Advanced C++

#2 Память.
Арифметика указателей.
Типы в C++.
Функции.

Menu

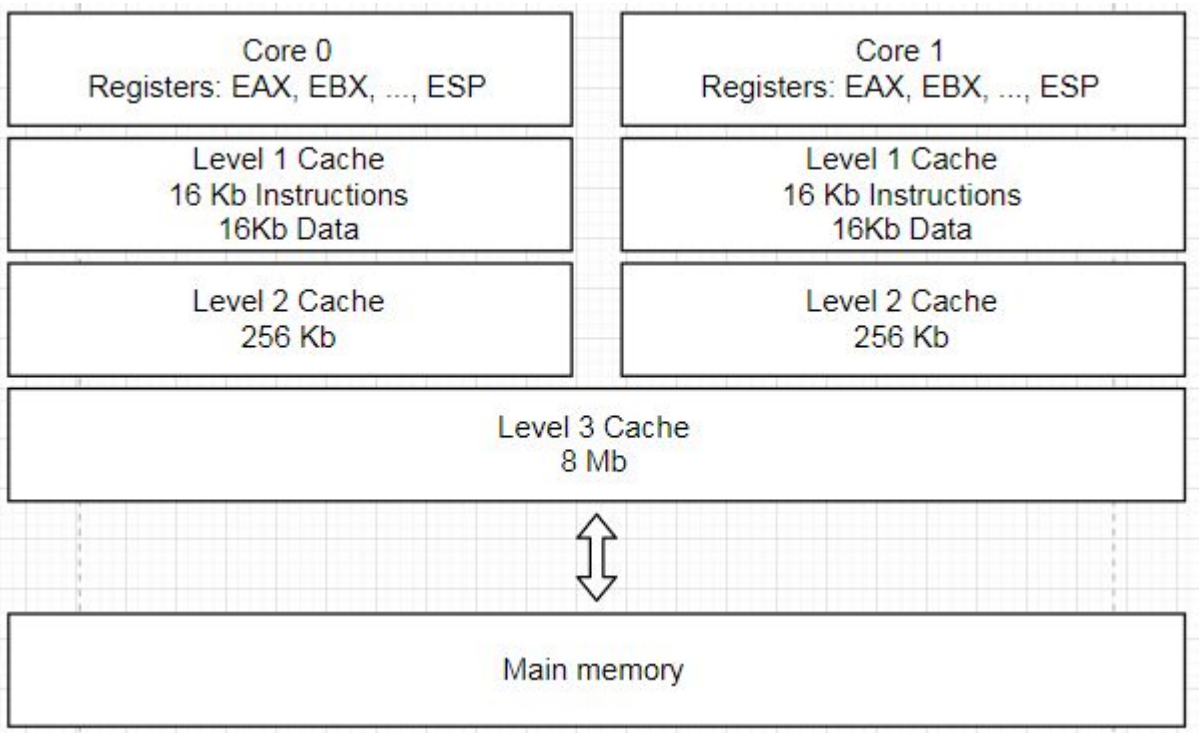
- Еще немного про память: кеши
- Представление памяти, виртуальная память

- Арифметика указателей
- Встроенные целочисленные типы

- Функции
- Конфликты имен, namespace
- Передача по ссылке и по значению

Кэш памяти. Указатели

Процессор


Core 0
Registers: EAX, EBX, ..., ESP
Core 1
Registers: EAX, EBX, ..., ESP
Level 1 Cache
16 Kb Instructions
16Kb Data
Level 1 Cache
16 Kb Instructions
16Kb Data
Level 2 Cache
256 Kb
Level 2 Cache
256 Kb
Level 3 Cache
8 Mb
Main memory

Представление памяти

0x0000	-6567478
...	
0x1000	1
0x1001	2
...	
0xfffffffff	0

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 указывает
// указатель
 char* charPtr = (char*) 0x1000;
```

Арифметика указателей

```
// & - 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
```

Арифметика указателей

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

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

Встроенные целочисленные типы

При этом, размер типов не регламентируется стандартом
Соблюдается только иерархия размеров типов.

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

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

Пример

```
#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;
}
```

Пример

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

Пример

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

400986
6022b4
18adc20
7ffd5591e7d0
```

Пример

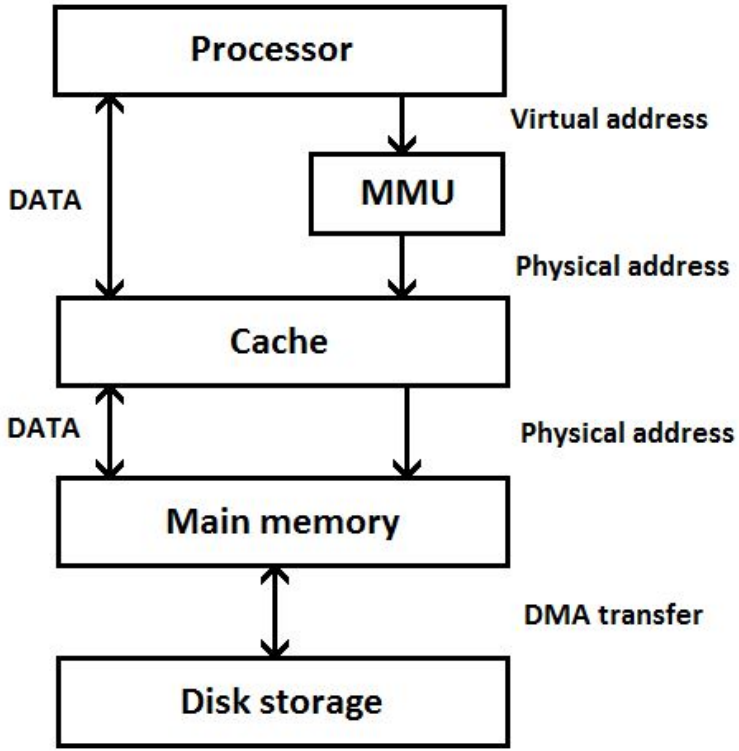
```
/proc/.../maps

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

  [stack]

7ffd55952000-7ffd55954000 r--p 00000000 00:00 0
```

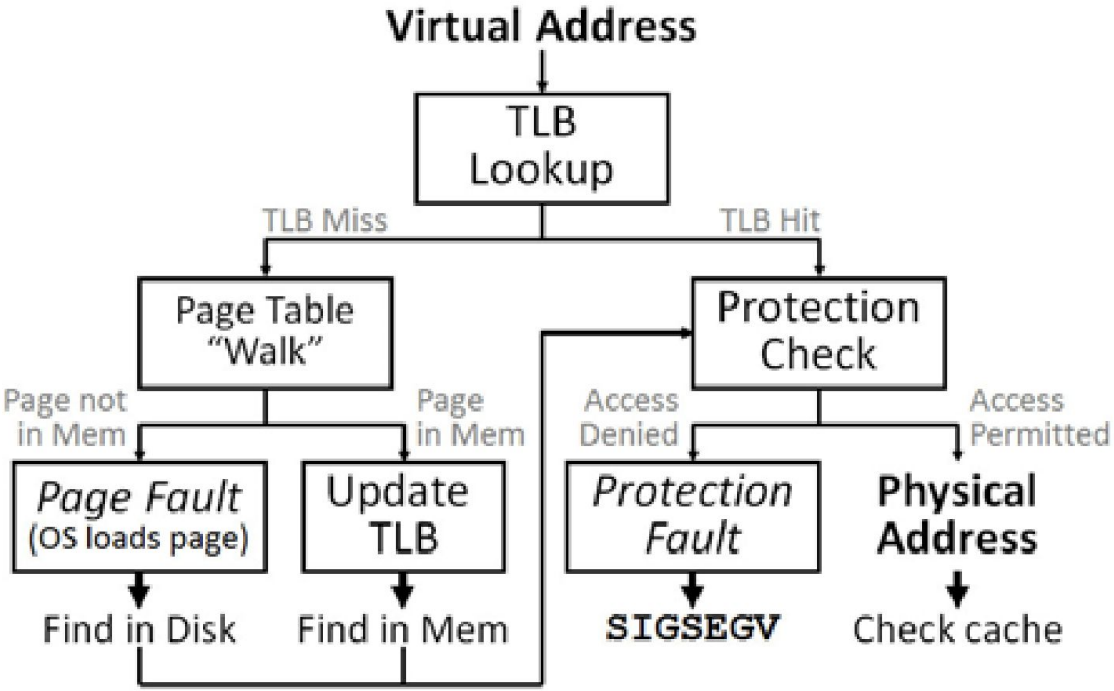
Виртуальная память



https://habr.com/ru/post/270009/

MMU & TLB

TLB - Translation Lookaside Buffer



Константы

1 такт = 1 / частота процессора
1 / 3 GHz = 0.3 ns

Branch mispredict	5	ns
L1 cache reference	0.5	ns
L2 cache reference	7	ns
Mutex lock/unlock	25	ns
Main memory reference	100	ns
Read 1 MB sequentially from SSD	1,000,000	ns
HDD seek	10,000,000	ns

Область Видимости. Lifetime

Характеристики

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 (register)

```
{
    int i = 5;
}

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

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

Автоматический (register)

```
{
     (auto)int i = 5;
}

if (true)
{
     register int j = 3; // хотим разместить в регистре
}

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

Время жизни	Область видимости	Связывание
Автоматическое (блок)	Блок	Отсутствует

Static without linkage

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

Время жизни	Область видимости	Связывание
Статическое	Блок	Отсутствует

Static without linkage

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

(Ленивые вычисления)

Время жизни	Область видимости	Связывание
Статическое	Блок	Отсутствует

Static internal linkage

```
static int j = 3;


(Main scope)
```

Время жизни	Область видимости	Связывание
Статическое	Файл	Внутреннее

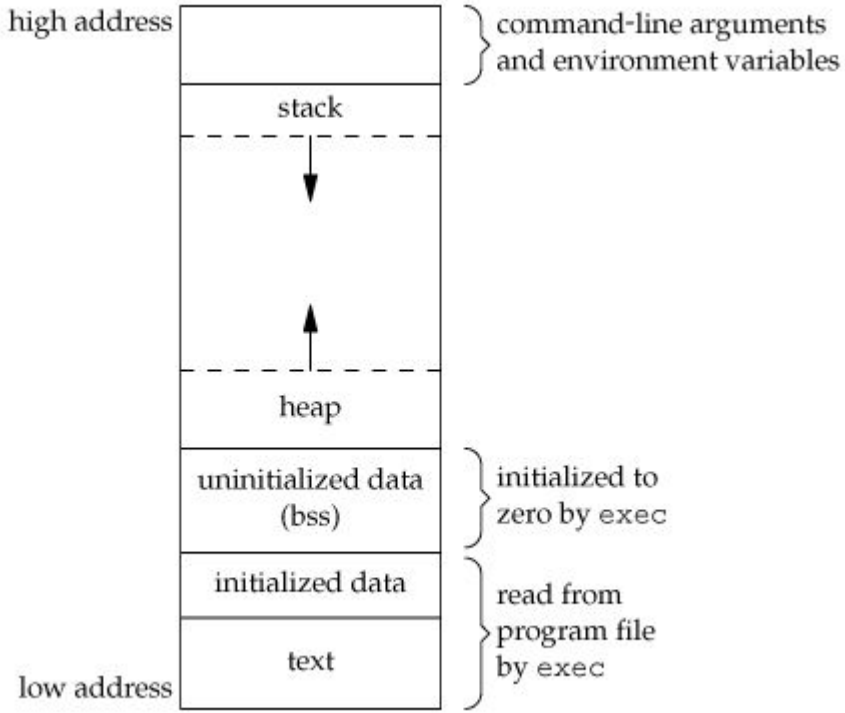
Static external linkage

```
extern int j; // *.h
int j = 3; // *.cpp
```

(Main scope)

Время жизни	Область видимости	Связывание
Статическое	Файл	Внутреннее

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

Valgrind

```
#include <cstdlib>

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

```
valgrind ./mem
```

Valgrind

```
#include <cstdlib>

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

```
valgrind ./mem
```

Namespace

Пространства имен

```
#include <cstdlib>

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

```
valgrind ./mem
```

Пространства имен

Проблема:

```
// math.h
 double cos(double x);


// ваш код
 double cos(double x);
```

Пространства имен

Решение в стиле C:

```
// ваш код
 double fastCos(double x);
```

Пространства имен

Решение в стиле C++:

```
namespace fast
{
    double cos(double x);
}

fast::cos(x);
cos(x); // вызов из math.h
```

Поиск имен

```
void foo() {} // ::foo

namespace A
{
    void foo() {} // A::foo

    namespace B
    {
        void bar() // A::B::foo
        {
            foo(); // A::foo
            ::foo(); // foo()
        }
    }
}
```

Using

```
void foo()
{
    using namespace A;
    // видимо все из A
}
```

```
void foo()
{
    using namespace A::foo;
    // видима только A::foo()
}
```

Using

```
void foo()
{
    namespace ab = A::B;
    ab::bar(); // A::B::bar()
}
```

Не используйте using namespace в заголовочных файлах!

Функции

Функции

```
int square(int x)
{
    int tmp = x * x;
    return tmp;
}
```

Конвенции вызова

```
void foo1(int x, int y, int z, int a)
{
}

void foo2(int x, int y, int z, int a)
{
}

void bar1()
{
    foo1(1, 2, 3, 4);
}

void bar2()
{
    foo2(5, 6, 7, 8);
}
```

Конвенции вызова(fastcall)

```
g++ -c test.cpp -o test.o -O0 -m32

objdump -d test.o
```

Через стек 1 и 2 аргумент, остальные -- через регистр

```
000005c8 <_Z4bar1v>:
 5c8:   6a 04                   push   $0x4
 5ca:   6a 03                   push   $0x3
 5cc:   ba 02 00 00 00          mov    $0x2,%edx
 5d1:   b9 01 00 00 00          mov    $0x1,%ecx
 5d6:   e8 b5 ff ff ff          call   590 <_Z4foo1iiii>
 5dd:   c3                      ret
```

Конвенции вызова (cdecl)

```
g++ -c test.cpp -o test.o -O0 -m32

objdump -d test.o
```

Все 4 аргумента через стек

```
000005eb <_Z4bar2v>:
 5eb:    6a 08                   push   $0x8
 5ed:    6a 07                   push   $0x7
 5ef:    6a 06                   push   $0x6
 5f1:    6a 05                   push   $0x5
 5f3:    e8 b3 ff ff ff          call   5ab <_Z4foo2iiii>
 5fd:    c3                      ret
```

Inline functions

```
inline void foo()
{
}

// ms vc
__forceinline void foo()
{
}

// gcc
__attribute__((always_inline)) void foo()
{}
```

Inline functions

```
inline void foo()
{
}

// ms vc
__forceinline void foo()
{
}

// gcc
__attribute__((always_inline)) void foo()
{}
```

Ссылки

```
int a = 1;
int b = 2;
int* ptr = nullptr;
ptr = &a;
ptr = &b;
int& ref; // Ошибка
int& ref = a; // ref ссылается на a
ref = 5; // Теперь a == 5
ref = b; // ref не стала указывать на b,
         // мы просто скопировали значение из b в a
ref = 7; // a == 7, b == 2
```

```
int a = 2;
int* ptr = nullptr;
int*& ptrRef = ptr; // ptrRef ссылается на ptr
  ptrRef = &a; // теперь ptr хранит адрес a
```

Ссылки

```
int a = 1;
int b = 2;
int* ptr = nullptr;
ptr = &a;
ptr = &b;
int& ref; // Ошибка
int& ref = a; // ref ссылается на a
ref = 5; // Теперь a == 5
ref = b; // ref не стала указывать на b,
         // мы просто скопировали значение из b в a
ref = 7; // a == 7, b == 2
```

```
int a = 2;
int* ptr = nullptr;
int*& ptrRef = ptr; // ptrRef ссылается на ptr
  ptrRef = &a; // теперь ptr хранит адрес a
```