ОТВЕТЫ НА ВОПРОСЫ ЭКЗАМЕНА 2

Условные обозначения:

⚠️ - информация неточная, нужно проверить
✅ - сделано
🅿️ - нужны примеры
⚛️ - в целом норм, но непонятно достаточно написано или нет
🆘 - ничего не понятно, но очень интересно

Содержание:

- ✅ Алгоритм работы утилиты make на примере простого сценария сборки.
- ✅ Ключи запуска утилиты make.
- ✅ Использование переменных. Примеры использования.
- ✅ Неявные правила и переменные.
- ✅ Автоматические переменные и их использование.
- ✅ Шаблонные правила. Примеры использование.
- ✅ Условные конструкции в сценарии сборки. Примеры использования.
- ✅ Переменные, зависящие от цели. Примеры использования.
- ✅ Автоматическая генерация зависимостей.

5. Динамические матрицы (32-36)
   - ✅ Представление динамической матрицы с помощью одномерного массива. Преимущества и недостатки.
   - ✅ Представление динамической матрицы с помощью массива указателей на строки/столбцы. Преимуществ и недостатки.
   - ✅ Объединенный подход для представления динамической матрицы (отдельное выделение памяти под массив указателей и массив данных). Преимуществ и недостатки.
   - ✅ Объединенный подход для представления динамической матрицы (массив указателей и массив данных располагаются в одной области). Преимуществ и недостатки.
   - ✅ Необходимо реализовать функцию, которая может обрабатывать как статические, так и динамические матрицы. Какими способами это можно сделать? *(меня пугает множественное число)*

6. Непонятная тема (37-47)
   - Умение читать сложные объявления и использовать это на практике.
   - Функции, возвращающие динамическую строку: strdup/strndup, getline, snprintf/asprintf.
   - Feature Test Macro.
   - Функции memcpy, memmove, memcmp, memset.
   - Структуры с полями указателями и особенности их использования.
   - «Поверхностное» копирование vs «глубокое» копирование.
   - «Рекурсивное» освобождение памяти для структур с динамическими полями.
   - Структуры переменного размера. Приведите примеры.
   - Что такое «flexible array member»? Какие особенности использование есть у этих полей? Для чего они нужны? Приведите примеры.
   - Flexible array member до C99.
   - Flexible array member vs поле-указатель.

7. Динамически расширяемый массив (48-57)
   - Дайте определение массива.
   - Описание типа.
   - Удаление элемента
   - Особенности использования

- Почему при добавлении нового элемента память необходимо выделять блоками, а не под один элемент?

8. Односвязный список (57-64)
    - Дайте определение линейного односвязного списка.
    - Описание типа.
    - Добавление нового элемента в начало/конец списка.
    - Вставка элемента перед/после указанного.
    - Удаление элемента из списка.
    - Обход списка
    - Удаление памяти из-под всего списка
    - Возможные улучшения "классической"реализации
    - Сравните массив и линейный односвязный список.

9. Двоичное дерево поиска (65-71)
    - Описание типа.
    - Добавление элемента.
    - Поиск элемента (рекурсивный и нерекурсивный варианты)
    - Обход дерева.
    - Освобождение памяти из-под всего дерева.
    - Язык DOT, примеры использования. Утилита GraphViz.

10. Области видимости (72-89)
    - Что такое область видимости имени?
    - Какие области видимости есть в языке Си? Приведите примеры.
    - Какие правила перекрытия областей видимости есть в языке Си? Приведите примеры.
    - Что такое блок?
    - Какие виды блоков есть в языке Си?
    - Что такое объявление? Приведите примеры.
    - Что такое определение? Приведите примеры.
    - Для чего нужны объявления?
    - Чем отличаются определения и объявления?
    - Что такое время жизни программного объекта?
    - Какие виды времени жизни есть у переменных?
    - Какие виды времени жизни есть у функций?
    - Как время жизни влияет на область памяти, в которой располагается программный объект?
    - Что такое связывание?
    - Какие виды связывания есть в языке Си?
    - Как связывание влияет на "свойства"объектного/исполняемого файла? Что это за "свойства"?
    - Какими характеристиками (область видимости, время жизни, связывание) обладает переменная в зависимости от места своего определения?
    - Какими характеристиками (область видимости, время жизни, связывание) обладает функция в зависимости от места своего определения?

11. Классы памяти (90-107)
    - Какие классы памяти есть в языке Си?
    - Для чего нужны классы памяти?
    - Какие классы памяти можно использовать с переменными? С функциями?
    - Сколько классов памяти может быть у переменной? У функции?
    - Какие классы памяти по умолчанию есть у переменной? У функции?
    - Расскажите о классе памяти auto.
    - Расскажите о классе памяти static.
    - Расскажите о классе памяти extern.
    - Расскажите о классе памяти register
    - Для чего используется ключевое слово extern?
    - Особенности совместного использования ключевых слов static и extern.
    - Как описать автоматическую глобальную переменную?
    - Какая переменная называется глобальной?
    - Какая переменная называется локальной?
    - Каким значением по умолчанию инициализируются автоматические переменные?
    - Каким значением по умолчанию инициализируются переменные с глобальным временем жизни?
    - Какими недостатки есть у использования глобальных переменных?

12. Объектный файл(107-114)
    - Объектный файл, секции, таблица символов
    - Что делает компоновщик?
    - Журналирование, подходы к реализации.
    - Процесс запуска программы («превращения в процесс»).
    - Абстрактное адресное пространство программы.
    - Опишите достоинства и недостатки локальных переменных.
    - Локальные переменные создаются в так называемой «автоматической памяти». Почему эта память так называется?

13. Стек и куча (114-125)
    - Для чего в программе используется аппаратный стек?
    - Что такое кадр стека?
    - Для чего в программе используется кадр стека? Приведите примеры.
    - Какие преимущества и недостатки есть у использования кадра стека?
    - Что такое соглашение о вызове?
    - Какое соглашение о вызове используется в языке Си? В чем оно заключается?
    - Что такое переполнение буфера? Чем оно опасно?
    - Почему нельзя из функции возвращать указатель на локальную переменную, определенную в этой функции?
    - Для чего в программе используется куча?
    - Происхождение термина «куча».
    - Свойства области памяти, которая выделяется динамически.
    - Как организована куча?

14. Выделение динамической памяти(126-132)
    - Алгоритм работы функции malloc.
    - Алгоритм работы функции free.
    - Какие гарантии относительно выделенного блока памяти даются программисту?
    - Что значит "освободить блок памяти"с точки зрения функции free?
    - Преимущества и недостатки использования динамической памяти.
    - Что такое фрагментация памяти?
    - Выравнивание блока памяти, выделенного динамически.

15. Массивы переменной длины(132-140)
    - Что такое variable length array?
    - Чем отличается статический массив от variable length array?
    - Какую операцию языка Си пришлось реализовывать по-другому (не как для встроенных типов) специально для variable length array?
    - Особенности использования variable length array.
    - Справедлива ли для variable length array адресная арифметика?
    - Как вы думаете почему variable length array нельзя инициализировать?
    - Для чего используется variable length array? Приведите примеры.
    - В какой области и «кем» выделяется память под массив переменной длины?

16. alloca (141-142)
    - Функция alloca.
    - alloca vs VLA.

17. Функции с переменным числом параметров (143-150)
    - Можно ли реализовать в языке Си функцию со следующим прототипом int f(. . .) ? Почему?
    - Покажите идею реализации функций с переменным числом параметров.
    - Почему для реализации функций с переменным числом параметров нужно использовать возможности стандартной библиотеки?
    - Опишите подход к реализации функций с переменным числом параметров с использованием стандартной библиотеки. Какой заголовочный файл стандартной библиотеки нужно использовать? Какие типы и макросы из этого файла вам понадобятся? Для чего?
    - Какая особенность языка Си упрощает реализацию функций (с точки зрения компилятора) с переменным числом параметров?
    - Почему при вызове va_arg(argp, short int) (или va_arg(argp, float)) выдается предупреждение?
    - Какая "опасность"существует при использовании функций с переменным числом параметров?
    - Как написать функцию, которая получает строку форматирования и переменное число параметров (как функция printf), и передает эти данные функции printf? (Подсказка: см. последний вариант реализации журналирования.)

18. Препроцессор (151-169, 173, 174)
    - Что делает препроцессор? В какой момент в процессе получения исполняемого файла вызывается препроцессор?
    - На какие группы можно разделить директивы препроцессора?
    - Какие правила справедливы для всех директив препроцессора?
    - Что такое простой макрос? Как такой макрос обрабатывается препроцессором? Приведите примеры.
    - Для чего используются простые макросы?
    - Что такое макрос с параметрами? Как такой макрос обрабатывается препроцессором? Приведите примеры.
    - Макросы с параметрами vs функции: преимущества и недостатки
    - Макросы с переменным числом параметров. Приведите примеры
    - Какими общими особенностями/свойствами обладают все макросы?
    - Объясните правила использования скобок внутри макросов. Приведите примеры.
    - Какие подходы к написанию "длинных"макросов вы знаете? Опишите их преимущества и недостатки. Приведите примеры.
    - Какие предопределенные макросы вы знаете? Для чего эти макросы могут использоваться?
    - Для чего используется условная компиляция? Приведите примеры.
    - Директива #if vs директива #ifdef.
    - Операция #. Примеры использования
    - Операция ##. Примеры использования.
    - Особенности использование операций
    - Директива #error. Примеры использования
    - Директива #pragma (на примере once и pack). Примеры использования.
    - В чем разница между использованием <> и "" в директиве include?
    - Можно ли операцию sizeof использовать в директивах препроцессора? Почему?

19. inline функции (170-172)
    - Ключевое слово inline.
    - Назовите основную причину, по которой ключевое слово inline было добавлено в язык Си.
    - Подходы к реализации ключевого слова inline компилятором. Проанализируйте их недостатки.

20. Библиотеки (175-191)
    - Что такое библиотека?
    - Какие функции обычно выносят в библиотеку?
    - В каком виде распространяются библиотеки? Что обычно входит в их состав?
    - Какие виды библиотек вы знаете?
    - Преимущества и недостатки, которые есть у статических/динамических библиотек.
    - Как собрать статическую библиотеку?

- Нужно ли "оформлять"каким-то специальным образом функции, которые входят в состав статической библиотеки?
- Как собрать приложение, которое использует статическую библиотеку?
- Нужно ли "оформлять"каким-то специальным образом исходный код приложения, которое использует статическую библиотеку?
- Как собрать динамическую библиотеку (Windows/Linux)?
- Нужно ли "оформлять"каким-то специальным образом функции, которые входят в состав динамической библиотеки (Windows/Linux)?
- Какие способы компоновки приложения с динамической библиотекой вы знаете? Назовите их преимущества и недостатки.
- Что такое динамическая компоновка?
- Что такое динамическая загрузка (Windows/Linux)?
- Нужно ли "оформлять"каким-то специальным образом исходный код приложения, которое использует динамическую библиотеку (Windows/Linux)?
- Особенности реализации функций, использующих динамическое выделение памяти, в динамических библиотеках.
- Ключи -I, -l, -L компилятора gcc.

21. Python + Си (192-194)
    - Проблемы использования динамической библиотеки, реализованной на одном языке программирования, и приложения, реализованного на другом языке программирования.
    - Модуль ctypes. Загрузка библиотеки. Представление стандартных типов языка Си. Импорт функций из библиотеки. Проблемы, которые при этом возникают.
    - Написание модуля расширения для Python (основные шаги).

22. Побочные эффекты (195-201)
    - Что такое побочный эффект?
    - Какие выражения стандарт c99 относит к выражениям с побочным эффектом?
    - Почему порядок вычисления подвыражений в языке Си неопределен?
    - Порядок вычисления каких выражения в языке Си определен?
    - Что такое точка следования?
    - Какие точки следования выделяет стандарт c99?
    - Почему необходимо избегать выражений, которые дают разный результат в зависимости от порядка их вычисления?

23. Неопределенное поведение (202-208)
    - Какие виды "неопределенного"поведения есть в языке Си?
    - Почему "неопределенное"поведение присутствует в языке Си
    - Какой из видов "неопределенного"поведения является самым опасным? Чем он опасен?
    - Как бороться с неопределенным поведением?
    - Приведите примеры неопределенного поведения.
    - Приведите примеры поведения, зависящего от реализации.
    - Приведите примеры неспецифицированного поведения.

24. Модуль (209-212, 217)
    - Что такое модуль?
    - Из каких частей состоит модуль? Какие требования предъявляются к этим частям?
    - Назовите преимущества модульной организации программы. Приведите примеры.
    - Какие виды модулей вы знаете? Приведите примеры.
    - Средства реализации модулей в языке Си.

25. Абстрактный тип данных (213-223)
    - Что такое тип данных?
    - Что такое абстрактный тип данных?
    - Какие требования выдвигаются к абстрактному типу данных?
    - Абстрактный объект vs абстрактный тип данных.
    - Что такое неполный тип данных в языке Си?
    - Приведите примеры описания неполного типа данных? (А кроме структур ;) ?)
    - Какие действия можно выполнять с неполным типом данных?
    - Для чего при реализации абстрактного типа данных используется неполный тип данных языка Си?
    - Проблемы реализации АТД на языке Си.
    - Есть ли в стандартной библиотеке языка Си примеры абстрактных типов данных?

1. (1-2) Указатель на void

   Обобщенный указатель (generic pointer) используется, если тип объекта не известен.
   - полезен для ссылки на произвольный участок памяти, независимо от размещенных там объектов;
     ```
     void *memcpy (void *dst, const void *src, size_t n);
     ```
   - позволяет передавать в функцию указатель на объект любого типа. (реализация обобщенной функции, например qsort)
     ```
     void qsort (void * first, size_t number, size_t size, int (* comparator)(const void *, const void *));
     ```

   Например, мы реализуем функцию, которая копирует одну область памяти в другую. В этом случае нас интересует только где эти области располагаются и размер данных.
   Также указатель на void используется как возвращаемое значение для функций выделения памяти, так как этим функциям не важно, что будет хранится в выделенной области.

   Особенности использования:
   - В языке С допускается присваивание указателя типа void указателю любого другого типа (и

```
double d = 5.0;
double *pd = &d;
void *pv = pd;

pd = pv;
```

наоборот) без явного преобразования типа указателя.

(как видим, указатель на void может стоять как слева, так и справа от операции присваивания)

- Указатель типа void нельзя разыменовывать (т.к. неизвестен размер того типа, на который указатель указывает)

```
/*
// error: dereferencing 'void *' pointer
// error: invalid use of void expression
printf("%d\n", *pv);
*/
```

- К указателям типа void неприменима адресная арифметика

```
/*
// error: dereferencing 'void *' pointer
pv++;
*/
```

(НО у компилятора gcc есть расширение, из-за которого он может работать с указателем на void, как с указателем на char, вследствие чего он может пропустить адресную арифметику с void. Чтобы этого избежать используется ключ pedantic)

2. (3-10) Функции выделения и освобождения памяти

Основной недостаток статического массива в том, что размер такого массива должен быть известен на этапе компиляции. Логично, что нам было бы удобнее “создавать” переменные в процессе работы программы.

Для выделения памяти в языке си необходимо вызвать одну из трех функций (C99 7.20.3):

- malloc (выделяет блок памяти и не инициализирует его)
- calloc (выделяет блок памяти и инициализирует его нулями)
- realloc (перевыделяет уже выделенный ранее блок памяти)

Все эти функции находятся в заголовочном файле stdlib.h.

Что общего у этих функций (особенности):

- Все функции не создают переменную, они только выделяют область памяти. В качестве результата возвращается адрес расположения этой области в памяти компьютера, т.е. указатель.

(Переменная = имя + тип + значение + адрес. Здесь имени нет)

- Все функции возвращают указатель на void (т.к. они не знают, что будет хранится в выделенное области)
- Все функции возвращают NULL, если выделить запрашиваемый блок памяти не удалось.
- После использования блока памяти он должен быть освобожден. Это делается с помощью функции free.

1. **void* malloc(size_t size); (C99 7.20.3.3)**
   - Выделяет блок памяти указанного размера size(байты)
   - Выделенный блок памяти не инициализируется (т.е. содержит мусор)
   - Для вычисления размера требуемой области необходимо использовать операцию sizeof.

```
int *a = NULL;
int n = 5;

// Выделение памяти
a = malloc(n * sizeof(int));
// Проверка успешности выделения
if (a == NULL)
{
    return ...
}

// Использование памяти
for (int i = 0; i < n; i++)
    a[i] = i;

// Освобождение памяти
free(a);
```

2. **void* calloc(size_t nmemb, size_t size); (C99 7.20.3.1)**
   - Выделяет блок памяти для массива из nmemb элементов, каждый из который имеет размер size.
   - Выделенная область памяти инициализируется таким образом, чтобы каждый бит имел значение 0.

     (НО не для всех типов это означает, что соответствующая переменная получит такое значение. Пример Лом не привел)

```
int *a;
int n = 5;

// Выделение памяти
a = calloc(n, sizeof(int));
// Проверка успешности выделения
if (a == NULL)
{
    return ...
}

// Использование памяти
for (int i = 0; i < n; i++)
    printf("%d ", a[i]);

// Освобождение памяти
free(a);
```

3. **void free(void *ptr); (C99 7.20.3.2, stdlib.h)**

- Освобождает (делает возможным повторное использование) ранее выделенный блок памяти, на который указывает ptr.
- Если значением ptr является нулевой указатель, то ничего не происходит.
- Если значением ptr является указатель, который не был получен с помощью одной из трех функций malloc, calloc или realloc, то поведение функции неопределено.

4. **void* realloc(void *ptr, size_t size); (C99 7.20.3.4)**
   “С точки зрения программной инженерии реализована неправильно, так как умеет делать очень многое” - цитатник Ломовского .
   “Самая затратная из трех”

   - ptr == NULL, size != 0 - malloc (выделяет память)
   - ptr != NULL, size == 0 - free (освобождает память)
   - ptr != NULL, size != 0 - перевыделение памяти

   В худшем случае перевыделения:
   - Выделить новую область
   - Скопировать данные из старой области в новую
   - Освободить старую область

   (В “худшем” потому что разработчики стандартной библиотеки люди умные, понимают, что выделение динамической памяти штука достаточно затратная и стараются с этой затратностью бороться. Поэтому и придумывают всякие обходные пути, типа выделить не столько памяти, сколько мы попросили, а чуть больше. Тогда при удачном запасе и удачном размере новой области никакого перевыделения делать не нужно будет. Аналогично можно делать, когда происходит уменьшение размера выделенной области. Область помечается как уменьшившаяся, но фактически ничего не уменьшается. НО закладываться на то, что такие оптимизации будут обязательно выполнены не стоит)

Типичная ошибка вызова realloc

Неправильно

```
// pbuf и n имеют корректные значения
pbuf = realloc(pbuf, 2 * n);
```

Что будет, если realloc вернет NULL?

Правильно

```
void *ptmp = realloc(pbuf, 2 * n);
if (ptmp)
    pbuf = ptmp;
else
    // обработка ошибочной ситуации
```

Явное приведение типа a = **(int*)** malloc(n * sizeof(int):

Преимущества:

- Программа собирается как си-компилятором, так и компилятором c++. (т.к. в язык c++ не может сам неявно преобразовать указатель на void к любому другому указателю)
- Программа собирается с использованием очень старого компилятора си (еще до принятия стандарта ANSI C). До этого стандарта у функции был другой прототип char* malloc(size_t size)
- Дополнительная проверка аргументов разработчиком

Недостатки:

- Начиная с ANSI C такое приведение не нужно
- Может скрыть ошибку, если не подключен stdlib.h (Лом рассказывает здесь страсти про разницу платформ. Лекция 1, 37 минута)
- В случае изменения типа указателя нужно изменить и тип в приведении

Что будет, если попытаться выделить 0 байт?

Implementation-defined (C99 7.20.3) - результат зависит от реализации компилятора.

Возможные варианты:

- вернется нулевой указатель
- вернется “нормальный” указатель, но его нельзя будет использовать для разыменования

ПОЭТОМУ перед вызовом этих функций необходимо убедится, что размер блока не равен 0.

Возвращение массива из функции

Прототип:

- Как возвращаемое значение `int* create(FILE *f, int *n);`
- Как параметр функции `int create(FILE *f, int *n, int **arr);`

Вызов:

- Как возвращаемое значение
  ```
  int n;
  int *arr = create(f, &n);
  ```
- Как параметр функции
  ```
  int n, rc;
  int *arr;
  rc = create(f, &n, &arr);
  ```

Сама функция:

```c
int* create_array(FILE *f, int *n)
{
    int *p = NULL;

    *n = get_count(f);
    if (*n)
    {
        p = malloc(*n * sizeof(int));
        if (p)
        {
            rewind(f);

            if (read_array(f, p, *n))
            {
                free(p);
                p = NULL;
            }
        }
    }

    return p;
}
```

```c
int get_count(FILE *f)
{
    int dummy, n = 0;

    while (fscanf(f, "%d", &dummy) == 1)
        n++;

    return n;
}
```

Пример функции, которую можно использовать как для статического, так и для динамического массива:

```c
int read_array(FILE *f, int *arr, int n)
{
    for (int i = 0; i < n; i++)
        if (fscanf(f, "%d", arr + i) != 1)
            return ERR_IO;

    return ERR_OK;
}
```

Типичные ошибки при работе с динамической памятью

- Неверный расчет количества выделяемой памяти

```c
int n_elems = 5;
int *arr = NULL;

arr = malloc(n_elems);
if (arr)
{
    for (int i = 0; i < n_elems; i++)
        arr[i] = i;
```

```c
struct date *p = NULL;

p = malloc(sizeof(p));
if (p)
{
    p->day = day;
```

- Отсутствие проверки успешности выделения памяти
- Утечка памяти

```c
int *p = NULL;

p = malloc(sizeof(int));
if (p)
{
    *p = 5;

    printf("%d\n", *p);

    p = malloc(sizeof(int));
    if (p)
    {
```

- Логические ошибки
    - Wild pointer - использование неинициализированного указателя

```c
int *p;

setbuf(stdout, NULL);

printf("Input n: ");
if (scanf("%d", p) == 1)
{
    printf("%d", *p);
}
```

    - Dangling pointer (висящий) - использование указателя сразу после освобождения памяти

```
free(p);

*p = 7;

printf("%d\n", *p);
```

- Изменение указателя, который вернула функция выделения памяти

```
pbeg = malloc(n_elems * sizeof(int));
if (pbeg)
{
    pend = pbeg + n_elems;

    while (pbeg < pend)
    {
        *pbeg = 0;
        pbeg++;
    }

    free(pbeg);
}
```

- Двойное освобождение памяти
- Освобождение невыделенной или нединамической памяти
- Выход за пределы динамического массива

Подходы к обработке ситуации отсутствия памяти (OOM)

- Возвращение ошибки (return value) - наш вариант
- Ошибка сегментации (segfault)

  (segfault - ошибка разыменования нулевого указателя).
  Плюс - очень дешево реализуется. Минус - проблемы с безопасностью
- Аварийное завершение (abort) - идея принадлежит Кернигану и Ричи (xmalloc)

Одна из первых реализаций:

```
#include <stdio.h>
extern char *malloc ();
void *
xmalloc (size)
    unsigned size;
{
  void *new_mem = (void *) malloc (size);
  if (new_mem == NULL)
    {
      fprintf (stderr, "fatal: memory exhausted (xmalloc of %u bytes).\n", size);
      exit (-1);
    }
  return new_mem;
}
```

- Восстановление (recovery) - xmalloc из git

  Основная идея: *“А давайте-ка я поскребу по сусекам, что-нибудь освобожу и вдруг выделится столько, сколько нужно”.*

```
Одна из реализация xmalloc из git

static void *do_xmalloc(size_t size, int gentle)
{
    void *ret;

    if (memory_limit_check(size, gentle))
        return NULL;
    ret = malloc(size);
    if (!ret && !size)
        ret = malloc(1);
    if (!ret) {
        try_to_free_routine(size);
        ret = malloc(size);
        if (!ret && !size)
            ret = malloc(1);
        if (!ret) {
            if (!gentle)
                die("Out of memory, malloc failed (tried to allocate %lu bytes)",
                    (unsigned long)size);
            else {
                error("Out of memory, malloc failed (tried to allocate %lu bytes)",
                      (unsigned long)size);
                return NULL;
            }
        }
    }
#ifdef XMALLOC_POISON
    memset(ret, 0xA5, size);
#endif
```

  Здесь интересно, как обработано выделение 0 байтов памяти - 0 не выделяется никогда, выделяется хотя бы 1.

ДОП. Отладчики использования памяти

Специальное программное обеспечение для обнаружения ошибок программы при работе с памятью, например, таких как утечки памяти или переполнение буфера.

- Dr.Memory (проблема с 32битной версией мсиса)

  Запуск - сначала обычная компиляция с ключом -ggdb, потом drmemory – ./app.exe

- valgrind

3. (11-16) Указатель на функцию

Описание, инициализация, вызов функции по указателю:

/* Объявление предоставляет основные свойства сущности: ее тип и название. Определение предоставляет все детали этой сущности — если это функция, что она делает; если это переменная, где эта переменная находится. Часто, компилятору нужно объявление, чтобы скомпилировать файл в объектный файл, так как компоновщик может найти определение из другого файла. */

1. Объявление указателя на функцию

```
double trapezium(double a, double b, int n,
                                    double (*func)(double));
```

2. Получение адреса функции

```
result = trapezium(0, 3.14, 25, &sin /* sin */);
```

3. Вызов функции по указателю

```
y = (*func)(x);  // y = func(x);
```

Функция qsort, примеры использования:

Определена в заголовочном файле stdlib.h.

```
void qsort(void *base, size_t nmemb, size_t size,
     int (*cmp)(const void*, const void*));
```

Пример:

Пусть необходимо упорядочить массив целых чисел по возрастанию.

```
int compare_int(const void* p, const void* q)
{
    const int *a = p;
    const int *b = q;
    return *a - *b;    // return *(int*)p - *(int*)q;
}
...
int a[10];
...
qsort(a, sizeof(a) / sizeof(a[0]), sizeof(a[0]),
                                        compare_int);
```

3

Использование указателей на функцию:

- Функции обратного вызова (callback)

- передача исполняемого кода в качестве одного из параметров другого кода.

Другими словами, функция обратного вызова - это “действие”, передаваемое в функцию в качестве аргумента, которое обычно используется:

- Для обработки данных внутри функции
- Для того, чтобы связываться с тем, кто вызвал функции при наступлении какого-то события.

- Таблицы переходов (jump table)
- Динамическое связывание (binding)

Простой пример:

```c
#include <stdio.h>

int add(int a, int b)
{
    return a + b;
}

int mul(int a, int b)
{
    return a * b;
}

typedef int (*ptr_action_t)(int, int);

int apply(int a, int b, ptr_action_t action)
{
    return action(a, b);
}

int main(void)
{
    ptr_action_t p_action = add; // или &add

    int a = 5, b = 4;

    printf("%d + %d = %d\n", a, b, apply(a, b, p_action));

    printf("%d * %d = %d\n", a, b, apply(a, b, mul));

    return 0;
}
```

Особенности использования указателей на функцию:

- Согласно C99 6.7.5.3 #8, выражение из имени функции неявно преобразуется в указатель на функцию. Операция & для функции возвращает указатель на функцию, но из-за этого пункта стандарта это лишняя операция.

```c
int add(int a, int b);
int (*p1)(int, int) = add;    ===    int(*p1)(int, int) = &add;
```

- Операция * для указателя на функцию возвращает саму функцию, которая неявно преобразуется в указатель на функцию.

```
int (*p1)(int, int) = *add;
int (*p1)(int, int) = *******add;
```

- Указатели на функции можно сравнивать
  `if (p1 == add) …`

- Указатель на функцию может быть типом возвращаемого значения функции.

```
int (*get_action(char ch))(int, int);

// typedef приходит на помощь :)
typedef int (*ptr_action_t)(int, int);

ptr_action_t get_action(char ch);
```

Указатель на функцию и адресная арифметика
Есть явное ощущение, что применять адресную арифметику к указателям нельзя, но в лекции об этом ничего нет

Указатель на функцию и указатель на void.

C99 6.3.2.3 #1

Указатель на void может быть преобразован в указатель или из указателя на любой неполный или объектный тип. Указатель на любой неполный или объектный тип может быть преобразован в указатель на пустоту и обратно; результат должен сравниваться с исходным указателем.

C99 6.3.2.3 #8
Указатель на функцию одного типа может быть преобразован в указатель на функцию другого типа и обратно; результат должен быть равен исходному указателю. Если преобразованный указатель используется для вызова функции, тип которой несовместим с типом указателя, поведение не определено.

Согласно C99 6.3.2.3 #1 и C99 6.3.2.3 #8, указатель на функцию **не** может быть преобразован к указателю на void и наоборот. (Функция - не объект в терминологии стандарта)

НО POSIX требует, чтобы такое преобразование было возможно при работе с динамическими библиотеками.

- C99 J.5.7 Function pointer casts (расширение стандарта)
- POSIX dlsym RATIONALE
- Generic Function Pointer C2X (будущее (?))

```
nataa@DESKTOP-BL12PJP MINGW64 /c/Users/nataa/OneDrive/Документы/src_02
$gcc -std=c99 -Wall -Werror -Wpedantic -Wextra -Wvla test_06.c -c
test_06.c: In function 'main':
test_06.c:22:8: error: ISO C forbids assignment between function pointer and 'vo
id *' [-Werror=pedantic]
   22 |     p2 = p1;
      |        ^
test_06.c:23:8: error: ISO C forbids assignment between function pointer and 'vo
id *' [-Werror=pedantic]
   23 |     p1 = p2;
      |        ^
cc1.exe: all warnings being treated as errors
```

```c
#include <stdio.h>
#include <stdlib.h>

int add(int a, int b)
{
    return a + b;
}

int main(void)
{
    int (*p1)(int, int) = add;
    void *p2 = NULL;
    void (*p3)(void) = NULL;
    int a =4, b = 5;

    printf("%d + %d = %d\n", a, b, p1(a, b));

    p2 = p1;
    p1 = p2;

    printf("%d + %d = %d\n", a, b, p1(a, b));

    p3 = (void (*)(void)) p1;
    p1 = (int (*)(int, int)) p3;

    printf("%d + %d = %d\n", a, b, p1(a, b));

    return 0;
}
```

4. (17-31) Make

/* Преимущества многофайловой организации проекта:

- Код программы более структурированный, так как в один файл, как правило, выносятся функции, которые логически друг с другом связаны
- Повторное использование кода
- Облегчает работу над проектом нескольким программистам
- Раздельная компиляция (сокращение времени компиляции)

Так плохо делать, потому что в этом случае мы теряем преимущества раздельной компиляции, так как мы пересобираем все файлы всегда. */

Компиляция

```
gcc -std=c99 -Wall -Werror -pedantic -c hello.c
gcc -std=c99 -Wall -Werror -pedantic -c bye.c
gcc -std=c99 -Wall -Werror -pedantic -c main.c
gcc -std=c99 -Wall -Werror -pedantic -c test.c
```

Компоновка

```
gcc -o greeting.exe hello.o bye.o main.o
gcc -o test_greeting.exe hello.o bye.o test.o
```

Почему плохо делать так?

```
gcc -std=c99 -Wall -Werror *.c -o app.exe
```

make - утилита, которая автоматизирует процесс преобразования файлов из одной формы в другую. (на самом деле не ограничивается только компиляцией, можно еще, например, pdf-файлы с ее помощью получать)

Разновидности:

- GNU Make
- BSD Make
- Microsoft Make (nmake)

Все разновидности основываются на одних и тех же принципах, но различаются синтаксисом, поэтому между собой они не совместимы.

Для своей работы утилита make использует так называемый “сценарий сборки”, в котором нужно описать зависимости между файлами проекта и каким образом на основе одних файлов можно получить другие.

Утилита make принимает решение о том, какие файлы нужно пересобрать на основе последнего времени модификации файла, которое она получает из файловой системы.

Сценарий сборки и простой сценарий сборки, правила

Формально состоит из правил. Каждое правило состоит из трех частей: цель + зависимости + команды. (команда обязательно начинается с табуляции, пробелы не подойдут!). Целью часто бывает название того файла, который мы хотим получить. Для того, чтобы создать файл нужны данные. Эти данные оформляются как зависимости.

цель: зависимость_1 ... зависимость_n
[tab]команда_1
[tab]команда_2
...
[tab]команда_m

что создать/сделать: из чего создать
как создать/что сделать

```
greeting.exe : hello.o bye.o main.o
   gcc -o greeting.exe hello.o bye.o main.o

test_greeting.exe : hello.o bye.o test.o
   gcc -o test_greeting.exe hello.o bye.o test.o

hello.o : hello.c hello.h
   gcc -std=c99 -Wall -Werror -pedantic -c hello.c

bye.o : bye.c bye.h
   gcc -std=c99 -Wall -Werror -pedantic -c bye.c

main.o : main.c hello.h bye.h
   gcc -std=c99 -Wall -Werror -pedantic -c main.c

test.o : test.c hello.h bye.h
   gcc -std=c99 -Wall -Werror -pedantic -c test.c

clean :
   rm *.o *.exe
```

Алгоритм работы make:

Первый запуск make:

- make читает сценарий сборки и начинает выполнять первое правило

```
greeting.exe : hello.o bye.o main.o
    gcc -o greeting.exe hello.o bye.o main.o
```

- для выполнения этого правила необходимо сначала обработать зависимости

```
hello.o bye.o main.o
```

- ищет правило соответствующей зависимости

```
hello.o : hello.c hello.h
    gcc -std=c99 -Wall -Werror -pedantic -c hello.c
```

- все зависимости есть, файла-цели нет => этот файл нужно создать => выполняем правило

```
gcc -std=c99 -Wall -Werror -pedantic -c hello.c
```

- так получаются все .o файлы
- теперь, когда все зависимости есть, можно выполнить правило для exe файла

```
gcc -o greeting.exe hello.o bye.o main.o
```

Второй запуск (один си-файл был изменен):

- make читает сценарий сборки и начинает выполнять первое правило

```
greeting.exe : hello.o bye.o main.o
 gcc -o greeting.exe hello.o bye.o main.o
```

- для этого нужно сначала обработать зависимости

```
hello.o bye.o main.o
```

- make ищет правило для первой зависимости

```
hello.o : hello.c hello.h
```

```
      gcc -std=c99 -Wall -Werror -pedantic -c hello.c
```

- из правила становится понятно, что время изменения o-файла меньше времени изменения c-файла => o-файл нужно пересоздать

```
gcc -std=c99 -Wall -Werror -pedantic -c hello.c
```

- для остальных o-файлов делается тоже самое, но у них время создания больше времени изменения их зависимостей, поэтому пересоздавать эти o-файлы не нужно
- все зависимости для exe-файла получены. Время его изменения меньше времени изменения его зависимостей => создаем новый.

```
gcc -o greeting.exe hello.o bye.o main.o
```

Ключи запуска:

- **-f** - для указания имени файла сценария сборки (по умолчанию makefile или Makefile)

  `make -f makefile_2`
- **-B** - для безусловного выполнения правил

  `make -B`
- **-n** вывод всех команд без их выполнения

  `make -n`
- **-i** игнорирование ошибок при выполнении команд

  `make -i`
- **-p** показывать неявные правила и переменные
- **-r** запрещает использовать неявные правила

Переменные и комментарии:

Комментарии - строки, которые начинаются с #

Переменные определяются с помощью := `VAR_NAME := value`

Получить значение переменной `$(VAR_NAME)`

(это не единственный способ описать переменную, за остальными велкам в документацию)

```
# Компилятор
CC := gcc

# Опции компиляции
CFLAGS := -std=c99 -Wall -Werror -pedantic

# Общие объектные файлы
OBJS := hello.o bye.o

greeting.exe : $(OBJS) main.o
   $(CC) -o greeting.exe $(OBJS) main.o

test_greeting.exe : $(OBJS) test.o
   $(CC) -o test_greeting.exe $(OBJS) test.o

hello.o : hello.c hello.h
   $(CC) $(CFLAGS) -c hello.c

bye.o : bye.c bye.h
   $(CC) $(CFLAGS) -c bye.c

main.o : main.c hello.h bye.h
   $(CC)  $(CFLAGS) -c main.c

test.o : test.c hello.h bye.h
   $(CC)  $(CFLAGS) -c test.c

clean :
   rm *.o *.exe
```

ДОП. Фиктивные цели (.PHONY)

- цели, которые не являются именами файлов. Такие цели используются для выполнения каких-то действий (очистки, установки и тд)

  Чтобы make не пытался сделать эти цели именами файлов, их помечают атрибутом .PHONY

  `.PHONY: clean`

Неявные правила и переменные

Понятно, что есть ряд действий, выполнение которых всем понятно. Разработчики make решили, что эти, всем известные, действия могут быть реализованы по умолчанию. Так появились неявные правила. Для того, чтобы выполнилось неявное правило ему нужен какой-то компилятор и какие-то опции компиляции. Все это очень удобно расположить в переменных. Отсюда появились неявные переменные.

Итак, неявные правила - это те умолчания, которые утилита make будет делать за вас, если вы сами их явно не сделаете.

```
$ make -f makefile_2
gcc -std=c99 -Wall -Werror -pedantic -c hello.
gcc -std=c99 -Wall -Werror -pedantic -c bye.c
gcc -std=c99 -Wall -Werror -pedantic -c main.c
gcc -o greeting.exe hello.o bye.o main.o

Dell@DESKTOP-67JGOMQ MINGW32 ~/2021_winter/18_
$ make -f makefile_2 clean
rm -f *.o *.exe

Dell@DESKTOP-67JGOMQ MINGW32 ~/2021_winter/18_
$ make -f makefile_3
cc    -c -o hello.o hello.c
cc    -c -o bye.o bye.c
cc    -c -o main.o main.c
cc -o greeting.exe hello.o bye.o main.o
```

`make -p > report.txt` (в каталоге, где нет makefile-а) - посмотреть все возможные неявные вещи

`make -r` - запрет использования неявных правил

Автоматические переменные

Автоматические переменные - переменные со специальными именами, которые “автоматически” принимают определенное значение перед выполнением описанных в правиле команд.

- **$^** - список зависимостей
- **$@** - имя цели
- **$<** - первая зависимость

Было

```
greeting.exe : $(OBJS) main.o
	$(CC) -o greeting.exe $(OBJS) main.o
```

Стало

```
greeting.exe : $(OBJS) main.o
	$(CC) -o $@ $^
```

Было

```
hello.o : hello.c hello.h
	$(CC) $(CFLAGS) -c hello.c
```

Стало

```
hello.o : hello.c hello.h
	$(CC) $(CFLAGS) -c $<
```

```
# Компилятор
CC := gcc

# Опции компиляции
CFLAGS := -std=c99 -Wall -Werror -pedantic

# Общие объектные файлы
OBJS := hello.o bye.o

greeting.exe : $(OBJS) main.o
	$(CC) $^ -o $@

test_greeting.exe : $(OBJS) test.o
	$(CC) $^ -o $@

hello.o : hello.c hello.h
	$(CC) $(CFLAGS) -c $<
```

Шаблонные правила

Шаблонные правила - обычные правила, которые можно применять не к одному файлу, а к группе. Группа файлов, как правило, задается расширением.

%.расш_файлов_целей : %.расш_файлов_зав
[tab]команда_1
[tab]команда_2
...
[tab]команда_m

```
# Компилятор
CC := gcc

# Опции компиляции
CFLAGS := -std=c99 -Wall -Werror -pedantic

# Общие объектные файлы
OBJS := hello.o bye.o

greeting.exe : $(OBJS) main.o
   $(CC) $^ -o $@

test_greeting.exe : $(OBJS) test.o
   $(CC) $^ -o $@

%.o : %.c *.h
   $(CC) $(CFLAGS) -c $<

.PHONY : clean
clean :
   $(RM) *.o *.exe
```

Проблема - нельзя корректно расписать зависимости h файлов. (решается ниже)

Условные конструкции, сборка программы с разными параметрами компиляции

Понятно, что для получения отладочной или релизной сборки мы можем написать отдельный makefile, но это очень неудобно, потому что в случае изменений их нужно будет делать в двух местах. Для решения этой проблемы в makefile-ах есть условные конструкции.
Сборка - `make mode=debug`

```
# Компилятор
CC := gcc

# Опции компиляции
CFLAGS := -std=c99 -Wall -Werror -pedantic

# Общие объектные файлы
OBJS := hello.o bye.o

ifeq ($(mode), debug)
    # Отладочная сборка: добавим генерацию отладочной информации
    CFLAGS += -g3
endif

ifeq ($(mode), release)
```

```
    # Финальная сборка: исключим отладочную информацию и
    # утверждения (asserts)
    CFLAGS += -DNDEBUG -g0
endif

greeting.exe : $(OBJS) main.o
   $(CC) $^ -o $@

test_greeting.exe : $(OBJS) test.o
   $(CC) $^ -o $@

%.o : %.c *.h
   $(CC) $(CFLAGS) -c $<

.PHONY : clean
clean :
   $(RM) *.o *.exe
```

ifeq - не единственная форма условного оператора, остальные в документации.

Переменные зависящие от цели

Условные операторы - это не единственный способ, которым можно влиять на особенности выполнения мейкфайла.

```
# Компилятор
CC := gcc

# Опции компиляции
CFLAGS := -std=c99 -Wall -Werror -pedantic

# Общие объектные файлы
OBJS := hello.o bye.o

debug : CFLAGS += -g3
debug : greeting.exe

release : CFLAGS += -DNDEBUG -g0
release : greeting.exe
```

```
greeting.exe : $(OBJS) main.o
    $(CC) $^ -o $@

test_greeting.exe : $(OBJS) test.o
    $(CC) $^ -o $@

%.o : %.c *.h
    $(CC) $(CFLAGS) -c $<

.PHONY : clean debug release
clean :
    $(RM) *.o *.exe
```

Автоматическая генерация зависимостей

С одной стороны, шаблонные правила использовать очень удобно, с другой, у таких правил есть свой недостаток, связанный с тем, что использование этих правил не позволяет нам корректно выполнить анализ зависимостей.

Утилита, которая знает все о зависимостях, - компилятор. Поэтому переложим все хозяйство на него с помощью специальных ключей.

```
$ gcc -M hello.c
hello.o: hello.c C:/msys64/mingw32/i686-w64-mingw32/include/stdio.h \
 C:/msys64/mingw32/i686-w64-mingw32/include/corecrt_stdio_config.h \
 C:/msys64/mingw32/i686-w64-mingw32/include/corecrt.h \
 C:/msys64/mingw32/i686-w64-mingw32/include/_mingw.h \
 C:/msys64/mingw32/i686-w64-mingw32/include/_mingw_mac.h \
 C:/msys64/mingw32/i686-w64-mingw32/include/_mingw_secapi.h \
 C:/msys64/mingw32/i686-w64-mingw32/include/vadefs.h \
 C:/msys64/mingw32/i686-w64-mingw32/include/sdks/_mingw_ddk.h \
 C:/msys64/mingw32/i686-w64-mingw32/include/_mingw_off_t.h \
 C:/msys64/mingw32/i686-w64-mingw32/include/swprintf.inl \
 C:/msys64/mingw32/i686-w64-mingw32/include/sec_api/stdio_s.h hello.h

Dell@DESKTOP-67JGOMQ MINGW32 ~/2021_winter/18_09/src_03
$ gcc -MM hello.c
hello.o: hello.c hello.h
```

Заметим, что вывод компилятора в этом случае ориентирован на утилиту make.

В итоге получаем новый makefile, где:

- **wildcard** - встроенная функция make, которая может находить все файлы, подходящие под правило)
- **%.d** - информация обо всех зависимостях, для соответствующего c-файла. (Запуск как на картинке выше, перенаправленный в файл с расширением d)
- **$(SRC: .c=.d)** - указание мейкфайлу поменять всем файлам в переменной SRC расширение с .c на .d
- **include** - директива мейкфайла, в целом работает также, как директива препроцессора, то есть включает файл с указанным именем. Но в отличие от препроцессора, если эта директива не может найти указанный файл, она заглядывает в сценарий сборки на предмет поиска правила, которое порождало бы этот файл, и потом снова выполняет директиву include еще раз.

```
# Компилятор
CC := gcc

# Опции компиляции
CFLAGS := -std=c99 -Wall -Werror -pedantic

# Общие объектные файлы
OBJS := hello.o bye.o

# Все c-файлы (или так SRCS := $(wildcard *.c))
SRCS := hello.c bye.c test.c main.c

greeting.exe : $(OBJS) main.o
    $(CC) $^ -o $@

test_greeting.exe : $(OBJS) test.o
    $(CC) $^ -o $@

%.o : %.c
    $(CC) $(CFLAGS) -c $<

%.d : %.c
    $(CC) -M $< > $@

# $(SRCS:.c=.d) - заменяет в переменной SRCS имена файлов с
# с расширением "c" на имена с расширением "d"
include $(SRCS:.c=.d)

.PHONY : clean
clean :
    $(RM) *.o *.exe *.d
```

Чет, если честно, нифига не понятно, почему это должно работать. Ну сгенерировали мы файлы с зависимостями, даже предположим, что у нас содержимое этих файлов вставилось в мейкфайл. В итоге ведь получится что-то очень странное, когда сначала мы объявили общее правило, что o зависит только от c и соответствующее правило для компиляции, а потом вставили странные конструкции .o : 1.c 2.c …

Особенности выполнения команд:

- Ненулевой код возврата может прервать выполнение сценария
- Каждая команда выполняется в своем shell

Не сработает:

```
run_test : test_greeting.exe
        ./test_greeting.exe
        cd test_data
        touch test.txt
        cd ..
```

5. (32-36) Динамические матрицы

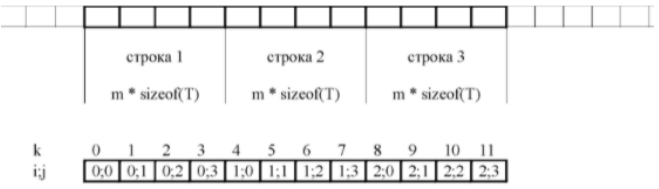
Как мы знаем, никакого специального типа для описание многомерных массивов в языке си нет. Язык си воспринимает матрицу, как массив, элементами которого являются другие массивы. Поэтому в случае статической матрицы компилятор располагает элементы статической матрицы по сути в одном большом одномерном массиве, где эти элементы хранятся построчно.

В случае динамической матрицы есть несколько способов ее хранения. В зависимости от способа мы получим разный набор преимуществ и недостатков.

Матрица как одномерный массив

Выделить память под такой массив достаточно просто. Нужно всего лишь узнать его размер, который рассчитывается как `i * j * sizeof(тип элемента)`. Сложность этого способа - вычисление индексов массива. Для этого придется применять конструкцию `i * m + j`.

n = 3 – количество строк
m = 4 – количество столбцов
T – тип элементов матрицы



k	0	1	2	3	4	5	6	7	8	9	10	11
i;j	0;0	0;1	0;2	0;3	1;0	1;1	1;2	1;3	2;0	2;1	2;2	2;3

a[i][j] ⇔ a[k], k = i * m + j

Преимущества:

- Простота выделения и освобождения памяти
- Возможность использовать, как одномерный массив

Недостатки:

- Отладчик использования памяти не может вычислить выход за пределы строки
- Нужно писать i * m + j

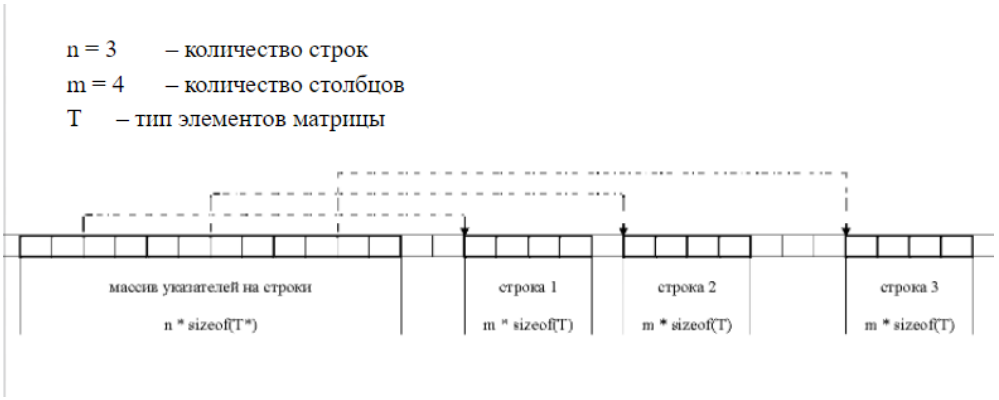
```
double *data;
int n = 3, m = 2;

// Выделение памяти под "матрицу"
data = malloc(n * m * sizeof(double));
if (data)
{
    // Работа с "матрицей"
    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
            // Обращение к элементу i, j
            data[i * m + j] = 0.0;

    // Освобождение памяти
    free(data);
}
```

Матрица как массив указателей

Основная идея: представим матрицу как набор строк. Каждая строка матрицы - это одномерный массив. Выделим память под каждую строку и сохраним в отдельный массив указатели на начало каждой строки.



Алгоритм выделения памяти:

Вход: количество строк n и количество столбцов m

Выход: указатель на массив строк матрицы p

1. Выделить память под массив указателей
2. В цикле по количеству строк матрицы
   a. Выделить память под i-тую строку q
   b. Обработать ошибку выделения памяти
   c. p[i] = q

```
double** allocate_matrix(int n, int m)
{
    double **data = calloc(n, sizeof(double*));
    if (!data)
        return NULL;
    for (int i = 0; i < n; i++)
    {
        data[i] = malloc(m * sizeof(double));
        if (!data[i])
        {
            free_matrix(data, n);
            return NULL;
        }
    }
    return data;
}
```

В этой реализации передается полное количество строк в случае аварийного освобождения, так как изначально мы используем calloc, который нам про инициализирует все значения указателей нулями и => мы сможем их освобождать функцией free.

Алгоритм освобождения памяти:

Вход: указатель на массив строк матрицы p и количество строк

1. В цикле по количеству строк матрицы освобождаем память из под i-той строки
2. Освобождаем память из под массива указателей p

```
void free_matrix(double **data, int n)
{
    for (int i = 0; i < n; i++)
       // free можно передать NULL
       free(data[i]);

    free(data);
}
```

Преимущества:

- Возможность обмена строками через обмен указателей.
- Отладчик использования памяти может отследить выход за пределы строки.

Недостатки:

- Сложность выделения и освобождения памяти
- Память под матрицу “не лежит” единым блоком

Объединение подходов - строки лежат единым блоком

Давайте хранить все в двух массивах

n = 3 – количество строк
m = 4 – количество столбцов
T – тип элементов матрицы

Алгоритм выделения памяти:
Вход: количество строк n и количество столбцов m
Выход: указатель на массив строк матрицы p

1. Выделить память под массив указателей на строки p
2. Обработать ошибку выделения памяти
3. Выделить память под данные (т.е. под строки q)
4. Обработать ошибку выделения памяти
5. В цикле по количеству строк матрицы
   p[i] = адрес i-ой строки в массиве q

```
double** allocate_matrix(int n, int m)
{
    double **ptrs, *data;
    ptrs = malloc(n * sizeof(double*));
    if (!ptrs)
        return NULL;
    data = malloc(n * m * sizeof(double));
    if (!data)
    {
        free(ptrs);
        return NULL;
    }
    for (int i = 0; i < n; i++)
        ptrs[i] = data + i * m;
    return ptrs;
}
```

Алгоритм освобождения памяти:
Вход: указатель на массив строк матрицы p

1. Освободить память из под данных
2. Освободить память из под массива указателей

```
void free_matrix(double **ptrs)
{
    free(ptrs[0]);

    free(ptrs);
}
```

ВНИМАНИЕ
Здесь скрывается потенциальная ошибка.

Ошибка связана с тем, что адрес нулевой строки может меняться, например, если мы захотим поменять строки местами. В этом случае такая очистка памяти будет некорректной.
Вариант решения - поиск минимального значения адреса.
Другой вариант - выделение в массиве указателей на один элемент больше, чтобы хранить в этом дополнительном элементе адрес начала массива.
Еще один - использовать структуру и в ней хранить доп. указатель.

Преимущества:

- Относительная простота выделения и освобождения памяти
- Возможность использовать как одномерный массив
- Перестановка строк через обмен указателей (но это может приводить к ошибкам, см. выше)

Недостатки:

- Относительная сложность начальной инициализации
- Отладчик использования памяти не может отследить выход за пределы строки.

Объединение подходов - храним все одним блоком

Алгоритм выделения памяти:
Вход: количество строк n и количество столбцов m
Выход: указатель на массив строк матрицы p

1. Выделить память под массив указателей на строки и элементы массива
2. Обработать ошибку выделения памяти
3. В цикле по количеству строк вычислить адрес i-ой строки и сохранить его p[i] = q

```
double** allocate_matrix(int n, int m)
{
    double **data = malloc(n * sizeof(double*) + n * m * sizeof(double));
    if (!data)
        return NULL;

    for(int i = 0; i < n; i++)
        data[i] = (double*)((char*) data + n * sizeof(double*) + i * m * sizeof(double));

    return data;
}
```

Другой вариант реализации:

```
double** allocate_matrix(int n, int m)
{
    double **data = malloc(n * sizeof(double*) + n * m * sizeof(double));
    if (!data)
        return NULL;

    double *elems = (double*) (data + n);
    for(int i = 0; i < n; i++)
        data[i] = elems + i * m;

    return data;
}
```

Алгоритм освобождения памяти:

1. Освободить соответствующий указатель

Преимущества:

- Простота выделения и освобождения памяти
- Возможность использовать как одномерный массив
- Перестановка строк через обмен указателей

Недостатки:

- Сложность начальной инициализации
- Отладчик использования памяти не может отследить выход за пределы строки

Функция, которая будет одинаково обрабатывать как статические, так и динамические матрицы

1. Указатель на массив

```c
void foo_1(int a[][M] /*int (*a)[M]*/, int n, int m);

int main(void)
{
    int a[N][M];        // статический массив
    int *b = NULL;      // матрица как одномерный массив
    int **c = NULL;     // матрица как массив указателей
    int n = N, m = M;

    foo_1(a, n, m);     // скомпилируется
    foo_1(b, n, m);     // нет
    foo_1(c, n, m);     // нет
```

2. Указатель на указатель

```c
void foo_2(int **a, int n, int m);

int main(void)
{
    int a[N][M];        // статический массив
    int *b = NULL;      // матрица как одномерный массив
    int **c = NULL;     // матрица как массив указателей
    int n = N, m = M;

    foo_2(a, n, m);     // нет
    foo_2(b, n, m);     // нет
    foo_2(c, n, m);     // скомпилируется
```

3. Одномерный массив

```c
void foo_3(int *a, int n, int m);

int main(void)
{
    int a[N][M];        // статический массив
    int *b = NULL;      // матрица как одномерный массив
    int **c = NULL;     // матрица как массив указателей
    int n = N, m = M;

    foo_3(a, n, m);     // нет
    foo_3(b, n, m);     // да
    foo_3(c, n, m);     // нет
```

Решение этой проблемы с передачей статической матрицы в динамическую функцию:

```c
void foo_2(int **a, int n, int m)
{
}

int main(void)
{
    int a[N][M], n = N, m = M;
    int* b[N] = {a[0], a[1], a[2]};

    foo_2(b, n, m);
```

6. (37-47) Непонятная тема