и др.

- void* malloc(size_t size)
  - резервирует непрерывный блок из size байтов и возвращает указатель на него
  - возвращает NULL, если резервирование невозможно
- void* calloc(size_t count, size_t size)
  - резервирует непрерывный блок из count · size байтов, заполняет нулями и возвращает указатель на него
  - возвращает NULL, если резервирование невозможно
- void* realloc(void* ptr , size_t size)
  - резервирует непрерывный блок из size байтов и возвращает указатель на него
  - переносит в новый блок min(size, размер блока по адресу ptr) байтов из блока по адресу ptr и освобождает его
- void free(void* ptr)

# Стандартные функции malloc, free и др.

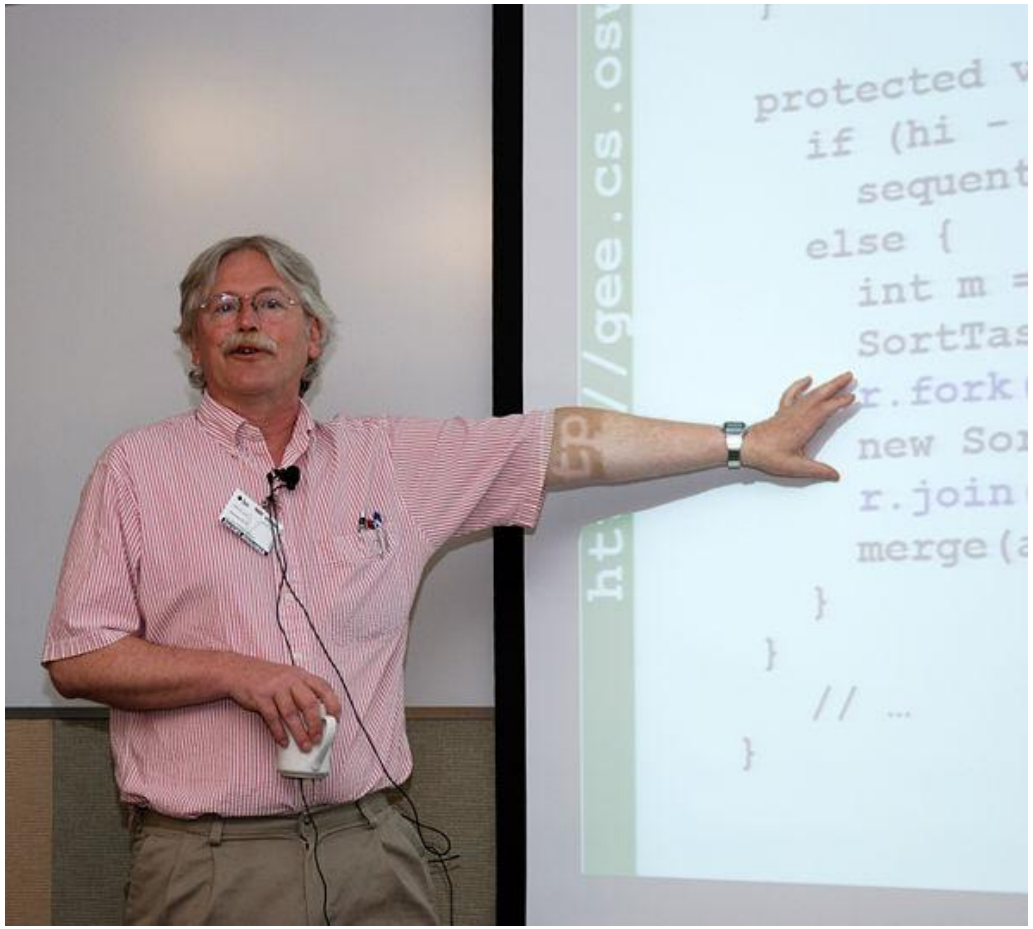
- void* malloc(size_t size)
  - резервирует непрерывный блок из size байтов и возвращает указатель на него
  - возвращает NULL, если резервирование невозможно
- void* calloc(size_t count, size_t size)
  - резервирует непрерывный блок из count · size байтов, заполняет нулями и возвращает указатель на него
  - возвращает NULL, если резервирование невозможно
- void* realloc(void* ptr , size_t size)
  - резервирует непрерывный блок из size байтов и возвращает указатель на него
  - переносит в новый блок min(size, размер блока по адресу ptr) байтов из блока по адресу ptr и освобождает его
  - возвращает NULL, если изменение размера невозможно
    - при этом блок по адресу ptr не освобождается, данные в нем сохраняются
- void free(void* ptr)

# Стандартные функции malloc, free и др.

- void* malloc(size_t size)
  - резервирует непрерывный блок из size байтов и возвращает указатель на него
  - возвращает NULL, если резервирование невозможно
- void* calloc(size_t count, size_t size)
  - резервирует непрерывный блок из count · size байтов, заполняет нулями и возвращает указатель на него
  - возвращает NULL, если резервирование невозможно
- void* realloc(void* ptr , size_t size)
  - резервирует непрерывный блок из size байтов и возвращает указатель на него
  - переносит в новый блок min(size, размер блока по адресу ptr) байтов из блока по адресу ptr и освобождает его
  - возвращает NULL, если изменение размера невозможно
    - при этом блок по адресу ptr не освобождается, данные в нем сохраняются
- void free(void* ptr)
  - освобождает ранее зарезервированный блок по адресу ptr

# Doug Lea’s malloc (dlmalloc)

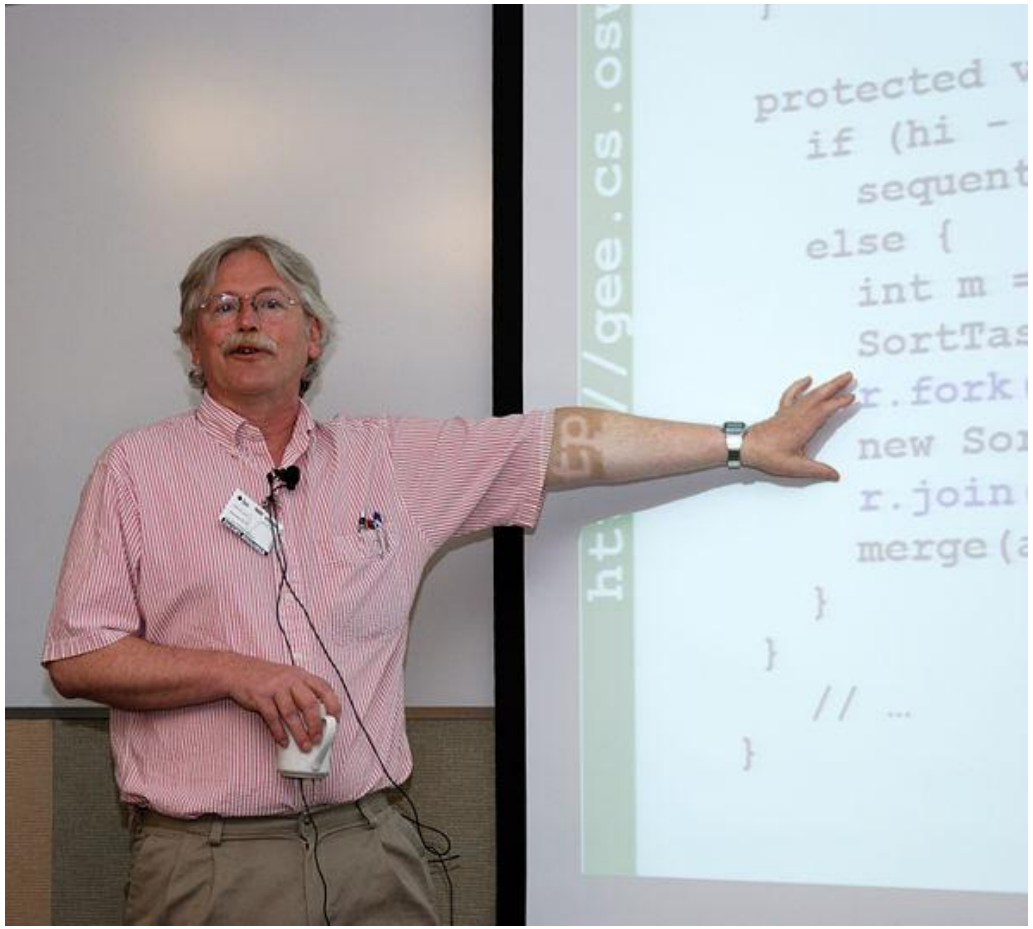
# Doug Lea’s malloc (dlmalloc)




Douglas (Doug) Lea, JVM Language Summit, 2010

# Doug Lea’s malloc (dlmalloc)

- Основа malloc в библиотеке GNU C (libc) для большинства версий Linux

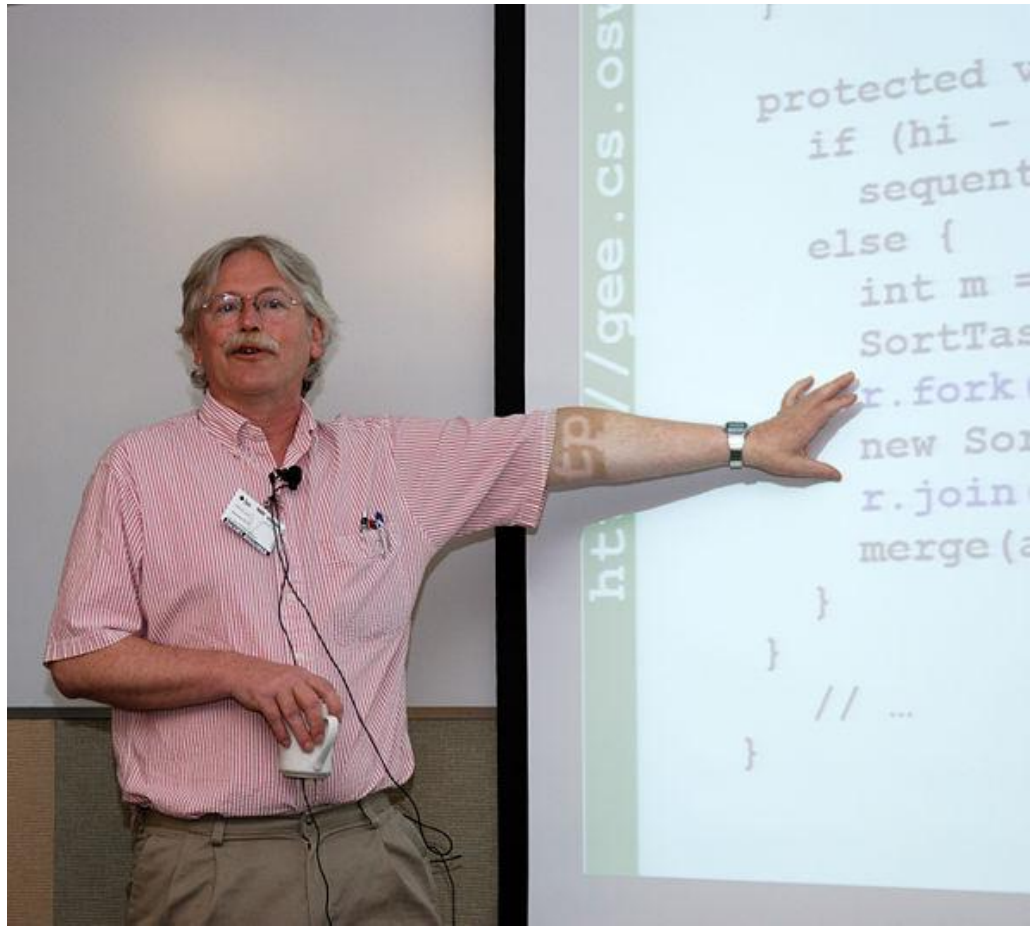



Douglas (Doug) Lea, JVM Language Summit, 2010

# Doug Lea’s malloc (dlmalloc)

- Основа malloc в библиотеке GNU C (libc) для большинства версий Linux

- http://gee.cs.oswego.edu/dl/html/malloc.html



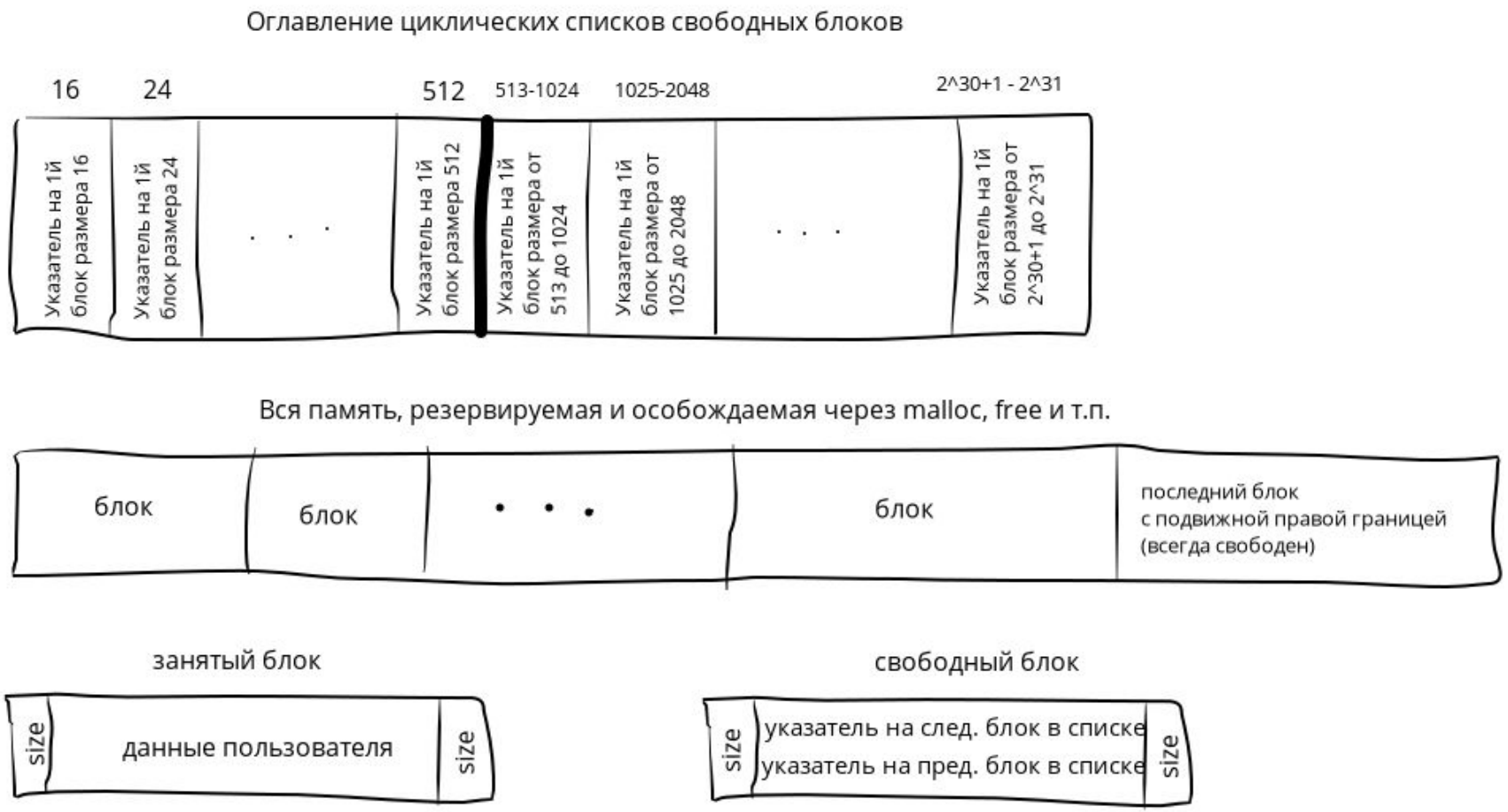

Douglas (Doug) Lea, JVM Language Summit, 2010

# Устройство «кучи»

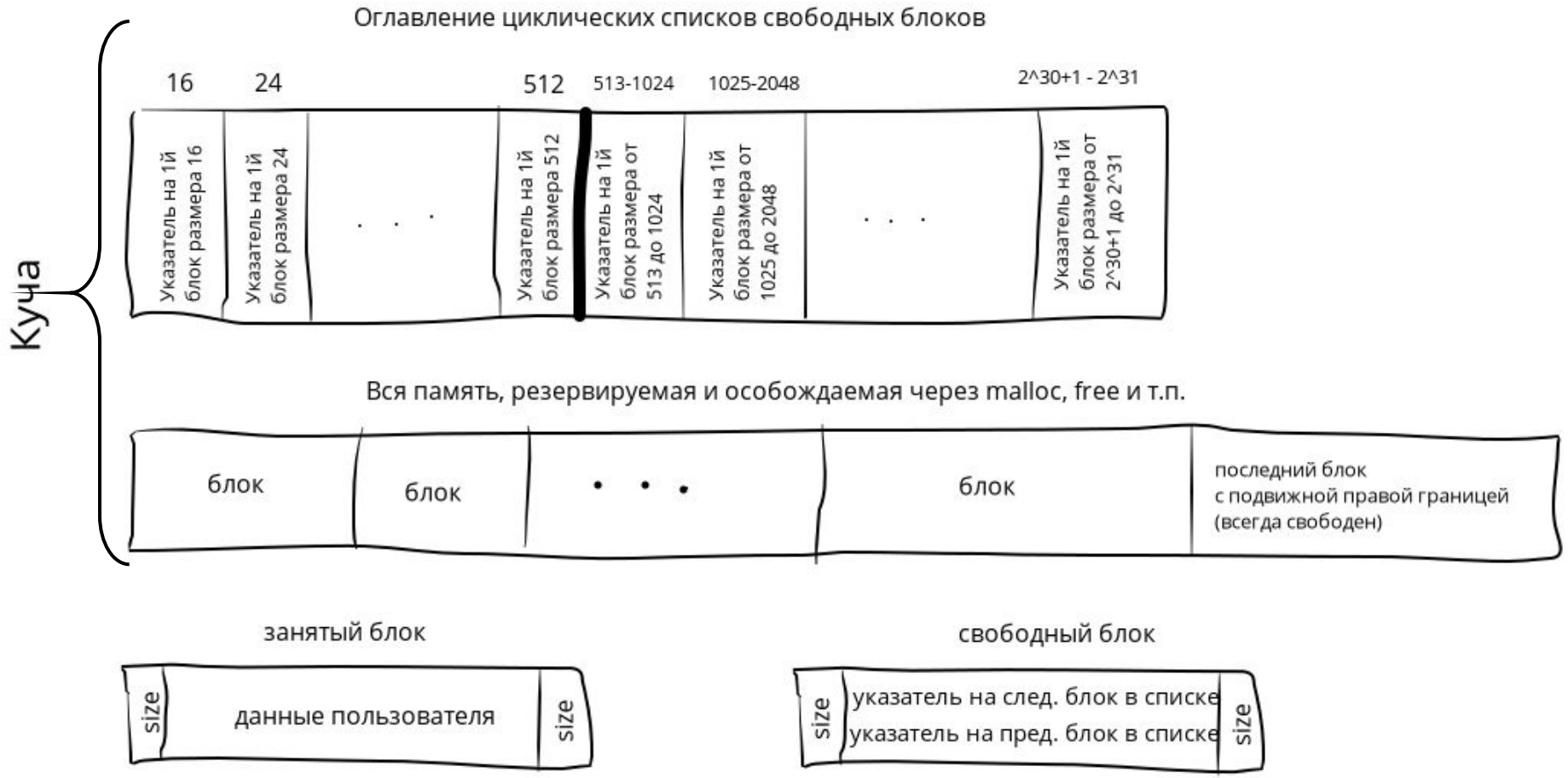
# Устройство «кучи»

# Устройство «кучи»

# Устройство «кучи»



# Устройство «кучи»



# Резервирование malloc(size)

# Резервирование malloc(size)

1. block = свободный блок min размера ≥ size

# Резервирование malloc(size)

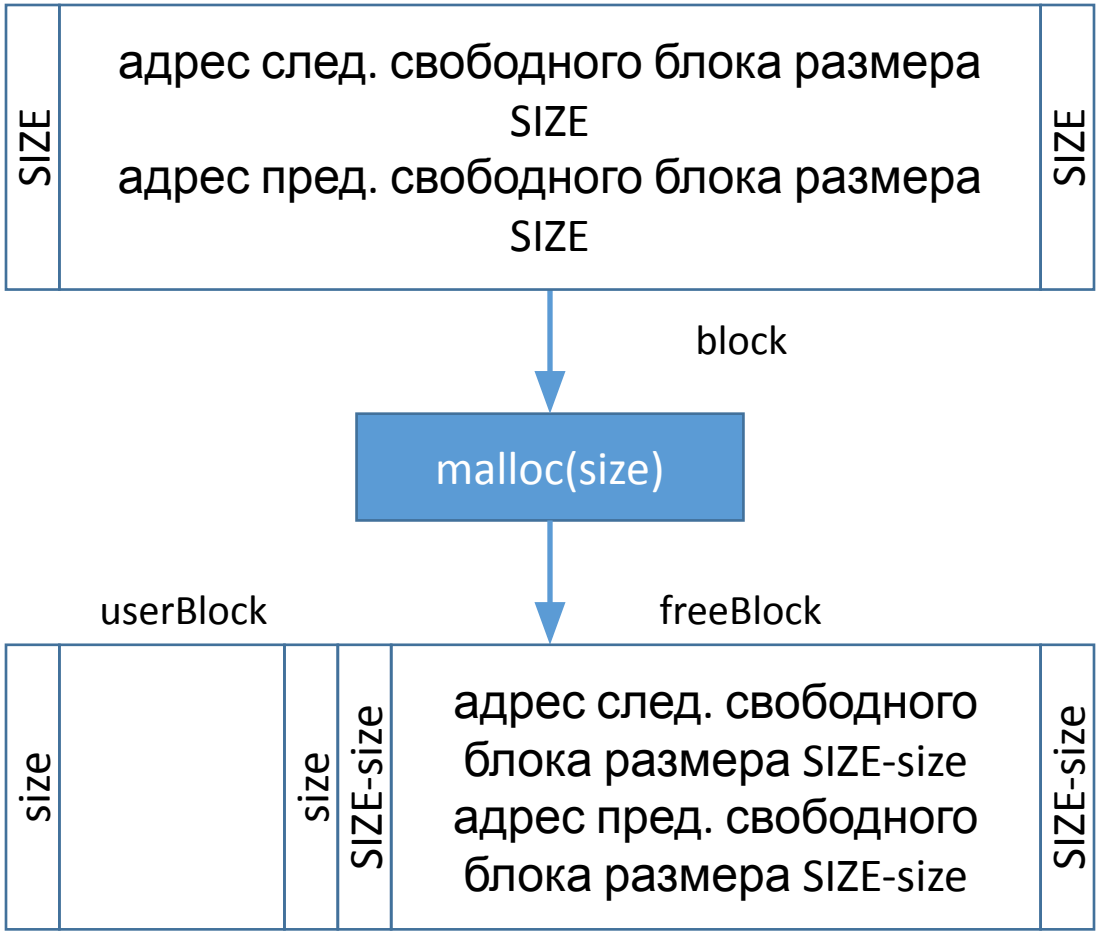
1. block = свободный блок min размера ≥ size
2. Если block не найден, то возвращаем NULL

# Резервирование malloc(size)

1. block = свободный блок min размера ≥ size
2. Если block не найден, то возвращаем NULL
3. Если размер(block) ≈ size, то возвращаем block + sizeof(size_t)

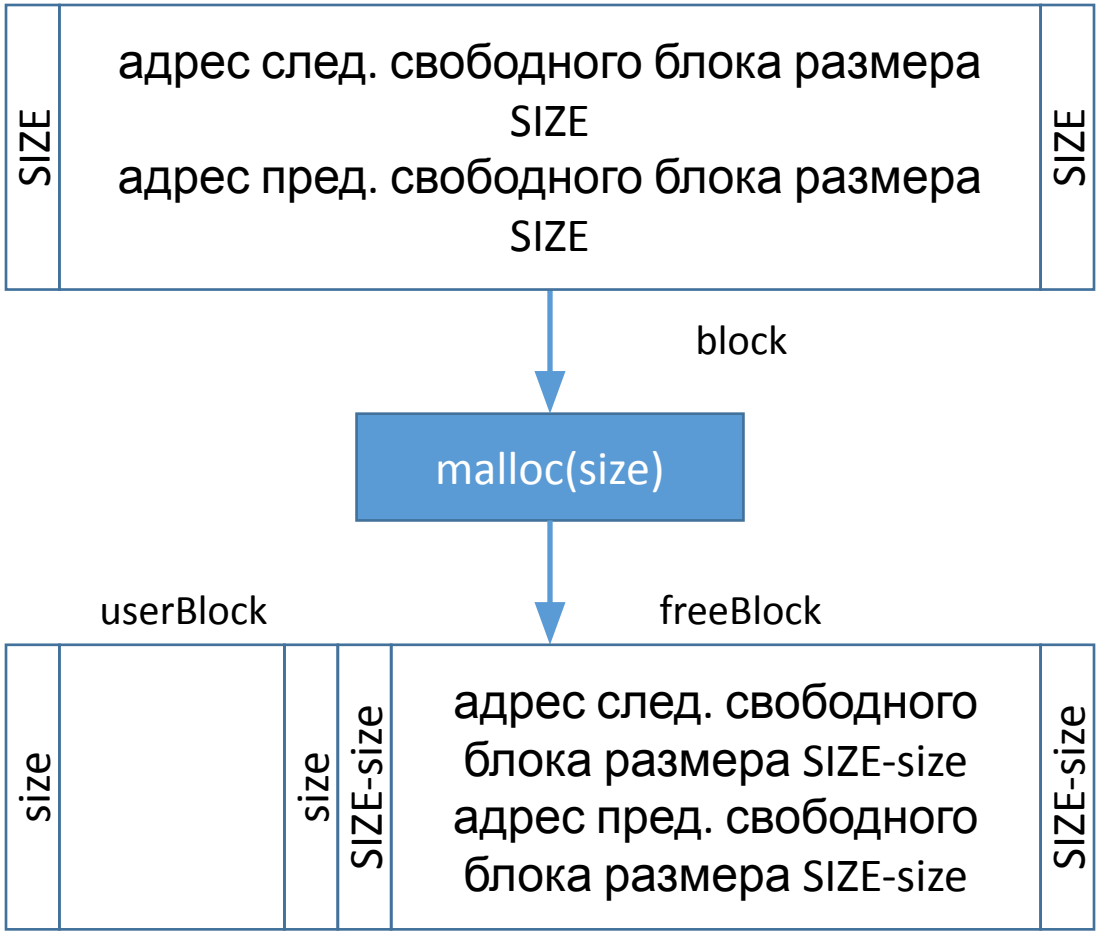
# Резервирование malloc(size)

1. block = свободный блок min размера ≥ size
2. Если block не найден, то возвращаем NULL
3. Если размер(block) ≈ size, то возвращаем block + sizeof(size_t)
4. Иначе режем block на userBlock и freeBlock, чтобы размер(userBlock ) ≈ size



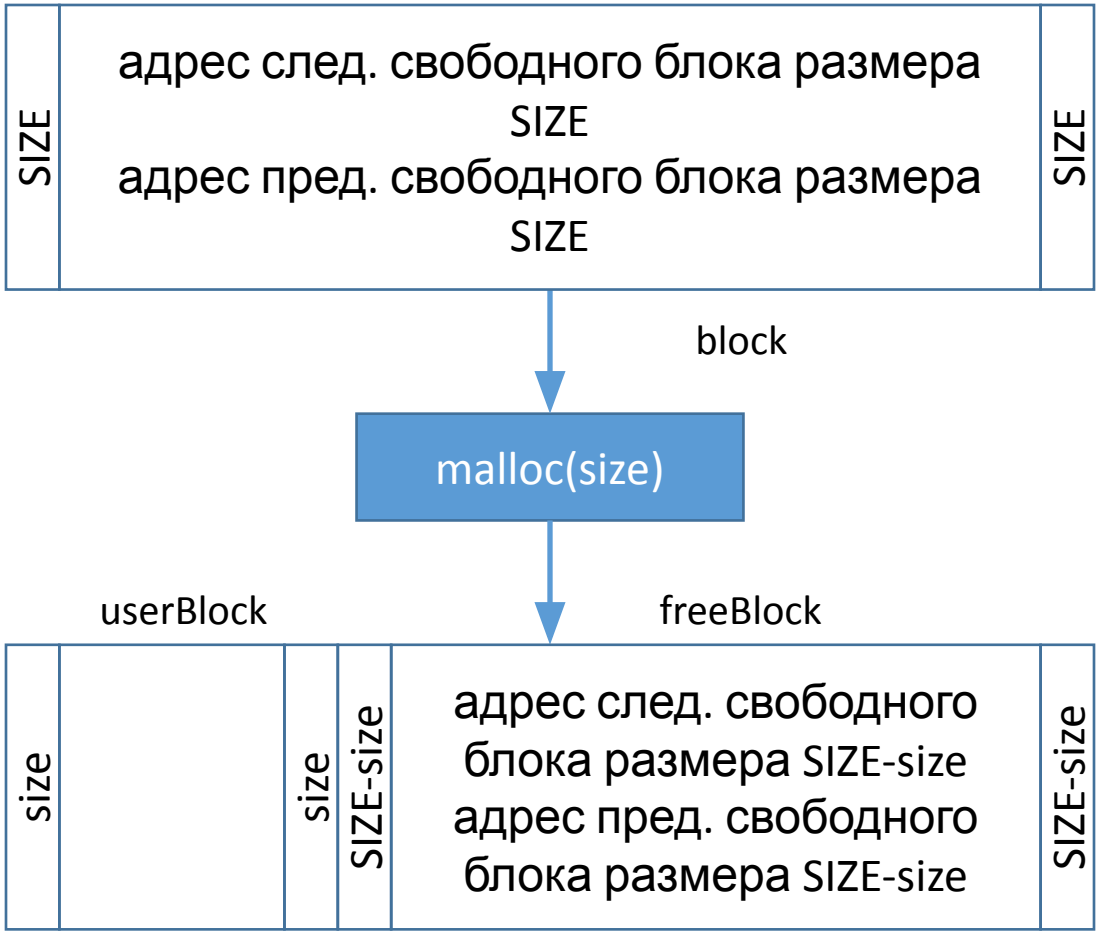
# Резервирование malloc(size)

1. block = свободный блок min размера ≥ size
2. Если block не найден, то возвращаем NULL
3. Если размер(block) ≈ size, то возвращаем block + sizeof(size_t)
4. Иначе режем block на userBlock и freeBlock, чтобы размер(userBlock ) ≈ size
5. Добавляем freeBlock в список свободных блоков размера размер(freeBlock)



# Резервирование malloc(size)

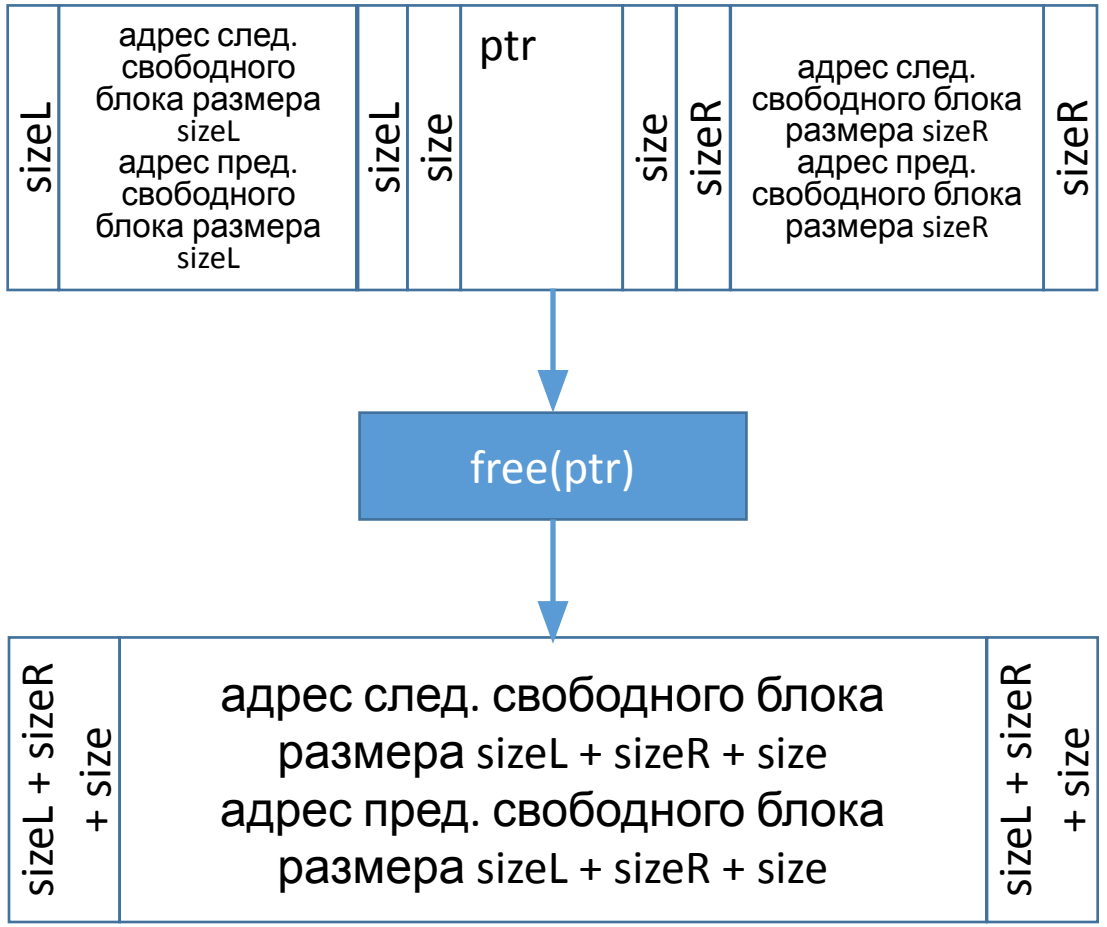
1. block = свободный блок min размера ≥ size
2. Если block не найден, то возвращаем NULL
3. Если размер(block) ≈ size, то возвращаем block + sizeof(size_t)
4. Иначе режем block на userBlock и freeBlock, чтобы размер(userBlock ) ≈ size
5. Добавляем freeBlock в список свободных блоков размера размер(freeBlock)
6. Возвращаем userBlock + sizeof(size_t)



# Освобождение free(ptr)

# Освобождение free(ptr)

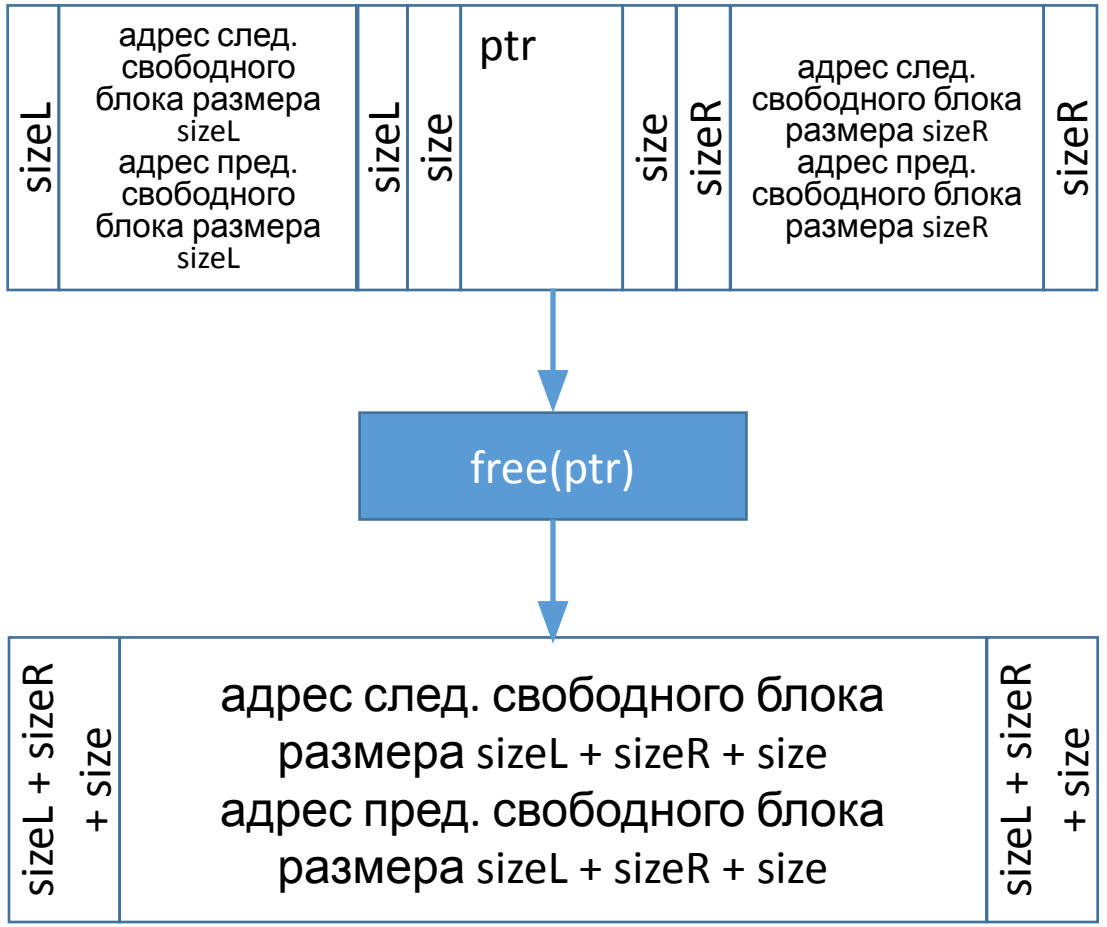
1. Объединяем с блоком по адресу ptr блок слева (если свободен) и блок справа (если свободен)
   - «слева» и «справа» по адресу в памяти, а не по связям в списке



# Освобождение free(ptr)

1. Объединяем с блоком по адресу ptr блок слева (если свободен) и блок справа (если свободен)
   - «слева» и «справа» по адресу в памяти, а не по связям в списке

2. Добавляем получившийся блок в список свободных блоков соотв. размера



# Накладные расходы при работе с кучей

# Накладные расходы при работе с кучей

- Поиск min свободного блока в malloc
  - malloc размера > 512 байтов и много свободных блоков в соотв. списке – большой накладной расход времени на просмотр списка свободных блоков

# Накладные расходы при работе с кучей

- Поиск min свободного блока в malloc
  - malloc размера > 512 байтов и много свободных блоков в соотв. списке – большой накладной расход времени на просмотр списка свободных блоков
- Дополнительные 2 · sizeof(size_t) байтов на каждый блок
  - Много malloc небольшого размера – большой накладной расход памяти

# Фрагментация кучи

# Фрагментация кучи

- Резервирование и освобождение блоков разного размера приводит к фрагментации

# Фрагментация кучи

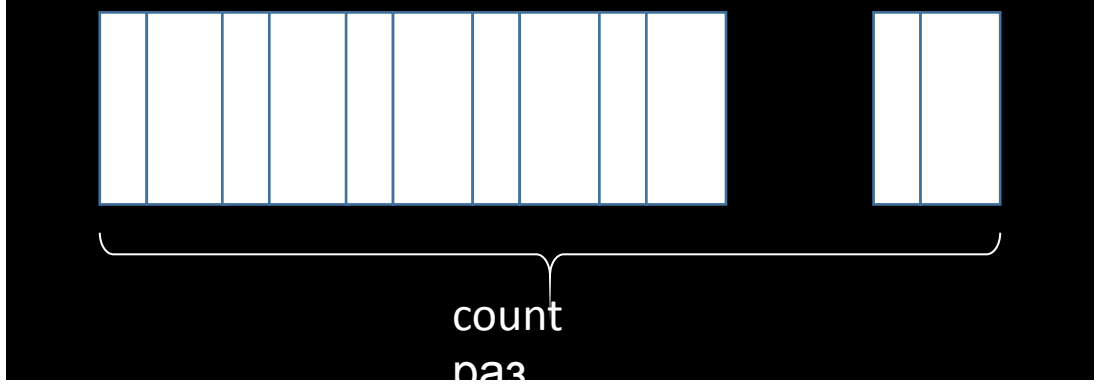
- Резервирование и освобождение блоков разного размера приводит к фрагментации

```
for (int i = 0; i < count; ++i) {
    void* small = malloc(8);
    bigger[i] = malloc(32);
    free(small);
}
```

# Фрагментация кучи

- Резервирование и освобождение блоков разного размера приводит к фрагментации

```
for (int i = 0; i < count; ++i) {
    void* small = malloc(8);
    bigger[i] = malloc(32);
    free(small);
}
```

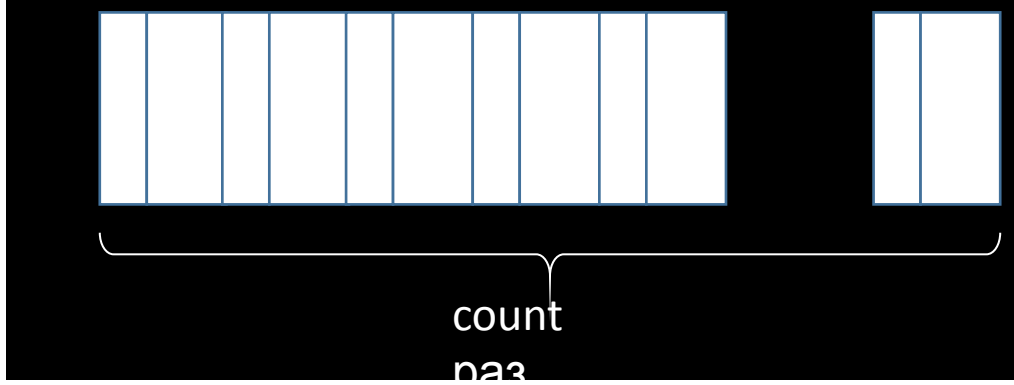


# Фрагментация кучи

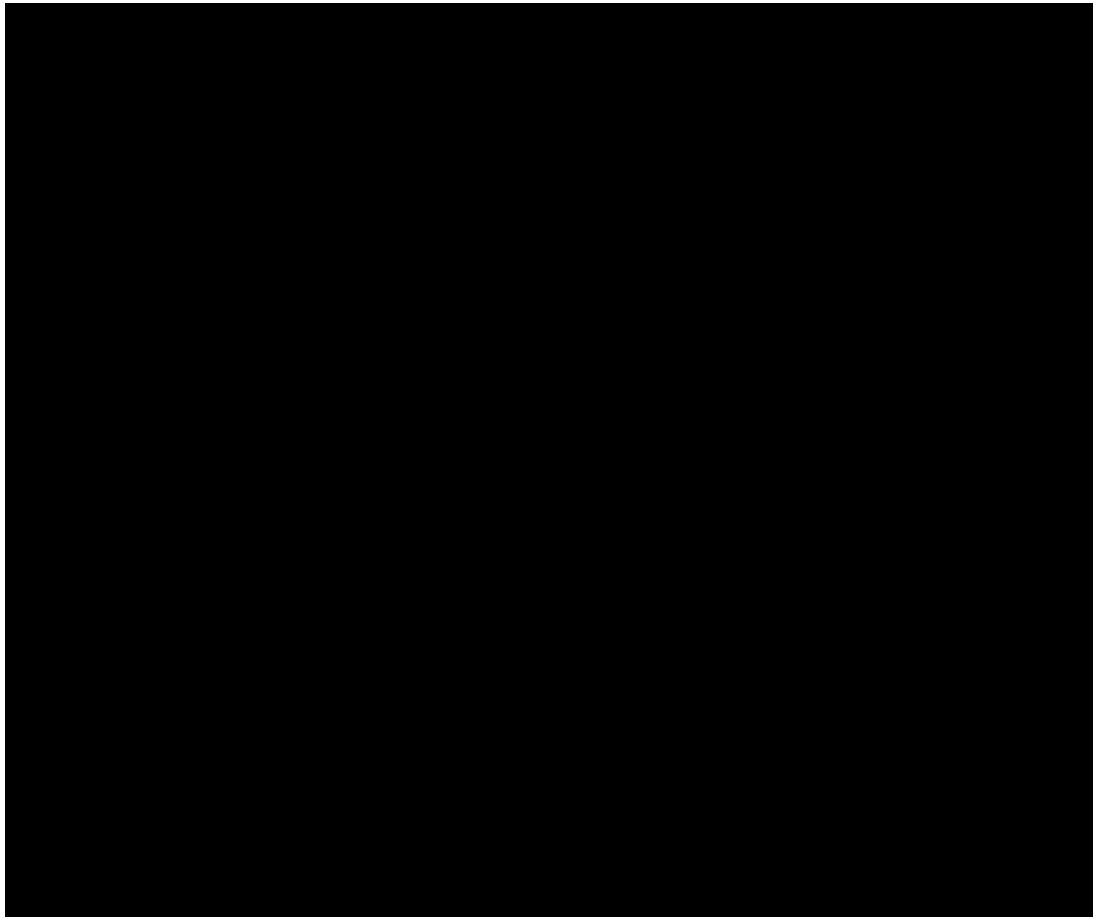
- Резервирование и освобождение блоков разного размера приводит к фрагментации

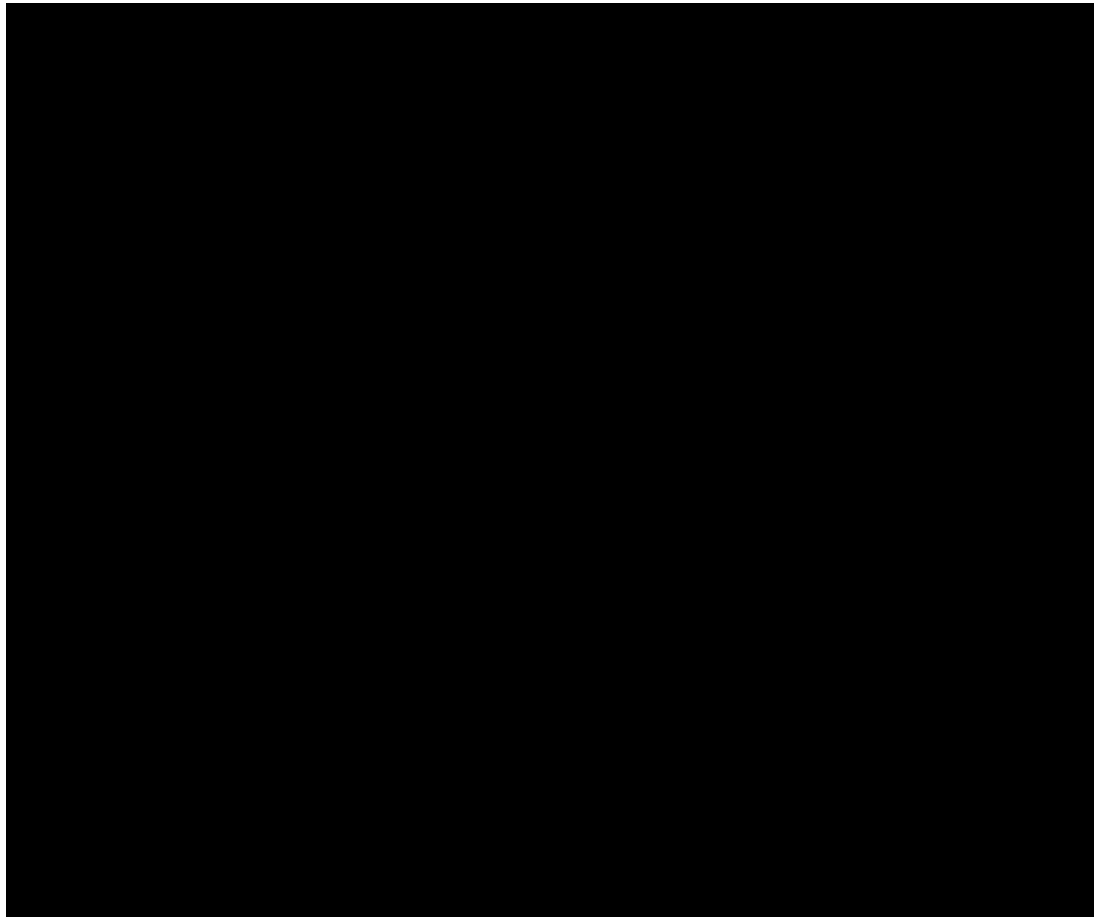
- Свободная память разбита на большое число мелких блоков и нет возможности зарезервировать блоки большего размера

```
for (int i = 0; i < count; ++i) {
    void* small = malloc(8);
    bigger[i] = malloc(32);
    free(small);
}
```



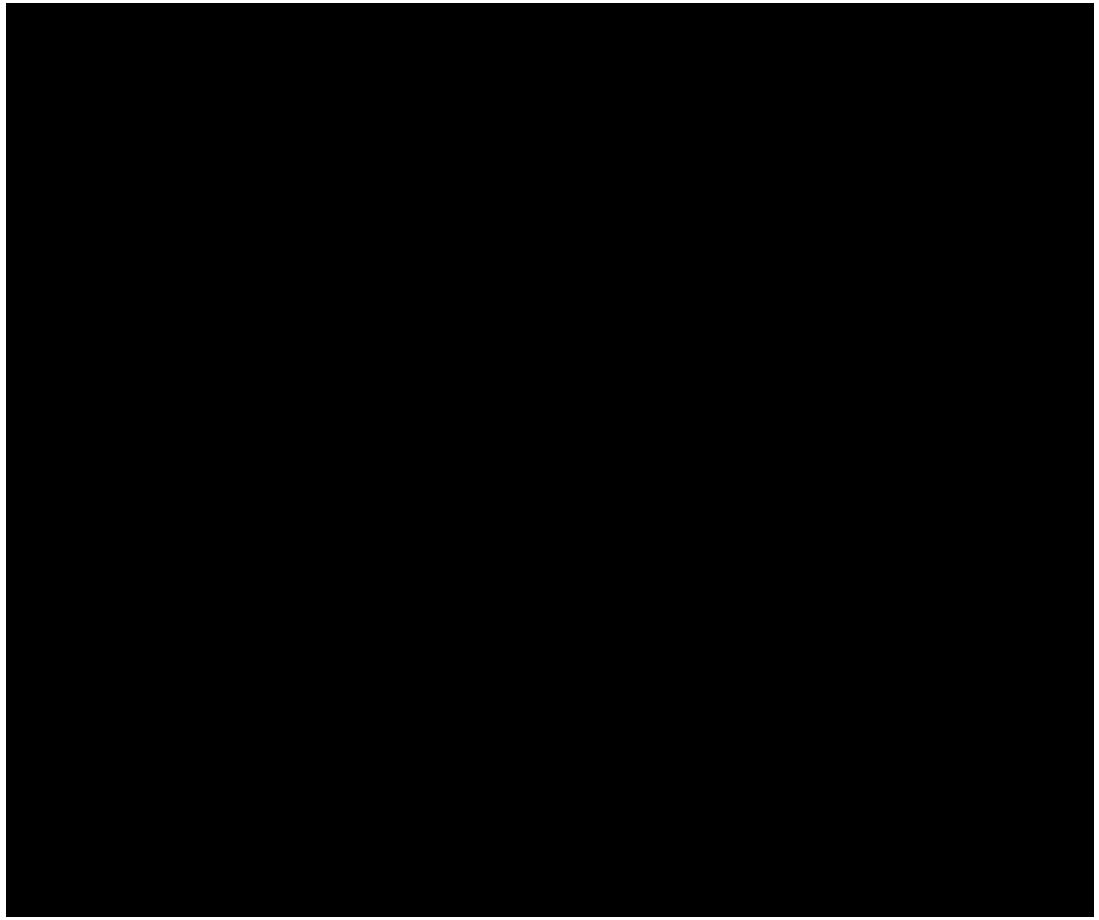
# Виды ошибок при работе с кучей





# Виды ошибок при работе с кучей

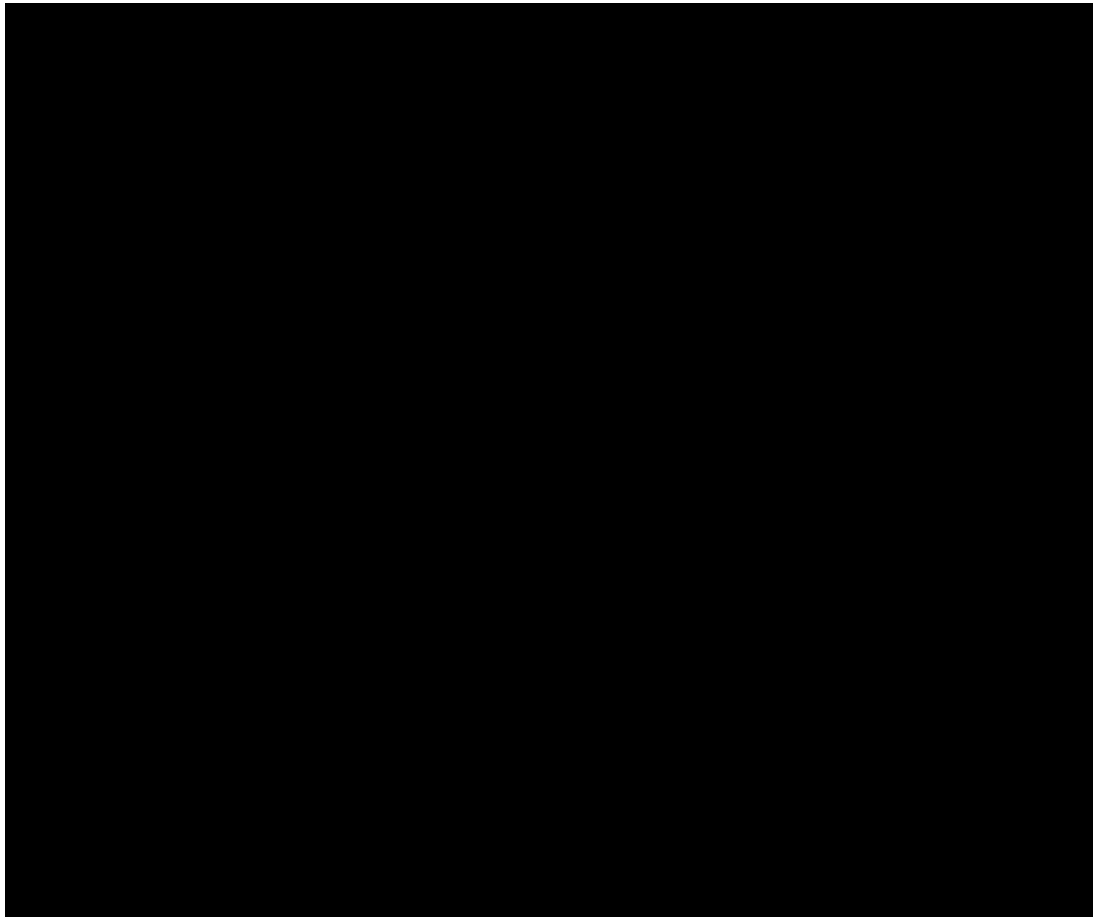
```
// missing NULL pointer check
int* ptr = malloc(4);
*ptr = 0; // <--
```



# Виды ошибок при работе с кучей

```
// missing NULL pointer check
int* ptr = malloc(4);
*ptr = 0; // <--

free(ptr);
*ptr = 0; // use after free
```

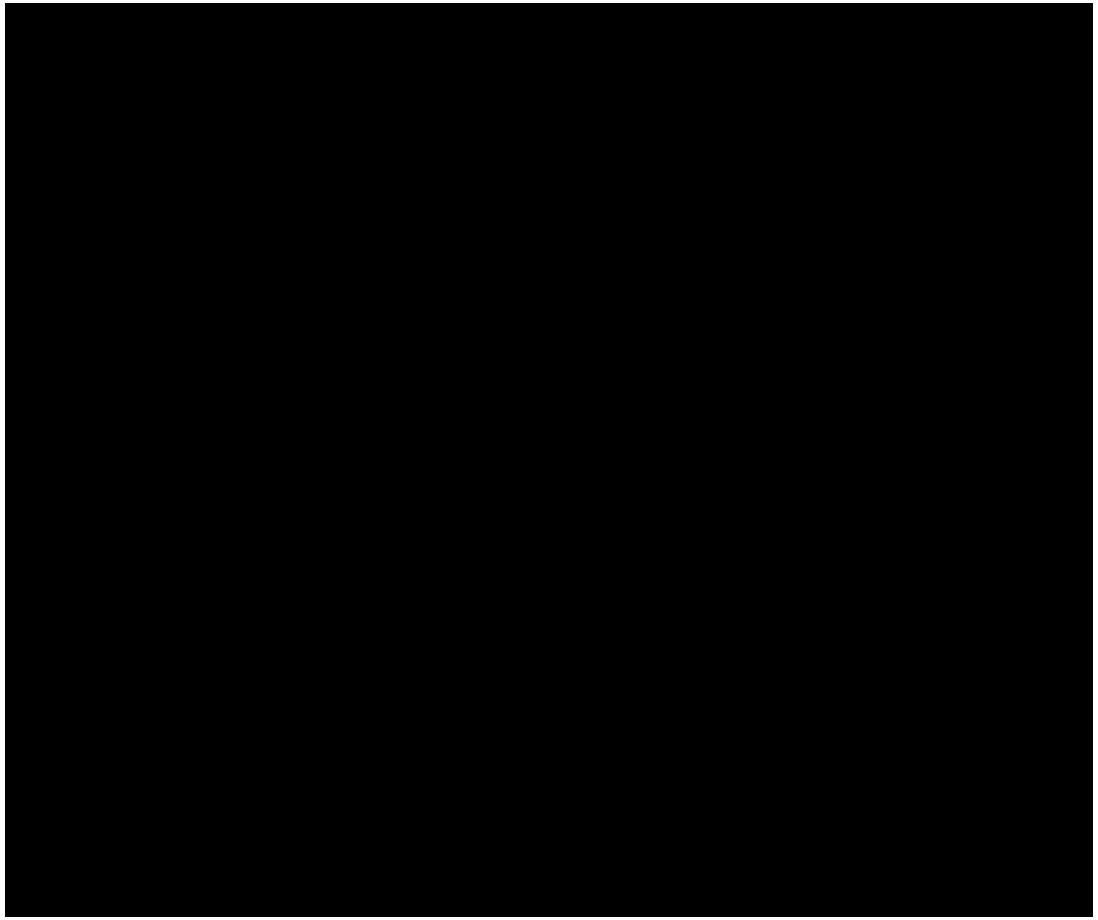


# Виды ошибок при работе с кучей

```
// missing NULL pointer check
int* ptr = malloc(4);
*ptr = 0; // <--

free(ptr);
*ptr = 0; // use after free

free(ptr); // double free
```



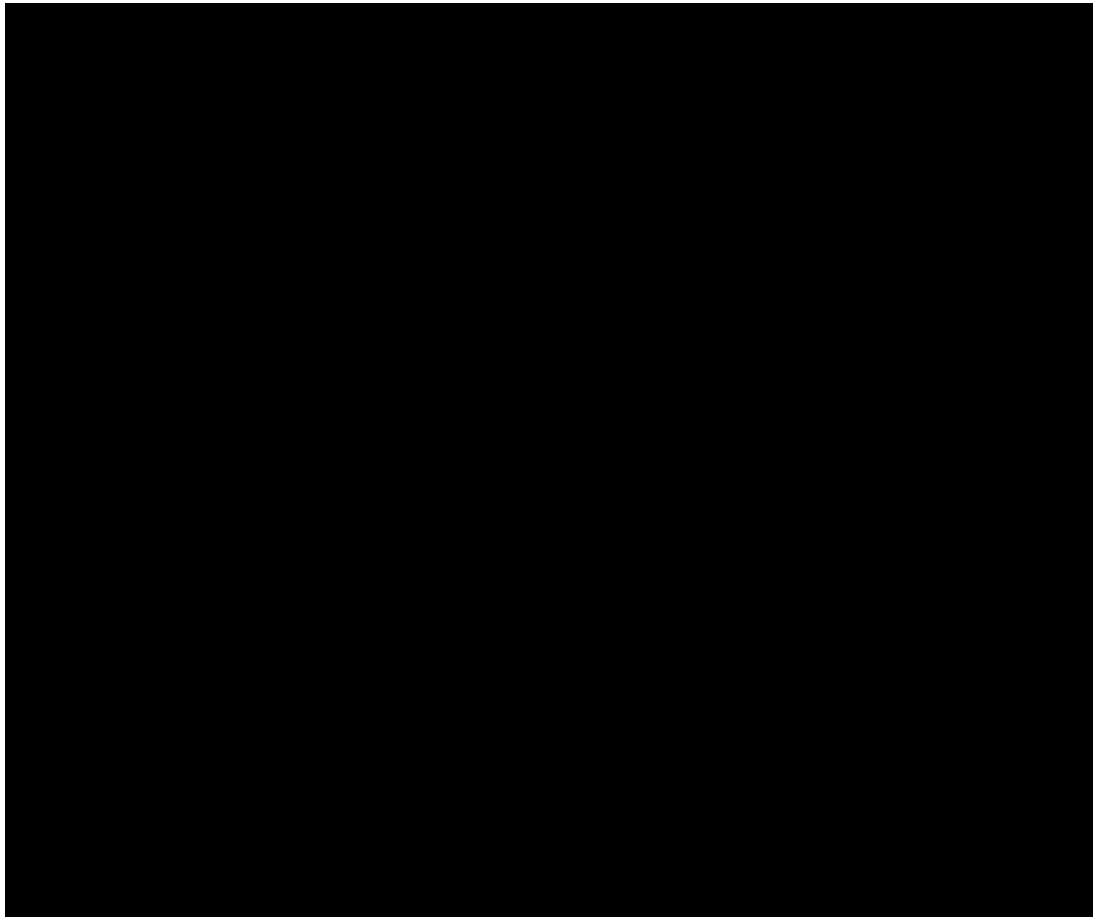
# Виды ошибок при работе с кучей

```
// missing NULL pointer check
int* ptr = malloc(4);
*ptr = 0; // <--

free(ptr);
*ptr = 0; // use after free

free(ptr); // double free

// freeing invalid pointer
ptr = malloc(8);
free(ptr + 4); // <--
```



# Виды ошибок при работе с кучей

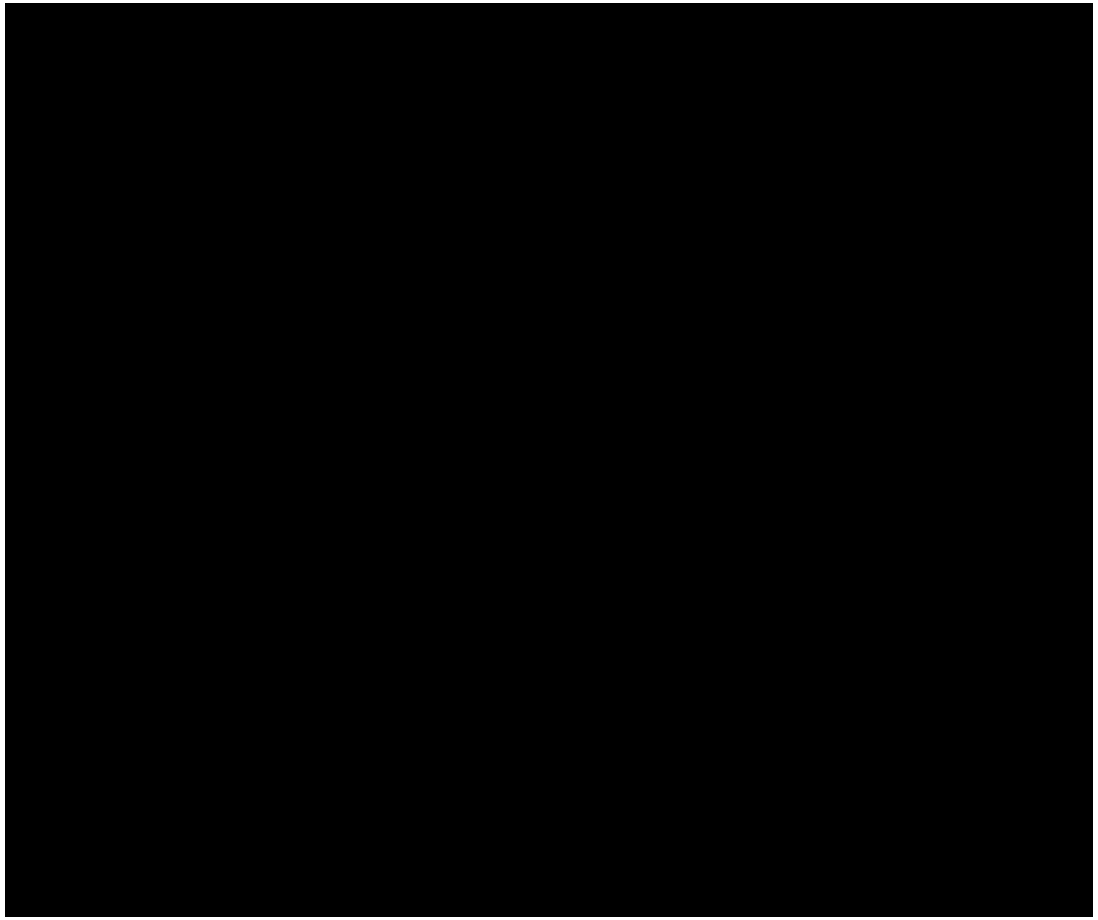
```
// missing NULL pointer check
int* ptr = malloc(4);
*ptr = 0; // <--

free(ptr);
*ptr = 0; // use after free

free(ptr); // double free

// freeing invalid pointer
ptr = malloc(8);
free(ptr + 4); // <--

free(&ptr);  // <--
```



# Виды ошибок при работе с кучей

```
// missing NULL pointer check
int* ptr = malloc(4);
*ptr = 0; // <--

free(ptr);
*ptr = 0; // use after free

free(ptr); // double free

// freeing invalid pointer
ptr = malloc(8);
free(ptr + 4); // <--

free(&ptr);  // <--
```

```
// memory leak
ptr = malloc(8);
ptr = malloc(8); // <--
```

# Виды ошибок при работе с кучей

```
// missing NULL pointer check
int* ptr = malloc(4);
*ptr = 0; // <--

free(ptr);
*ptr = 0; // use after free

free(ptr); // double free

// freeing invalid pointer
ptr = malloc(8);
free(ptr + 4); // <--

free(&ptr);  // <--
```

```
// memory leak
ptr = malloc(8);
ptr = malloc(8); // <--

// memory leak
ptr = realloc(ptr, 32); // <-- if OOM
```

# Виды ошибок при работе с кучей

```
// missing NULL pointer check
int* ptr = malloc(4);
*ptr = 0; // <--

free(ptr);
*ptr = 0; // use after free

free(ptr); // double free

// freeing invalid pointer
ptr = malloc(8);
free(ptr + 4); // <--

free(&ptr);  // <--
```

```
// memory leak
ptr = malloc(8);
ptr = malloc(8); // <--

// memory leak
ptr = realloc(ptr, 32); // <-- if OOM

// heap corruption
ptr = malloc(4);
for (int i = 0; i < 10; ++i) {
    ptr[i - 1] = 0; // <-- if i = 0
}
```

# Address sanitizer

# Address sanitizer

# Address sanitizer

- Константин Серебряный[1]
- Derek Bruening[2]
- Александр Потапенко[3]
- Дмитрий Вьюков[4]

1

2

3

4

# Address sanitizer

- Константин Серебряный[1]
- Derek Bruening[2]
- Александр Потапенко[3]
- Дмитрий Вьюков[4]
- AddressSanitizer: A Fast Address Sanity Checker, 2012
  - https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/37752.pdf

1

2

3

4

# Use after free

```
==5587==ERROR: AddressSanitizer: heap-use-after-free on address 0x61400000fe44 at pc 0x47b55f bp 0x7ffc36b28200 sp 0x7ffc36b281f8
READ of size 4 at 0x61400000fe44 thread T0
    #0 0x47b55e in main /home/test/example_UseAfterFree.cc:7
    #1 0x7f15cfe71b14 in __libc_start_main (/lib64/libc.so.6+0x21b14)
    #2 0x47b44c in _start (/root/a.out+0x47b44c)

0x61400000fe44 is located 4 bytes inside of 400-byte region [0x61400000fe40,0x61400000ffd0)
freed by thread T0 here:
    #0 0x465da9 in operator delete[](void*) (/root/a.out+0x465da9)
    #1 0x47b529 in main /home/test/example_UseAfterFree.cc:6

previously allocated by thread T0 here:
    #0 0x465aa9 in operator new[](unsigned long) (/root/a.out+0x465aa9)
    #1 0x47b51e in main /home/test/example_UseAfterFree.cc:5

SUMMARY: AddressSanitizer: heap-use-after-free /home/test/example_UseAfterFree.cc:7 main
```

```
int main(int argc, char **argv) {
  int *array = new int[100];
  delete [] array;
  return array[argc];  // BOOM
}
```

# Buffer overflow (heap corruption)

```
==25372==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x61400000ffd4 at pc 0x0000004ddb59 bp 0x7fffea6005a0 sp 0x7fffea600598
READ of size 4 at 0x61400000ffd4 thread T0
    #0 0x46bfee in main /tmp/main.cpp:4:13

0x61400000ffd4 is located 4 bytes to the right of 400-byte region [0x61400000fe40,0x61400000ffd0)
allocated by thread T0 here:
    #0 0x4536e1 in operator delete[](void*)
    #1 0x46bfb9 in main /tmp/main.cpp:2:16
```

```
int main(int argc, char **argv) {
  int *array = new int[100];
  array[0] = 0;
  int res = array[argc + 100];  // BOOM
  delete [] array;
  return res;
}
```

# Заключение

- Размещение данных в стековом кадре
  - Выравнивание
  - Связь выравниваний производного типа и его элементов
  - Выравнивающие байты
- Динамическое распределение памяти
  - Стандартные функции языка Си malloc, free и др.
    - Doug Lea’s malloc
  - Накладные расходы, фрагментация
  - Виды ошибок и address sanitizer