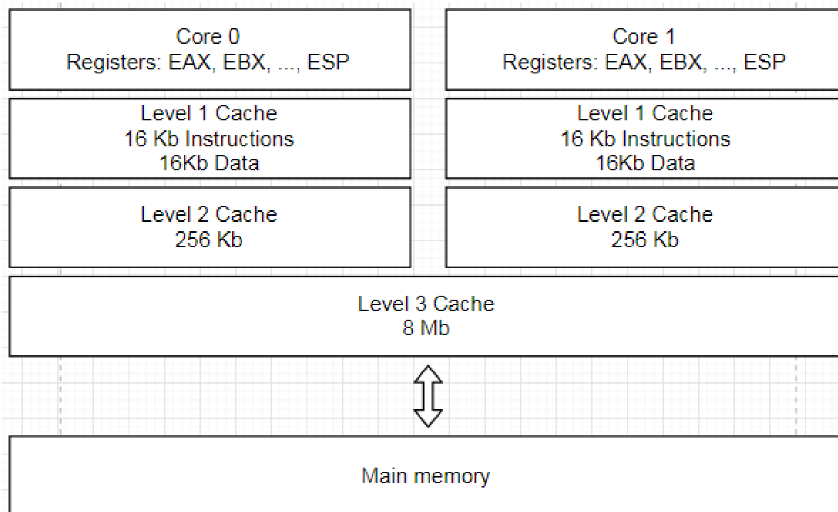
02.memory.md

Память в C++

Кеш, оперативная память, стек и куча, выделение и освобождение памяти

Процессор



Линейное представление памяти

Адрес	Значение (1 байт)
0x0000	...
...	...
0x1000	1
0x1001	2
0x1002	3
0x1003	4
...	...
0xffffffffff	...

Арифметика указателей

```
// Просто хранит какой-то адрес
void* addr = 0x1000;

// Если указатель никуда не ссылается,
// надо использовать nullptr
void* invalid = nullptr;

// Размер указателя, например, 4 - это количество
// байт необходимых для размещения адреса
size_t size = sizeof(addr); // size == 4

// Теперь мы говорим компилятору как
// интерпретировать то, на что указывет
// указатель
char* charPtr = (char*) 0x1000;

// Разыменование - получение значения, находящегося
```

```
// по указанному адресу
char c = *charPtr; // c == 1

// & - взятие адреса, теперь в charPtrPtr находится
// адрес charPtr
char** charPtrPtr = &charPtr;

int* intPtr = (int*) addr;
int i = *intPtr; // i == 0x04030201 (little endian)

int* i1 = intPtr;
int* i2 = i1 + 2;

ptrdiff_t d1 = i2 - i1; // d1 == 2

char* c1 = (char*) i1;
char* c2 = (char*) i2;

ptrdiff_t d2 = c2 - c1; // d2 == 8
```

```
T* + n -> T* + sizeof(T) * n
T* - n -> T* - sizeof(T) * n
```

> C-cast использовать в C++ нельзя! Как надо приводить типы в C++ и надо ли вообще будет в другой лекции

Целочисленные типы

Знаковые	Беззнаковые
char	unsigned char
short	unsigned short
int	unsigned или unsigned int
long	unsigned long

> Стандарт не регламентирует размер типов

```
#include <cstdint>
```

Размер, бит	Тип
8	int8_t, int_fast8_t, int_least8_t
16	int16_t, int_fast16_t, int_least16_t
32	int32_t, int_fast32_t, int_least32_t
64	int64_t, int_fast64_t, int_least64_t

Беззнаковая (unsigned) версия - добавление префикса `u`

mem.cpp

```
#include <iostream>
#include <cstdint>

int global = 0;

int main()
{
    int* heap = (int*) malloc(sizeof(int));

    std::cout << std::hex << (uint64_t) main << '\n';
    std::cout << std::hex << (uint64_t) &global << '\n';
    std::cout << std::hex << (uint64_t) heap << '\n';
    std::cout << std::hex << (uint64_t) &heap << '\n';

    char c;
```

```
    std::cin >> c;
    return 0;
}
```

```
g++ -O0 mem.cpp -o mem --std=c++11
./mem
```

```
400986
6022b4
18adc20
7ffd5591e7d0
```

/proc/.../maps

```
ps ax | grep mem
```

```
00400000-00401000 r-xp 00000000 08:01 2362492
        /home/mt/work/tmp/mem
00601000-00602000 r--p 00001000 08:01 2362492
        /home/mt/work/tmp/mem
00602000-00603000 rw-p 00002000 08:01 2362492
        /home/mt/work/tmp/mem
0189c000-018ce000 rw-p 00000000 00:00 0
        [heap]
7f66aaa53000-7f66aabc5000 r-xp 00000000 08:01 6826866
        /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.21
7f66aadc5000-7f66aadcf000 r--p 00172000 08:01 6826866
        /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.21
7f66aadcf000-7f66aadd1000 rw-p 0017c000 08:01 6826866
        /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.21
7ffd55900000-7ffd55921000 rw-p 00000000 00:00 0
        [stack]
7ffd55952000-7ffd55954000 r--p 00000000 00:00 0
        [vvar]
7ffd55954000-7ffd55956000 r-xp 00000000 00:00 0
        [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0
        [vsyscall]
```

Память разбита на сегменты:

- кода (CS)
- данных (DS)
- стека (SS)

Регистр сегмента (CS, DS, SS) указывают на дескриптор.

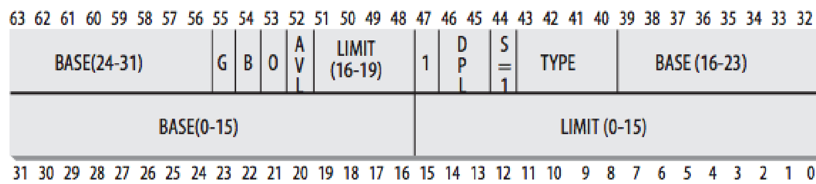
Для инструкций и стека на смещение в сегменте указывает регистр:

- кода (EIP)
- стека (ESP)

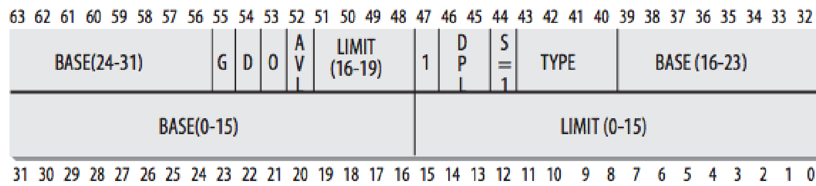
Линейный адрес - это сумма базового адреса сегмента и смещения.

Дескриптор

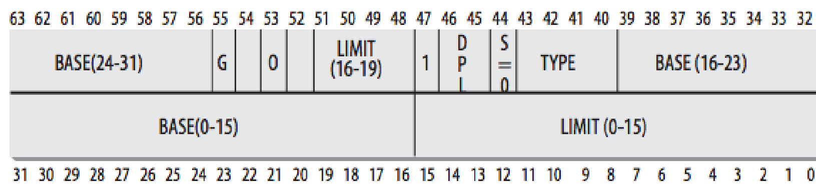
Data Segment Descriptor



Code Segment Descriptor



System Segment Descriptor



Segment limit (20 bit) - размер сегмента, 55-й бит G определяет гранулярность размера:

- байты, если 0
- страницы, если 1 (размер страницы обычно 4Кб)

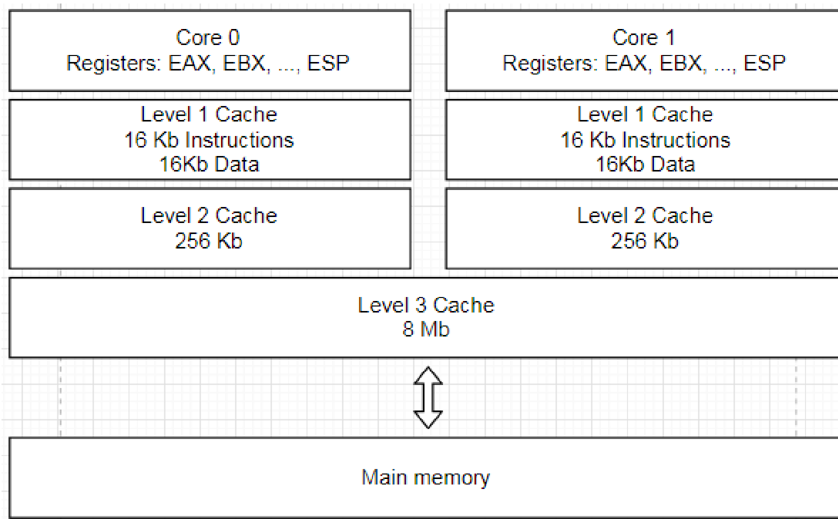
Бит 41-43:

- 000 - сегмент данных, только чтение
- 001 - сегмент данных, чтение и запись
- 010 - сегмент стека, только чтение
- 011 - сегмент стека, чтение и запись
- 100 - сегмент кода, только выполнение
- 101- сегмент кода, чтение и выполнение

Виртуальная память

- Память делится на страницы
- Страница может находится в оперативной памяти или на внешнем носителе
- Трансляция из физического адреса в виртуальный и обратно выполняется через специальные таблицы: PGD (Page Global Directory), PMD (Page Middle Directory) и PTE (Page Table Entry). В PTE хранятся физические адреса страниц
- Для ускорения трансляции адресов процессор хранит в кеше таблицу TLB (Translation lookaside buffer)
- Если обращение к памяти не может быть оттранслировано через TLB, процессор обращается к таблицам страниц и пытается загрузить PTE оттуда в TLB. Если загрузка не удалась, процессор вызывает прерывание Page Fault
- Обработчик прерывания Page Fault находится в подсистеме виртуальной памяти ядра ОС и может загрузить требуемую страницу с внешнего носителя в оперативную память

Процессор



Важные константы

```
1 такт = 1 / частота процессора
1 / 3 GHz = 0.3 ns
                                             0.3 ns
L1 cache reference                           0.5 ns
Branch mispredict                            5   ns
```

> Неудачный if ()

```
L2 cache reference                           7   ns
Mutex lock/unlock                           25   ns
Main memory reference                      100   ns
```

> Кроме задержки (latency) есть понятие пропускной способности (throughput, bandwidth). В случае чтения из RAM - 10-50 Gb/sec

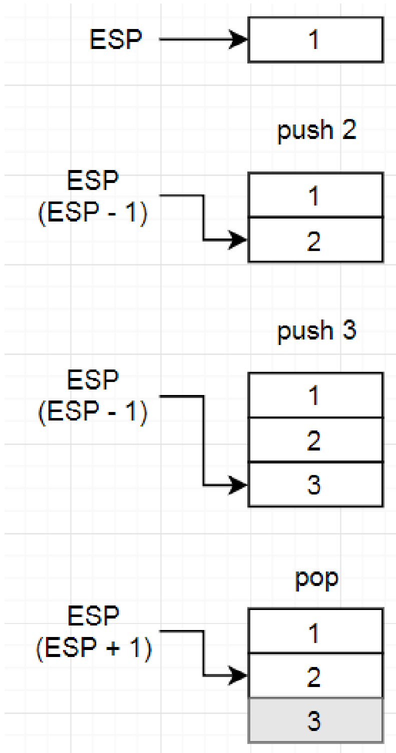
```
Compress 1K bytes with Zippy             3,000   ns
Send 1K bytes over 1 Gbps network       10,000   ns
Read 4K randomly from SSD              150,000   ns
Read 1 MB sequentially from memory     250,000   ns
Round trip within same datacenter      500,000   ns
Read 1 MB sequentially from SSD      1,000,000   ns
HDD seek                            10,000,000   ns
Read 1 MB sequentially from HDD     20,000,000   ns
Send packet CA->Netherlands->CA    150,000,000   ns
```

Источник: https://gist.github.com/jboner/2841832

Выводы из таблицы

1. Стараться укладывать данные в кеш
2. Минимизировать скачки по памяти
3. В условиях основной веткой делать ветку которая выполняется с большей вероятностью

Стек



Классы управления памятью и областью видимости в C++

Характеризуются тремя понятиями:

1. **Время жизни**

> Продолжительность хранения данных в памяти

2. **Область видимости**

> Части кода из которых можно получить доступ к данным

3. **Связывание (linkage)**

> Если к данным можно обратиться из другой единицы трансляции — связывание внешнее (external), иначе связывание внутреннее (internal)

Автоматический/регистровый (register)

Время жизни	Область видимости	Связывание
Автоматическое (блок)	Блок	Отсутствует

```
{
        int i = 5;
}

if (true)
{
        register int j = 3;
}

for (int k = 0; k < 7; ++k)
{
}
```

Статический без связывания

Время жизни	Область видимости	Связывание
Статическое	Блок	Отсутствует

```
void foo()
{
        static int j = 3;
}
```

> Инициализируется при первом обращении

Статический с внутренним связыванием

Время жизни	Область видимости	Связывание
Статическое	Файл	Внутреннее

```
static int i = 5;
```

> Инициализируется до входа в main

Статический с внешним связыванием

Время жизни	Область видимости	Связывание
Статическое	Файл	Внешнее

```
// *.cpp
int i = 0;
```

```
// *.h
extern int i;
```

Типы памяти

Стек (Stack)

```
int i = 5;
std::string name;
char data[5];
```

> Выделение памяти на стеке очень быстрая, но стек не резиновый

Куча (Heap)

```
int* i = (int*) malloc(sizeof(int));
std::string* name = new std::string();
char* data = new char[5];
...
free(i);
delete(name);
delete[] data;
```

Память в куче выделяют new и malloc, есть сторонние менеджеры памяти.

Основное:

- new то же, что и malloc, только дополнительно вызывает конструкторы
- Внутри malloc есть буфер, если в буфере есть место, ваш вызов может выполниться быстро
- Если памяти в буфере нет, будет запрошена память у ОС (sbrk, VirtualAlloc) - это дорого
- ОС выделяет память страницами от 4Кб, а может быть и все 2Мб
- Стандартные аллокаторы универсальные, то есть должны быть потокобезопасны, быть одинаково эффективны для блоков разной длины, и 10 байт и 100Мб. Плата за универсальность - быстродействие

valgrind

```
#include <cstdlib>

int main()
{
    int* data = (int*) malloc(1024);
    return 0;
}
```

```
valgrind ./mem
```

```
==117392== Memcheck, a memory error detector
==117392== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==117392== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==117392== Command: ./mem
==117392==
==117392==
==117392== HEAP SUMMARY:
==117392==     in use at exit: 1,024 bytes in 1 blocks
==117392==   total heap usage: 1 allocs, 0 frees, 1,024 bytes allocated
==117392==
==117392== LEAK SUMMARY:
==117392==    definitely lost: 1,024 bytes in 1 blocks
==117392==    indirectly lost: 0 bytes in 0 blocks
==117392==      possibly lost: 0 bytes in 0 blocks
==117392==    still reachable: 0 bytes in 0 blocks
==117392==         suppressed: 0 bytes in 0 blocks
==117392== Rerun with --leak-check=full to see details of leaked memory
==117392==
==117392== For counts of detected and suppressed errors, rerun with: -v
==117392== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Глобальная память (data segment)

```
static const int i = 5;
static std::string name;
extern char data[5];
```

> Если не удастся разместить блок глобальной памяти, то программа даже не запустится

Массивы

```
T array[maxColumns];
T value = array[x];
```

> Значение в квадратных скобках должно быть известно на этапе компиляции, увы

```
int data[] = { 1, 2, 3 };
int i = data[2];
```

Фактически - это вычисление смещения:

```
ptr = data;
ptr = ptr + 2 * sizeof(int);
i = *ptr;
```

Массив - непрерывный блок байт в памяти, sizeof(data) вернет размер массива в байтах (не элементах!). Размер массива в элементах можно вычислить: sizeof(data) / sizeof(data[0])

```
int* data = new int[10];
int i = data[2];
delete[] data;
```

Массив <-> указатель

```
int i[] = { 1, 2, 3 };
int* j = i;
using array = int*;
array k = (array) j;
```

Двумерные массивы

```
T array[maxRows][maxColumns];
T value = array[y][x];
```

```
int data[][2] = { { 1, 2 },  { 3, 4 }, { 5, 6 } };
int i = data[2][1];
```

Фактически:

```
ptr = data;
ptr = ptr + maxColumns * sizeof(int) * 2 + 1;
i = *ptr;
```

Массив <-> указатель

```
int i[][2] = { { 1, 2 }, { 3, 4 }, { 5, 6 } };
int* j = (int*) i;
using matrix = int(*)[2];
matrix k = (matrix) j;
```

Измеряем скорость работы (benchmark)

1. Измерений должно быть много
2. Одному прогону верить нельзя
3. Компилятор оптимизирует, надо ему мешать
4. Перед тестами надо греться

Пример "вредной" оптимизации

```
int main()
{
    Timer t;
    for (int i = 0; i < 100 * 1000 * 1000; ++i)
        int a = i / 3;
    return 0;
}
```

Не даем компилятору оптимизировать

```
int main()
{
    Timer t;
    for (int i = 0; i < 100 * 1000 * 1000; ++i)
        volatile int a = i / 3;
    return 0;
}
```

Класс Timer

```
#include <chrono>
#include <iostream>

class Timer
{
    using clock_t = std::chrono::high_resolution_clock;
    using microseconds = std::chrono::microseconds;
```

```
public:
    Timer()
        : start_(clock_t::now())
    {
    }

    ~Timer()
    {
        const auto finish = clock_t::now();
        const auto us =
            std::chrono::duration_cast<microseconds>
                (finish - start_).count();
        std::cout << us << " us" << std::endl;
    }

private:
    const clock_t::time_point start_;
};
```

Практическая часть

Написать две программы суммирующие элементы двумерного массива (матрицы) целых чисел. Одна программа суммирует по столбцам, вторая по строкам. Измерить время работы в обоих случаях, сравнить результаты. Для замеров можно использовать класс Timer. В репозитории в директории homework создать директорию со своей фамилией, внутри этой директории создать директорию 01, файлы с решением положить внутрь. Программу компилировать с влюченной оптимизацией,, например, так:

```
g++ sum_by_rows.cpp -o sum_by_rows --std=c++11 -O2
```

Оба решения можно запустить через valgrind и посмотреть количество промахов в кеш:

```
valgrind --tool=cachegrind your_program
```

Подумать.

EOF