17.03.2025

Структура памяти процесса. Стек и куча. Интегральные вычисления. Форматы хранения данных.

Филиппов Михаил Витальевич

m.filippov@g.nsu.ru

89232283872

Императивное программирование, 2024-2025

Новосибирский государственный университет

*НАСТОЯЩАЯ НАУКА

Давайте познакомимся

Филиппов Михаил Витальевич

- Окончил магистратуру ФФ НГУ
- Окончил аспирантуру ИТ СО РАН
- Являюсь м.н.с. ИТ СО РАН
- 7+ лет опыт в программировании C/C++

Введение

Структура памяти процесса	Стек и куча	Интегральные вычисления	Форматы хранения данных
30 минут	30 минут	15 минут	15 минут

Введение

Структура памяти процесса	Стек и куча	Интегральные вычисления	Форматы хранения данных
30 минут	30 минут	15 минут	15 минут

Введение

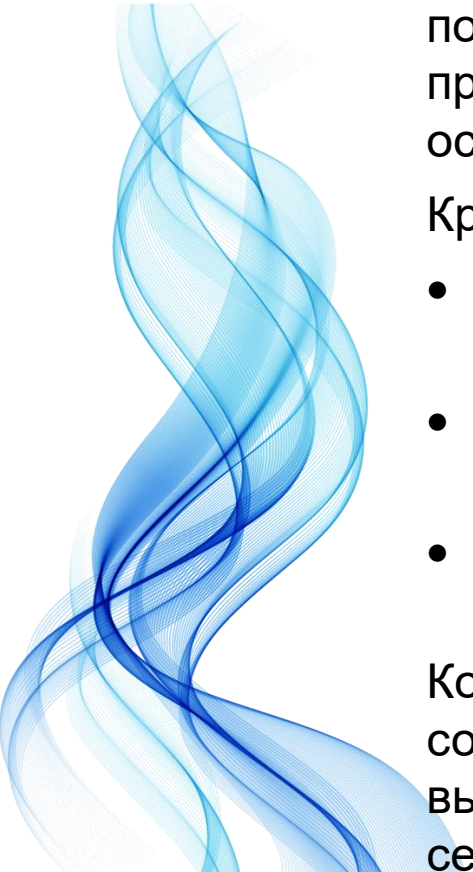
Управление памятью — крайне важная тема для любого программиста на языке C, и, чтобы применять рекомендуемые методики, необходимо иметь общее представление о ее структуре. На самом деле это касается не только C. Использование многих языков программирования, таких как C++ или Java, возможно при наличии базового понимания устройства и принципа работы памяти; в противном случае вы столкнетесь с серьезными проблемами, которые будет непросто выявить и исправить.

В языке C управление памятью полностью ручное. Более того, вся ответственность за выделение областей памяти и их освобождение после того, как они больше не нужны, ложится на программиста.

В высокоуровневых языках программирования, таких как Java или C#, управление памятью происходит иначе и частично выполняется внутренней платформой языка — например, Java Virtual Machine (JVM) в случае с Java. В таких языках программист занимается лишь выделением памяти, но не ее освобождением. Ресурсы освобождаются автоматически с помощью компонента под названием «сборщик мусора».

Поскольку сборщиков мусора в C и C++ нет, о концепциях и проблемах, относящихся к управлению памятью, необходимо поговорить отдельно.

Внутреннее устройство памяти процесса



При каждом запуске исполняемого файла операционная система создает новый процесс. Процесс — активная запущенная программа, которая загружена в память и имеет уникальный идентификатор (process identifier, PID). ОС полностью контролирует создание и загрузку новых процессов.

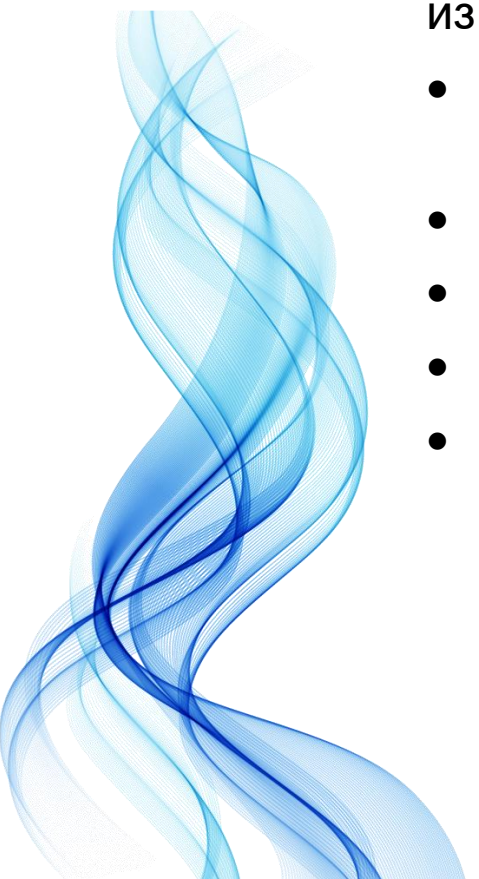
Процесс перестает быть активным либо в результате нормального завершения, либо при получении сигнала наподобие SIGTERM, SIGINT или SIGKILL, который в итоге заставляет его прекратить работу. Сигналы SIGTERM и SIGINT можно игнорировать, но SIGKILL останавливает процесс немедленно и принудительно.

Краткое описание упомянутых выше сигналов:

- SIGTERM — сигнал, запрашивающий завершение; дает возможность процессу подготовиться к выходу;
- SIGINT — сигнал прерывания, который обычно отправляется активным процессам путем нажатия Ctrl+C;
- SIGKILL — сигнал немедленного завершения; принудительно закрывает процесс, не давая ему возможности подготовиться.

Когда операционная система создает процесс, одно из первых действий, которые она совершает, — выделяет участок памяти с заранее определенной внутренней структурой. Та выглядит более или менее похоже в разных ОС, особенно в тех, которые принадле семейству Unix.

Внутреннее устройство памяти процесса



Память типичного процесса делится на несколько частей, которые называются сегментами. Каждый из них представляет собой область памяти с определенной задачей, предназначенную для хранения данных конкретного типа. Ниже приведен список сегментов, из которых состоит память активного процесса:

- сегмент неинициализированных данных или BSS (block started by symbol — блок, начинающийся с символа);
- сегмент данных;
- текстовый сегмент или сегмент кода;
- сегмент стека;
- сегмент кучи.

Исследование структуры памяти

Unix-подобные операционные системы предоставляют набор инструментов для исследования сегментов памяти процесса. Исполняемый объектный файл и процесс — две разные вещи, поэтому неудивительно, что для их исследования применяются разные инструменты.

Исполняемый объектный файл содержит машинные инструкции и за его создание отвечает компилятор. Но процесс — активная программа, созданная путем запуска исполняемого объектного файла; она занимает участок основной памяти, а центральный процессор постоянно извлекает и выполняет ее инструкции.

Процесс — динамическая сущность, выполняемая внутри операционной системы, в то время как **исполняемый объектный файл** — просто набор данных с подготовленной начальной структурой, и на ее основе создается будущий процесс. Действительно, некоторые сегменты в структуре памяти активного процесса берутся непосредственно из исполняемого файла, а остальные создаются динамически во время его загрузки. Первые называют статической схемой размещения в памяти, а вторые — динамической.

Статическая и динамическая схемы включают определенный набор сегментов. Содержимое статической заранее записывается в исполняемый объектный файл во время компиляции исходного кода. А динамическая схема формируется инструкциями процесса, которые выделяют память для переменных и массивов и изменяют ее в соответствии с логикой программы.

Исследование статической схемы размещения в памяти

Инструменты, которые используются для исследования статической памяти, обычно рассчитаны на объектные файлы. Минимальная программа на C:

```
int main(int argc, char **argv) { return 0; }
```

С помощью команды size можно вывести статическую схему размещения в памяти исполняемого объектного файла.

```
$ gcc main.c -o a.out
$ size a.out
   text    data     bss     dec     hex filename
   1235     544       8    1787     6fb a.out
```

```
$ clang main.c -o a.out
$ size a.out
   text    data     bss     dec     hex filename
   1218     528       8    1754     6da a.out
```

Сегменты Text, Data и BSS содержат в статической схеме размещения.

Сегмент BSS

Название BSS расшифровывается как block started by symbol (блок, начинающийся с символа). С давних пор так обозначают области памяти, зарезервированные для неинициализированных машинных слов. Сегмент BSS фактически предназначен для хранения либо неинициализированных, либо обнуленных глобальных переменных.

```
int global_var1;
int global_var2;
int global_var3 = 0;
int main(int argc, char **argv) { return 0; }
```

Изменение размера сегмента BSS

```
$ clang main.c -o a.out
$ size a.out
   text    data     bss     dec     hex filename
   1218     528      16    1762     6e2 a.out
```

Объявление неинициализированных или обнуленных глобальных переменных увеличивает сегмент BSS. Эти особые глобальные переменные — часть статической схемы размещения, и место для них выделяется во время загрузки процесса; кроме того, они не покидают память, пока процесс не завершит работу. Иными словами, имеют статический жизненный цикл.

Сегмент Data

Объявим больше глобальных переменных, однако на сей раз инициализируем их с помощью ненулевых значений.

```
int global_var1;
int global_var2;
int global_var3 = 0;
double global_var4 = 4.5;
char global_var5 = 'A';
int main(int argc, char **argv) { return 0; }
```

Изменение размера сегмента Data

```
$ clang main.c -o a.out
$ size a.out
   text    data     bss     dec     hex filename
   1218     537      20    1775     6ef a.out
```

Сегмент Data предназначен для хранения инициализированных глобальных переменных с ненулевыми значениями. Если сравнить вывод команды size в примерах, то в глаза сразу бросается то, что сегмент Data увеличился на 9 байт; это общий размер двух добавленных нами переменных (восьмибайтной типа double и однобайтной типа char).

Сегмент Data

Чтобы просмотреть то, что хранится внутри сегментов объектного файла, каждая операционная система предоставляет свои инструменты. Например, в Linux содержимое ELF-файла можно проанализировать с помощью команд readelf и objdump, которые также позволяют исследовать статическую схему размещения внутри объектных файлов.
Помимо глобальных, есть также статические переменные, объявленные внутри функций. При многократном вызове одной и той же функции эти переменные не меняют свои значения. В зависимости от платформы и наличия значения они могут храниться как в сегменте Data, так и в BSS.

```
void func() {
    static int i;
    static int j = 1;
    ...
}
```

Переменные i и j — статические. Первая не инициализирована, а вторая имеет значение 1.
На этапе выполнения эти переменные находятся либо в сегменте Data, либо в BSS (которые обладают статическим жизненным циклом), и функция func имеет к ним доступ. Поэтому их, в сущности, и называют статическими. Мы знаем, что переменная j хранится в сегменте Data, просто потому, что имеет начальное значение; в то же время переменная i не инициализирована, поэтому должна находиться в сегменте BSS.

Сегмент Data

Теперь познакомимся со второй командой для исследования содержимого сегмента BSS в Linux, objdump (macOS – gobjdump). С ее помощью можно выводить сегменты памяти объектных файлов. В, но там ее нужно самостоятельно установить.

```
int x = 33; // 0x00000021
int y = 0x12153467;
char z[6] = "ABCDE";
int main(int argc, char **argv) { return 0; }
```

Использование команды objdump

```
$ gcc main.c -o a.out
$ objdump -s -j .data a.out

a.out:     file format elf64-x86-64

Contents of section .data:
 4000 00000000 00000000 08400000 00000000  .........@......
 4010 21000000 67341512 41424344 4500      !...g4..ABCDE.
```

Обратите внимание на параметры -s и -j .data: первый сообщает команде objdump о том, что нужно вывести все содержимое заданной секции, а второй указывает секцию .data.
Двоичное содержимое объектного файла можно читать с помощью таких утилит, как hexdump.

Сегмент Data

Теперь познакомимся со второй командой для исследования содержимого сегмента BSS в Linux, objdump (macOS – gobjdump). С ее помощью можно выводить сегменты памяти объектных файлов. В, но там ее нужно самостоятельно установить.

```
int x = 33; // 0x00000021
int y = 0x12153467;
char z[6] = "ABCDE";
int main(int argc, char **argv) { return 0; }
```

Использование команды objdump

```
$ gcc main.c -o a.out
$ objdump -s -j .data a.out

a.out:     file format elf64-x86-64

Contents of section .data:
 4000 00000000 00000000 08400000 00000000  .........@......
 4010 21000000 67341512 41424344 4500      !...g4..ABCDE.
```

Обратите внимание на параметры -s и -j .data: первый сообщает команде objdump о том, что нужно вывести все содержимое заданной секции, а второй указывает секцию .data.
Двоичное содержимое объектного файла можно читать с помощью таких утилит, как hexdump.

Сегмент Text

Итоговый исполняемый файл записываются инструкции машинного уровня. Поскольку все машинные инструкции программы находятся в сегменте Text (или Code), он должен находиться в исполняемом объектном файле — а именно, в его статической схеме размещения. Процессор извлекает эти инструкции и выполняет их во время работы процесса.

```c
int main(int argc, char **argv) {
    return 0;
}
```

Здесь показано несколько секций с машинными инструкциями, включая .text, .init и .plt. Все вместе они делают возможными загрузку и выполнение программы. Каждая из этих секций — часть сегмента Text, который входит в статическую схему размещения в исполняемом объектном файле.

Перед вызовом функции main выполняется другая логика. В Linux эти функции обычно заимствуются из библиотеки glibc и используются для формирования готовой программы.

Использование objdump для вывода содержимого секции относящейся к функции main

```
$ gcc main.c -o a.out
$ objdump -S a.out
a.out:     file format elf64-x86-64
Disassembly of section .init:

0000000000001000 <_init>:
    1000:       f3 0f 1e fa             endbr64
…
    101a:       c3                      ret
Disassembly of section .plt:
0000000000001020 <.plt>:
…
0000000000001129 <main>:
    1129:       f3 0f 1e fa             endbr64
    112d:       55                      push   %rbp
    112e:       48 89 e5                mov    %rsp,%rbp
    1131:       89 7d fc                mov    %edi,-0x4(%rbp)
    1134:       48 89 75 f0             mov    %rsi,-0x10(%rbp)
    1138:       b8 00 00 00 00          mov    $0x0,%eax
    113d:       5d                      pop    %rbp
    113e:       c3                      ret
Disassembly of section .fini:
0000000000001140 <_fini>:
    1140:       f3 0f 1e fa             endbr64
…
    114c:       c3                      ret
```

Исследование динамической схемы размещения в памяти

Динамическая схема размещения находится в памяти процесса и существует до тех пор, пока он не завершится. Процедурой запуска исполняемого объектного файла занимается программа под названием «загрузчик». Он создает новый процесс и его начальную схему размещения в памяти, которая должна быть динамической. Для этого копируются сегменты, найденные в статической схеме размещения исполняемого объектного файла. Затем к ним добавляется два новых сегмента. И только после этого процесс может приступить к выполнению.

Если коротко, то в памяти активного процесса должно быть пять сегментов, три из которых копируются непосредственно из статической схемы размещения исполняемого объектного файла, а остальные два создаются с нуля и называются стеком и кучей. Последние являются динамическими сегментами и существуют только в период работы процесса. Это значит, вы не найдете никакого упоминания о них в исполняемом объектном файле.

Стек — область памяти, в которой по умолчанию выделяется место для переменных. Она имеет ограниченный размер, поэтому большие объекты в ней хранить нельзя. Для сравнения, **куча** — более крупная и гибкая область памяти, в которой могут поместиться большие объекты и массивы. Для работы с кучей нужен отдельный API.

Отражение памяти

Код, представленный ниже, выполняется бесконечно долго. Таким образом, мы получим процесс, который никогда не завершается, что позволит нам изучить структуру его памяти. Закончив исследование, мы сможем от него избавиться с помощью команды kill.

```c
#include <unistd.h> // Needed for sleep function
// #include <windows.h> в windows
int main(int argc, char **argv) {
    // Бесконечный цикл
    while (1)
        sleep(1); // Засыпаем на 1 секунду
    return 0;
}
```

Запуск примера в фоновом режиме

```
$ gcc main.c -o a.out
$ ./a.out &
[1] 641
```

PID (process ID — идентификатор процесса), вы можете легко завершить процесс, используя команду kill. Например, если значение PID равно 641, то следующая команда будет работать в любой операционной системе семейства Unix: kill -9 641. **PID** — идентификатор, который мы используем для исследования памяти процесса. Обычно система предоставляет собственные механизмы получения различных свойств активных процессов на основе их PID.

Отражение памяти

Чтобы получить список отражений, нужно просмотреть содержимое файла maps в каталоге /proc/641. Выведем файл /proc/402/maps с помощью команды cat

```
cat /proc/641/maps
555594f09000-555594f0a000 r--p 00000000 00:4e 87820192733770704          /mnt/…/lection_23/a.out
555594f0a000-555594f0b000 r-xp 00001000 00:4e 87820192733770704          /mnt/…/lection_23/a.out
555594f0b000-555594f0c000 r--p 00002000 00:4e 87820192733770704          /mnt/…/lection_23/a.out
555594f0c000-555594f0d000 r--p 00002000 00:4e 87820192733770704          /mnt/…/lection_23/a.out
555594f0d000-555594f0e000 rw-p 00003000 00:4e 87820192733770704          /mnt/…/lection_23/a.out
7f59fc65f000-7f59fc662000 rw-p 00000000 00:00 0
7f59fc662000-7f59fc68a000 r--p 00000000 08:20 22554                      /usr/lib/x86_64-linux-gnu/libc.so.6
7f59fc68a000-7f59fc81f000 r-xp 00028000 08:20 22554                      /usr/lib/x86_64-linux-gnu/libc.so.6
7f59fc81f000-7f59fc877000 r--p 001bd000 08:20 22554                      /usr/lib/x86_64-linux-gnu/libc.so.6
7f59fc877000-7f59fc878000 ---p 00215000 08:20 22554                      /usr/lib/x86_64-linux-gnu/libc.so.6
7f59fc878000-7f59fc87c000 r--p 00215000 08:20 22554                      /usr/lib/x86_64-linux-gnu/libc.so.6
7f59fc87c000-7f59fc87e000 rw-p 00219000 08:20 22554                      /usr/lib/x86_64-linux-gnu/libc.so.6
7f59fc87e000-7f59fc88b000 rw-p 00000000 00:00 0
7f59fc894000-7f59fc896000 rw-p 00000000 00:00 0
7f59fc896000-7f59fc898000 r--p 00000000 08:20 22551        /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
7f59fc898000-7f59fc8c2000 r-xp 00002000 08:20 22551        /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
7f59fc8c2000-7f59fc8cd000 r--p 0002c000 08:20 22551        /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
7f59fc8ce000-7f59fc8d0000 r--p 00037000 08:20 22551        /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
7f59fc8d0000-7f59fc8d2000 rw-p 00039000 08:20 22551        /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
7ffdba85d000-7ffdba87f000 rw-p 00000000 00:00 0                          [stack]
7ffdba8b4000-7ffdba8b8000 r--p 00000000 00:00 0                          [vvar]
7ffdba8b8000-7ffdba8ba000 r-xp 00000000 00:00 0                          [vdso]
```

Отражение памяти

Каждая из строк представляет отражение памяти с диапазоном выделенных адресов (областью); она привязана к определенному файлу или сегменту в динамической схеме размещения процесса. У каждого отражения есть поля, разделенные одним или несколькими пробелами.

- **Диапазон адресов** — начальный и конечный адреса отраженного диапазона. Если данная область отражена в файл, то путь к нему указан рядом. Это элегантный способ отражения одного и того же загруженного разделяемого объектного файла в разных процессах.
- **Разрешения** — этот столбец определяет, можно ли выполнять (x), читать (r) или изменять (w) содержимое. Область памяти также можно разделять (s) с другими процессами или изолировать (p) ее в рамках процесса, которому она принадлежит.
- **Сдвиг** — если область отражена в файл, то это сдвиг относительно начала данного файла. Если не отражена, то данный столбец обычно содержит 0.
- **Устройство** — если область отражена в файл, то здесь указан номер устройства (в формате m:n), которое хранит данный файл. Это, к примеру, может быть номер жесткого диска, на котором находится разделяемый объектный файл.
- **inode** — файл, в который отражена область памяти, должен находиться в файловой системе и иметь в ней свой номер inode. Последний указывается в данном столбце. Файл inode — абстракция в файловых системах наподобие ext4, которые в основном используются в ОС семейства Unix. Номер inode может относиться как к файлам, так и к каталогам, и его используют для доступа к их содержимому.
- **Путь или описание** — если область отражена в файл, то здесь указывается путь к нему. В противном случае данный столбец может быть пустым или описывать назначение области.

Например, метка [stack] говорит о том, что это стек.

Стек

Стек - ключевая часть динамической памяти любого процесса, предусмотренная почти во всех существующих архитектурах. В отражениях памяти она помечена как [stack].

Стек и куча хранят динамические данные, которые постоянно меняются во время выполнения программы. Просмотреть динамическое содержимое этих сегментов не слишком просто; в большинстве случаев для этого необходим отладчик наподобие gdb, который читает байты памяти активного процесса.

Стек обычно имеет ограниченный размер и плохо подходит для хранения крупных объектов. Если в этом сегменте закончится свободное место, то процесс больше не сможет вызывать функции, поскольку стек активно используется механизмом вызова. В таких случаях процесс принудительно завершается операционной системой. Переполнение стека — широко известная ошибка, которая возникает при полном заполнении стека.

Стек — область памяти, в которой по умолчанию выделяется место для переменных.

```
void func() {
    // память, необходимая для следующей переменной, выделяется в стеке
    int a;
    ...
}
```

Стек

Термин **«стек»** происходит от английского слова stack (стопка, штабель). Когда вы объявляете локальную переменную, она создается на вершине стека. При выходе из функции компилятор снимает со стека ее локальные переменные, в результате чего на вершину поднимаются значения внешней области видимости.

В абстрактном виде стек представляет собой структуру данных **FILO** (first in, last out — «первым пришел, последним вышел») или **LIFO** (last in, first out — «последним пришел, первым вышел»). В стеке хранятся не только переменные. При каждом вызове функции на стек кладется новая запись под названием «стековый фрейм». Иначе вы бы не смогли вернуться в предыдущую функцию или вернуть результат вызывающему коду.

Наличие надежного механизма стека — неотъемлемая часть рабочей программы. Поскольку стек ограничен в размере, в нем рекомендуется объявлять небольшие переменные, создавать слишком много стековых фреймов.

С другой стороны, **стек** — область памяти, в которой программист хранит данные и объявляет локальные переменные для своих алгоритмов. Операционная система, отвечающая за запуск программ, использует стек в целях размещения данных, необходимых ее внутренним механизмам.

Содержимое стека не так просто просматривать извне, используя только те инструменты, с которыми мы познакомились при исследовании статической схемы размещения. Эта область памяти содержит приватные данные, которые могут требовать деликатного обращения. Кроме того, она доступна лишь одному процессу.

Куча

```c
#include <unistd.h> // для функции sleep
#include <stdlib.h> // для функции malloc
#include <stdio.h>  // для printf
int main(int argc, char **argv)
{   // выделяем для кучи 1 Кбайт
    void *ptr = malloc(1024);
    printf("Address: %p\n", ptr);
    fflush(stdout); // для принудительного вывода
    // бесконечный цикл
    while (1)
        sleep(1); // засыпаем на 1 секунду
    return 0;
}
```

Запуск примера в фоновом режиме

```
$ gcc main.c -o a.out
$ ./a.out &
[1] 17374
Address: 0x55c45aa012a0
```

malloc – это основной механизм выделения дополнительной памяти в сегменте кучи. Она принимает количество байтов, которые нужно выделить, и возвращает обобщенный указатель.

Обобщенный (пустой) указатель содержит адрес памяти, но его нельзя разыменовать или использовать напрямую. Вначале его следует привести к определенному типу.

Запущенный в фоне процесс получил PID 17374.

Куча

```c
#include <stdio.h>  // для функции printf
#include <stdlib.h> // для функций malloc и free
void fill(char *ptr) {
    ptr[0] = 'H';
    ptr[1] = 'e';
    ptr[2] = 'l';
    ptr[3] = 'l';
    ptr[5] = 0;
}
int main(int argc, char **argv) {
    void *gptr = malloc(10 * sizeof(char));
    char *ptr = (char *)gptr;
    fill(ptr);
    printf("%s!\n", ptr);
    free(ptr);
    return 0;
}
```

Эта программа выделяет в куче 10 байт, используя функцию malloc. Та принимает количество байтов, которое нужно выделить, и возвращает обобщенный указатель на первый байт выделенного блока памяти.

Обратите внимание: переменные локальных указателей, gptr и ptr, выделяются в стеке. Им нужно место для хранения своих значений, и оно находится в стековой памяти. Но адреса, на которые они указывают, хранятся в куче. В этом нет ничего необычного.

Когда речь идет о куче, за выделение памяти отвечает программа или, скорее, программист. Вдобавок программа должна сама освобождать память, которая ей больше не нужна. Наличие участка кучи, к которому нельзя обратиться, называется утечкой памяти. Это значит, у нас нет указателя для работы с данной областью.

Введение

Структура памяти процесса	Стек и куча	Интегральные вычисления	Форматы хранения данных
30 минут	30 минут	15 минут	15 минут

Стек

Процесс может продолжать выполнение и без кучи, но без стека работать не будет. **Стек** — главный аспект метаболизма процесса. Это продиктовано тем, как происходит вызов функций. В предыдущей главе я уже упоминал, что функцию можно вызвать, только используя стек. Без этого сегмента нельзя выполнить ни одну функцию, что делает невозможной работу программы в целом.

Стек и его содержимое тщательно спроектированы для обеспечения бесперебойной работы процесса. Поэтому беспорядок в стеке может нарушить и прервать выполнение программы. Выделение памяти в стеке происходит быстро и не требует применения никаких специальных функций. Более того, освобождение ресурсов и все действия по управлению памятью происходят автоматически. Все описанное звучит заманчиво и может подстегнуть вас к излишнему использованию стека.

```c
#include <string.h>
int main(int argc, char **argv)
{
    char str[10];
    strcpy(str, "akjsdhkhqiueryo34928739r27yeiwuyfiusdciuti7twe79ye");
    return 0;
}
```

Если запустить этот код, то программа, скорее всего, аварийно завершится. Дело в том, что функция strcpy переполняет содержимое стека.

Исследование содержимого стека

Стек — изолированная область памяти, читать и модифицировать которую имеет право только ее владелец. Чтобы работать со стеком, необходимо быть частью процесса, которому он принадлежит.

Отладчик – программа, которая подключается к стороннему процессу и позволяет его отлаживать. При этом программисты обычно занимаются отслеживанием и изменением различных сегментов памяти. Читать и модифицировать приватные блоки памяти можно только в ходе отладки. Отладчик также позволяет управлять порядком выполнения программных инструкций.

```c
#include <stdio.h>
int main(int argc, char **argv) {
    char arr[4];
    arr[0] = 'A';
    arr[1] = 'B';
    arr[2] = 'C';
    arr[3] = 'D';
    return 0;
}
```

Память, необходимая для массива arr, выделяется в стеке, а не в куче просто потому, что мы не использовали функцию malloc. Чтобы выделить место в куче, необходимо воспользоваться функцией malloc или ее аналогом, таким как calloc. В противном случае память будет выделена в стеке, а точнее, на его вершине.

Чтобы программу можно было отлаживать, ее двоичный файл должен быть собран соответствующим образом. То есть нам нужно сообщить компилятору о том, что в исполняемом файле должны быть отладочные символы. Они будут использоваться для поиска строчек кода, которые выполняются или приводят к сбою.

Исследование содержимого стека

```c
#include <stdio.h>
int main(int argc, char **argv) {
    char arr[4];
    arr[0] = 'A';
    arr[1] = 'B';
    arr[2] = 'C';
    arr[3] = 'D';
    return 0;
}
```

Компиляция с отладочным параметром -g

```
$ gcc -g main.c -o a.out
```

Параметр **-g** говорит компилятору о том, что итоговый исполняемый объектный файл должен содержать отладочную информацию. *Обратите внимание:* он влияет на размер программы.

```
$ gcc main.c -o a.out
$ ls -al a.out
-rwxrwxrwx 1 mikhail mikhail 15968 Feb 22 13:55 a.out
$ gcc -g main.c -o a.out
$ ls -al a.out
-rwxrwxrwx 1 mikhail mikhail 17240 Feb 22 13:56 a.ouv
```

Итак, у нас есть исполняемый файл с отладочными символами. Теперь мы можем запустить его с помощью отладчика. В нашем примере для отладки используется gdb.

Исследование содержимого стека

```
$ gdb a.out
GNU gdb (Ubuntu 12.1-0ubuntu1~22.04.2) 12.1
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
<http://gnu.org/licenses/gpl.html>
...
Reading symbols from a.out...
(gdb) run
Starting program: /mnt/c/.../lection_23_1/a.out
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-
gnu/libthread_db.so.1".
[Inferior 1 (process 3745) exited normally]
(gdb) break main
Breakpoint 1 at 0x55555555515c: file main.c, line 3.
(gdb)
```

Отладчик gdb предоставляет интерфейс командной строки, который позволяет выполнять отладочные команды. Введите команду r (или run), чтобы запустить исполняемый объектный файл, поданный на вход отладчику.

В этом терминале после ввода команды run отладчик запускает процесс, подключается к нему и дает возможность программе выполнить свои инструкции вплоть до завершения. Он не прерывает выполнение, поскольку мы не указали точку останова. Она говорит gdb о том, что процесс нужно приостановить и ждать дальнейших инструкций. Создадим точку останова в функции main, используя команду b (или break). В результате gdb остановит выполнение, когда программа войдет в main.

Исследование содержимого стека

```
(gdb) r
Starting program: /mnt/c/…/lection_23_1/a.out
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-
gnu/libthread_db.so.1".

Breakpoint 1, main (argc=1, argv=0x7fffffffdaf8) at
main.c:3
3  {
(gdb) n
5           arr[0] = 'A';
(gdb) n
6           arr[1] = 'B';
(gdb) n
7           arr[2] = 'C';
(gdb) next
8           arr[3] = 'D';
(gdb) next
9           return 0;
(gdb) print arr
$1 = "ABCD"
```

Как видите, выполнение остановилось на строчке 3, с которой начинается функция main. После этого отладчик ждет ввода следующей команды. Мы можем попросить gdb выполнить следующую строчку и снова остановиться. Иными словами, программу можно выполнить пошагово, строчка за строчкой. Таким образом, у вас будет достаточно времени для того, чтобы оглядеться и проверить значения переменных в памяти. На самом деле с помощью именно этого метода мы и будем исследовать стек и кучу.

Команда n (или next) используется для выполнения следующих строчек кода

Вывод содержимого массива с помощью отладчика gdb (print arr)

Исследование содержимого стека

```
(gdb) x/4b arr
0x7fffffffd9d4: 65      66      67      68
(gdb) x/8b arr
0x7fffffffd9d4: 65      66      67      68
0       107     -85     -99
(gdb) x/4b arr
0x7fffffffd9d4: 65      66      67      68
(gdb) set arr[1] = 'F'
(gdb) x/4b arr
0x7fffffffd9d4: 65      70      67      68
(gdb) print arr
$2 = "AFCD"
(gdb)
```

Первая команда, x/4b, выводит 4 байта, хранящиеся на том участке, на который указывает arr.
Вторая команда, x/8b, выводит 8 байт, идущих после arr.
В массиве arr находятся значения A, B, C и D. Вы должны знать, что массив хранит не сами символы, а их ASCII-коды. Например, ASCII-код A равен 65 в десятеричной системе или 0x41 в шестнадцатеричной. В случае с B это 66 или 0x42. Таким образом, gdb выводит те значения, которые мы сохранили в массив arr.

Но что насчет остальных 4 байт во второй команде? Они находятся в стеке и, вероятно, содержат данные последнего стекового фрейма, созданного во время вызова функции main.

Обратите внимание: по сравнению с другими сегментами стек заполняется в обратном порядке.

Другие области памяти заполняются, начиная с младшего адреса, но со стеком все не так.

Заполнение стека происходит от старших адресов к младшим. Отчасти это связано с историей развития современных компьютеров, а отчасти — с некоторыми возможностями сегмента стека (который ведет себя как одноименная структура данных).

Если вы хотите изменить значения внутри стека, то вам нужно использовать команду set. Это позволит модифицировать существующие ячейки памяти.

Исследование содержимого стека

```
(gdb) x/20x arr
0x7fffffffd9f4: 0x41    0x46    0x43    0x44    0x00    0xd0    0x76    0x42
0x7fffffffd9fc: 0x00    0x7a    0x16    0x1f    0x01    0x00    0x00    0x00
0x7fffffffda04: 0x00    0x00    0x00    0x00
(gdb) set *(0x7fffffffda01) = 0xff
(gdb) x/20x arr
0x7fffffffd9f4: 0x41    0x46    0x43    0x44    0x00    0xd0    0x76    0x42
0x7fffffffd9fc: 0x00    0x7a    0x16    0x1f    0x01    0xff    0x00    0x00
0x7fffffffda04: 0x00    0x00    0x00    0x00
(gdb) c
Continuing.
*** stack smashing detected ***: .../a.out terminated
Program received signal SIGABRT, Aborted.
0x00007ffff7a42428 in __GI_raise (sig=sig@entry=6) at ../sysdeps/Unix/sysv/
linux/raise.c:54
54 ../sysdeps/Unix/sysv/linux/raise.c: No such file or directory.
```

Вот и все. Мы только что записали значение 0xff по адресу 0x7fffffffda01, который находится вне массива arr. Вполне вероятно, что этот байт принадлежит стековому фрейму, сохраненному еще до входа в функцию main. Мы только что сломали стек! Изменение его адреса, который выделяли не вы, даже если речь идет об 1 байте, может быть очень опасным и обычно приводит к сбою или внезапному завершению. Если вы закончили отладку и хотите покинуть gdb, можете использовать команду **q** (или **quit**).

Исследование содержимого стека

```c
#include <stdio.h>
#include <string.h>
int main(int argc, char **argv)
{
    char str[10];
    strcpy(str, argv[1]);
    printf("Hello %s!\n", str);
}
```

Следует отметить, что запись непроверенных значений в буфер (байтовый или символьный массив), выделенный на вершине стека (а не в куче), считается уязвимостью. Злоумышленник может тщательно подготовить массив байтов и «скормить» его программе, чтобы получить контроль за ее выполнением. Обычно это называют эксплойтом в результате переполнения буфера.

Этот код не проверяет содержимое и размер ввода argv[1], копируя его прямо в массив arr, выделенный на вершине стека.

Если вам повезет, то вы отделаетесь лишь сбоем в работе. Но иногда это может привести к вредоносной атаке.

```
$ gcc main.c -o main.out
$ ./main.out
Segmentation fault
$
```

Рекомендации по использованию стековой памяти

```
int main(int argc, char** argv)
{
    int a;
    ...
    return 0;
}
```

Каждая переменная имеет свою область видимости, которая определяет ее время жизни. Это значит, что вместе с данной областью исчезают и все переменные, которые были в ней созданы.

Кроме того, только переменные, находящиеся в стеке, выделяются и освобождаются автоматически. Автоматическое управление памятью обусловлено природой стекового сегмента.

Любая переменная, которую вы объявляете в стеке, автоматически создается на его вершине. Выделение памяти происходит автоматически и может считаться началом жизни переменной. После этого поверх нее будет записано множество других переменных и стековых фреймов. Но пока она находится в стеке и над ней есть другие переменные, ее существование продолжается.

Однако рано или поздно программа должна завершиться, и перед ее выходом стек должен быть пустым. Поэтому в какой-то момент наша переменная будет извлечена из стека.

Тем не менее глобальное значение всегда остается в памяти, даже когда завершаются главная функция и сама программа, в то время как наша переменная будет извлечена из стека.

Рекомендации по использованию стековой памяти

```
int *get_integer() {
    int var = 10;
    return &var;
}
int main(int argc, char **argv) {
    int *ptr = get_integer();
    *ptr = 5;
    return 0;
}
```

Возвращение адреса локальной переменной внутри функции.
Функция get_integer возвращает адрес локальной переменной var, объявленной в ее области видимости. Затем функция main пытается разыменовать указатель и обратиться к соответствующей области памяти. Как видите, мы получили предупреждение.
Здесь видна ошибка сегментации, что можно интерпретировать как отказ программы. Обычно это происходит при доступе к некорректной области памяти, которая уже освободилась.
Если вы хотите, чтобы компилятор gcc считал ошибками любые предупреждения, то передайте ему параметр -Werror. Можно сделать и для отдельных предупреждений; например, в предыдущем случае достаточно указать параметр -Werror=return-local-addr.

```
$ gcc main.c -o main.out
main.c: In function ‘get_integer’:
main.c:4:12: warning: function returns address of local
variable [-Wreturn-local-addr]
    4 |     return &var;
      |            ^~~~
$ ./main.out
$
```

Рекомендации по использованию стековой памяти

```
$ gcc -g main.c -o a.out
main.c: In function ‘get_integer’:
...
(gdb) run
Starting program: /mnt/.../a.out
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Program received signal SIGSEGV, Segmentation fault.
0x000055555555551ab in main (argc=1, argv=0x7fffffffdb18) at main.c:9
9           *ptr = 5;
(gdb) q
$
```

Если запустить программу с помощью отладчика gdb, то можно узнать больше подробностей о ее сбое. ***Но помните:*** ее нужно скомпилировать с параметром -g, иначе от gdb будет мало пользы.
Вывод отладчика gdb говорит о том, что источник сбоя находится в строчке 9 внутри функции main — именно там, где программа пытается выполнить запись по возвращенному адресу путем разыменования полученного указателя. Переменная var была локальной для функции get_integer и перестала существовать просто потому, что в строчке 9.

Рекомендации по использованию стековой памяти

Ключевые характеристики стека:

- стековая память имеет ограниченный размер, поэтому не подходит для хранения крупных объектов;
- адреса в стеке увеличиваются в обратном порядке, поэтому, перемещаясь вперед по стеку, мы читаем уже сохраненные байты;
- управление памятью в стеке происходит автоматически. Это касается как выделения, так и освобождения места;
- каждая переменная в стеке обладает областью видимости, которая определяет ее время жизни. Это следует учитывать при проектировании логики программы. Вы не можете контролировать этот механизм;
- указатели должны ссылаться только на те стековые переменные, которые все еще находятся в области видимости;
- освобождение памяти стековых переменных происходит автоматически, прямо перед исчезновением области видимости, и вы не можете на это повлиять;
- указатели на переменные, существующие в текущей области видимости, можно передавать в другие функции в качестве аргументов, но только если вы уверены в том, что в момент использования этих указателей область видимости будет оставаться на месте. Данное правило может не распространяться на программы с параллельной логикой.

Куча

Почти любой код, написанный на любом языке программирования, так или иначе использует кучу. Это вызвано тем, что она имеет уникальные преимущества. Но у этого сегмента памяти есть и недостатки. Например, выделение места в нем происходит медленней, чем в стеке.

1. В куче никакие блоки памяти не выделяются автоматически. Вместо этого для выделения каждого участка памяти программист должен использовать malloc или аналогичную функцию.

2. Куча большая. В то время как стек имеет ограниченный размер и не подходит для размещения крупных объектов, куча позволяет хранить огромные объемы данных, достигающие десятков гигабайтов.

3. Выделением и освобождением памяти в куче занимается программист. Это значит, на программиста ложится вся ответственность за выделение памяти и последующее ее освобождение, когда она больше не нужна.

4. Переменные, выделенные в куче, не имеют никакой области видимости, в отличие от стековых переменных. Такое свойство можно считать отрицательным, поскольку оно существенно усложняет управление памятью.

5. К блокам кучи можно обращаться только с помощью указателей. Иными словами, нет такого понятия, как «переменная кучи». Для навигации по куче используются указатели.

6. Поскольку куча доступна только владеющему ей процессу, для ее исследования нужен отладчик. К счастью, в языке С указатели работают одинаково как со стеком, так и с блоками кучи. Данная абстракция работает очень хорошо, и благодаря ей мы можем использовать одни и те же указатели для обращения к этим двум сегментам памяти.

Выделение и освобождение памяти

```
#include <stdio.h>  // для функции printf
#include <stdlib.h> // для функций по работе с кучей
void print_mem_maps()
{
#ifdef __linux__
    FILE *fd = fopen("/proc/self/maps", "r");
    if (!fd)
    {
        printf("Could not open maps file.\n");
        exit(1);
    }
    char line[1024];
    while (!feof(fd))
    {
        fgets(line, 1024, fd);
        printf("> %s", line);
    }
    fclose(fd);
#endif
}
```

В куче необходимо выделять и освобождать вручную. Для этого программист должен использовать набор функций или API (функции для работы с памятью из стандартной библиотеки C).

Эти функции существуют, и их определения находятся в заголовке stdlib.h. Для получения блока памяти в куче используются функции malloc, calloc и realloc, а для освобождения памяти предусмотрена только одна функция: free.. Выделение динамической памяти — то же самое, что выделение кучи.

Приведенный выше код является кросс-платформенным и может быть скомпилирован в большинстве операционных систем семейства Unix. Но функция ***print_mem_maps*** работает только в Linux, поскольку макрос __linux__ определен лишь в этой ОС.

Выделение и освобождение памяти

```c
int main(int argc, char **argv) {
    // выделяем 10 байт без инициализации
    char *ptr1 = (char *)malloc(10 * sizeof(char));
    printf("Address of ptr1: %p\n", (void *)&ptr1);
    printf("Memory allocated by malloc at %p: ", (void *)ptr1);
    for (int i = 0; i < 10; i++)
        printf("0x%02x ", (unsigned char)ptr1[i]);
    printf("\n");
    // выделяем 10 байт, каждый из которых обнулен
    char *ptr2 = (char *)calloc(10, sizeof(char));
    printf("Address of ptr2: %p\n", (void *)&ptr2);
    printf("Memory allocated by calloc at %p: ", (void *)ptr2);
    for (int i = 0; i < 10; i++)
        printf("0x%02x ", (unsigned char)ptr2[i]);
    printf("\n");
    print_mem_maps();
    free(ptr1);
    free(ptr2);
    return 0;
}
```

Программа выводит адреса указателей ptr1 и ptr2. Указатели выделены в стеке, но ссылаются на какой-то другой сегмент памяти — в данном случае на кучу. Использование стекового указателя для работы с блоком кучи — распространенная практика.

Имейте в виду: указатели ptr1 и ptr2 находятся в одной области видимости и освобождаются при завершении функции main, однако блоки памяти, полученные в куче, не входят ни в какую область видимости. Они будут существовать, пока их не освободят вручную.

Выделение и освобождение памяти

```
$ gcc main.c -o main.out
$ ./main.out
Address of ptr1: 0x7ffc13e22f18
Memory allocated by malloc at 0x5597131712a0: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
Address of ptr2: 0x7ffc13e22f20
Memory allocated by calloc at 0x5597131716d0: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
> 559711191000-559711192000 r--p 00000000 00:4e 27021597764844883      /mnt/…/main.out
> ...
> 559711195000-559711196000 rw-p 00003000 00:4e 27021597764844883      /mnt/…/main.out
> 559713171000-559713192000 rw-p 00000000 00:00 0                       [heap]
> 7fc09b245000-7fc09b248000 rw-p 00000000 00:00 0
> 7fc09b248000-7fc09b270000 r--p 00000000 08:20 22554      /usr/lib/x86_64-linux-gnu/libc.so.6
...
> 7fc09b47e000-7fc09b4a8000 r-xp 00002000 08:20 22551      /usr/lib/ld-linux-x86-64.so.2
o.2
> 7fc09b4b6000-7fc09b4b8000 rw-p 00039000 08:20 22551      /usr/lib/ld-linux-x86-64.so.2
> 7ffc13e04000-7ffc13e26000 rw-p 00000000 00:00 0                   [stack]
> 7ffc13f3a000-7ffc13f3e000 r--p 00000000 00:00 0                   [vvar]
> 7ffc13f3e000-7ffc13f40000 r-xp 00000000 00:00 0                   [vdso]
> 7ffc13f3e000-7ffc13f40000 r-xp 00000000 00:00 0                   [vdso]
$
```

Выделение и освобождение памяти

Если вернуться к функциям выделения памяти в куче, то calloc расшифровывается как clear and allocate («очистить и выделить»), а malloc — memory allocation («выделение памяти»). То есть calloc очищает выделенный блок памяти, а malloc оставляет его инициализацию самой программе.

Если присмотреться, то можно заметить, что в блоке памяти, который выделила функция malloc, находятся ненулевые байты, а блок, выделенный функцией calloc, состоит из одних нулей. Что же делать? Можно ли исходить из того, что malloc в Linux всегда выделяет обнуленные блоки?

Если вы собираетесь писать кросс-платформенную программу, то вам следует руководствоваться спецификацией языка C. В спецификации сказано: функция malloc не инициализирует выделяемый блок памяти.

Но даже если вы пишете свою программу только для Linux и вас не интересуют другие операционные системы, то имейте в виду: в будущем компиляторы могут изменить свое поведение.

Обратите внимание: благодаря этому malloc обычно работает быстрее, чем calloc. На самом деле некоторые реализации malloc откладывают выделение блока памяти до тех пор, пока к нему кто-то не обратится (для чтения или записи). Таким образом, ускоряется выделение памяти.

```
> gcc main.c -o a.exe
> ./a.exe
Address of ptr1: 00000017c8dff6c0
Memory allocated by malloc at 0000011bc3102470: 0x30 0x66 0x10 0xc3 0x1b 0x01 0x00 0x00 0x50 0x01
Address of ptr2: 00000017c8dff6b8
Memory allocated by calloc at 0000011bc3102490: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
>
```

Выделение и освобождение памяти

```c
#include <stdlib.h> // для malloc
#include <string.h> // для memset
int main(int argc, char **argv) {
    char *ptr = (char *)malloc(16 * sizeof(char));
    memset(ptr, 0, 16 * sizeof(char));    // заполняем нулями
    memset(ptr, 0xff, 16 * sizeof(char)); // заполняем байтами 0xff
    ...
    free(ptr);
    return 0;
}
```

Использование функции memset для инициализации блока памяти

```c
int main(int argc, char** argv) {
    char* ptr = (char*)malloc(16 * sizeof(char));
    ...
    ptr = (char*)realloc(32 * sizeof(char));
    ...
    free(ptr);
    return 0;
}
```

Функция realloc не изменяет содержимое выделенной памяти, а просто расширяет старый блок. Если это не удается сделать из-за фрагментации, то она находит другой блок подходящего размера и копирует в него данные из старого блока. В этом случае последний освобождается.

Выделение и освобождение памяти

```c
#include <stdlib.h> // для функций по работе с кучей
int main(int argc, char **argv) {
    char *ptr = (char *)malloc(16 * sizeof(char));
    return 0;
}
```

free освобождает уже выделенный блок кучи, принимая его адрес в виде указателя.
У данной программы есть утечка памяти, поскольку после ее завершения в куче остается неосвобожденный блок размером 16 байт.

```
$ gcc -g main.c -o a.out
$ valgrind ./a.out
...
==654== Command: ./a.out
==654== HEAP SUMMARY:
==654==     in use at exit: 16 bytes in 1 blocks
==654==   total heap usage: 1 allocs, 0 frees, 16 bytes allocated
==654== LEAK SUMMARY:
==654==    definitely lost: 16 bytes in 1 blocks
==654==    indirectly lost: 0 bytes in 0 blocks
==654==      possibly lost: 0 bytes in 0 blocks
==654==    still reachable: 0 bytes in 0 blocks
==654==         suppressed: 0 bytes in 0 blocks
==654== Rerun with --leak-check=full to see details of leaked memory
==654== For lists of detected and suppressed errors, rerun with: -s
==654== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

В разделе HEAP SUMMARY указано, что наша программа выделила один блок, а освободила 0; в результате в момент выхода выделенными оставались 16 байт. Если опуститься чуть ниже к разделу LEAK SUMMARY, то можно увидеть, что эти 16 байт безвозвратно потеряны, то есть произошла утечка памяти!

Выделение и освобождение памяти

Как видите, мы передали утилите valgrind параметр --leak-check=full, и теперь она показывает строчку кода, в которой был выделен потерянный блок кучи. Это строчка 4 — то есть вызов malloc. Таким образом, вы можете проследить за этим блоком и найти место, где его следует освободить.

```
$ valgrind --leak-check=full ./a.out
...
==2116== Command: ./a.out
==2116==
==2116==
==2116== HEAP SUMMARY:
==2116==     in use at exit: 16 bytes in 1 blocks
==2116==   total heap usage: 1 allocs, 0 frees, 16 bytes allocated
==2116==
==2116== 16 bytes in 1 blocks are definitely lost in loss record 1 of 1
==2116==    at 0x4848899: malloc (in
/usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==2116==    by 0x109165: main (main.c:4)
...
```

Выделение и освобождение памяти

```c
#include <stdlib.h> // для функций по работе с кучей
int main(int argc, char **argv) {
    char *ptr = (char *)malloc(16 * sizeof(char));
    free(ptr);
    return 0;
}
```

Обратите внимание на сообщение All Heap blocks were freed (Все блоки кучи были освобождены). Оно фактически означает, что в программе не осталось утечек памяти.

```
$ gcc -g main.c -o a.out
$ valgrind --leak-check=full ./a.out
==2850== Memcheck, a memory error detector
==2850== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==2850== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright
info
==2850== Command: ./a.out
==2850== HEAP SUMMARY:
==2850==     in use at exit: 0 bytes in 0 blocks
==2850==   total heap usage: 1 allocs, 1 frees, 16 bytes allocated
==2850== All heap blocks were freed -- no leaks are possible
==2850== For lists of detected and suppressed errors, rerun with: -s
==2850== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Выделение и освобождение памяти

Программа, запущенная с помощью упомянутого профилировщика, может существенно замедлить свою работу (в 10–50 раз), но благодаря этому вы сможете легко выявить проблемы с памятью. Запускать код внутри valgrind, чтобы обнаружить утечки памяти на максимально раннем этапе, — рекомендованный подход.

Помимо valgrind, существуют другие профилировщики памяти. Среди самых известных можно выделить LLVM ASAN (Address Sanitizer) и MemProf. Профилировщики могут анализировать использование памяти и операции выделения с помощью различных методов. Часть из них перечислены ниже:

- Некоторые профилировщики могут служить изолированной средой для запуска программ и мониторинга их операций с памятью.
- Еще один метод заключается в использовании библиотек, которые предоставляют некоторые профилировщики в качестве оберток для системных вызовов, относящихся к управлению памятью. Таким образом, итоговый двоичный файл будет содержать всю логику, необходимую для профилирования. Этот метод требует повторной компиляции и даже некоторой модификации исходного кода.
- Программы также могут предварительно загружать разные библиотеки с функциями, которые меняют поведение операций по выделению памяти из стандартной библиотеки C. Таким образом, вам не обязательно компилировать свой исходный код. Вы можете просто указать библиотеки таких профайлеров в переменной среды LD_PRELOAD, и они будут загружены вместо стандартных библиотек libc.

Подстановочная функция — это обертка вокруг стандартной функции. Она находится в динамической библиотеке, загружается перед оригиналом и перенаправляет ему поступающие вызовы.

Принцип работы кучи

Куча имеет несколько отличий от стека, поэтому при работе с ней нужно руководствоваться отдельными правилами.

Каждый блок памяти (или переменная) в стеке имеет область видимости, поэтому определить его время жизни не составляет труда. Каждый раз, когда мы покидаем область видимости, все ее переменные исчезают. Но с кучей все намного сложнее.

У блока кучи нет никакой области видимости, и потому его время жизни является неочевидным и должно быть определено вручную. Именно поэтому в современных языках программирования, таких как Java, применяется автоматическое освобождение или сборка мусора с поддержкой поколений объектов. Сама программа и библиотеки, которые она использует, не в состоянии определить время жизни кучи, вследствие чего вся ответственность за это ложится на программиста.

Когда дело доходит до принятия конкретных решений, особенно в этом случае, сложно предложить какой-то универсальный ответ. Любое мнение является предметом дискуссии и может вести к компромиссу.

Одна из лучших стратегий по преодолению сложностей, связанных с временем жизни кучи, состоит в определении владельца блока памяти, а не области видимости, которая вмещает в себя данный блок. Конечно, это нельзя назвать полноценным решением.

Владелец полностью ответственен за управление жизненным циклом блока кучи; он его изначально выделяет и затем, когда нужда в нем отпадает, освобождает.

Принцип работы кучи 1/2

```
#define QUEUE_MAX_SIZE 100
typedef struct {
    int front;
    int rear;
    double *arr;
} queue_t;
void init(queue_t *q) {
    q->front = q->rear = 0;
    // выделенными здесь блоками кучи владеет объект очереди
    q->arr = (double *)malloc(QUEUE_MAX_SIZE * sizeof(double));
}
void destroy(queue_t *q) {
    free(q->arr);
}
int size(queue_t *q) {
    return q->rear - q->front;
}
void enqueue(queue_t *q, double item) {
    q->arr[q->rear] = item;
    q->rear++;
}
double dequeue(queue_t *q) {
    double item = q->arr[q->front];
    q->front++;
    return item;
}
```

Принцип работы кучи 2/2

```c
int main(int argc, char **argv) {
    // выделенными здесь блоками кучи владеет функция main
    queue_t *q = (queue_t *)malloc(sizeof(queue_t));
    // выделяем необходимую память для объекта очереди
    init(q);
    enqueue(q, 6.5);
    enqueue(q, 1.3);
    enqueue(q, 2.4);
    printf("%f\n", dequeue(q));
    printf("%f\n", dequeue(q));
    printf("%f\n", dequeue(q));
    // освобождаем ресурсы, полученные объектом очереди
    destroy(q);
    // освобождаем память, выделенную для объекта кучи, принадлежащего функции main
    free(q);
    return 0;
}
```

Помимо стратегии владения, можно также использовать сборщик мусора — автоматический механизм, встроенный в программу, пытающийся освобождать блоки памяти, на которые не ссылается ни один указатель. В языке C одним из традиционных и широко известных инструментов этого типа является консервативный сборщик мусора Бема — Демерса — Вайзера; он предоставляет набор функций для выделения памяти, которые нужно вызывать вместо malloc и других стандартных операций языка C.

Принцип работы кучи

Еще один подход к управлению временем жизни блоков кучи — использование объекта RAII (resource acquisition is initialization — «получение ресурса есть инициализация»). Это идиома объектно-ориентированного программирования, согласно которой время жизни ресурса (такого как выделенный блок кучи) можно привязать к времени жизни объекта. Иными словами, мы задействуем объект, который во время своего создания инициализирует ресурс, а во время своего уничтожения — освобождает его. К сожалению, данный метод нельзя реализовать в C, поскольку в этом языке программиста не уведомляют об уничтожении объектов.

Факты и рекомендации, которые нужно учитывать при работе с кучей:

- Выделение памяти в куче отнимает определенные ресурсы. Разные функции выделения памяти имеют разные накладные расходы. Самой «дешевой» является функция malloc.
- Все блоки памяти, выделенные в пространстве кучи, должны быть освобождены либо сразу после того, как в них пропадает необходимость, либо перед завершением программы.
- Поскольку у блоков кучи нет области видимости, во избежание потенциальных утечек программа должна уметь управлять памятью.
- Как показывает практика, при выделении каждого блока кучи следует придерживаться выбранной стратегии управления памятью.
- Выбранную стратегию и предположения, на которых она основана, следует документировать на любом участке кода, где происходит доступ к блоку, чтобы в будущем программисты об этом знали.
- В определенных языках программирования, таких как C++, управлять ресурсами (такими как, к примеру, блоки кучи) можно с помощью объектов RAII.

Управление памятью в средах с ограниченными ресурсами

Существуют среды, в которых память является ценным и зачастую ограниченным ресурсом. В ряде систем ключевым фактором выступает производительность, и программы, которые в них выполняются, должны работать быстро, вне зависимости от количества доступной памяти. Каждая среда требует применения определенных методик, позволяющих избежать нехватки памяти и снижения производительности.

Ограниченность ресурсов еще не говорит о малом объеме памяти. Ограничения обычно касаются того, как программа может использовать данную память. Это могут быть жесткие лимиты, установленные вашим клиентом, определенная аппаратная конфигурация или отсутствие поддержки больших объемов памяти в вашей операционной системе (как, например, в MS-DOS).

Но даже при отсутствии подобных ограничений программист должен пытаться использовать как можно меньше памяти и делать это наиболее оптимальным образом. ***Потребление памяти*** — одно из ключевых нефункциональных требований к проекту, и потому его нужно тщательно отслеживать и оптимизировать.

Среды с ограниченной памятью

В таких средах ограничением является память, и ваши алгоритмы должны уметь справляться с ее нехваткой. К данной категории обычно относятся встраиваемые системы с памятью размером десятки или сотни мегабайт. Существует несколько приемов по управлению памятью в подобных средах, однако ни один из них не сравнится по своей эффективности с хорошо оптимизированным алгоритмом. В этом случае обычно используются алгоритмы с низким потреблением памяти, которое обычно выливается в увеличение времени работы.

Каждый алгоритм обладает временной сложностью и сложностью памяти. Первая описывает отношение между размером ввода и временем, необходимым для завершения алгоритма. Вторая описывает отношение между размером ввода и объемом памяти, который нужен алгоритму для выполнения работы.

В идеале алгоритм должен иметь низкую временную сложность и низкую сложность памяти. Иными словами, быстрая работа и умеренное потребление памяти крайне желательны, но в реальности подобное сочетание встречается редко. Точно так же мы не ожидаем низкой производительности от алгоритма, которому нужно много памяти.

В большинстве случаев мы имеем дело с неким балансом между памятью и скоростью (то есть временем работы). Например, если один алгоритм сортировки работает быстрее другого, то ему обычно нужно больше памяти, хотя оба делают одно и то же.

Упакованные структуры

Один из самых простых способов уменьшить потребление памяти — использовать упакованные структуры. Они игнорируют выравнивание в памяти и применяют более компактную схему размещения своих полей.
На самом деле это компромиссное решение. Ввиду отказа от выравнивания чтение структуры занимает больше времени, что замедляет программу.

Сжатие

Это эффективная методика, особенно если программа работает с большим количеством текстовых данных, которые нужно хранить в памяти. Текстовые данные, по сравнению с двоичными, имеют высокую степень сжатия. Таким образом, программа может хранить текст в сжатом виде и экономить много памяти.

Но за все приходится платить: алгоритмы сжатия требуют интенсивных вычислений и сильно нагружают процессор, поэтому конечная производительность ухудшается. Данный метод идеально подходит для программ, которые хранят много текстовых данных, но нечасто их используют; в противном случае понадобится много операций сжатия/разжатия, что в конечном счете затруднит или сделает невозможным применение программы.

Внешнее хранилище данных

Использование внешнего хранилища данных в виде сетевого сервиса, облачной инфраструктуры или обычного жесткого диска — очень распространенный и действенный способ борьбы с нехваткой памяти. Обычно предполагается, что программа может выполняться в среде с ограниченными ресурсами, поэтому существует много программ, которые используют данный подход, даже когда памяти в достатке.

Использование этой методики обычно предполагает, что оперативная память играет роль кэша. Еще одно предположение — все данные в память не поместятся и загружать их нужно по частям или постранично.

Эти алгоритмы не решают проблемы с нехваткой памяти как таковые, но пытаются решить проблемы с медленным внешним хранилищем данных. Внешние хранилища всегда очень медленные по сравнению с оперативной памятью. Поэтому алгоритмы должны балансировать между чтением из внутренней памяти и из внешнего хранилища. Данный подход используют все базы данных, включая PostgreSQL и Oracle.

Высокопроизводительные среды

Быстрым алгоритмам свойственно повышенное потребление памяти. Поговорим об этом более подробно.
Наглядной иллюстрацией этого утверждения может быть использование кэша для увеличения производительности. Кэширование данных означает повышенный расход памяти, но взамен мы получаем повышенную скорость, если кэш используется должным образом
Но расширение памяти — не всегда оптимальный путь увеличения производительности. Существуют другие методы, которые имеют прямое или опосредованное отношение к памяти и могут существенно повлиять на скорость работы алгоритма.

Дружественный к кэшированию код

При выполнении инструкции процессору сначала нужно получить все необходимые данные. Они находятся в оперативной памяти по определенному адресу, который указан в инструкции. Перед вычислением данные должны быть скопированы в регистры процессора. Однако он обычно копирует больше блоков, чем ожидалось, и помещает их в свой кэш.

В следующий раз, если ему понадобится значение, находящееся недалеко от предыдущего адреса, он сможет найти его в кэше и избежать обращения к оперативной памяти, что намного повысит скорость чтения.

Если значение не удается найти, то это промах кэша, в результате которого процессору придется считывать и копировать нужный адрес из оперативной памяти, что довольно медленно. В целом, чем выше частота попаданий, тем быстрее работает код.

Зачем процессор копирует соседние значения, находящиеся поблизости от искомого адреса? Это связано с принципом локальности. В компьютерных системах доступ к данным, находящимся в одном районе, обычно происходит чаще. Процессор следует данному принципу и загружает дополнительные данные из той же местности. Если алгоритм умеет извлекать пользу из такого поведения, то будет выполняться быстрее. Поэтому такие алгоритмы называют дружественными к кэшированию.

Дружественный к кэшированию код

Программа 23_7 вычисляет и выводит сумму всех элементов матрицы, однако имеет еще одну особенность. Пользователь может изменять ее поведение путем передачи параметров.

```
$ ./main.out print 2 3
Matrix:
1 1 1
2 2 2
Flat matrix: 1 1 1 2 2 2
$ time ./main.out friendly-sum 20000 20000
Friendly sum: 1585447424
real    0m2.986s
user    0m2.251s
sys     0m0.730s
$ time ./main.out not-friendly-sum 20000
20000
Not friendly sum: 1585447424
real    0m12.173s
user    0m11.537s
sys     0m0.630s
$
```

Данный вывод состоит из двух разных представлений матрицы: двумерного и плоского. Как видите, элементы матрицы развертываются в памяти по строкам. Это значит, они сохраняются строка за строкой. Поэтому если процессор копирует из строки один элемент, то все остальные элементы в данной строке тоже, скорее всего, копируются в кэш. И, как следствие, сложение лучше выполнять по строкам, а не по столбцам.

Если еще раз взглянуть на код, то можно заметить, что в функции friendly_sum сложение происходит построчно, а в friendly_sum — по столбцам.

Разница во времени составила около 10 секунд!

Накладные расходы на выделение и освобождение памяти

В отличие от стека, в котором выделение памяти происходит относительно быстро и не требует получения дополнительного диапазона адресов, куча должна сначала найти свободный блок памяти достаточного размера, и это может быть затратной операцией.

Для выделения и освобождения существует множество алгоритмов, и между этими операциями всегда приходится находить разумный баланс. Если вы хотите быстро выделять память, то для этого необходимо потреблять больше блоков. И наоборот, если вы хотите использовать меньше памяти, то ваш алгоритм выделения будет работать медленней.

Помимо функций malloc и free, которые входят в стандартную библиотеку, в языке C можно использовать другие аллокаторы. В их число входят ptmalloc, tcmalloc, Haord и dlmalloc.

Как же решить эту скрытую проблему? Очень просто: реже выделять и освобождать память. В ряде программ, которым необходимо часто выделять место в куче, такое решение может показаться недостижимым. В подобных случаях в куче обычно выделяют один большой блок и пытаются работать с ним самостоятельно. В результате получается еще один слой логики для выделения/освобождения памяти (может быть, не такой сложный, как функции malloc и free), управляющий этим большим блоком.

Пулы памяти

Как вы уже знаете, выделение и освобождение памяти — затратные операции. Чтобы сократить их количество и повысить производительность, можно использовать пул заранее выделенных блоков кучи фиксированного размера. Обычно каждый блок имеет идентификатор, который можно получить с помощью API, предназначенного для управления пулом. Позже, когда блок больше не нужен, его можно освободить. Поскольку объем выделенной памяти не меняется, это отличное решение для алгоритмов, стремящихся к предсказуемому поведению в средах с ограниченными ресурсами.

Введение

Структура памяти процесса	Стек и куча	Интегральные вычисления	Форматы хранения данных
30 минут	30 минут	15 минут	15 минут

Введение

Зачем это нужно?

Математическое моделирование: физика (траектории частиц), инженерия (расчет материалов), компьютерная графика (рендеринг кривых).

Реальные примеры:

- Длина кривой: измерение пути спутника по орбите.
- Площадь: вычисление объема жидкости под кривой поверхности.
- Поверхность площади вращения
- Объем тел вращения
- Масса криволинейных объектов

Практическая ценность: автоматизация расчетов, где аналитическое решение невозможно или слишком сложно.

Почему C?

- Высокая производительность для численных расчетов.
- Прямой контроль над памятью и вычислениями.
- Широкое использование в научных и инженерных задачах.

Основные задачи

Что будем вычислять?

1. Длина кривой:

- Формула: $L = \int_a^b \sqrt{1 + (f'(x))^2}dx$
- Геометрический смысл: сумма длин маленьких отрезков вдоль кривой.
- Пример: y= x^2 на ([0, 1]), аналитически L≈1.147

2. Площадь под кривой:

- Формула: $A = \int_a^b f(x)dx$
- Применение: объем под графиком, работа силы вдоль пути.
- Пример: y = x^2 на ([0, 1]), A = 1/3

3. Площадь поверхности вращения (дополнительно):

- Формула: $S = \int_a^b f(x)\sqrt{1 + (f'(x))^2}dx$ (вращение вокруг оси X).
- Пример: вращение y = sqrt{x} вокруг оси X.

Общий подход:

- Аналитическое решение часто недоступно (сложные функции, нет явной формулы).
- Численное интегрирование: разбиение интервала на малые участки. Ошибка уменьшается с увеличением числа участков ((n)).

Метод прямоугольников

Описание:

Разбиваем интервал ([a, b]) на (n) равных частей.

Площадь: $A \approx \sum_{i=0}^{n-1} f(x_i)\Delta x$, где Δx=(b−a)/n

Для длины кривой:

Приближаем кривую отрезками, длина: $L \approx \sum_{i=0}^{n-1} \sqrt{\Delta x^2 + \left(f(x_{i+1}) - f(x_i)\right)^2}$

Плюсы: Простота реализации.

Минусы: Низкая точность при малом (n).

```c
double rectangle_area(double a, double b, int n, double (*f)(double)) {
    double dx = (b - a) / n;
    double sum = 0;
    for (int i = 0; i < n; i++)
        sum += f(a + i * dx);
    return sum * dx;
}
```

Метод трапеций

Описание:

Площадь: $A \approx \frac{\Delta x}{2}(f(a) + 2\sum_{i=1}^{n-1} f(x_i) + f(b))$

Плюсы: Более точный, чем прямоугольники.

Минусы: Ошибка $O(\Delta x^2)$, требует вычисления разностей.

```c
double trapezoid_curve_length(double a, double b, int n, double (*f)(double)) {
    double dx = (b - a) / n;
    double sum = 0;
    for (int i = 0; i < n; i++) {
        double x0 = a + i * dx;
        double x1 = a + (i + 1) * dx;
        double dy = f(x1) - f(x0);
        sum += sqrt(dx * dx + dy * dy);
    }
    return sum;
}
```

Метод Симпсона

Описание для площади:

Площадь: $A \approx \frac{\Delta x}{3}(f(a) + 4\sum_{i=1}^{n/2} f(x_{2i-1}) + 2\sum_{i=1}^{n/2-1} f(x_{2i}) + f(b))$.

(n) — четное, Δx=(b−a)/n

Для длины кривой:

Формула: $L = \int_a^b \sqrt{1 + (f'(x))^2}dx$

Определим: $g(x) = \sqrt{1 + (f'(x))^2}$

Численное приближение методом Симпсона:

$L = \frac{\Delta x}{3}(g(a) + 4\sum_{i=1}^{n/2} g(x_{2i-1}) + 2\sum_{i=1}^{n/2-1} g(x_{2i}) + g(b))$.

Производная (f'(x)) аппроксимируется как $f'(x_i) \approx \frac{f(x_i+h)-f(x_i-h)}{2h}$ (центральная разность), где (h) — малый шаг.

Плюсы: Очень высокая точность (ошибка $O(\Delta x^4)$).

Минусы: Требует вычисления (f'(x)), сложнее реализация.

Метод Симпсона

```c
// Функция для численной производной
double derivative(double x, double (*f)(double), double h) {
    return (f(x + h) - f(x - h)) / (2 * h);
}
// Функция g(x) = sqrt(1 + (f'(x))^2)
double g(double x, double (*f)(double), double h) {
    double df = derivative(x, f, h);
    return sqrt(1 + df * df);
}
// Метод Симпсона для длины кривой
double simpson_curve_length(double a, double b, int n, double (*f)(double)) {
    if (n % 2 != 0) n++; // n должно быть четным
    double dx = (b - a) / n;
    double h = dx / 2.0; // шаг для производной
    double sum = g(a, f, h) + g(b, f, h); // края
    for (int i = 1; i < n; i++) {
        double x = a + i * dx;
        sum += (i % 2 == 0 ? 2 : 4) * g(x, f, h);
    }
    return sum * dx / 3;
}
```

Введение

Структура памяти процесса	Стек и куча	Интегральные вычисления	Форматы хранения данных
30 минут	30 минут	15 минут	15 минут

Введение

Что такое формат хранения данных?

- Способ организации и представления данных в памяти компьютера или на носителях.
- Влияет на скорость доступа, объем памяти и совместимость.

Зачем они нужны?

- Эффективная работа алгоритмов (например, сортировка, поиск).
- Оптимизация программ на языках вроде Си (ручное управление памятью).

Основные виды форматов

1. Текстовые форматы

- Данные хранятся в читаемом виде (ASCII, UTF-8).
- Примеры: TXT, CSV, INI.
- Плюсы: Простота, читаемость человеком.
- Минусы: Больший объем, медленный парсинг.

2. Бинарные форматы

- Данные в машинном виде (байты, биты).
- Примеры: BIN, DAT, исполняемые файлы (.exe).
- Плюсы: Компактность, быстрая обработка.
- Минусы: Нечитаемы без специальных инструментов.

3. Структурированные форматы

- Иерархическая организация (ключ-значение, деревья).
- Примеры: JSON, XML, YAML.
- Плюсы: Гибкость, поддержка сложных структур.
- Минусы: Overhead (дополнительные метаданные).

Откуда взялись форматы?

Исторический контекст:

- 1950-е: Появление первых компьютеров → необходимость стандартов (ASCII, 1963).
- 1970-е: Рост вычислительных задач → бинарные форматы для скорости.
- 1990-е: Интернет и обмен данными → JSON (2000-е), XML (1998).

Эволюция:

- От простых (битовые строки) к сложным (базы данных, облачные форматы).
- Влияние языков программирования: Си → работа с сырыми данными, что дало толчок бинарным форматам.

Пример из Си:

Структуры (struct) — основа для создания собственных форматов.

Откуда взялись форматы?

Исторический контекст:

- 1950-е: Появление первых компьютеров → необходимость стандартов (ASCII, 1963).
- 1970-е: Рост вычислительных задач → бинарные форматы для скорости.
- 1990-е: Интернет и обмен данными → JSON (2000-е), XML (1998).

Эволюция:

- От простых (битовые строки) к сложным (базы данных, облачные форматы).
- Влияние языков программирования: Си → работа с сырыми данными, что дало толчок бинарным форматам.

Пример из Си:

Структуры (struct) — основа для создания собственных форматов.

Структурированные форматы

Что это? Данные организованы в иерархическую или логическую структуру (ключ-значение, деревья, таблицы). Используются для представления сложных объектов и связей. Основные примеры:

JSON (JavaScript Object Notation):

- Легкий, текстовый, популярен в вебе.
- Плюсы: Читаемость, поддержка массивов и вложенности.
- Минусы: Больше overhead, чем у бинарных форматов.

XML (eXtensible Markup Language):

- Теговая структура, гибкость через схемы (XSD).
- Плюсы: Поддержка метаданных, стандартизация.
- Минусы: Громоздкость, медленный парсинг.

YAML (YAML Ain’t Markup Language):

- Читаемый человеком, минималистичный.
- Плюсы: Удобство для конфигураций.
- Минусы: Чувствительность к отступам.

Protobuf (Protocol Buffers):

- Бинарный, разработан Google.
- Плюсы: Высокая скорость, компактность.
- Минусы: Требует схемы, не читаем напрямую.

SQL

```
{
  "student": {
    "id": 12345,
    "name": "Alexey Petrov",
    "grades": [85, 90, 78],
    "active": true
  }
}
```