

Лекция № 1

Углубленное программирование на языке C / C++

Алексей Петров



Модуль №1. Углубленное программирование на языке C. Управление памятью

Лекция №1. Цели и задачи курса. Организация и использование оперативной памяти в программах на языке C

Практикум №1. Адресная арифметика. Одно- и многомерные массивы и строки. Алгоритмы их обработки

Лекция №2. Организация и использование сверхоперативной памяти. Основы многопоточного программирования. Вопросы качества структурного программного кода

Практикум №2. Составные типы языка C. Алгоритмы их обработки. Взаимодействие с ОС



Модуль №2. Объектная модель языка C++. Обобщенное и безопасное программирование

Лекция №3. Основные вопросы объектно-ориентированного программирования (ООП) на языке C++

Лекция №4. Дополнительные вопросы ООП на языке C++. Динамическая идентификация типов (RTTI)

Практикум №3. Проектирование полиморфной иерархии классов повышенного уровня сложности

Лекция №5. Шаблоны классов и методов. Обработка исключительных ситуаций. Обобщенное и безопасное программирование

Практикум №4. Разработка и обеспечение безопасности полиморфной иерархии с шаблонами классов



Модуль №3. Библиотеки для промышленной разработки ПО: STL, Boost

Лекция №6. Практическое введение в STL

Лекция №7. Функциональное программирование в C++11.
Практическое введение в Boost

Практикум №5. *Оптимизация полиморфной иерархии классов с использованием элементов библиотек STL и Boost*



Модуль №4. Шаблоны ОО-проектирования. Основы промышленной разработки ПО

Лекция №8. Принципы и шаблоны объектно-ориентированного проектирования. Базовые шаблоны, шаблоны GoF

Практикум №6. *Оптимизация полиморфной иерархии классов с использованием шаблонов объектно-ориентированного проектирования однопоточных приложений*

Лекция №9. Идиоматика C++. Основы рефакторинга и качество исходного кода. Стандарты кодирования и методологии разработки ПО

Практикум №7. *Инспекция и рефакторинг объектно-ориентированного исходного кода*

Лекция №1. Цели и задачи курса. Организация и использование ОЗУ и СОЗУ в программах на языке С



1. Цели, задачи, структура курса.
Язык С в современной промышленной разработке.
2. Организация оперативной памяти. Одно- и многомерные массивы, строки и указатели.
3. Выравнивание и упаковка переменных простых и составных типов.
4. Выделение и освобождение памяти, управление памятью и производительность кода.
5. Стандарт POSIX и переносимый исходный код.
6. Оптимизация работы с кэш-памятью ЦП ЭВМ.
7. **Постановка ИЗ к практикуму №1.**

Цель и структура курса



Цель курса — сформировать практические навыки и умения, необходимые специалистам по разработке ПО UNIX-подобных операционных систем (ОС) для участия в проектах **промышленной разработки** среднего уровня сложности, в том числе для замещения стажерских должностей разработчиков **серверной части высоконагруженных приложений**.

Состав курса (весна 2014 / 2015 уч. г.) — **9 лекций, 7 практикумов**.

Для сравнения:

- осень-весна 2013 / 2014 — 9 лекций, 7 практикумов;
- весна 2013 — 10 лекций, 6 практикумов;
- осень 2012 — 12 лекций, 4 практикума.

Общая аудиторная нагрузка — 64 акад. часа (лекции — 36 акад. часов, практика — 28 акад. часов).

Чему научимся?

Практический результат (1 / 2)



Обязательно:

- моделировать систему при помощи **UML-диаграмм**;
- разрабатывать код на языке C / C++ с элементами **C++11**;
- использовать инструменты анализа кода: **Valgrind, dwarves** и др.;
- создавать качественный код в **структурной и объектно-ориентированной парадигме**;
- использовать приемы **обобщенного и безопасного программирования**;
- применять промышленные библиотеки **STL, Boost**;
- внедрять в продукт классические архитектурные **шаблоны GoF**;
- оценивать **качество** и выполнять **рефакторинг** исходного программного кода;
- презентовать и защищать свои разработки перед аудиторией.

Чему научимся?

Практический результат (2 / 2)



По желанию:

- моделировать **варианты использования** продукта;
- проектировать и реализовывать слой данных продукта;
- выполнять **кодогенерацию** по UML-моделям;
- писать **многопоточные** приложения;
- создавать **POSIX**-совместимый **переносимый** исходный код;
- реализовывать графический интерфейс пользователя в **Qt**;
- использовать систему контроля версий: **Git** или аналогичную.

Организационные положения



Расписание занятий:

- постановка задач к практикумам — через блог дисциплины и на лекциях №№1, 2, 3, 4, 6, 8 и 9 (работа выполняется индивидуально и в группах!).

Правила поведения и регламент:

- приходить **вовремя**, самостоятельно проходить **электронную регистрацию**;
- общезначимые вопросы задавать **во время пауз** преподавателя или по поднятию руки, индивидуальные — **в перерыве** или после занятия;
- средства связи и ноутбуки использовать только в беззвучном режиме или режиме текстовых сообщений;
- продолжительность занятий — 4 акад. часа с 1 или 2 перерывами общей продолжительностью до 10 минут.

Знакомство с аудиторией:

- известные языки программирования (C, C++, Java);
- опыт разработки, известные среды и технологии.

Web-ресурсы и онлайн-книги



- Официальный Web-сайт проекта Boost: <http://www.boost.org/>.
- Официальный Web-сайт проекта Eclipse: <http://www.eclipse.org/>.
- Официальный Web-сайт проекта Qt: <http://qt-project.org/>.
- Справка по языкам C / C++: <http://en.cppreference.com/w/>.
- C Programming: http://en.wikibooks.org/wiki/C_Programming.
- Google C++ Style Guide: <http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml>
- More C++ Idioms:
http://en.wikibooks.org/wiki/More_C%2B%2B_Idioms.
- Schäling, B. *The Boost C++ Libraries*:
<http://en.highscore.de/cpp/boost/>.
- Stack Overflow — <http://stackoverflow.com/>.
- Standard C++ — <http://isocpp.org/>.
- The C++ Resources Network — <http://www.cplusplus.com/>.

Блог дисциплины



- **Размещен** по адресу: <https://tech-mail.ru/blog/cpp/>
- **Что делать:**
 - подписаться на обновления;
 - изучить более ранние записи;
 - задавать вопросы;
 - участвовать в опросах и обсуждениях.

Рекомендуемая литература:

модуль №1



- Керниган Б., Ритчи Д. Язык программирования С. — Вильямс, 2012. — 304 с.
- Прата С. Язык программирования С. Лекции и упражнения. — Вильямс, 2013. — 960 с.
- Шилдт Г. Полный справочник по С. — Вильямс, 2009. — 704 с.
- Butenhof, D. *Programming with POSIX Threads* (Addison-Wesley, 1997).
- Fog, A. Optimizing Software in C++: An Optimization Guide for Windows, Linux and Mac platforms (Oct. 2013). URL: http://www.agner.org/optimize/optimizing_cpp.pdf.
- Intel® 64 and IA-32 Architectures Optimization Reference Manual (July 2013). URL: <http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf>.
- King, K. C Programming: A Modern Approach, 2nd ed. (W. W. Norton & Co., 2008).
- Meyers, S. CPU Caches and Why You Care. URL: <http://aristeia.com/TalkNotes/PDXCodeCamp2010.pdf>.

Язык C сегодня



- **2015** — активное применение языка в практике программирования:

- ядра ОС:    

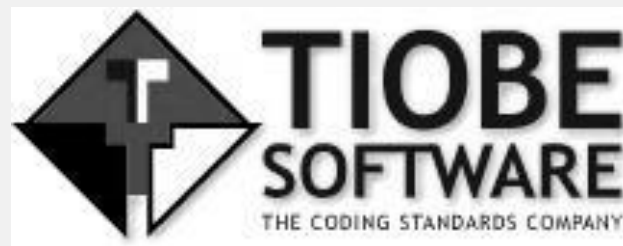
- инструментальные средства:    

- системы управления БД и Web-серверы и пр.



- проекты Mail.Ru Group: tarantool, Почта@Mail.Ru и пр.

- В феврале 2015 г. TIOBE Programming Community Index языка C составляет 16,488% (1-е место), а сам язык в нем занимает 1 – 2-е места с 1989 г. (конкурируя, главным образом, с Java).

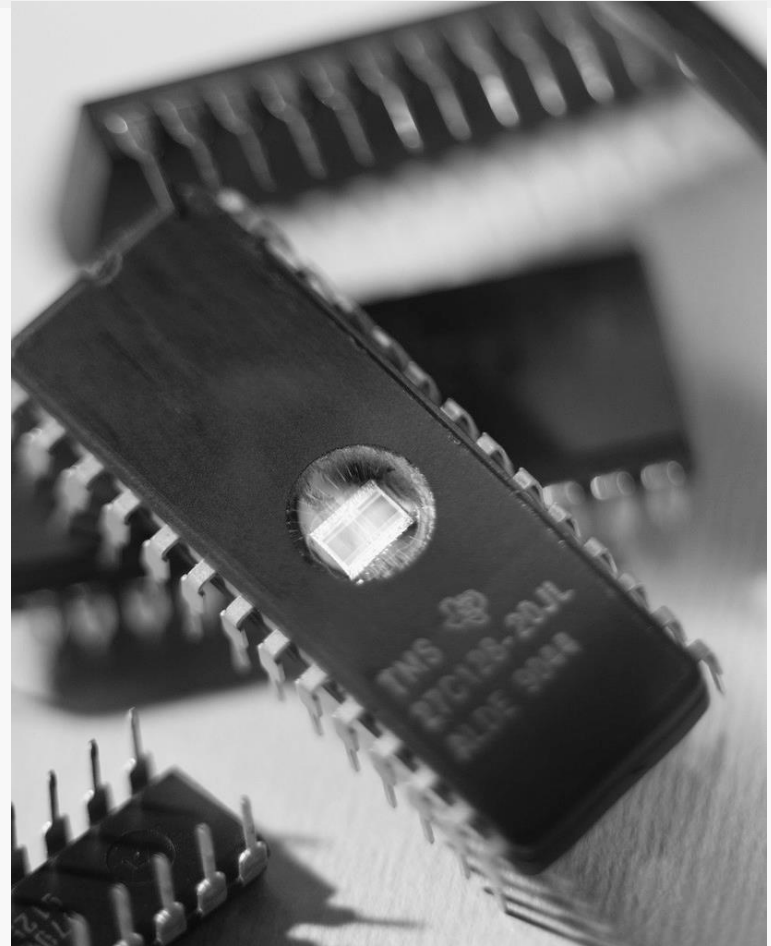


Вопросы управления памятью и производительность кода: зачем?



Неоптимальная работа с памятью становится **ограничивающим фактором** для большинства программ.

Проблему усугубляет **рост сложности подсистемы памяти**, в частности — механизмов кэширования и пр.



Модели управления памятью и области видимости объектов данных



- Предлагаемые языком С модели управления объектами данных (переменными) закреплены в понятии **класса памяти**, которое охватывает:
 - **время жизни** — продолжительность хранения объекта в памяти;
 - **область видимости** — части исходного кода программы, из которых можно получить доступ к объекту по идентификатору;
 - **связывание** — части исходного кода, способные обращаться к объекту по его имени.
- Для языка С характерны **три области видимости**:
 - **блок** — фрагмент кода, ограниченный фигурными скобками (напр. составной оператор), либо заголовок функции, либо заголовок оператора **for**, **while**, **do while** и **if**;
 - **прототип функции**;
 - **файл**.

Классы памяти в языке C



Класс памяти	Время жизни	Область видимости	Тип связывания	Точка определения
Автоматически ий	Автоматическое	Блок	Отсутствует	В пределах блока,
Регистровый	Автоматическое	Блок	Отсутствует	В пределах блока,
Статический, без связывания	Статическое	Блок	Отсутствует	В пределах блока,
Статические, с внешним связыванием	Статическое	Файл	Внешнее	Вне функций
Статические, с внутренним связыванием	Статическое	Файл	Внутреннее	Вне функций, <code>static</code>

Размещение объектов данных на регистрах процессора



- Применение ключевого слова **register** для активно используемых переменных:
 - несет все риски «ручной оптимизации» кода и полезно преимущественно для встроенных систем и архитектур, не имеющих компиляторов C с долгой историей (gcc разрабатывается с 1987 г.);
 - относится к регистрам ЦП (в x86/x86-64: AX, EBX, RCX и т.д.), но не кэш-памяти ЦП 1-го или 2-го уровня;
 - является **рекомендацией**, но **не требованием** к компилятору;
 - вполне может игнорироваться компилятором, который будет действовать «на свое усмотрение» (например, разместит переменную на регистре, потребность в котором возникнет позднее всего).
- **Операция взятия адреса переменной** со спецификатором **register недопустима** вне зависимости от того, размещена ли она фактически на регистре.

Операция `sizeof` и тип `size_t`



- Унарная операция `sizeof`:
 - допускает скобочную и бесскобочную (только для переменных) нотацию: `sizeof a` или `sizeof(T)`;
 - возвращает объем памяти, выделенной под объект простого или составного типа, в байтах как значение переносимого типа `size_t`, являющегося псевдонимом одного из базовых беззнаковых целых типов (ср. `int32_t` и пр.);
 - не учитывает возможного выравнивания объекта.
- Использование вычисляемых компилятором конструкций вида `sizeof(T)` не влияет на производительность кода, но **повышает переносимость**.

Указатели и арифметика указателей.

Тип `ptrdiff_t`



- Стандартные указатели типа T^* как составной тип языка C и символический способ использования адресов можно условно считать «шестым классом памяти», важной особенностью которого является поддержка специфической арифметики.
- Пусть p , $p2$ — указатели типа T^* , а n — значение целого типа (желательно — `ptrdiff_t`). Тогда:
 - $p + n$ либо $n + p$ — адрес, смещенный относительно p на n единиц хранения размера `sizeof(T)` в направлении увеличения адресов;
 - $p - n$ — адрес, смещенный относительно p на n единиц хранения размера `sizeof(T)` в направлении уменьшения адресов;
 - $p++$ либо $++p$, $p--$ либо $--p$ — аналогичны $p + 1$ и $p - 1$;
 - $p - p2$ — разность содержащихся в указателях адресов, выраженная в единицах хранения и имеющая тип `ptrdiff_t`. Разность положительна при условии, что p расположен в пространстве адресов «правее» $p2$.

Одномерные массивы (строки)



- Для одномерного массива **T a[N]** в языке C справедливо:
 - массивы поддерживают полную и частичную инициализацию, в том числе с помощью выделенных инициализаторов;
 - в частично инициализированных массивах опущенные значения трактуются как нули;
 - элементы массивов размещаются в памяти непрерывно и занимают смежные адреса, для обхода которых может использоваться арифметика указателей;
 - строки **char c[N]** конструктивно являются частными случаями массивов, при этом в корректных строках **c[sizeof(c) - 1] == '\0'**;
 - **sizeof(a)** возвращает размер массива в байтах (не элементах!);
 - **sizeof(a[0])** возвращает размер элемента в байтах.
- Принятая система обозначения массивов является лишь особым способом применения указателей.



Одномерные массивы (строки): пример



```
// с освобождением квадратных скобок
int a[] = {1, 2, 3};
// эквивалентно int a[3] = {1, 2, 3};

// с частичной неявной инициализацией
int b[5] = {1, 2, 3};
// эквивалентно:
// int b[5] = {1, 2, 3, 0, 0};

// с выделенными инициализаторами
int c[7] = {1, [5] = 10, 20, [1] = 2};
// эквивалентно:
// int c[7] = {1, 2, 0, 0, 0, 10, 20};
```

Одномерные массивы (строки) и указатели



- Пусть $T\ a[N]$ — массив. Тогда:
 - имя массива является **константным указателем на 0-й элемент**:
 $a == \&a[0];$
 - для любых типов и длин массивов **справедливо**:
 $\&a[i] == a + i$ и $a[i] == *(a + i);$
- С учетом этого эквивалентны прототипы:
 - $\text{int foo(double } [], \text{int});$
 - $\text{int foo(double } *, \text{int});$
- Передать массив в функцию можно так, как показано выше, или как **пару указателей: на 0-й и N-й элементы** (обращение к элементу $a[N]$ без его разыменования допустимо):
 - $\text{int foo(double } *, \text{double } *);$

Макроопределение NULL



- Стандартное макроопределение NULL расширяется препроцессором до константы с семантикой **«пустого» указателя**, который...
 - является константным целочисленным выражением, вычисляемым в длинный или короткий нуль (`0L` или `0`), либо
 - выступает как результат приведения такого значения к `void*` (напр. `(void*)0`).
- Значение NULL приводимо к любому типу-указателю и может использоваться в конструкциях вида:
 - `if (p != NULL) // ...`
 - `if (q == NULL) // ...`

Вопросы безопасного программирования



- **Инициализировать указатели** во время определения:
 - допустимый адрес;
 - 0, (void*)0 или NULL.
- **Проверять:**
 - значения указателей перед их разыменованием;
 - значения индексов элементов массивов перед использованием;
 - возвращаемые значения стандартных функций после их вызова.

Стандартные функции ввода-вывода



Имя функции	Назначение функции	Причины ошибок	POSIX-совместима?
int scanf(const char *restrict format, ...);			
scanf	Осуществляет форматный ввод с консоли — чтение из стандартного входного потока <code>stdin</code> . Возвращает количество успешно считанных элементов ввода	Некорректная входная последовательность (EILSEQ) Недостаточно аргументов (EINVAL)	Да
int printf(const char *restrict format, ...);			
printf	Осуществляет форматный вывод в консоль — запись в стандартный выходной поток <code>stdout</code> . Возвращает количество переданных в поток байт	EILSEQ, EINVAL и др.	Да

Стандартные функции для работы с динамической памятью (1 / 2)



Имя функции	Назначение функции	Причины ошибок	POSIX-совместима ?
void *malloc(size_t size);			
malloc	Выделяет неиспользуемый участок памяти объекту данных размера <code>size</code> байт, не меняя содержимое указанного участка	Недостаточно памяти (ENOMEM)	Да
void *calloc(size_t nelem, size_t elsize);			
calloc	Выделяет неиспользуемый участок памяти массиву из <code>nelem</code> элементов размера <code>elsize</code> байт каждый и выполняет его поразрядное обнуление	Недостаточно памяти (ENOMEM)	Да

Стандартные функции для работы с динамической памятью (2 / 2)



Имя функции	Назначение функции	Причины ошибок	POSIX-совместима ?
void *realloc(void *ptr, size_t size);			
realloc	Изменяет размер объекта данных, на который указывает ptr, до size. Если указатель ptr пуст, вызов эквивалентен malloc. Если size == 0, эквивалентен malloc. Если size == 0,	Недостаточно памяти (ENOMEM)	Да
void free(void *ptr);			
free	Вызывает освобождение памяти, на которую указывает ptr, делая ее доступной для нового выделения. Дальнейшее использование ptr влечет неопределенное поведение	Нет	Да

Выравнивание объектов, размещаемых статически. GCC-атрибут `aligned` (1 / 2)



- Одним из способов повышения производительности программы является такое размещение данных в ОЗУ, при котором они эффективно загружаются в кэш-память ЦП. Для этого данные должны быть, как минимум, **выровнены на границу линии кэш-памяти данных 1-го уровня (L1d)**. Выравнивание объекта в ОЗУ обычно определяется характеристиками выравнивания, которые имеет соответствующий тип данных. При этом:
 - выравнивание **скалярного объекта** определяется собственной характеристикой выравнивания приписанного ему базового типа;
 - выравнивание **массива**, — если размер каждого элемента не кратен величине выравнивания, — распространяет свое действие только на элемент с индексом 0;
 - выравнивание объектов в программе на языке C может регулироваться **на уровне отдельных переменных и типов данных.**, для чего в компиляторе GCC служит атрибут `aligned`.

Выравнивание объектов, размещаемых статически. GCC-атрибут `aligned` (2 / 2)



- Выравнивание статически размещаемых переменных имеет силу как для глобальных, так и для автоматических переменных. При этом характеристика выравнивания, присущая типу объекта, полностью игнорируется.
- При выравнивании массивов гарантированно выравнивается только начальный, нулевой элемент массива.



Выравнивание объектов, размещаемых статически. Атрибут `aligned`. пример



```
// выравнивание, регулируемое на уровне объекта  
// переменная qwd выравнивается на границу 64 байт  
uint64_t qwd __attribute__((aligned(64)));
```

```
// выравнивание, регулируемое на уровне типа  
// переменные типа al128int_t (синоним int)  
// выравниваются на границу 128 байт  
typedef int __attribute__((aligned(128))) al128int_t;  
al128int_t aln;
```

Выравнивание объектов, размещаемых динамически. Функция `posix_memalign`



- Функция **`posix_memalign`**:
 - определена в стандарте POSIX 1003.1d;
 - имеет прототип
`int posix_memalign(void **memptr, size_t alignment, size_t size);`
 - выделяет неиспользуемый участок памяти размера `size` байт, выровненный на границу `alignment`, и возвращает указатель на него в `memptr`;
 - допускает освобождение выделенной памяти функцией `free()`.
- Требования к значению **`alignment`**:
 - кратно `sizeof(void*)`;
 - является целочисленной степенью числа 2.
- Ошибки (**`EINVAL`**, **`ENOMEM`**):
 - значение `alignment` не является кратной `sizeof(void*)` степенью 2;
 - недостаточно памяти.



posix_memalign: пример (1 / 2)



```
int b[7] = {1, [5] = 10, 20, [1] = 2}; // массив-источник
int *p = NULL,                          // массив-приемник
    errflag;                            // код ошибки posix_memalign

// установить размер линии кэш-памяти данных 1-го уровня
// (L1d); типичное значение: 64 байта
long l1dcls = sysconf(_SC_LEVEL1_DCACHE_LINESIZE);
// проверить, удался ли вызов sysconf()
if (l1dcls == -1)
// если вызов sysconf() неудачен, использовать значение
// выравнивания по умолчанию
    l1dcls = sizeof(void*);
```



posix_memalign: пример (2 / 2)



```
// выделить память с выравниванием на границу строки L1d
errflag = posix_memalign((void**)&p, l1dcls, sizeof b);
if(!errflag)// в случае успеха posix_memalign возвращает 0
{
    printf("\nL1d cache line size is %ld\n", l1dcls);
    printf("p and &p are %p and %p\n", p, &p);
    p = memcpy(p, b, sizeof(b));
    // ...
    free(p);
}
else
    printf("posix_memalign error: %d\n", errflag);
```

Многомерные массивы

- **Двумерный массив** — объект данных $T\ a[N][M]$, который:
 - содержит N последовательно расположенных в памяти строк по M элементов типа T в каждой;
 - в общем и целом инициализируется аналогично одномерным массивам;
 - по характеристикам выравнивания идентичен объекту $T\ a[N * M]$, что сводит его двумерный характер к удобному умозрительному приему, упрощающему обсуждение и визуализацию порядка размещения данных.
- Массивы размерности больше двух считаются **многомерными**, при этом $(N + 1)$ -мерные массивы индуктивно определяются как линеаризованные массивы N -мерных массивов, для которых справедливо все сказанное об одно- и двумерных массивах.



Многомерные массивы: пример



// определение двумерных массивов

```
int a[2][3] = {  
    {0, 1},    // частичная инициализация строки  
    {2, 3, 4}}; // полная инициализация строки  
int b[2][3] = {0, 1, 2, 3, 4};
```

// результаты:

// a: {0, 1, 0, 2, 3, 4}; b: {0, 1, 2, 3, 4, 0}

// определение массивов размерности больше 2

```
double d[3][5][10];  
int32_t k[5][4][3][2];
```

Многомерные массивы и указатели



- Для многомерных массивов справедлив ряд тождеств, отражающих эквивалентность соответствующих выражений языка C. Так, для двумерного массива $T\ a[N][M]$ справедливо:
 - $a == \&a[0]; a + i == \&a[i];$
 - $*a = a[0] == \&a[0][0];$
 - $**a == *\&a[0][0] == a[0][0];$
 - $a[i][j] == (*(a + i) + j).$
- Использование операции разыменования $*$ не имеет каких-либо преимуществ перед доступом по индексу, и наоборот. Трансляция и первой, и второй формы записи в объектный код приводит в целом к одинаковым результатам.



Многомерные массивы и указатели: пример



// указатели на массивы и массивы указателей

int k[3][5];

int (*pk)[5]; // указатель на массив int[5]

int *p[5]; // массив указателей (int*)[5]

// примеры использования (все — допустимы)

pk = k; // аналогично: pk = &k[0];

pk[0][0] = 1; // аналогично: k[0][0] = 1;

***pk[0] = 2; // аналогично: k[0][0] = 2;**

****pk = 3; // аналогично: k[0][0] = 3;**

Совместимость указателей



- Общеизвестно, что для любого конкретного типа T , не тождественного ему типа Y и указателей T^*pt и Y^*py недопустимо присваивание:

```
pt = py; // T — не void
```

- Представленный пример является частным случаем более общего правила совместимости указателей, расширением которого является запрет на присваивание значения указателя на константу «обычному» указателю.



Совместимость указателей: пример



// определения

```
double *pd, **ppd;
```

```
double (*pda)[2];
```

```
double dbl32[3][2];
```

```
double dbl23[2][3];
```

// допустимые примеры использования

```
pd = &dbl32[0][0]; // double* -> double*
```

```
pd = dbl32[1];      // double[] -> double*
```

```
pda = dbl32;          // double(*)[2] -> double(*)[2]
```

```
ppd = &pd;            // double** -> double**
```

// недопустимые примеры использования

```
pd = dbl32;           // double[][] -> double*
```

```
pda = dbl23;          // double(*)[3] -> double(*)[2]
```


Указатели на константы и константные указатели



- Различное положение квалификатора `const` в определении указателя позволяет вводить в исходном коде программ четыре разновидности указателей:
 - «обычный» указатель (**вариант 1**) — изменяемый указатель на изменяемую через него область памяти;
 - указатель на константу (**вариант 2**) — изменяемый указатель на неизменяемую через него область памяти;
 - константный указатель (**вариант 3**) — неизменяемый указатель на изменяемую через него область памяти;
 - константный указатель на константу (**вариант 4**) — неизменяемый указатель на неизменяемую через него область памяти.



Указатели на константы и константные указатели: пример



// пример 1: определения

int *p1; // обычный указатель

const int *pc2; // указатель на константу

int *const cp3; // константный указатель

const int *const cpc4; // константный указатель на константу

// пример 2: совместимость T*, const T* и const T**

int *pi;

const int *pci;

const int **ppci;

pci = pi; // допустимо: int* -> const int*

pi = pci; // недопустимо: const int* -> int*

ppci = π // недопустимо: int -> const int****

Указатели и квалификатор `restrict`



- Использование ключевого слова `restrict` допустимо **строго в отношении указателей** и...
 - расширяет возможности компилятора по оптимизации некоторых видов кода путем поиска сокращенных методов вычислений;
 - означает, что указатель — **единственное** (других нет) **исходное** (первоначальное) **средство доступа** к соответствующему объекту.
- В отсутствие квалификатора `restrict` компилятор вынужден прибегать к «пессимистичной» стратегии оптимизации.
- Квалификатор `restrict` широко применяется в отношении формальных параметров функций стандартной библиотеки языка C. Например:
 - `void *memcpy(void *restrict s1, const void *restrict s2, size_t n);`
 - `int sscanf(const char *restrict s, const char *restrict format, ...);`



Указатели и квалификатор

restrict: пример



```
// restrict: оптимистичная стратегия

int *restrict a = (int*)malloc(N * sizeof(int));

a[k] *= 2;           // (1)

// здесь следует иной код без участия a[k]

a[k] *= 3;           // (2)

// строки (1) и (2) вкупе эквивалентны: a[k] *= 6;


// пессимистичная стратегия

double d[N];  // d – не единственное средство доступа

double *p = d; // p – не исходное средство доступа (см. d)

d[k] *= 2;      // (3)

// здесь следует иной код без участия d[k]

d[k] *= 3;      // (4)

// стр. (3) и (4) не могут быть объединены ввиду существ. p
```

Многомерные массивы и функции



- В общем случае для передачи функции двумерного массива
`T a[N][M]` используется формальный параметр `T **pa`. При этом информация о размерности массива закреплена в типе (`T**`), а размеры передаются как **дополнительные параметры**.
- Существование в языке объектов вида `T (*pt) [M]` открывает возможность **встраивания** информации о длине строк массива в тип данных и определения функций с прототипами вида:
 - `void foo(T a[][M], size_t N);`
 - `void foo(T (*a)[M], size_t N);` // комплект `[]` трактуется как указатель
- Для многомерных массивов аналогично имеем:
 - `void bar(T b[][SZ2][SZ3][SZ4], size_t sz1);`

Массивы переменной длины (1 / 2)



- **Массивы переменной длины** — введенное в стандарте C99 языковое средство **динамического** (а не традиционно статического) распределения локальной памяти функций. Размеры таких массивов по каждому измерению остаются **неизвестными** до момента исполнения кода функции:
 - концепция массивов переменной длины (VLA, variable-length array) — ответ разработчиков C99 на требование научного сообщества обеспечить переносимость на язык C наработанной десятилетиями базы на языке FORTRAN, более гибком в части работы функций с массивами.
- Размер массива переменной длины:
 - **определяется переменными** (отсюда название);
 - **остается постоянным** до уничтожения объекта.

Массивы переменной длины (2 / 2)



- С учетом ограничений C99 массивы переменной длины:
 - предполагают использование класса автоматической памяти;
 - определяются внутри блоков и прототипов функций;
 - не допускают инициализации при создании.
- Идентификатор массива переменной длины, как и традиционного массива, является **указателем**.
- Использование массива переменной длины как формального параметра функции означает передачу данных **по указателю**.



Массивы переменной длины: пример



```
// прототип функции
int foo(size_t rows, size_t cols, double a[rows][cols]);
int foo(size_t, size_t, double a[*][*]); // имена опущены
// определение функции
int foo(size_t rows, size_t cols, double a[rows][cols])
{ /* ... */ }

int bar()
{
    int n = 3, m = 4;
    int var[n][m];    // массив как локальная переменная
    // ...
}
```


Упаковка переменных составных типов (1 / 2)



- Для структур данных актуален также не характерный для массивов и скаляров вопрос **упаковки данных**, обусловленный наличием у элементов структур индивидуальных характеристик выравнивания.
- **Проблема упаковки структур** заключается в том, что смежные (перечисленные подряд) элементы часто физически не «примыкают» друг к другу в памяти.

Упаковка переменных составных типов (2 / 2)



- Например (для x86):

```
typedef struct {  
    int      id;           // 4 байта  
    char     name[15];     // 15 байт  
    double   amount;       // 8 байт  
    _Bool    active;       // 1 байт  
} account;                // 28 байт (не 32 байта!)
```

- Наличие лакун, аналогичных выявленным 4-байтовым (13%) потерям в структуре `account`, вызывается совокупностью факторов:
 - архитектура процессора (напр., x86 или x86-64);
 - оптимизирующие действия компилятора;
 - выбранный программистом порядок следования элементов.

Реорганизация структур данных: рекомендации



- Реорганизация структур для повышения эффективности использования кэш-памяти должна идти **по 2 направлениям**:
 - декомпозиция тяжеловесных («божественных») структур на более мелкие, узкоспециализированные структуры, которые при решении конкретной задачи используются полностью либо не используются вообще;
 - устранение в структурах лакун, обусловленных характеристиками выравнивания типов их элементов ([см. ранее](#)).
- При прочих равных условиях крайне желательно:
 - переносить наиболее востребованные элементы структуры к ее началу ([при загрузке в кэш-память такие элементы структуры могут становиться «критическими словами», доступ к которым должен быть самым быстрым](#));
 - обходить структуру в порядке определения элементов, если



Реорганизация структур данных: рекомендации: пример



```
typedef struct { // вар. 1: 28/32 байт (x86: 13% потерь)
    int      id;           // 4 байта
    char     name[15];     // 15 байт
    /* лакуна — 1 байт выравнивания */
    double   amount;       // 8 байт
    _Bool    active;       // 1 байт
    /* лакуна — 3 байта выравнивания */
} account_1;             // 32 байта

typedef struct { // вар. 2: 28/28 байт (x86: 0% потерь)
    int      id;           // 4 байта
    char     name[15];     // 15 байт
    _Bool    active;       // 1 байт
    double   amount;       // 8 байт
} account_2;             // 28 байт
```

Реорганизация структур данных: недостатки и альтернатива решения

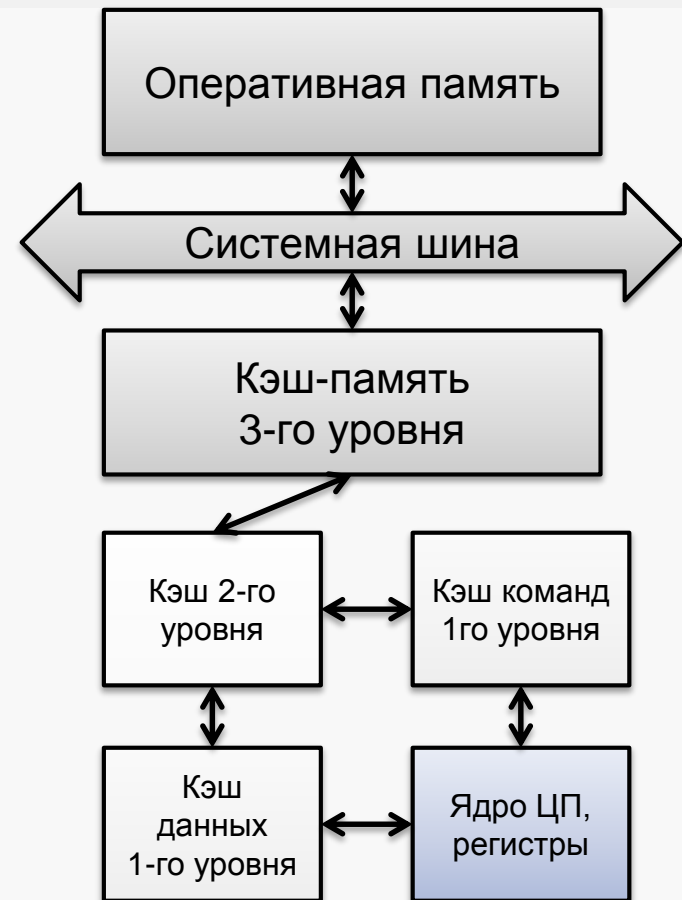


- Недостатки реорганизации:
 - снижение удобства чтения и сопровождения исходного кода;
 - риск размещения совместно используемых элементов (**напр., длины вектора и адреса его начального элемента**) на разных линиях кэш-памяти.
- Основная альтернатива реорганизации — замена стихийно выбранных типов данных наиболее адекватными по размеру, вплоть до использования битовых полей данных.

Кэш-память в архитектуре современных ЭВМ



- **Проблема** — отставание системной шины [и модулей оперативной памяти (DRAM)] от ядра ЦП по внутренней частоте; простой ЦП.
- **Решение** — включение в архитектуру небольших модулей сверхоперативной памяти (SRAM), полностью контролируемой ЦП.
- **Условия эффективности** — локальность данных и кода в пространстве-времени.



Кэш-память в архитектуре современных ЭВМ: что делать?



- Обеспечивать **локальность данных и команд** в пространстве и времени:
 - совместно хранить совместно используемые данные или команды;
 - не нарушать эмпирические правила написания эффективного кода.
- Обеспечивать **эффективность загрузки** общей (L2, L3) и отдельной кэш-памяти данных (L1d) и команд (L1i):
 - полагаться на оптимизирующие возможности компилятора;
 - помогать компилятору в процессе написания кода.
- Знать **основы организации** аппаратного обеспечения.
- Экспериментировать!

Эффективный обход двумерных массивов



- Простейшим способом повышения эффективности работы с двумерным массивом является **отказ от его обхода по столбцам в пользу обхода по строкам**:
 - для массивов, объем которых превышает размер (выделенной процессу) кэш-памяти данных самого верхнего уровня (напр., L2d), время инициализации по строкам приблизительно втрое меньше времени инициализации по столбцам вне зависимости от того, ведется ли запись в кэш-память или в оперативную память в обход нее (У. Дреппер, 2007).
- Дальнейшая оптимизация может быть связана с анализом и переработкой решаемой задачи в целях снижения частоты кэш-промахов или использования векторных инструкций процессора (SIMD — Single Instruction, Multiple Data).

Эффективный обход двумерных массивов: постановка задачи



- Задача.** Рассчитать сумму столбцов заданной целочисленной матрицы. Оптимизировать найденное решение с точки зрения загрузки кэш-памяти данных.

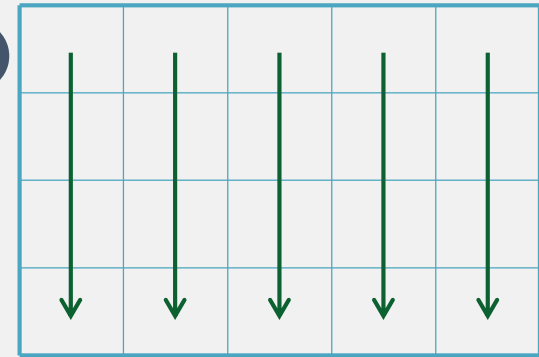
Дано:

$$A = (a_{ij})$$

Найти:

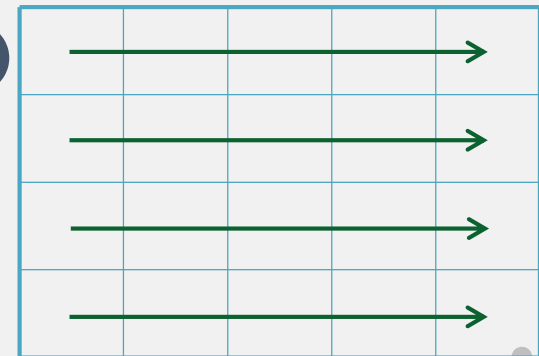
$$B = (b_j)$$

1



или

2



Эффективный обход двумерных массивов: анализ вариантов

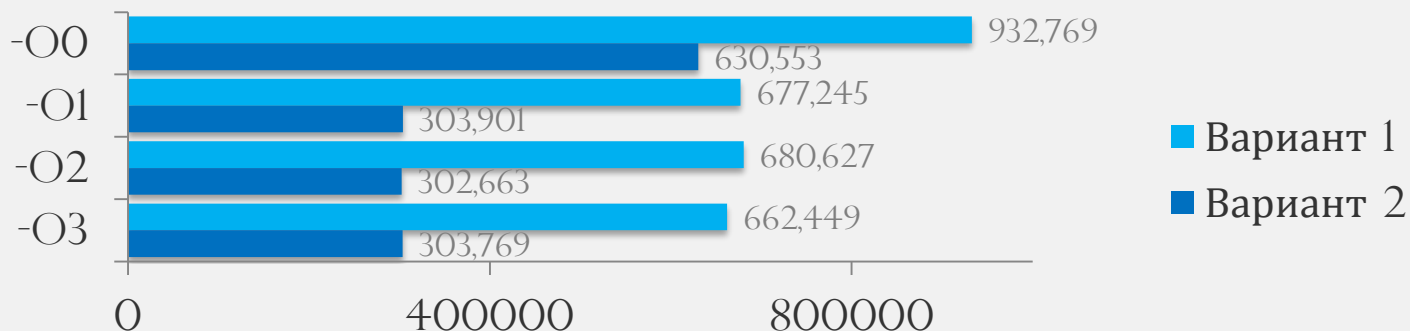


- **Размерность задачи:**
 - **набор данных:** $2^8 \times 2^8$ (2^{16} элементов, 2^{18} байт);
 - **количество тестов:** 100;
 - **выбираемое время:** минимальное.
- **Инфраструктура тестирования:**
 - **x86:** Intel® Core™ i5 460M, 2533 МГц, L1d: 2 x 32 Кб;
 - **x86-64:** AMD® E-450, 1650 МГц, L1d: 2 x 32 Кб / ядро;
 - **ОС:** Ubuntu Linux 12.04 LTS; **компилятор:** GCC 4.8.x.
- **Порядок обеспечения независимости тестов —**
2-фазная программная инвалидация памяти L1d:
 - последовательная запись массива (10^6 элементов, $\approx 2^{22}$ байт);
 - рандомизированная модификация элементов.

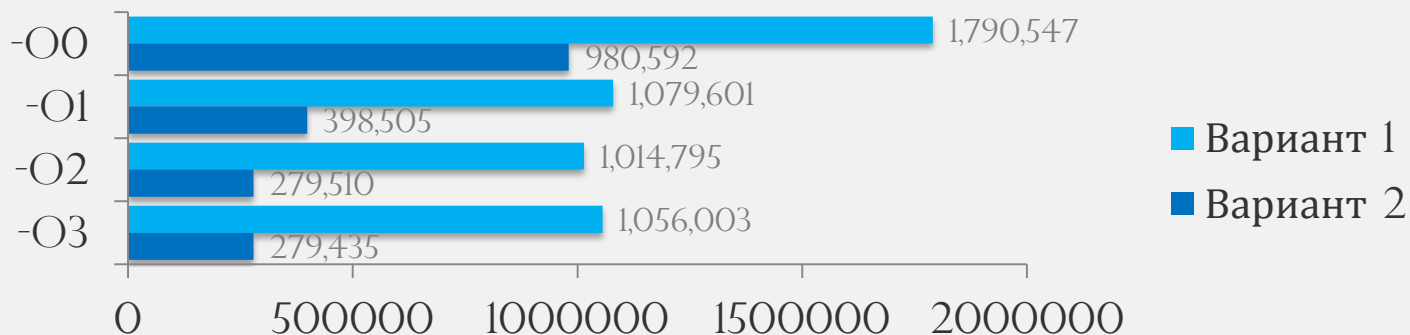
Эффективный обход двумерных массивов: сравнительная эффективность



Эффективность вариантов, такты ЦП INTEL X86



Эффективность вариантов, такты ЦП AMD X86-64



Эффективный обход двумерных массивов: результаты оптимизации



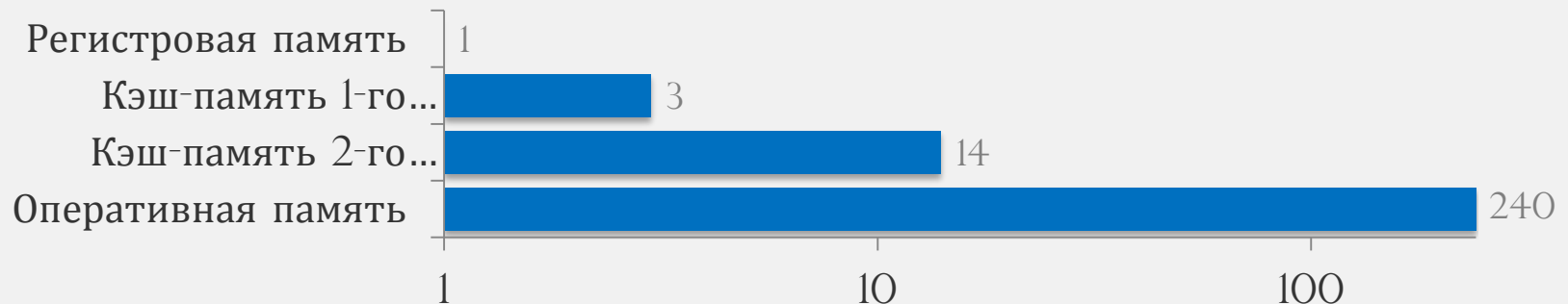
- Время решения задачи за счет оптимизации обхода данных (без применения SSE-расширений) **снижается в 1,8 – 2,4 раза**:
 - для ЦП **INTEL X86**: от 1,5 до 2,2 раза;
 - для ЦП **AMD X86-64**: от 1,8 до 3,8 раза.
- При компиляции с флагами **-O0**, **-O1** результат характеризуется **высокой повторяемостью** на ЦП с выбранной архитектурой (INTEL X86 / AMD X86-64):
 - относительный рост эффективности колеблется в пределах **20 % – 25 %**.
- Применение векторных SIMD-инструкций из наборов команд SSE, SSE2, SSE3 позволяет улучшить полученный результат еще на **15 % – 20 %**.

Почему оптимизация работы с кэш-памятью того стоит?



- Несложная трансформация вычислительноемких фрагментов кода позволяет добиться **серьезного роста** скорости выполнения:
 - разбиение на квадраты (SQUARE BLOCKING) и пр.
- Причина — высокая «стоимость» кэш-промахов:
 - на рис. — оценки для одного из ЦП INTEL.

Время доступа, такты ЦП



Что еще можно оптимизировать?



- **Предсказание переходов:**
 - устранение ветвлений;
 - развертывание (линеаризация) циклов;
 - встраивание функций (методов) и др.
- **Критические секции:**
 - устранение цепочек зависимости для внеочередного исполнения инструкций;
 - использование поразрядных операций, INC (++), DEC (--), векторизация (SIMD) и т.д.
- **Обращение к памяти:**
 - выполнение потоковых операций;
 - выравнивание и упаковка данных и пр.

Домашнее задание № 1



- Решить индивидуальную задачу №1 в соответствии с формальными требованиями.
- Для этого:
 - выполнить действия, указанные в письме-приглашении в систему автоматизированного тестирования задач (АСТС);
 - авторизоваться в АСТС и узнать в ней постановку задач.

Срок сдачи

Весь семестр



**Спасибо за
внимание!**

Алексей Петров

Приложения



Приложение А. Двумерные массивы и векторы векторов





Двумерные массивы и векторы

векторов: пример



```
// создание вектора векторов
int **v = (int**)malloc(N * sizeof(int*));
for(int i = 0; i < N; i++)
    // NB: в каждой строке значение M может быть разным
    v[i] = (int*)malloc(M * sizeof(int));

// ...

// удаление вектора векторов
for(int i = 0; i < N; i++)
    free(v[i]);
free(v);
```

Приложение Б. Выравнивание переменных составных типов



- Объекты составных типов языка C выравниваются в соответствии с характеристикой, **наибольшей** среди характеристик выравнивания всех своих элементов, которая в большинстве случаев не достигает длины линии кэш-памяти.
- Другими словами, даже при «подгоне» элементов структуры под линию L1D размещенный в ОЗУ объект может не обладать требуемым выравниванием.
- Принудительное выравнивание статически и динамически размещаемых структур выполняется аналогично выравниванию скалярных переменных и начальных элементов массивов:
 - GCC-атрибут `ALIGNED` в определении типа или объекта данных;
 - функция `POSIX_MEMALIGN`.

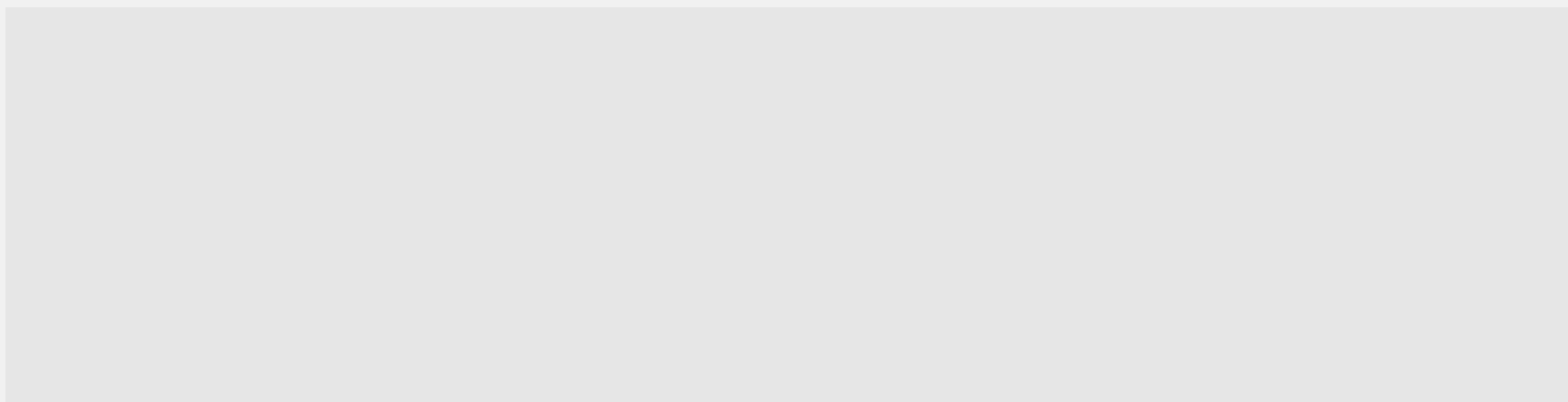
Приложение В. Утилита pahole



- Самый доступный способ изучить физическое размещение элементов структуры в оперативной памяти и сопоставить их с загрузкой линий кэш-памяти L1D — утилита **PAHOLE**, входящая в пакет **DWARVES**.
- Использование утилиты PAHOLE позволяет:
 - **проанализировать размещение** элементов структур относительно линий кэш-памяти данных;
 - **получить варианты** реорганизации структуры (**параметр `-REORGANIZE`**).

Приложение Г.

Задача об умножении матриц (1 / 2)



Задача об умножении матриц (2 / 2)



Приложение Д. Эффективный обход массивов: эффективность оптимизации



Эффективность оптимизации

