Advanced C++

#1 Вводное занятие.
Память, компиляция,
ассемблер.
C и C++.

Intro

Махмуд
Sberbank, DSE
MIPT, APTD

Tg: @mipt_user

Константин
1C, SWE
MIPT, APTD

Tg: @konstantinleladze

Rating

5 мини-проектов
10 маленьких домашних заданий

Competitive-система: рейтинг.

Каждый проект в 10 баллов. Каждая домашка в 5 баллов. Суммарно -- 100 баллов.
Сдача проектов в 2 этапа: **local-тесты + review**

Все коды заливаются в гитхаб-репозиторий.
В **assignee** указываем ассистентов. А также продублировать в портал MADE.

Branch в формате **[Surname]_[Name]_[Group№]**

Menu

- C vs C++, C++ standards
- Recap
- About C: IO-потоки, scanf, printf, fflush, format
- Memory models: адресное пространство, stack vs heap
- Malloc, free, allocators

- Compilation
- Assembler
- Кодогенерация инструкций в asm-код
- Вопросы/пожелания

Recap.
C vs C++

C vs C++

C++ вытек из C. (Стандарт ANSI C)

Exceptions, Classes, Move-семантика.

```
int f();
Int f(void);

goto

main()

etc...
```

C++11

auto types:

```
map<int, pair<string, double>>::reverse_iterator it;
```

Instead:

```
auto it = a.begin();
```

C++11

Range-based cycles (like in python):

```
vector<int> a;
for (int elem: a)
    cout << elem << endl;
```

То же самое по ссылке:

```
vector<int> a;
for (auto & elem: a)
    ++elem;
```

C++11

Универсальная инициализация:

В языке C: `int a[3] = {1, 2, 3};`

В C++11:

```
struct point {
    int x, y;
};
point P;
P = {1, 2};

Либо: point P{1, 2};
```

C++11

Структура tie:

Пример. Пусть функция f возвращает пару значений, то есть структуру pair или tuple. Хочется записать эти значения в две переменные. Раньше мы писали так:

```
auto res = f();
a = res.first;
b = res.second;
```

С использованием tie это можно сделать так:

```
tie(a, b) = f();
```

Quick Recap

Инструкции C++:

```
; // пустая инструкция

if ( ival0 > ival1 ) {
    // составная инструкция, состоящая
    // из объявления и двух присваиваний
    int temp = ival0;
  ival0 = ival1;
  ival1 = temp;
}
```

Quick Recap

Инструкции C++: Switch

```
switch ( ch ) {
   case 'a':
       ++aCnt;
       break;
   case 'e':
       ++eCnt;
       break;
   case 'i':
       ++iCnt;
       break;
}
```

Quick Recap

Инструкции C++: Goto

Инструкция goto обеспечивает *безусловный* переход к другой инструкции внутри той же функции, поэтому современная практика программирования выступает против ее применения.
Синтаксис goto следующий:

```
goto метка;
```

Quick Recap

Управляющие конструкции:

```
if (условие)
    Действие;

if (условие)
    действие1;
else
    действие2;
```

Quick Recap

Выражения:

Выражение состоит из одного или более операндов, в простейшем случае – из одного литерала или объекта. Результатом такого выражения является r-значение его операнда. Например:

```
void mumble() {
   3.14159;
   "melancholia";
   upperBound;
}
```

C basics

IO streams

Функция scanf() является процедурой ввода общего назначения, считывающей данные из потока stdin. Она может считывать данные всех базовых типов и автоматически конвертировать их в нужный внутренний формат.

Функция scanf корректно считывает целые числа, если они начинаются с символа 0, или со знака "+". То есть числа "+123" или "0123" будут корректно считаны по форматной строке "%d", никаких дополнительных параметров задавать не нужно.

In C++:
Оператор извлечения `>>` используется для извлечения значений из потока. (istream)
Оператор вставки `<<` используется для помещения значений в поток.(ostream)
Класс iostream может обрабатывать как ввод, так и вывод данных, что позволяет ему осуществлять двунаправленный ввод/вывод.

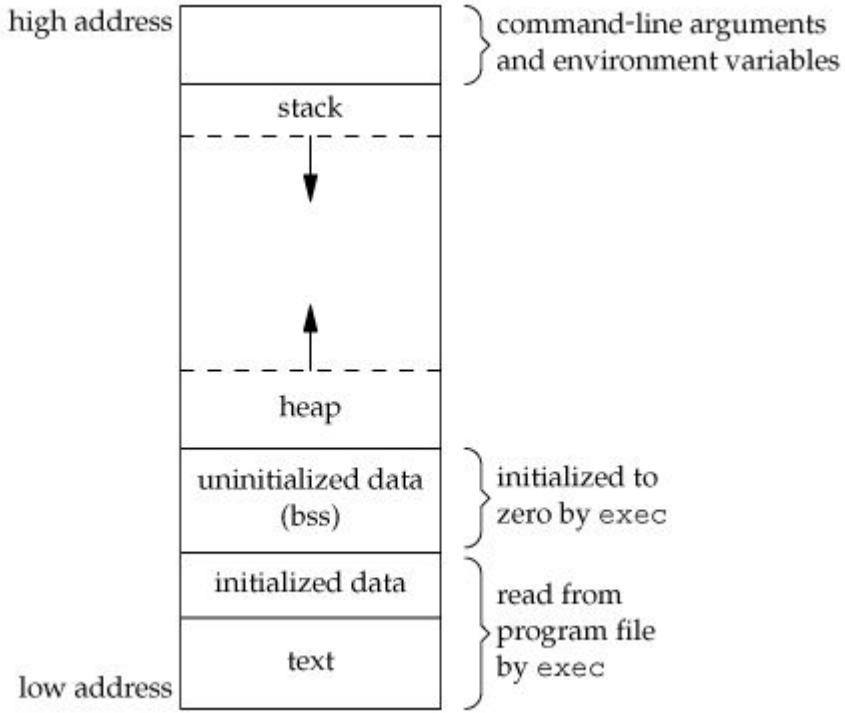
Memory

Динамическая память с произвольным доступом (DRAM).

SAM -- ячейки памяти выбираются (считываются) последовательно, одна за другой, в очерёдности их расположения. Вариант такой памяти — стековая память.

RAM -- вычислительное устройство может обратиться к произвольной ячейке памяти по любому адресу.

Memory


high address
command-line arguments
and environment variables
stack
heap
uninitialized data
(bss)
initialized to
zero by exec
initialized data
read from
program file
by exec
text
low address

Text segment

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

int main() {

    while (1) {
        printf("\tAddress of main: %p\n", &main);
        printf("\tMy process ID : %d\n", getpid());
        sleep(10);
    };

    return 0;
}
```

Text segment

```
Address of main: 0x560d08b2e6da
My process ID : 19752
```

```
$ cat /proc/19752/maps | head -n 1
```

560d08b2e000-560d08b2f000 r-xp 00000000 fe:01 16384016 …

адрес - это начальный и конечный адрес региона в адресном пространстве процесса.

Text segment

```
Address of main: 0x560d08b2e6da
My process ID : 19752


$ cat /proc/19752/maps | head -n 1

560d08b2e000-560d08b2f000 r-xp 00000000 fe:01 16384016 …
```

адрес - это начальный и конечный адрес региона в адресном пространстве процесса.

```
$ echo 'ibase=16;560D08B2E000-560D08B2F000' | bc

-4096        # байт
```

Text segment

```
Address of main: 0x560d08b2e6da
My process ID : 19752


$ cat /proc/19752/maps | head -n 1

560d08b2e000-560d08b2f000 r-xp 00000000 fe:01 16384016 …
```

адрес - это начальный и конечный адрес региона в адресном пространстве процесса.

```
$ echo 'ibase=16;560D08B2E000-560D08B2F000' | bc

-4096        # байт (размер одной страницы, PAGESIZE)
```

Text segment

```
Address of main: 0x560d08b2e6da
My process ID : 19752


$ cat /proc/19752/maps | head -n 1

560d08b2e000-560d08b2f000 r-xp 00000000 fe:01 16384016 …
```

адрес - это начальный и конечный адрес региона в адресном пространстве процесса.

```
$ echo 'ibase=16;560D08B2E000-560D08B2F000' | bc

-4096        # байт (размер одной страницы, PAGESIZE)
```

Data segment (Initialized)

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

int global_in_init = 10;

int main() {

    static const char somedata[8192] = "somedata";

    while (1) {
        printf("\tAddress of main: %p\n", &main);
        printf("\tMy process ID : %d\n", getpid());
        printf("\tThe global var's address is: %p\n", &global_in_init);
        sleep(10);
    };

    return 0;
}
```

Data segment (Initialized)

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

int global_in_init = 10;

int main() {

    static const char somedata[8192] = "somedata";

    while (1) {
        printf("\tAddress of main: %p\n", &main);
        printf("\tMy process ID : %d\n", getpid());
        printf("\tThe global var's address is: %p\n", &global_in_init);
        sleep(10);
    };

    return 0;
}
```

Data segment (Initialized)

```
$ gcc mem_lay.c -o mem_lay
$ size mem_lay

text    data     bss     dec     hex filename
10013     612       4   10629    2985 mem_lay
```

Предыдущий результат:

```
$ size mem_lay

text    data     bss     dec     hex filename
9949     608       8   10565    2945 mem_lay
```

Data segment (Uninitialized)

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

int global_in_init = 10;
int global_uninit;

int main() {

    static const char somedata[8192] = "somedata";

    while (1) {
        printf("\tAddress of main: %p\n", &main);
        printf("\tMy process ID : %d\n", getpid());
        printf("\tThe global var's address is: %p\n", &global_in_init);
        printf("\tThe uninit var's address is: %p\n", &global_uninit);
        sleep(10);
    };

    return 0;
}
```

Data segment (Uninitialized)

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

int global_in_init = 10;
int global_uninit;

int main() {

    static const char somedata[8192] = "somedata";

    while (1) {
        printf("\tAddress of main: %p\n", &main);
        printf("\tMy process ID : %d\n", getpid());
        printf("\tThe global var's address is: %p\n", &global_in_init);
        printf("\tThe uninit var's address is: %p\n", &global_uninit);
        sleep(10);
    };

    return 0;
}
```

Stack

```
$ ulimit -s
8192
```

https://gribblelab.org/CBootCamp/7_Memory_Stack_vs_Heap.html#orga366562

Heap

Сегмент памяти используемый под динамически выделяемые участки. Он начинается от **BSS** сегмента и растёт вверх, и используется такими функциями как `malloc()`, `calloc()` и `realloc()`, а для удаления данных из него — используется `free()`.

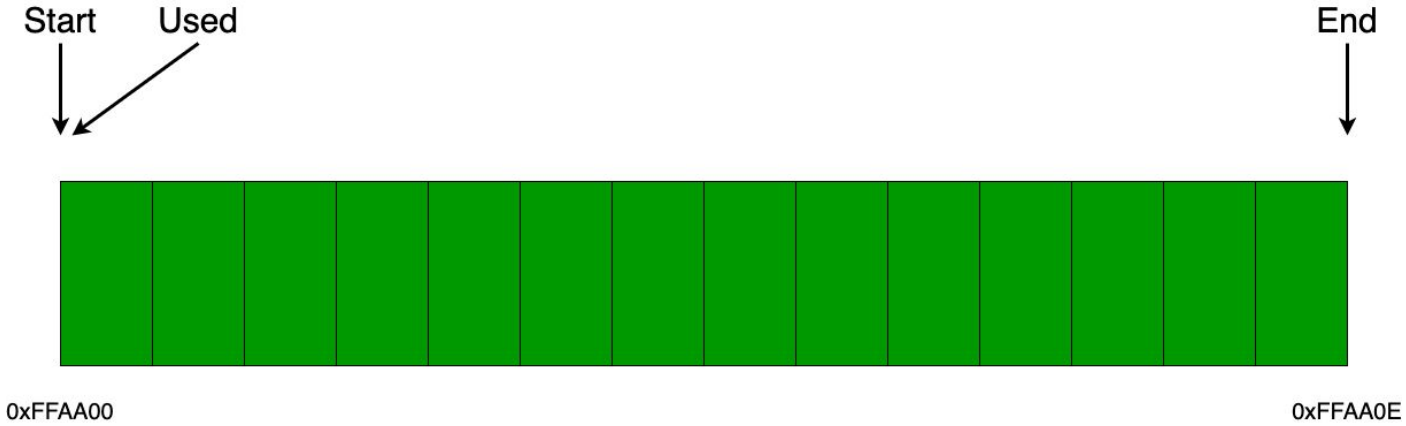
Allocators

Allocations

- *create* – создает аллокатор и отдает ему в распоряжение некоторый объем памяти;
- *allocate* – выделяет блок определенного размера из области памяти, которым распоряжается аллокатор;
- *deallocate* – освобождает определенный блок;
- *free* – освобождает все выделенные блоки из памяти аллокатора (память, выделенная аллокатору, не освобождается);
- *destroy* – уничтожает аллокатор с последующим освобождением памяти, выделенной аллокатору.

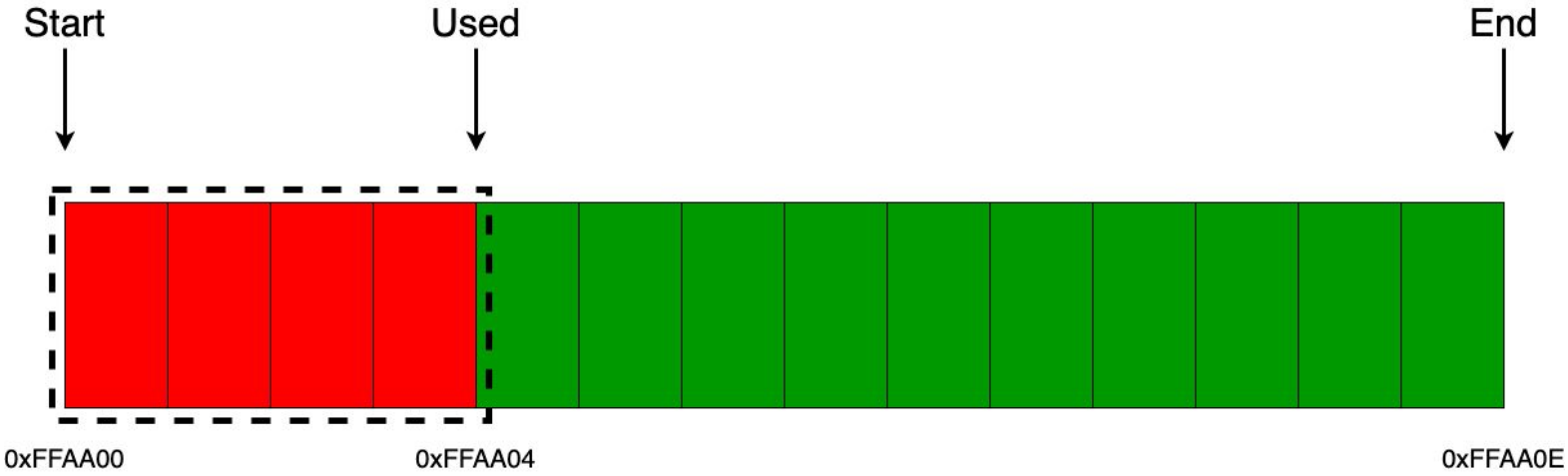
Linear Allocator

- проверить достаточно ли памяти для выделения;
- сохранить текущий указатель used, который в дальнейшем будет отдан пользователю, как указатель на блок выделенной памяти из аллокатора;
- сместить указатель used на величину равную объему выделенного блока памяти, т.е. на 4 байта.



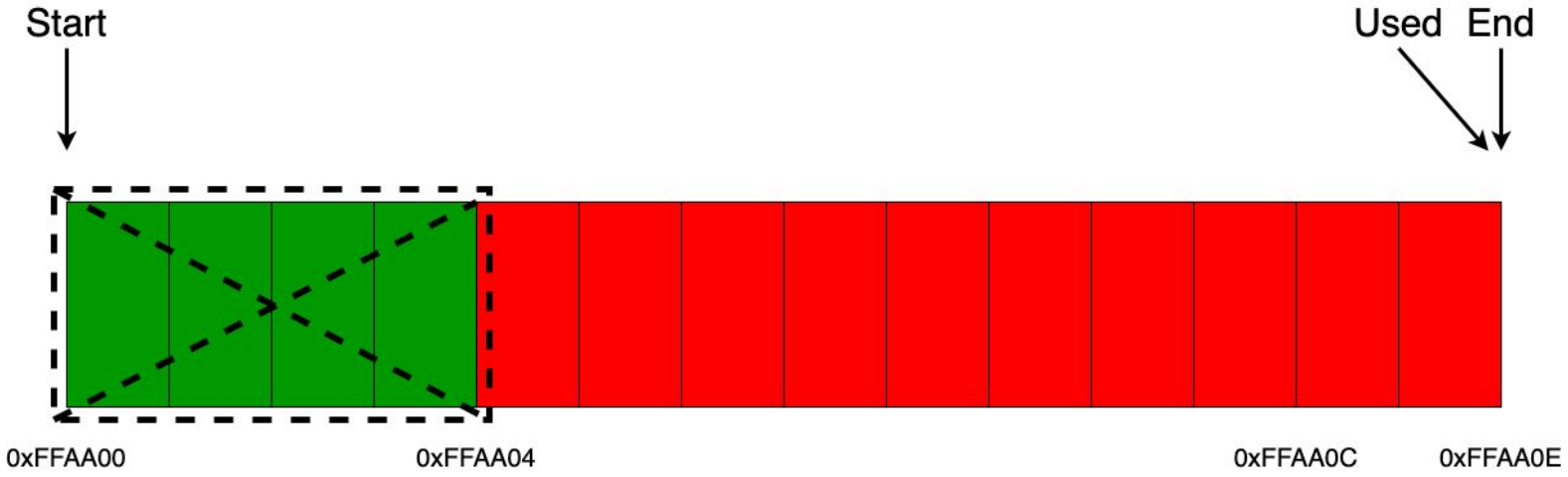
Linear Allocator

- проверить достаточно ли памяти для выделения;
- сохранить текущий указатель used, который в дальнейшем будет отдан пользователю, как указатель на блок выделенной памяти из аллокатора;
- сместить указатель used на величину равную объему выделенного блока памяти, т.е. на 4 байта.



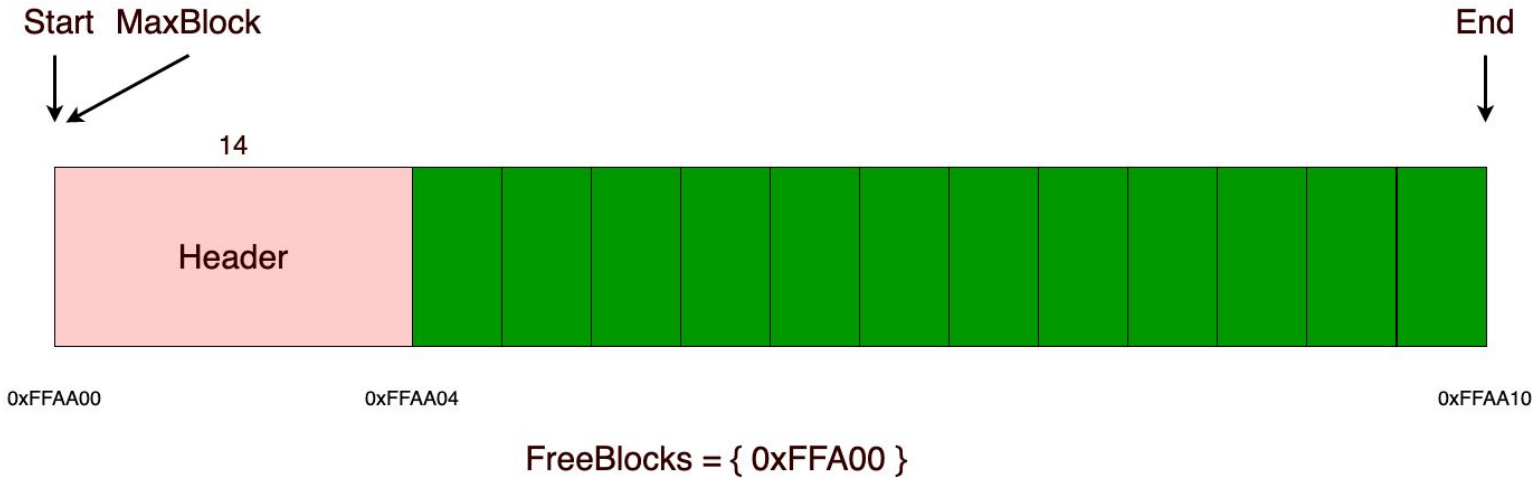
Linear Allocator

Данный вид аллокоторов не поддерживает выборочное освобождение определенных блоков памяти.



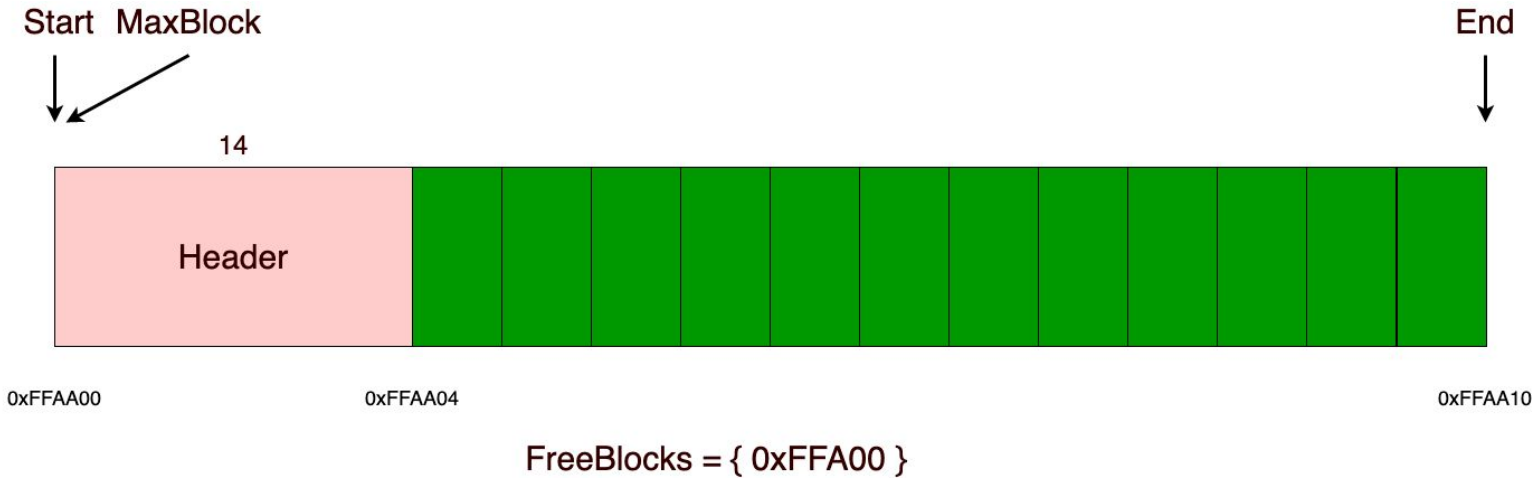
Примитивный STL Allocator

Внутри себя он будет содержать указатели на начало (*start*) и конец (*end*) памяти, указатель на максимальный блок памяти (*maxblock*) и множество (*freeblocks*), в котором будут храниться заголовки свободных блоков.



Примитивный STL Allocator

Внутри себя он будет содержать указатели на начало (*start*) и конец (*end*) памяти, указатель на максимальный блок памяти (*maxblock*) и множество (*freeblocks*), в котором будут храниться заголовки свободных блоков.

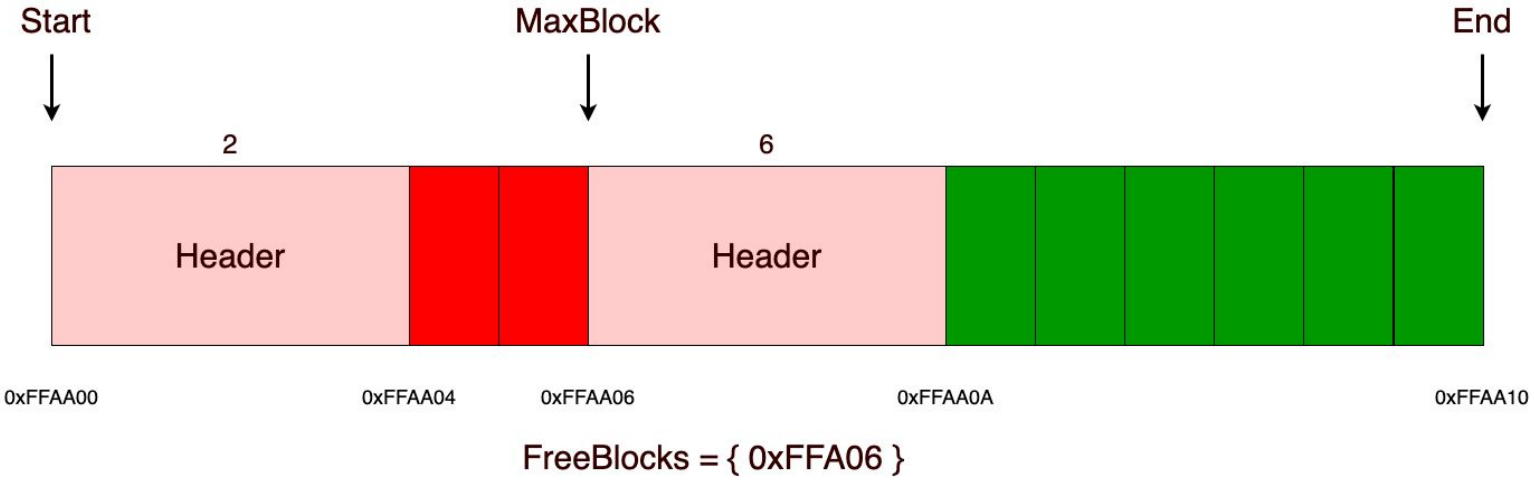


Примитивный STL Allocator

Прежде всего нужно проверить, достаточно ли памяти.

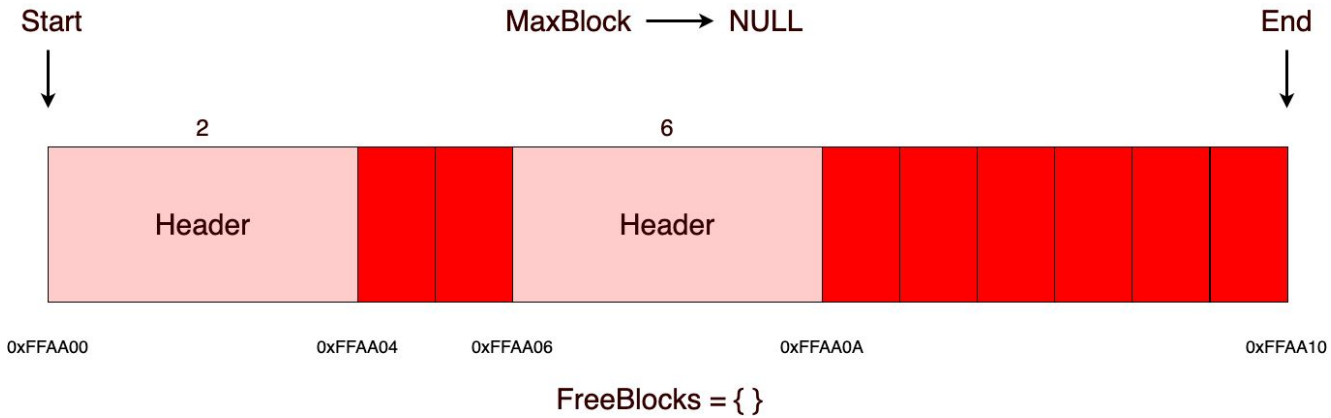
(Сравниваем с maxblock)

Если памяти достаточно, то мы просто отдаем адрес памяти, следующий за заголовком, как и в стековом аллокаторе, а также удаляем прошлый заголовок из множества свободных блоков и уже только после этого добавляем туда новый



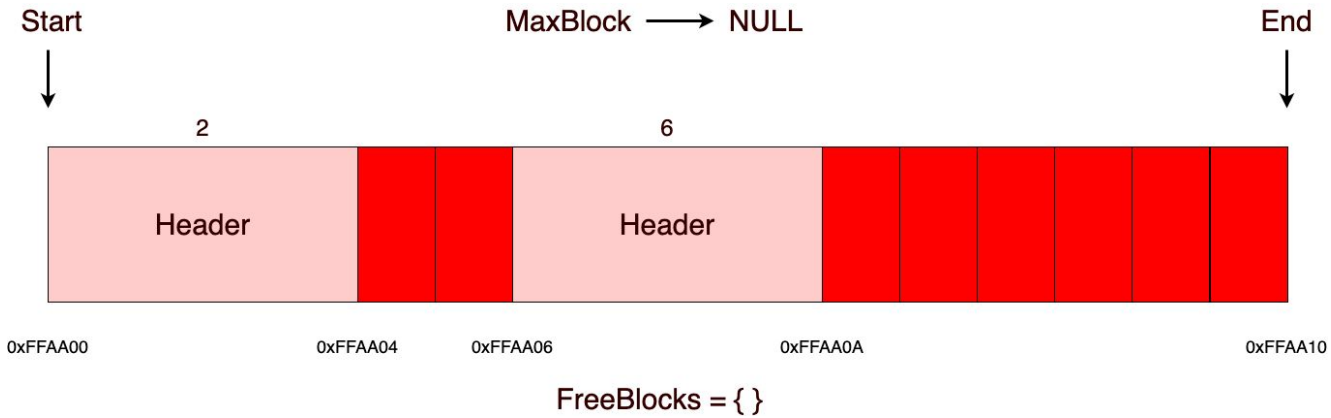
Примитивный STL Allocator

Но вот как только память в участке закончилась, аллокатор берет и просто создает еще один участок того же или большего размера (в данной реализации все участки одинакового размера). Стоит также следить за тем, чтобы размер возможного блока для выделения не превышал размер участка с вычетом размера заголовка.



Примитивный STL Allocator

Но вот как только память в участке закончилась, аллокатор берет и просто создает еще один участок того же или большего размера (в данной реализации все участки одинакового размера). Стоит также следить за тем, чтобы размер возможного блока для выделения не превышал размер участка с вычетом размера заголовка.

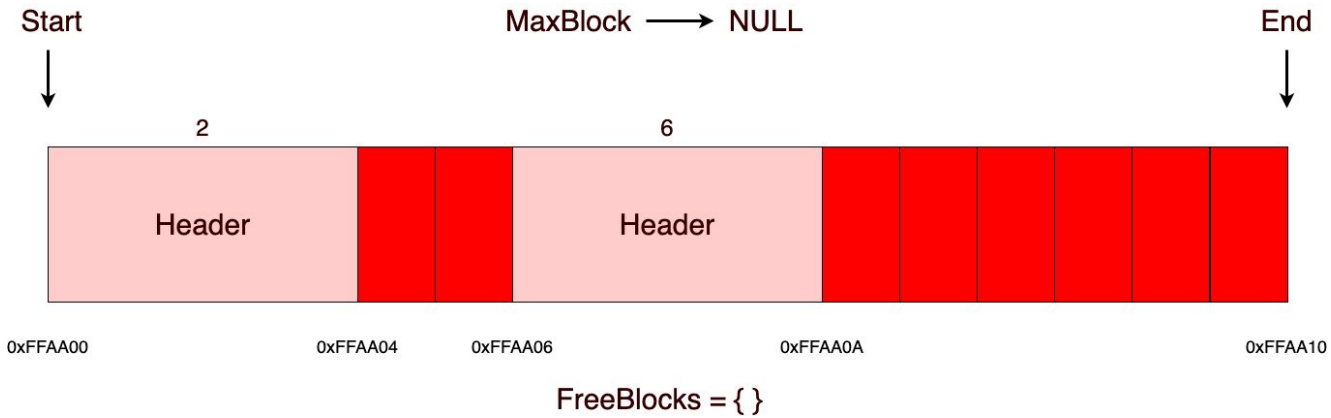


Malloc vs New
Delete vs Free

- `new` является типобезопасным, `malloc` возвращает объекты типа `void*`
- `new` создает исключение при ошибке, `malloc` возвращает `NULL` и устанавливает errno
- `new` является оператором и может быть перегружен, `malloc` является функцией и не может быть перегружен
- `new[]`, который выделяет массивы, является более интуитивным и типобезопасным, чем `malloc`
- `malloc`-производные распределения могут быть изменены с помощью `realloc` , `new`-производные распределения не могут быть изменены
- `malloc` может выделить N-байтовый кусок памяти, `new` необходимо попросить выделить массив, скажем, типов `char`

Примитивный STL Allocator

Но вот как только память в участке закончилась, аллокатор берет и просто создает еще один участок того же или большего размера (в данной реализации все участки одинакового размера). Стоит также следить за тем, чтобы размер возможного блока для выделения не превышал размер участка с вычетом размера заголовка.



Compilation

Компилятор

компилятор [опции] файлы [библиотеки]

Здесь **компилятор** - команда вызова компилятора;
основные опции:
-o - создать выходной файл с заданным именем (без опции создается a.out);
-c - не изготавливать исполняемый модуль (при компиляции подпрограмм);
-0 -01,-02,-03 - задание уровня оптимизации;
-g - выполнить компиляцию в отладочном режиме;
файлы - компилируемые файлы;;
библиотеки - подключаемые библиотеки.

Компилятор

cpp – препроцессор

as – ассемблер

g++ – сам компилятор

ld – линкер

Этапы компиляции

example.cpp:

```
#include <iostream>
using namespace std;
#define RETURN return 0

int main() {
    cout << "Hello, world!" << endl;
    RETURN;
}
```

Этапы компиляции

Препроцессинг:

```
g++ -E example.cpp -o example.ii
```

Этапы компиляции

Препроцессинг:

```
g++ -E example.cpp -o example.ii
```

example.ii:

```
int main() {
    cout << "Hello, world!" << endl;
    return 0;
}
```

Этапы компиляции

Компиляция :

```
$ g++ -S example.ii -o example.s
```

```
.file   "example.cpp"
    .local  ZStL8  ioinit
    .comm   ZStL8  ioinit,1,1
    .section    .rodata
.LC0:
    .string "Hello, world!"
    .text
    .globl  main
    .type   main, @function
main:
.LFB1021:
    .cfi_startproc……
```

Этапы компиляции

Ассемблирование:

```
$ as driver.s -o driver.o
```

Но на данном шаге еще ничего не закончено, ведь объектных файлов может быть много и нужно их всех соединить в единый исполняемый файл с помощью компоновщика (линкера). Поэтому мы переходим к следующей стадии.

Этапы компиляции

Компоновка

Компоновщик (линкер) связывает все объектные файлы и статические библиотеки в единый исполняемый файл, который мы и сможем запустить в дальнейшем. Для того, чтобы понять как происходит связка, следует рассказать о *таблице символов*.

Этапы компиляции

```
Загрузка (запуск)

$ ./driver
// Hello, world!
```

ASM

QEMU и ARM

Процесс сборки программ, предназначенных для другой процессорной архитектуры или операционной системы называется кросс-компиляцией.

Для этого необходимо специальная версия компилятора `gcc`, предназначенного для другой платформы. Во многих дистрибутивах существуют отдельные пакеты компилятора для других платформ, включая ARM.

QEMU и ARM

Процесс сборки программ, предназначенных для другой процессорной архитектуры или операционной системы называется кросс-компиляцией.

Для этого необходимо специальная версия компилятора `gcc`, предназначенного для другой платформы. Во многих дистрибутивах существуют отдельные пакеты компилятора для других платформ, включая ARM.

Выполнение программ, предназначенных для других архитектур, возможно только интерпретацией инородного набора команд. Для этого предназначены специальные программы - *эмуляторы*.

QEMU и ARM

Идеальный вариант для тестирования и отладки - это использовать настоящее железо, например Raspberry Pi.

Если под рукой нет компьютера с ARM-процессором, то можно выполнять эмуляцию ПК с установленной системой Raspbian.

arm.com

Написание и компиляция кода

Программы на языка ассемблера для компилятора GNU сохраняются в файле, имя которого оканчивается на .s или .S. Во втором случае (с заглавной буквой) подразумевается, что текст программы может быть обработан препроцессором.

Общий синтаксис

```
// Это комментарий, как в C++

    .text       // начало секции .text с кодом программы
    .global f   // указание о том, что метка f
                // является доступной извне (аналог extern)

f:              // метка (заканчивается двоеточием)

     // последовательность команд
     mul   r0, r0, r3
     mul   r0, r0, r3
     mul   r1, r1, r3
     add   r0, r0, r1
     add   r0, r0, r2
     mov   r1, r0
    bx    lr
```

Регистры

Процессор может выполнять операции только над *регистрами* - 32-битными ячейками памяти в ядре процессора. У ARM есть 16 регистров, доступных программно: `r0`, `r1`, ... ,`r15`.

У регистров `r13`...`r15` имеются специальные назначения и дополнительные имена:

- `r15` = `pc`: Program Counter - указатель на текущую выполняемую инструкцию
- `r14` = `lr`: Link Register - хранит адрес возврата из функции
- `r13` = `sp`: Stack Pointer - указатель на вершину стека.

Регистры

Процессор может выполнять операции только над *регистрами* - 32-битными ячейками памяти в ядре процессора. У ARM есть 16 регистров, доступных программно: `r0`, `r1`, ... ,`r15`.

У регистров `r13`...`r15` имеются специальные назначения и дополнительные имена:

- `r15` = `pc`: Program Counter - указатель на текущую выполняемую инструкцию
- `r14` = `lr`: Link Register - хранит адрес возврата из функции
- `r13` = `sp`: Stack Pointer - указатель на вершину стека.

Флаги

Выполнение команд может приводить к появлению некоторой дополнительной информации, которая хранится в *регистре флагов*. Флаги относятся к последней выполненной команде. Основные флаги, это:

- `C`: Carry - возникло беззнаковое переполнение
- `V`: oVerflow - возникло знаковое переполнение
- `N`: Negative - отрицательный результат
- `Z`: Zero - обнуление результата.

Команды

Архитектура ARM-32 подразумевает, что почти команды могут иметь *условное выполнение*. Условие кодируется 4-мя битами в самой команде, а с точки зрения синтаксиса ассемблера у команд могут быть суффиксы.

Таким образом, каждая команда состоит из двух частей (без разделения пробелами): сама команда и её суффикс.

Команды

`AND regd, rega, argb` // regd ← rega & argb

`EOR regd, rega, argb` // regd ← rega ^ argb

`SUB regd, rega, argb` // regd ← rega − argb

`RSB regd, rega, argb` // regd ← argb - rega

`ADD regd, rega, argb` // regd ← rega + argb

`TST rega, argb` // set flags for rega & argb

`TEQ rega, argb` // set flags for rega ^ argb

`CMP rega, argb` // set flags for rega − argb

`MOV regd, arg` // regd ← arg

`BIC regd, rega, argb` // regd ← rega & ~argb

`MVN regd, arg` // regd ← ~argb

Условия

EQ	equal (Z)
NE	not equal (!Z)
CS or HS	carry set / unsigned higher or same (C)
CC or LO	carry clear / unsigned lower (!C)
MI	minus / negative (N)
PL	plus / positive or zero (!N)
VS	overflow set (V)
VC	overflow clear (!V)

Переходы

Счетчик `pc` автоматически увеличивается на 4 при выполнении очередной инструкции. Для ветвления программ используются команды:

- `B label` - переход на метку; используется внутри функций для ветвлений, связанных с циклами или условиями
- `BL label` - сохранение текущего `pc` в `lr` и переход на `label`; обычно используется для вызова функций
- `BX register` - переход к адресу, указанному в регистре; обычно используется для выхода из функций.

Работа с памятью

Процессор может выполнять операции только над регистрами. Для взаимодействия с памятью используются отдельные инструкции загрузки/сохранения регистров.

- `LDR regd, [regaddr]` - загружает машинное слово из памяти по адресу, хранящимся в regaddr, и сохраняет его в регистре regd
- `STR reds, [regaddr]` - сохраняет в памяти машинное слово из регистра regs по адресу, указанному в регистре regaddr.

Кодогенерация

- https://godbolt.org/

Вопросы