



# ОТВЕТЫ НА ВОПРОСЫ ЭКЗАМЕНА 2

Первую часть можно найти [здесь](#)

Условные обозначения:

- ⚠ - информация неточная, нужно проверить
- ✓ - сделано
- P - нужны примеры
- ⚙ - в целом норм, но непонятно достаточно написано или нет
- sos - ничего не понятно, но очень интересно

Содержание:

1. [Указатель на void. \(1-2\)](#) ✓
  - ✓ Для чего используется указатель на void? Приведите примеры.
  - ✓ Каковы особенности использования указателя на void? Приведите примеры
2. [Функции выделения и освобождения памяти \(3-10\)](#) ✓
  - ✓ Функции для выделения и освобождения памяти malloc, calloc, free. Порядок работы и особенности использования этих функций.
  - ✓ Функция realloc. Особенности использования.
  - ✓ Общие «свойства» функций malloc, calloc, realloc.
  - ✓ Функция выделения памяти и явное приведение типа: за и против
  - ✓ Особенности выделения 0 байт памяти.
  - ✓ Способы возвращения динамического массива из функции.
  - ✓ Типичные ошибки при работе с динамической памятью (классификация, примеры).
  - ✓ Подходы к обработке ситуации отсутствия свободной памяти при выделении.
3. [Указатель на функцию \(11-16\)](#) sos
  - P Для чего используется указатель на функцию? Приведите примеры.
  - ✓ Указатель на функцию: описание, инициализация, вызов функции по указателю.
  - ⚙ Функция qsort, примеры использования.
  - ✓ Особенности использования указателей на функцию.
  - ⚠ Указатель на функцию и адресная арифметика.
  - ✓ Указатель на функцию и указатель на void.
4. [make \(17-31\)](#) ✓
  - ✓ Утилита make: назначение, входные данные, идея алгоритма работы
  - ✓ Разновидности утилиты make.
  - ✓ Сценарий сборки проекта: название файла, структура сценария сборки.
  - ✓ Правила: составные части, особенности использования правил в зависимости от составных частей.



- ✓ Особенности выполнения команд.
- ✓ Простой сценарий сборки.
- ✓ 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 использования.
- ✓ 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. Преимущества и недостатки.
- ✓ Объединенный подход для представления динамической матрицы (отдельное выделение памяти под массив указателей и массив данных). Преимущества и недостатки.
- ✓ Объединенный подход для представления динамической матрицы (массив указателей и массив данных располагаются в одной области). Преимущества и недостатки.
- ✓ Необходимо реализовать функцию, которая может обрабатывать как статические, так и динамические матрицы. Какими способами это можно сделать? *(меня пугает множественное число)*

6. [Сложные объявления \(37\)](#)



- ✓ Умение читать сложные объявления и использовать это на практике.

7. [Строки и динамическое выделение памяти \(38-39\)](#)



- ✓ Функции, возвращающие динамическую строку: strdup/strndup, getline, snprintf/asprintf.
- ✓ Feature Test Macros.

8. [Структуры и динамическое выделение памяти \(40-47\)](#)



- P Функции memcpu, memmove, memcpy, memset.
- ⚠ Структуры с полями указателями и особенности их использования.
- ✓ «Поверхностное» копирование vs «глубокое» копирование.
- ⚠ «Рекурсивное» освобождение памяти для структур с динамическими полями.
- SOS Структуры переменного размера. Приведите примеры. *(протокол...)*
- ✓ Что такое «flexible array member»? Какие особенности использования есть у этих полей? Для чего они нужны? Приведите примеры.
- ✓ Flexible array member до C99.
- ✓ Flexible array member vs поле-указатель.



9. [Динамически расширяемый массив \(48-56\)](#)



- Дайте определение массива.
- Описание типа.
- Добавление элемента
- Удаление элемента
- Особенности использования
- Почему при добавлении нового элемента память необходимо выделять блоками, а не под один элемент?

10. [Односвязный список \(57-64\)](#)

- Связный список
- 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.

11. [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.

12. [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?

### 13. 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 недостатками есть у использования глобальных переменных?

### 14. 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?

### 15. 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?

16. [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.

17. [Массивы переменной длины\(132-140\)](#)

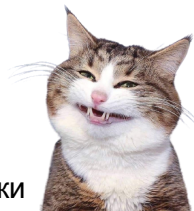
- Что такое variable length array?
- Чем отличается статический массив от variable length array?
- Какую операцию языка Си пришлось реализовывать по-другому (не как для встроенных типов) специально для variable length array?
- Особенности использования variable length array.
- Справедлива ли для variable 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?




18. [alloca \(141-142\)](#)

- Функция alloca.
- alloca vs VLA.

19. [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.)

## 20. [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 predefined 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?

## 21. [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.



## 22. [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 -l, -L компилятора gcc.

## 23. [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).

## 24. [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?





25. [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.

26. [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.

27. [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 храниться в выделенной области.

Особенности использования:

- В языке C допускается присваивание указателя типа `void` указателю любого другого типа (и наоборот) без явного преобразования типа указателя.

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

```
pd = pv;
```

(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 храниться в выделенной 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 которых 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)

Ошибки использования функции realloc:

- Не использовать доп указатель (приводит к утечке памяти, если realloc вернет NULL)
- Нерациональное использование realloc (например, в данном случае мы ухудшили сложность алгоритма с  $O(n)$  до  $O(n^2)$ )

Неправильно

```
int *p = malloc(10 * sizeof(int));  
  
p = realloc(p, 20 * sizeof(int));  
// А если realloc вернула NULL?
```

Правильно

```
int *p = malloc(10 * sizeof(int)), *tmp;  
  
tmp = realloc(p, 20 * sizeof(int));  
if (tmp)  
    p = tmp;  
else  
    // обработка ошибки
```

```
int* select_positive(const int *a, int n, int *k)  
{  
    int m = 0;  
    int *p = NULL;  
  
    for (int i = 0; i < n; i++)  
        if (a[i] > 0)  
        {  
            m++;  
            p = realloc(p, m * sizeof(int));  
            p[m-1] = a[i];  
        }  
  
    *k = m;  
    return p;  
}
```

**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 убедиться, 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:

```
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;
}
```

```
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:

```
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

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

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

```
struct date *p = NULL;

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

- Отсутствие проверки успешности выделения памяти
- Утечка памяти

```
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

```
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 XMALLOCS_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:

```
#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.

```
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 (?))

```
#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;
}
```

```
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 'void *' [-Werror=pedantic]
    22 |     p2 = p1;
        |         ^
test_06.c:23:8: error: ISO C forbids assignment between function pointer and 'void *' [-Werror=pedantic]
    23 |     p1 = p2;
        |         ^
cc1.exe: all warnings being treated as errors
```

## 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)

### 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
```

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



**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 о-файла меньше времени изменения с-файла => о-файл нужно пересоздать  

```
gcc -std=c99 -Wall -Werror -pedantic -c hello.c
```
- для остальных о-файлов делается тоже самое, но у них время создания больше времени изменения их зависимостей, поэтому пересоздавать эти о-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.o
gcc -std=c99 -Wall -Werror -pedantic -c bye.c
gcc -std=c99 -Wall -Werror -pedantic -c main.o
gcc -o greeting.exe hello.o bye.o main.o

De11@DESKTOP-67JG0MQ MINGW32 ~/2021_winter/18_
$ make -f makefile_2 clean
rm -f *.o *.exe

De11@DESKTOP-67JG0MQ 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-a) - 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
OBS := hello.o bye.o

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

test_greeting.exe : $(OBS) 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
OBS := 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.

<pre># 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</pre>	<pre>greeting.exe : \$(OBJS) main.o \$(CC) \$^ -o \$@  test_greeting.exe : \$(OBJS) test.o \$(CC) \$^ -o \$@  %.o : %.c *.h \$(CC) \$(CFLAGS) -c \$&lt;  .PHONY : clean debug release clean : \$(RM) *.o *.exe</pre>
--	--

## 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 с-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
OBS := hello.o bye.o

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

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

test_greeting.exe : $(OBS) 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
```

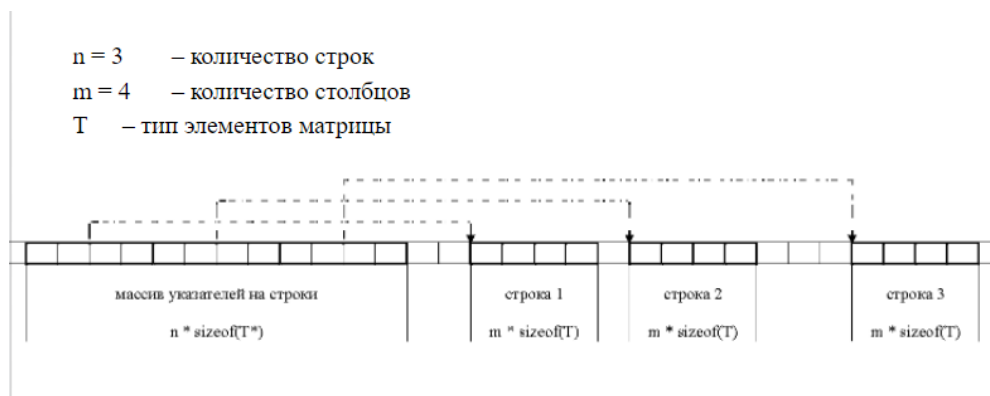
*Чет, если честно, нифига не понятно, почему это должно работать. Ну сгенерировали мы файлы с зависимостями, даже предположим, что у нас содержимое этих файлов вставилось в мейкфайл. В итоге ведь получится что-то очень странное, когда сначала мы объявили общее правило, что о зависит только от с и соответствующее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





## 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 проинициализирует 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



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**

$n = 3$  — 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 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

```
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

```
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

```
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:

```
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) Сложные объявления

Не возникает проблем с чтением простых объявлений.

```
int foo[5];  
char *foo;  
double foo(void);
```

Но как только объявления становится сложнее становится трудно сказать, что это

```
char *(*(**foo[][8])())[8];
```

### Замена элементов объявлений фразами

- `[]` - массив типа...
- `[N]` - массив из N элементов типа...
- `(type)` - функция, принимает аргумент типа type и возвращает...
- `*` - указатель на...

### Основные правила:

- Декодирование всегда происходит изнутри наружу. При этом отправной точкой является идентификатор.
- Предпочтение отдается скобкам, а не \*, т.е.  
*\*name[] - массив типа, а не указатель на*  
*\*name() - функция, принимающая, а не указатель на*
- `()` могут использоваться для изменения приоритета

```
int *(*x[10])(void);
```

1. Идентификатор - x
2. `x[10]` - массив из 10 элементов
3. `*` - типа указатель на
4. `(void)` - функцию, которая ничего не получает и возвращает
5. `int *` - указатель на int

Итог: x - массив из 10 элементов типа указатель на функцию, которая ничего не получает и возвращает указатель на int.



```
int (*foo(char))(int, int);
```

1. Идентификатор - foo
2. foo(char) - функция, принимает аргумент типа char
3. \* - и возвращает указатель на
4. int ... (int, int) - функцию, принимающую два инта и возвращающую инт

Итог: foo - это функция, которая принимает аргумент типа char и возвращает указатель на функцию, которая принимает два аргумента типа int и возвращает int.

```
typedef int (*p)(int, int);
```

1. Идентификатор p
2. \* - указатель на
3. (int, int) - функцию, которая получает два аргумента типа int и возвращает int

Итог: p - указатель на функцию, которая получает два аргумента типа int и возвращает int.

### Ограничения (чего быть не может):

- Нельзя создавать массив функций 

```
int a[10](int);
```
- Функция не может возвращать функцию 

```
int g(int)(int);
```
- Функция не может вернуть массив 

```
int f(int)[];
```
- В массиве только левая лексема [] может быть пустой
- Тип void ограниченный 

```
void x; void x[5];
```

(Но в первых двух случаях не возбраняется аналогичная работа с указателями)

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

int main(void)
{
    ptr_action_t actions[] = {add, sub, mul, div };
}
```

### Использование typedef для упрощения:

```
int (*x[10])(void);           // было
typedef int* func_t(void);     // функция (не факт, что
                               // переменные такого типа можно создать)
typedef func_t* func_ptr;     // указатель на функцию
```



```
typedef func_ptr* funt_ptr_arr[10]; // массив из 10 элементов типа
    указатель функцию
funt_ptr_arr x; // стало
```

## 7. (38-39) Строки и динамическое выделение памяти

С точки зрения языка си строка - это последовательность символов, которая заканчивается символом с кодом 0. Исходя из этого определения нам в принципе абсолютно все равно, в какой памяти строка будет располагаться.

Пример создания динамической строки (+1 для нулевого символа):

```
// Тут нужны include-ы

#define NAME "Bauman Moscow State Technical University"

int main(void)
{
    char *name = malloc((strlen(NAME) + 1) * sizeof(char));

    if (name)
    {
        strcpy(name, NAME);
        printf("%s\n", name);
        free(name);
    }
    else
        printf("Cant allocate memory\n");

    return 0;
}
```

(На размер char-а лучше умножать, так как могут быть проблемы с “широкими строками”, когда тип символа другой.)

Все тоже самое, но с функцией **strdup** (создает динамическую копию переданной строки, ее нет в стандарте си, только в стандарте posix):

```
// Тут нужны include-ы
// Для компиляции -std=gnu99

#define NAME "Bauman Moscow State Technical University"

int main(void)
{
    char *name = strdup(NAME); // string.h, POSIX (+ strdup)

    if (name)
    {
        printf("%s\n", name);
        free(name);
    }
    else
        printf("Cant allocate memory\n");

    return 0;
}
```





**getline** - прочитать строку из текстового файла целиком, независимо от длины строки. Возвращает размер (пере)выделенного буфера `n` и количество прочитанных символов `ssize_t`. В случае ошибки возвращает `-1`.

```
#include <stdio.h>
ssize_t getline(char **lineptr, size_t *n, FILE *stream); // POSIX
```

Можно передать как уже созданный буфер, тогда, если его недостаточно он будет перевыделен, так и `NULL` с `n = 0`, тогда функция сама выделит буфер с помощью `malloc` (именно поэтому все, что это функция вернет нужно будет почистить).

! Причем чистить буфер, который передается ей первым параметром, нужно даже в случае, когда функция вернула ошибку (Ломовской 6 лекция 34:12)

```
FILE *f;
char *line = NULL;
size_t len = 0;
ssize_t read;

// ...

f = fopen(argv[1], "r");
if (f)
{
    while ((read = getline(&line, &len, f)) != -1)
    {
        printf("len %d, read %d\n", (int) len, (int) read);
        printf("%s", line);
    }

    free(line);
    fclose(f);
}
```

В линуксе, для того чтобы собрать программу с `getline`, придется использовать **Feature Test Macros** - используется для управления доступностью имен в заголовочных файлах. Идея в том, чтобы указать компилятору, какой набор функций мы хотим использовать. Этот макрос располагается перед инклюдами программы.

```
// C glibc 2.10
#define _POSIX_C_SOURCE 200809L
// До glibc 2.10
#define _GNU_SOURCE
```



Msys-у этот макрос “вроде как” не помогает. Здесь решение проблемы - сборка программы в posix-ной оболочке msys-a.

**snprintf** - использование в динамике (сначала узнаем количество символов в строке, потом выделим и запишем снова)

```
int n, m;

n = snprintf(NULL, 0, "My name is %s. I live in %s.", NAME, CITY);
if (n > 0)
{
    char *line = malloc((n + 1) * sizeof(char));
    if (line)
    {
        m = snprintf(line, n + 1, "My name is %s. I live in %s.", NAME, CITY);

        printf("n = %d, m = %d\n", n, m);
        printf("%s\n", line);

        free(line);
    }
}
```

Другой аналогичный подход, выделять хоть что-нибудь, запускать функцию, смотреть на код ошибки и при необходимости запускать еще раз.

**asprintf** - улучшенная версия программы выше с snprintf. Сама вычисляет необходимый буфер. Функция не относится ни к стандарту, ни к posix. Лежит в стандарте расширения gnu. Чтобы эта функция стала доступна, обязательно нужно определять feature test macros GNU\_SOURCE, о котором было сказано выше.

```
#define _GNU_SOURCE
#include <stdio.h>

// ...
{
    char *line = NULL;
    int n;

    n = asprintf(&line, "My name is %s. I live in %s.", NAME, CITY);
    if (n > 0)
    {
        printf("n = %d\n", n);
        printf("%s\n", line);

        free(line);
    }
}
```



## 8. (40-47) Структуры и динамическое выделение памяти

### Функции `memcpy`, `memmove`, `memcmp`, `memset`

[memcpy](#)      Определен в `<string.h>`

- До стандарта c99  

```
void * memcpy ( void * dest, const void * src, size_t count ) ;
```
- Начиная со стандарта c99  

```
void * memcpy ( void * restrict dest, const void * restrict src, size_t count ) ;
```

Функция копирует `count` символов из объекта, на который указывает `src`, в объект на который указывает `dest`. Оба объекта интерпретируются как массивы беззнаковых символов (`unsigned char`).

Поведение не определено, если доступ происходит за пределами конца `dest` массива. Если объекты перекрываются (что является нарушением ограничивающего контракта) (начиная с C99) , поведение не определено. Поведение не определено , если либо `dest` или `src` является недействительным или нулевой указатель.

Возвращает копию `dest`.

`memcpy` это **самая быстрая библиотечная процедура** для копирования из памяти в память. Обычно он более эффективен, чем `strcpy` , который должен сканировать данные, которые он копирует, или `memmove` , который должен принимать меры для обработки перекрывающихся входных данных.

[memmove](#)      Определен в `<string.h>`

```
void * memmove ( void * dest, const void * src, size_t count ) ;
```

Копирует `count` символов из объекта, на который указывает, `src` в объект, на который указывает `dest`. Оба объекта интерпретируются как массивы беззнаковых символов (`unsigned char`).

**Объекты могут перекрываться:** копирование происходит так, как если бы символы были скопированы во временный массив символов, а затем символы были скопированы из массива в `dest`.

Поведение не определено, если доступ происходит за пределами конца массива `dest`. Поведение не определено , если либо `dest` или `src` является недействительным или нулевыми указателями.

Возвращает копию `dest`.



[memcmp](#)      Определен в <string.h>

```
int memcmp ( const void * lhs, const void * rhs, size_t count )
```

Сравнивает первые count символов объектов, на которые указывают lhs и rhs. Сравнение проводится лексикографически.

Знак результата - это знак разницы между значениями первой пары байтов (оба интерпретируются как unsigned char), которые различаются сравниваемыми объектами.

Поведение не определено, если доступ происходит за пределами любого объекта, на который указывают lhs и rhs. Поведение не определено, если любой из указателей lhs/rhs является нулевым указателем.

Возвращает:

- <0, если оно lhs указано rhs в лексикографическом порядке.
- 0, если lhs и rhs сравнение равно, или если счетчик равен нулю.
- >0, если lhs указано после rhs в лексикографическом порядке.

/\*

Эта функция считывает представления объектов , а не значения объектов, и обычно имеет смысл только для массивов байтов: структуры могут иметь байты заполнения, значения которых не определены, значения любых байтов за пределами последнего сохраненного члена в объединении являются неопределенными, а тип может иметь два или более представления для одного и того же значения (разные кодировки для +0 и -0 или для +0,0 и -0,0, неопределенные биты заполнения внутри типа).

\*/

[memset](#)      Определен в <string.h>

```
void *memset( void *dest, int ch, size_t count );
```

Копирует значение ch (после преобразования в unsigned char как если бы(unsigned char)ch) в каждый из первых count символов объекта, на который указывает dest.

Поведение не определено, если доступ происходит за пределами конца массива dest. Поведение не определено, если dest это нулевой указатель.

Возвращает копию dest.

/\*

memset может быть оптимизирован (в соответствии с правилами « как если бы »), если к объекту, измененному этой функцией, не будет получен доступ снова в течение оставшейся части его жизненного цикла (например, ошибка gcc 8537 ). По этой причине эту функцию нельзя использовать для очистки памяти (например, для заполнения массива, в котором хранится пароль, нулями). Эта оптимизация запрещена для memset\_s: гарантировано выполнение записи в память. Сторонние решения для этого включают FreeBSD explicit\_bzero или Microsoft SecureZeroMemory .

\*/



## Структуры с указателями и особенности их использования

Пример структуры с полями-указателями:

```
struct tree_node
{
    const char *name;
    struct tree_node *left;
    struct tree_node *right;
};
```

Особенность в том, что плохо копируется.

### «Рекурсивное» освобождение памяти для структур с динамическими полями.

Судя по всему имелся в виду тот факт, что когда мы освобождаем память из под структуры с динамическими полями нам важно не забыть сначала освободить память из под самих полей структуры.

Пример соответствующей функции:

```
void book_free_ex(struct book_t **pbook)
{
    if (!(*pbook))
        return;

    free((*pbook)->title);
    free(*pbook);

    *pbook = NULL;
}
```

### Поверхностное VS глубокое копирование

Из прошлого семестра мы знаем, что для структурных переменных одного и того же типа определена операция присваивания. Эта операция представляет собой фактически побитовое копирование области памяти, которую занимает одна переменная, в область памяти, которую занимает другая. Такой подход к копированию называется “**поверхностным копированием**” (shallow coping). У него есть как преимущества (простота), так и недостатки (копируется содержимое структурной переменной, но не копируется то, на что ссылаются поля этой переменной). Проблемы, которые могут возникать в этом случае:



- утечка памяти (присвоили, а старое значение переменной почистить забыли)
- двойное освобождение памяти (две структуры содержат указатели на одну область памяти после поверхностного копирования и обе эти области мы пытаемся почистить)

Соответствующий пример из лекции:

```
struct book_t
{
    char *title;
    int year;

} a = { 0 }, b = { 0 };

a.title = strdup("Book a");
a.year = 2000;

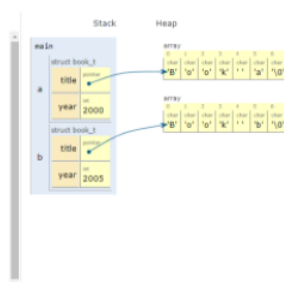
b.title = strdup("Book b");
b.year = 2005;

a = b;    // потеряли память

free(a.title);
free(b.title);    // много потеряли
```

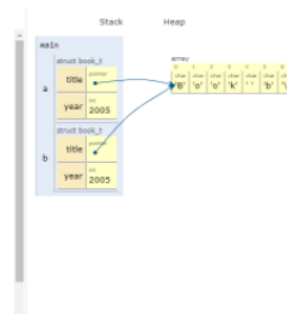
До присваивания

```
C (gcc 4.8, C11)
(Known limitations)
1 #include <string.h>
2 #include <stdlib.h>
3
4 int main(void)
5 {
6     struct book_t
7     {
8         char *title;
9         int year;
10    } a = { 0 }, b = { 0 };
11
12    a.title = strdup("Book a");
13    a.year = 2000;
14
15    b.title = strdup("Book b");
16    b.year = 2005;
17
18    a = b;
19
20
```



После присваивания

```
C (gcc 4.8, C11)
(Known limitations)
1 #include <string.h>
2 #include <stdlib.h>
3
4 int main(void)
5 {
6     struct book_t
7     {
8         char *title;
9         int year;
10    } a = { 0 }, b = { 0 };
11
12    a.title = strdup("Book a");
13    a.year = 2000;
14
15    b.title = strdup("Book b");
16    b.year = 2005;
17
18    a = b;
19
20    free(a.title);
21    free(b.title);
22
```



Чтобы как-то решать эти проблемы реализуется стратегия “**глубокого копирования**” (deep coping). Она подразумевает создание копий объектов, на которые ссылаются поля структуры.



```
int book_copy(struct book_t *dst, const struct book_t *src)
{
    char *ptmp = strdup(src->title);
    if (ptmp)
    {
        free(dst->title);
        dst->title = ptmp;
        dst->year = src->year;

        return 0;
    }

    return 1;
}
```

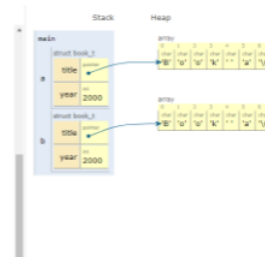
До копирования

После копирования

```
1  C (gcc 4.8, C11)
2  (OpenSSL 1.0.1.4)
3
4  23 dst->year = src->year;
5  24 return 0;
6  25 }
7
8  26 int main(void)
9  27 {
10 28 struct book_t a = { 0 }, b = { 0 };
1129 a.title = strdup("Book a");
1230 a.year = 2000;
1331 b.title = strdup("Book b");
1432 b.year = 2005;
1533 book_copy(&b, &a);
1634 free(a.title);
1735 free(b.title);
18...
```



```
1  C (gcc 4.8, C11)
2  (OpenSSL 1.0.1.4)
3
4  23 dst->year = src->year;
5  24 return 0;
6  25 }
7
8  26 int main(void)
9  27 {
1028 struct book_t a = { 0 }, b = { 0 };
1129 a.title = strdup("Book a");
1230 a.year = 2000;
1331 b.title = strdup("Book b");
1432 b.year = 2005;
1533 book_copy(&b, &a);
1634 free(a.title);
1735 free(b.title);
18...
```



Раньше мы всегда подразумевали, что структурные переменные имеют один и тот же размер. На самом деле на практике часто возникают ситуации, когда нужно работать со **структурами переменного размера**.

Существует такая схема кодирования, которая называется TLV - схема кодирования произвольных данных в некоторых телекоммуникационных протоколах. Данные в этой схеме кодирования описываются тройками:

- Type - описание назначения данных
- Length - размер данных (обычно в байтах)
- Value - данные

Первые два поля имеют фиксированный размер.

Идея в том, что если описывать это структурной переменной на языке си, то описание типа для разных структурных переменных будет одинаковым, а размер области памяти, которую будет занимать такая переменная, будет разным.

TLV-кодирование используется в:

- семействе протоколов TCP/IP





- спецификация PC/SC (smart cards) (для чтения или записи данные нужно закодировать с помощью TLV)
- ASN.1 (нотация для описания сертификатов)
- Реестры в винде (некоторые специальные базы данных, которые хранят настройки как системы в целом, так и отдельных приложений, эти базы реализованы на представлении объектов, которые кодируются в TLV)
- Графический формат tiff, в представлении объектов которого тоже используется TLV (Лом не был уверен в этой инфе)

Преимущества TLV-кодирования:

- простота разбора
- тройки TLV с неизвестным типом (тегом) могут быть опущены при разборе
- тройки TLV могут помещаться в произвольном порядке
- тройки TLV обычно кодируются двоично, что позволяет выполнять разбор быстрее и требует меньше объема по сравнению с кодированием, основанном на текстовом представлении

### Flexible array member

В 99 стандарте для описания структур переменного размера в структуры было добавлено поле, которое называется **flexible array member** (поле типа гибкий массив).

```
struct  
{  
    int n;  
    double d[];  
}
```

Особенности этого поля и соответствующей структуры:

- Такое поле должно быть последним
- Нельзя создать массив структур с таким полем (т.к. массив - это последовательность элементов одного типа и эти элементы занимают одинаковое количество байт, а здесь у нас элементы будут занимать разное количество памяти)
- Структура с таким полем не может использоваться как член в "середине" другой структуры (обязательно в конце)
- Операция sizeof не учитывает размер этого поля (возможно, за исключением выравнивания)
- Если в этом массиве нет элементов (не выделена память под него), то обращение к его элементам - неопределенное поведение



Алгоритм создания такой структуры:

```
struct s* create_s(int n, const double *d)
{
    assert(n >= 0);

    // посчитать размер структуры, как сумму фиксированной части и размера массива
    struct s *elem = malloc(sizeof(struct s) + n * sizeof(double));

    if (elem)
    {
        elem->n = n;
        memmove(elem->d, d, n * sizeof(double));
    }

    return elem;
}
```

```
// Чтобы успокоить valgrind
struct s *elem = calloc(sizeof(struct s) + n * sizeof(double), 1);
```

Сохранить структуру в файл:

```
int save_s(FILE *f, const struct s *elem)
{
    int size = sizeof(*elem) + elem->n * sizeof(double);
    int rc = fwrite(elem, 1, size, f);

    if (size != rc)
        return 1;

    return 0;
}
```



Прочитать структуру из файла

```
struct s* read_s(FILE *f)
{
    struct s part_s;
    struct s *elem;
    int size;

    // прочитали количество элементов, которое содержит структура
    size = fread(&part_s, 1, sizeof(part_s), f);
    if (size != sizeof(part_s))
        return NULL;

    // выделяем память под всю структуру
    elem = malloc(sizeof(part_s) + part_s.n * sizeof(double));
    if (!elem)
        return NULL;

    // сохраняем уже прочитанную фиксированную часть
    memmove(elem, &part_s, sizeof(part_s));

    // чтение самого массива
    size = fread(elem->d, sizeof(double), elem->n, f);
    if (size != elem->n)
    {
        free(elem);
        return NULL;
    }
    return elem;
}
```

/\*

Если вдруг захочется посмотреть, как устроена память в проге, то нужно будет вызвать компиляцию с ключом -ggdb, потом запустить прогу, получится файл с расширением dat, в котором будет лежать примерно такая таблица (но надеюсь, вам не захочется)

```
C:\msys64\home\Dell\2021_winter\09_10\src_06_1\test.dat
0000000000: 03 00 00 00 00 00 00 00 00 00 00 F0 3F  ▼  @  0?
0000000010: 00 00 00 00 00 00 00 40 00 00 00 08 40  @  0?
0000000020: 01 00 00 00 00 00 00 00 00 00 00 F0 3F  @  0?
0000000030: 00 00 00 00 00 00 00 00 05 00 00 00 00 00  +  @
0000000040: 00 00 00 00 00 00 F0 3F 00 00 00 00 00 40  0?  @
0000000050: 00 00 00 00 00 00 08 40 00 00 00 00 10 40  0?  @
0000000060: 00 00 00 00 00 00 14 40 00 00 00 00 00 00  0?  @
```

\*/



## Как люди работали со структурами переменной длины до 99 стандарта

```
struct s
{
    int n;
    double d[1];
};
```

В структуру помещался массив размера 1 (минимальный размер, т.к. массив нулевого размера объявить нельзя). При расчете памяти для создания структуры нужно учитывать, что один элемент в структуре уже есть.

Алгоритм создания структуры:

```
struct s* create_s(int n, const double *d)
{
    assert(n >= 0);

    struct s *elem = calloc(sizeof(struct s) +
                             (n > 1 ? (n - 1) * sizeof(double) : 0), 1);

    if (elem)
    {
        elem->n = n;
        memmove(elem->d, d, n * sizeof(double));
    }

    return elem;
}
```

Если верить Ричи, то такая конструкция никогда в языке си не задумывалась. Она появилась случайно. Она с одной стороны не противоречит синтаксису языка, с другой более менее правильно работает, поэтому почему бы и нет.

## Чем лучше использование поля гибкий массив по сравнению с обычным указателем:

- Экономия памяти (если мы заводим указатель, то это дополнительные 4 или 8 байт)
- Локальность данных data locality (программа “ждет” обращение в близкую область памяти после очередного обращения)
- Атомарность выделения памяти (в случае гибкого массива для структуры память будет выделяться один раз)
- Не требует “глубокого” копирования и освобождения



## 9. (48-57) Динамически расширяемый массив

Массив - агрегированный тип данных, обладающий следующими свойствами:

- линейность (все элементы расположены по порядку, единым блоком)
- однородность (все элементы одного типа и одного объема)
- произвольный доступ (скорость доступа к любому элементу не зависит от его расположения)

Динамически расширяемый массив - массив, размер которого автоматически увеличивается в течение выполнения программы.

// здесь обязательно нужно сначала рассказать про функцию [realloc](#) и про ее основные ошибки

Для уменьшения потерь при распределении памяти изменение размера должно происходить достаточно крупными блоками (решение [второй ошибки](#)).

Массив будет описываться тремя сущностями - сам массив, количество элементов в нем и фактический размер массива (размер выделенной под него области памяти). Поэтому для представления массива будет использоваться структура.

```
struct dyn_array
{
    int len;          // кол-во элементов
    int allocated;    // кол-во выделенной памяти
    int step;         // коэффициент для очередного перевыделения
    int *data;        // сам массив
};
```

Функция начальной инициализации структуры:

```
void init_dyn_array(struct dyn_array *d)
{
    d->len = 0;
    d->allocated = 0;
    d->step = 2;
    d->data = NULL;
}
```



Добавление элемента (можно переписать так, чтобы использовать realloc и в случае, когда массив не существует)

```
int append(struct dyn_array *d, int item)
{
    if (!d->data)           // создан ли массив
    {
        d->data = malloc(INIT_SIZE * sizeof(int));
        if (!d->data)
            return -1;
        d->allocated = INIT_SIZE;
    }
    else
        if (d->len >= d->allocated)    // хватит ли места
        {
            int *tmp = realloc(d->data,
                               d->allocated * d->step * sizeof(int));
            if (!tmp)
                return -1;
            d->data = tmp;
            d->allocated *= d->step;
        }
    d->data[d->len] = item;
    d->len++;
    return 0;
}
```

Важные особенности:

- Удвоение размера массива при каждом вызове realloc сохраняет средние “ожидаемые” затраты на копирование элемента
- Поскольку адрес элемента при очередном вызове realloc может измениться программа должна обращаться к элементам массива по индексам (например, случаи, когда нам нужно запоминать позиции элементов)
- Имеет смысл на начальном этапе делать маленький размер INIT\_SIZE (например, 1). В этом случае будет просто сразу проверить код, который реализует выделение памяти. Позже значение этой константы можно заменить на более подходящее для задачи.



## Удаление элемента

```
int delete(struct dyn_array *d, int index)
{
    if (index < 0 || index >= d->len)
        return -1;

    // сдвиг элементов, которые расположены за удаляемым на один вперед
    memmove(d->data + index, d->data + index + 1, (d->len - index - 1) * sizeof(int));
    d->len--;

    return 0;
}
```

Важные особенности этой функции:

- Если нам не важен порядок элементов в массиве, то можно просто обменять удаляемый элемент с последним и уменьшить счетчик количества элементов на один.
- Второй вариант ставить себе где-то флажки, что данный элемент удален
- В функции используется `memmove`, а не `memcpy`, так как несмотря на то, что `memcpy` быстрее, только `memmove` умеет безопасно копировать перекрывающиеся области. `for` не использовали просто потому что.

Достоинства и недостатки массивов:

- + Просто использовать, минимальные накладные расходы (для работы с массивом нужно знать только, где этот массив находится в памяти и какое количество элементов в нем находится, + размер памяти, если массив динамический)
- + Константное время доступа к любому элементу
- + Не тратятся лишние ресурсы
- + Хорошо сочетается с двоичным поиском
- Хранение меняющегося набора значений (сохранение порядка элементов дорого обходится)





## 10. (58-64) Односвязный список

**Связный список** - это набор элементов, причем каждый из них является частью узла, который также содержит ссылку на следующий и/или предыдущий узел.

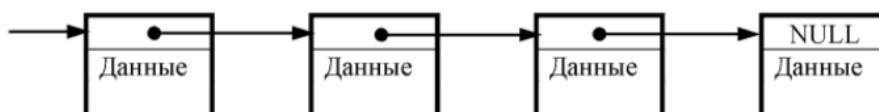
Связный список, как и массив, хранит набор элементов одного типа, но использует абсолютно другую стратегию выделения памяти: память под каждый элемент выделяется отдельно и лишь тогда, когда это нужно. (Седжвик)

(То есть однородность да, линейность и произвольный доступ – нет)

**Узел** - единица хранения данных, несущая в себе ссылки на связанные с ней узлы. *(эх, не оценит Костер такое рекурсивное определение)*

Узел обычно состоит из двух частей:

- информационная часть (данные)
- ссылочная часть (связь с другими узлами)



**Преимущества и недостатки связного списка по сравнению с массивом:**

- + Возможность эффективного изменения расположения элементов
- Скорость доступа к произвольному элементу (единственный способ получить доступ к элементу - это отследить все связи от начала списка)

**Линейный односвязный список** - структура данных, состоящая из узлов, каждый из которых ссылается на следующий узел списка.





Особенности:

- Узел, на который нет указателя является первым элементом списка.  
Обычно этот узел называют головой списка
- Последний элемент списка никуда не ссылается (ссылается на NULL).  
Обычно этот узел называют хвостом списка
- Передвигаться по такому списку можно только от головы к хвосту
- Находясь в  $i$ -том узле списка, мы знаем, где находится следующий узел, но не знаем где находится предыдущий

Базовые операции над линейными списками:

- Добавить элемент в начало или конец
- Найти указанный элемент
- Добавить новый элемент до/после указанного
- Удалить элемент

Описание типа (элемент списка):

```
typedef struct node_t
{
    struct node_t *next;
    void *data;
} node_t;
```

Функция создания узла списка:

```
node_t *create_node(void *data)
{
    node_t *new_node = malloc(sizeof(node_t));
    if(!new_node)
        return NULL;

    new_node->next = NULL;
    new_node->data = data;

    return new_node;
}
```

...Запланировано обновление)



## 11.(65-71) Двоичное дерево поиска

### Двоичное дерево поиска

**Дерево** - связный (в графе есть путь между любыми двумя вершинами) ациклический (без циклов) граф.

**Двоичное дерево поиска** - все вершины которого упорядочены, каждая вершина имеет не более 2х потомков (назовем их левыми и правыми), и все вершины, кроме корня, имеют родителя.

**Отношение порядка:** все вершины левого поддерева относительно некоей вершины будут меньше, чем значение в рассматриваемой вершине, значения в вершинах правого поддерева - больше.

### Описание типа

**Отношение порядка:** все вершины левого поддерева относительно некоей вершины будут меньше, чем значение в рассматриваемой вершине, значения в вершинах правого поддерева - больше.

### Базовые операции над деревом:

- добавление узла
- поиск узла
- удаление узла
- обход дерева

### Добавление элемента

Элемент (узел) дерева состоит из информационной и ссылочной частей.

1. Дерево пустое
  - Выделить память под узел, который станет корнем
2. Дерево не пустое
  - Выделить память под узел
  - Найти место для узла в дереве (согласно отношению порядка)
  - Добавить узел в дерево

```
struct tree_node* insert(struct tree_node *tree, struct tree_node *node)
```



```
{
    int cmp;

    if (tree == NULL)
        return node;

    cmp = strcmp(node->name, tree->name);
    if (cmp == 0)
        assert(0);
    else if (cmp < 0)
        tree->left = insert(tree->left, node);
    else
        tree->right = insert(tree->right, node);

    return tree;
}
```

Уже существующий в дереве элемент добавить не выйдет.

(тут он рассказывает про сбалансированные деревья: что они помогают решать проблему того, что если поступают упорядоченные данные и дерево вытягиваются в кишку, сбалансированное дерево это отрегулирует)

### Поиск элемента (рекурсивный и нерекурсивный варианты)

Поскольку алгоритмы над деревьями основаны на отношении порядка, все они похожи как “близнецы братья”.

Рекурсия - красиво и удобно, но затратно с точки зрения производительности.

Потому что тратится большое количество памяти на очередной рекурсивный вызов. Здесь - хвостовая рекурсия, поэтому ее легко можно преобразовать в итерацию.

#### 1. Рекурсивный вариант

```
struct tree_node* lookup_1(struct tree_node *tree, const char *name)
{
    int cmp;

    if (tree == NULL)
        return NULL;

    cmp = strcmp(name, tree->name);
    if (cmp == 0)
        return tree;
    else if (cmp < 0)
        return lookup_1(tree->left, name);
    else
        return lookup_1(tree->right, name);
}
```

#### 2. Нерекурсивный вариант

```
struct tree_node* lookup_2(struct tree_node *tree, const char *name)
{
    int cmp;

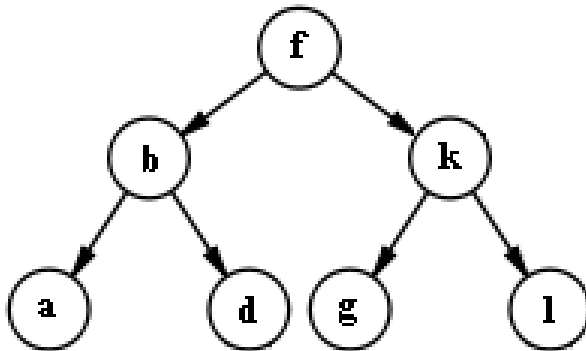
    while (tree != NULL)
    {
        cmp = strcmp(name, tree->name);
        if (cmp == 0)
            return tree;
        else if (cmp < 0)
            tree = tree->left;
    }
}
```



```
        else
            tree = tree->right;
    }

    return NULL;
}
```

## Обход дерева



- Прямой (pre-order) - начиная от корня, для каждого узла сначала влево, потом вправо  
– **f b a d k g l**
- Фланговый или поперечный (in-order) - начиная от левого узла  
– **a b d f g k l**
- Обратный (post-order) - левое поддереву корня, правое поддереву корня  
– **a d b g l k f**

Самый непопулярный вариант. Используется, например, для создания копии дерева.

Самый популярный вариант. Возвращает упорядоченное значение вершин.

Используется, когда нужна полная информация о левом и о правом поддеревьях. Например, надо определить высоту дерева.

```
void apply(struct tree_node *tree,
           void (*f)(struct tree_node*, void*), void *arg)
{
    if (tree == NULL)
        return;

    // pre-order
    // f(tree, arg);
    apply(tree->left, f, arg);
    // in-order
    f(tree, arg);
    apply(tree->right, f, arg);
    // post-order
    // f(tree, arg);
}
```



// Тут странная солянка всех обходов от Лома, возможно нужно вставить 3 функции обхода

## Освобождение памяти из-под всего дерева

Освобождение узла списка.

```
void node_free(struct tree_node_t *tree, void *param)
{
    free(node);
}
```

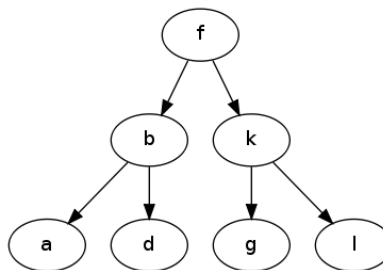
Эта функция посылается в функцию apply, чтобы освободить память из-под всего дерева. Применимо только при обратном обходе дерева.

## Язык DOT. Примеры использования. Утилита Graphviz.

DOT - язык описания графов.

- Граф, описанный на языке DOT, обычно представляет собой текстовый файл с расширением .gv в понятном для человека и обрабатывающей программы формате.
- В графическом виде графы, описанные на языке DOT, представляются с помощью специальных программ, например Graphviz.

```
// Описание дерева на
DOT
digraph test_tree {
f -> b;
f -> k;
b -> a;
b -> d;
k -> g;
k -> l;
}
```



Функции перевода в язык DOT

```
void to_dot(struct tree_node *tree, void *param)
{
    FILE *f = param;

    if (tree->left)
        fprintf(f, "%s -> %s;\n", tree->name, tree->left->name);

    if (tree->right)
        fprintf(f, "%s -> %s;\n", tree->name, tree->right->name);
}

void export_to_dot(FILE *f, const char *tree_name,
                  struct tree_node *tree)
{
    fprintf(f, "digraph %s {\n", tree_name);
```



```
    apply_pre(tree, to_dot, f);  
  
    fprintf(f, "}\n");  
}
```

## 12. (72-89) Области видимости

### Что такое область видимости имени?

*Область видимости (scope) имени* – это часть текста программы, в пределах которой имя может быть использовано.

### Какие области видимости есть в языке Си? Приведите примеры.

В языке Си выделяют следующие области видимости

- блок - (только?) переменные обладают этой областью видимости
- Переменная, определенная внутри блока, имеет область видимости в пределах блока.
- Формальные параметры функции имеют в качестве области видимости блок, составляющий тело функции.

```
double f(double a)  // начало области видимости переменной a  
{  
    double b;        // начало области видимости переменной b  
    ...  
    return b;  
}                    // конец области видимости переменных a и b
```

- файл - любое имя, описанное вне функции, имеет область видимости файл;
- Область видимости в пределах файла имеют имена, описанные за пределами какой бы то ни было функции.
- Переменная с областью видимости в пределах файла видна на протяжении от точки ее описания и до конца файла, содержащего это определение.
- Имя функции всегда имеет файловую область видимости.

```
#include <stdio.h>  
  
int max; // область видимости - файл  
  
void f(void)  
{  
    printf("%d\n", max);  
}  
  
int main(void)  
{  
    max = 5;  
    f();  
    ...  
}
```





- функция - метки (goto);
- Метки - это единственные идентификаторы, область действия которых - функция.
- Метки видны из любого места функции, в которой они описаны.
- В пределах функции имена меток должны быть уникальными.

(у нас на кафедре goto не любят, да и вы морщитесь, когда я про него вспоминаю, поэтому никакого примера нет)

```
int main()
{
    int i, j;

    for (i = 0; i < 10; i++)
    {
        printf("%d\n", i);
        if ( i == 5 )
            goto stop;
    }
    stop: printf( "Jumped to stop. i = %d\n", i );
}
```

- прототип функции - самая маленькая область видимости.

Область видимости в пределах прототипа функции применяется к именам переменных, которые используются в прототипах функций.

```
int f(int i, double d);
```

Область видимости в пределах прототипа функции простирается от точки, в которой объявлена переменная, до конца объявления прототипа.

```
int f(int, double);          // ок
int f(int i, double i);     // ошибка компиляции (имена должны быть уникальны)
```

**Какие правила перекрытия областей видимости есть в языке Си? Приведите примеры.**

*1 правило:*

Переменные, определенные внутри некоторого блока, будут доступны из всех блоков, вложенных в данный.

```
{
    int a = 1;
    ...
    {
        int b = 2;
        ...
    }
}
```



```
    printf("%d %d\n", a, b); // ок
}

printf("%d %d\n", a, b); // ошибка компиляции
}
```

*2 правило:*

Возможно определить в одном из вложенных блоков переменную с именем, совпадающим с именем одной из "внешних" переменных.

```
{
    int a = 1;

    {
        double a = 2.0;
        printf("%g\n", a);          // 2
    }

    printf("%d\n", a);              // 1
}
```

### Что такое блок?

В языке Си *блоком* считается последовательность объявлений, определений и операторов, заключенная в фигурные скобки

Блоки могут включать в себя составные операторы, но не определения функций.

### Какие виды блоков есть в языке Си?

Существуют два вида блоков:

- составной оператор;
- определение функции.

Операторы могут вкладываться друг в друга, а функции - нет. Функцию внутри другой функции описать нельзя, поэтому функции никогда не обладают областью видимости блок. Ей в СИ обладают только переменные.

### 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?

*Время жизни (storage duration)* – интервал времени выполнения программы, в течение которого «программный объект» существует.

В языке Си время жизни «программного объекта» делится на три категории

- глобальное (по стандарту - статическое (англ. static));
  - Если «программный объект» имеет глобальное время жизни, он существует на протяжении выполнения всей программы.
  - Примерами таких «программных объектов» могут быть функции и переменные, определенные вне каких либо функций.



- локальное (по стандарту - автоматическое (англ. automatic));
  - Локальным временем жизни обладают «программные объекты», область видимости которых ограничена блоком.
  - Такие объекты создаются при каждом входе в блок, где они определяются. Они уничтожаются при выходе из этого «родительского» блока.
  - Примерами таких переменных являются локальные переменные и параметры функций.
- динамическое (по стандарту - выделенное (англ. allocated))
  - Время жизни «выделенных» объектов длится с момента выделения памяти и заканчивается в момент ее освобождения.
  - В Си нет переменных, обладающих динамическим временем жизни.
  - Динамическое выделение выполняется программистом «вручную» с помощью соответствующих функций. Единственный способ «добраться» до выделенной динамической памяти – использование указателей.

### **Какие виды времени жизни есть у переменных?**

Глобальное (переменная определена вне функции) и локальное. Динамическое - никогда!

### **Какие виды времени жизни есть у функций?**

Только глобальное.

### **Как время жизни влияет на область памяти, в которой располагается программный объект?**

#### **Что такое связывание?**

*Связывание (linkage)* определяет область программы (функция, файл, вся программа целиком), в которой «программный объект» может быть доступен другим функциям программы.

### **Какие виды связывания есть в языке Си?**

Стандарт языка Си определяет три формы связывания:

- внешнее (external) - объект доступен во всей программе (можно обратиться из любой функции, даже если объект определен в другой файле);
- внутреннее (internal) - к объекту могут иметь доступ все функции только из файла, в котором он определен;
- никакое (none) - объект принадлежит одной и функции и вообще не может раздаться другими функциями.

### **Как связывание влияет на "свойства" объектного/исполняемого файла? Что это за "свойства"?**

Объектный файл включает машинный код, имена, которые требуются файлу и которые он предоставляет наружу. Два последних составляют таблицу символов. Связывание влияет на те свойства, которыми имена будут обладать в этой таблице символов.

### **Какими характеристиками (область видимости, время жизни, связывание) обладает переменная в зависимости от места своего определения?**

Время жизни, область видимости и связывание переменной зависят от места ее



определения. По умолчанию

```
int i; // глобальная переменная
```

- Глобальное время жизни
- Файловая область видимости
- Внешнее связывание

```
{  
    int i;    // локальная переменная  
    ...
```

- Локальное время жизни
- Видимость в блоке
- Отсутствие связывания

Чтобы изменить свойства по умолчанию - классы памяти.

**Какими характеристиками (область видимости, время жизни, связывание) обладает функция в зависимости от места своего определения?**

### 13. (90-107) Классы памяти

**Какие недостатки есть у использования глобальных переменных.**

Недостатки использования функций, которые используют глобальные переменные:

- Если глобальная переменная получает неверное значение, трудно понять какая функция работает неправильно.
- Изменение глобальной переменной требует проверки правильности работы всех функций, которые ее используют.
- Функции, которые используют глобальные переменные, трудно использовать в других программах.
- Увеличивается связность в программе (чаще ни к чему хорошему не приводит).

### 14. (107-114) Объектный файл

**Объектный файл, секции, таблица символов**

**Объектный файл** представляют собой блоки машинного кода и данных с неопределенными адресами ссылок на данные и подпрограммы в других объектных модулях, а также список своих подпрограмм и данных.



Объектный файл состоит из секций, который содержит разные данные в широком смысле этого слова:

- заголовки (метаинформация, необходимая для организации самого файла);
- код (.text);
- данные (.data, .rodata, .bss);
- таблицу символов (.symtab);

Буква	Расположение
B, b	Секция неинициализированных данных (.bss).
D, d	Секция инициализированных данных (.data).
R, r	Секция данных только для чтения (.rodata)
T, t	Секция кода (.text)
U	Символ не определен, но ожидается, что он появится.

Строчные буквы означают «локальные» символы – используются только внутри объектного файла и наружу не экспортируются, «заглавные» - внешние (глобальные) – экспортируются из объектного файла.

Секции:

.text – машинный код

.data – глобальные инициализированные данные

.bss - глобальные неинициализированные данные

.rodata - константы

### Что делает компоновщик?

Этапы до компоновки: препроцессинг – трансляция с языка си на язык ассемблера – с языка ассемблера в машинный код, т.е. на этом этапе мы получили программу в виде набора объектных файлов, в каждом из которых есть набор секций.

Далее компоновка. Что делает компоновщик:

1. Перемещение. Склеивание однотипных секций в одну.
2. Разрешение ссылок.
  - Поиск кода, который вызывает что-то за пределами своей исходной секции.
  - Поиск места, где теперь располагается вызываемый код.
  - Замена «поддельного» адреса на настоящий.

(Ссылки внутри каждого объектного файла уже разрешены)



*Пример:* функция мэйн вызывает функцию а, которая в этом объектном файле отсутствует. На первом шаге компоновщик получает секцию .text, тут он уже знает, где внутри исполняемого файла располагается функция а. Тогда в мэйн вместо а компоновщик поставит адрес, по которому располагается функция.

### **Журналирование, подходы к реализации.**

Журналирование – в процессе выполнения программы необходимо информацию о ее выполнении необходимо записывать (в журнал).

#### *1 подход:*

Есть глобальная файловая переменная, функция открывания журнала и его закрытия. (Файловая переменная беззащитная, кто-то может изменить файл).

#### *2 подход:*

Есть статическая файловая переменная (к ней нет доступа из других файлов проекта). Но тогда мы лишаем себя возможности записывать что-то в журнал, поэтому надо создать функцию, которая возвращает значение файловой переменной. (Если класс статик, то по имени к этой переменной из других файлов нельзя обратиться по имени, из-за внутреннего связывания, но никто не запрещает написать функцию, которая возвращает адрес переменной. После этого можно работать с ней и из других файлов.)

#### *3 подход:*

Есть статическая файловая переменная также, но можно обойтись без функции, возвращающей файловую переменную, если в журнал пишутся только строки. Вместо этой функции можно написать функцию, которая будет сохранять в файл строку. И мы никак не будем соприкасаться с файловой переменной. Этот вариант плох тем, что при использовании журнала мы будем вынуждены сначала сформировать строку, а потом поместить ее в журнал. Тогда журналирование будет требовать гораздо больше и времени, и ресурса.

#### *4 подход:*

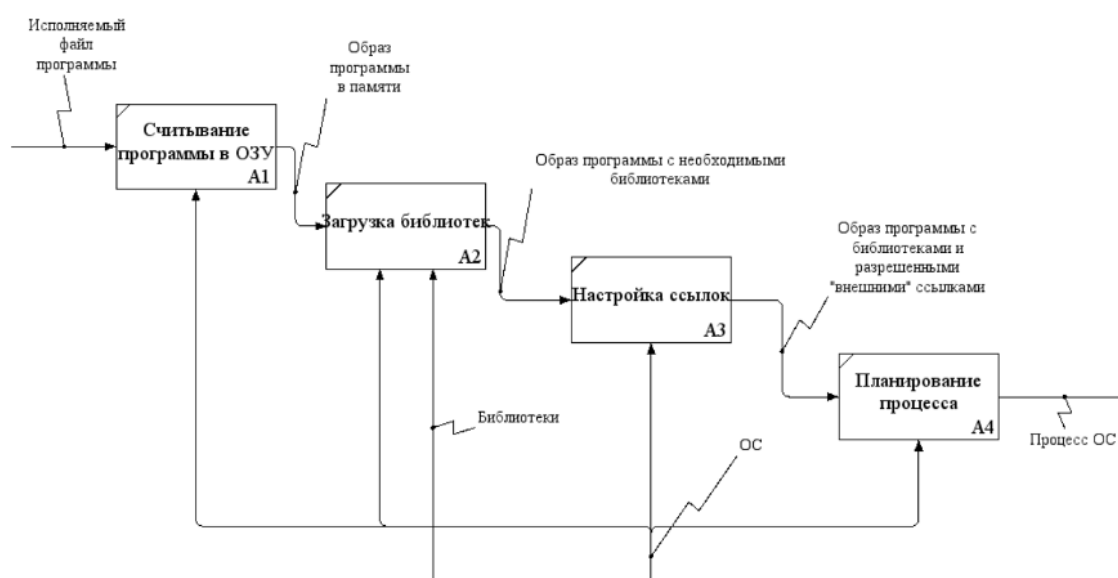
Есть статическая файловая переменная также, а строки записываются в журнал через функцию, которая работает по принципу printf – функция с переменным числом параметров, ее работа управляется строкой форматирования.

### **Процесс запуска программы («превращения в процесс»).**

Формат исполняемого файла: заголовки, секции, таблица импорта.



Таблица импорта нужна для того, чтобы можно было использовать функции из динамической библиотеки.

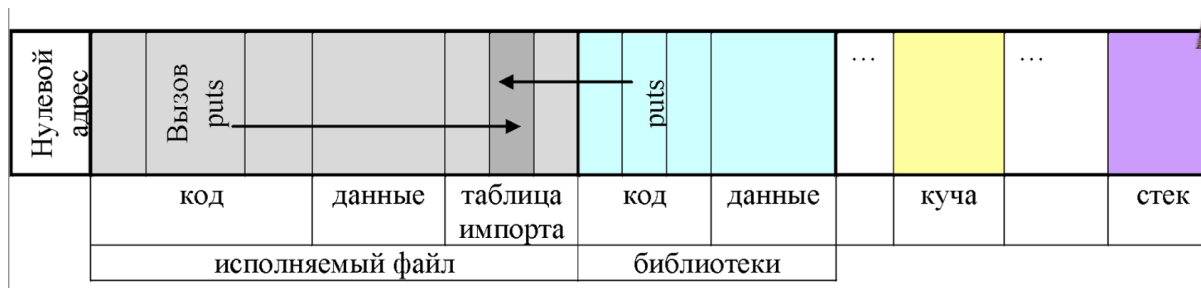


### Абстрактное адресное пространство программы.

Когда загрузчик загружает программу в память компьютера, она попадает в абстрактную память. Программе кажется, что в этой памяти располагается только она и код библиотек, которые ей нужны. В этой памяти гарантируется что ни одна функция и ни одна переменная не может располагаться по нулевому адресу. Это помогает ввести понятие нулевого адреса и помогает отслеживать обращение по неправильным адресам памяти.

Таблица импорта – ассоциативный массив (индекс – имя функции, элемент – адрес этой функции)

Загрузчик: загружает программу -> загружает библиотеку -> смотрит в таблицу импорта и понимает, какая функция нужна программе -> зная, где эта функция в библиотеке, вставляет в соответствующую ячейку таблицы импорта вставляет адрес.



### Опишите достоинства и недостатки локальных переменных.

Для хранения локальных переменных используется так называемая автоматическая память.

+

- Память под локальные переменные выделяет и освобождает компилятор

-

- Время жизни локальной переменной "ограничено" блоком, в котором она определена.
- Размер размещаемых в автоматической памяти объектов должен быть известен на этапе компиляции.
- Размер автоматической памяти в большинстве случаев ограничен.

**Локальные переменные создаются в так называемой «автоматической памяти».**

**Почему эта память так называется?**

Автоматическая память – память, выделением и освобождением которой управляет компилятор. (плюсы и минусы автоматической памяти по факту совпадают с 112 вопросом)

```
void f_1(int a)
{
    char b;
    // ...
}
void f_2(double c)
{
    int d = 1;
    f_1(d);
    // ...
}
int main(void)
{
    double e = 1.0;
    f_2(e);
    // ...
}
```





1. **Вызов main**
2. Создание e
3. **Вызов f\_2**
4. Создание c
5. Создание d
6. **Вызов f\_1**
7. Создание a
8. Создание b
9. **Завершение f\_1**
10. Разрушение b
11. Разрушение a
12. **Завершение f\_2**
13. Разрушение d
14. Разрушение c
15. **Завершение main**
16. Разрушение e

## 15. (114-125) Стек и куча

### Для чего в программе используется аппаратный стек?

1. Вызов функций.

Машинная команда call (name)

- Поместить в стек адрес команды, следующей за командой call
- Передать управление по адресу метке call

2. Возврат управления из функций

Машинная команда get

- Извлекает адрес возврата из стека
- Передает управление по адресу

3. Передача параметров в функцию

*Соглашение о вызове*

- расположение входных данных;
- порядок передачи параметров;
- какая из сторон очищает стек и т.п.

*cdecl*

- аргументы передаются через стек, справа налево;
- очистку стека производит вызывающая сторона;
- результат функции возвращается через регистр EAX

(но возвращаемый объект может быть очень большой и не помещаться в регистр, тогда появляются разные детали - нас детали не особо волнуют в этом курсе)

4. Выделение и освобождение памяти под локальные переменные

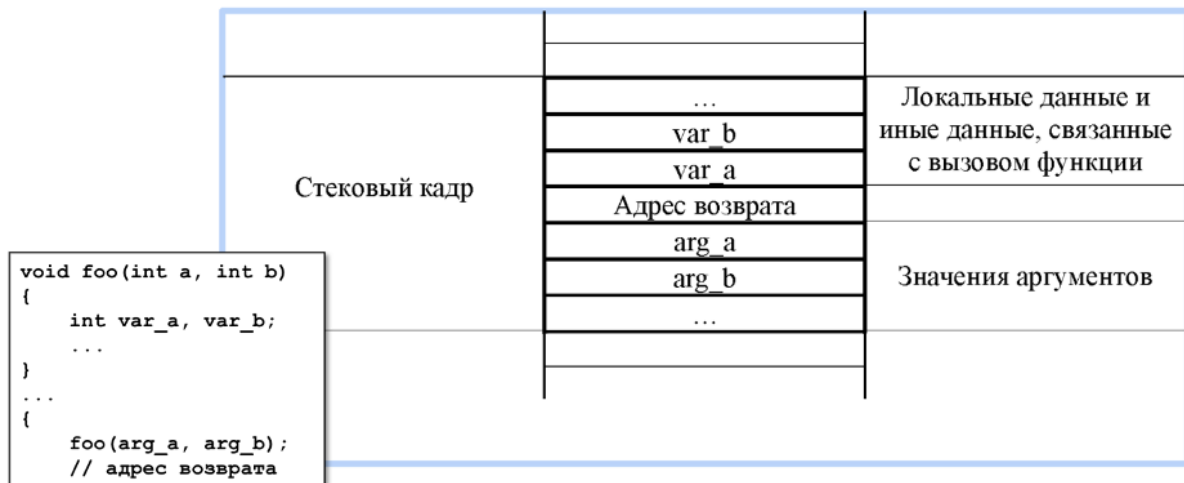


Аппаратный стек растет от старших адресов к младшим.

### Что такое кадр стека?

Стековый кадр (фрейм) – механизм передачи аргументов и выделения временной памяти с использованием аппаратного стека.

“Структура, благодаря которой можно передать параметры в функцию и выделить память под локальные переменные”



В стековом кадре размещаются:

- значения фактических аргументов функции;
- адрес возврата;
- локальные переменные;
- иные данные, связанные с вызовом функции.

**Для чего в программе используется кадр стека? Приведите примеры.**



C (gcc 4.8, C11) ( <a href="#">known limitations</a> )		Stack	Heap									
<pre>1 int sum(int x, int y) 2 { 3     int s = x + y; 4 5     return s; 6 } 7 8 void foo(void) 9 { 10     int a = 1, b = 5; 11     int s = sum(a, b); 12 } 13 14 int main(void) 15 { 16     foo(); 17 18     return 0; 19 }</pre>		<div>main</div> <div>foo</div> <table><tr><td>a</td><td>int</td><td>1</td></tr><tr><td>b</td><td>int</td><td>5</td></tr><tr><td>s</td><td>int</td><td>?</td></tr></table>	a	int	1	b	int	5	s	int	?	
a	int	1										
b	int	5										
s	int	?										
		<div>sum</div> <table><tr><td>x</td><td>int</td><td>1</td></tr><tr><td>y</td><td>int</td><td>5</td></tr><tr><td>s</td><td>int</td><td>6</td></tr></table>	x	int	1	y	int	5	s	int	6	
x	int	1										
y	int	5										
s	int	6										

На картинке три кадра стека.

При вызове функции выделяется специальная область памяти – кадр стека, где размещаются параметры функции и ее переменные. + см предыдущий вопрос

**Какие преимущества и недостатки есть у использования кадра стека?**

+

- Удобство и простота использования.

-

- Производительность
- Передача данных через память без необходимости замедляет выполнение программы.
- Безопасность
- Стековый кадр перемещает данные приложения с критическими данными - указателями, значениями регистров и адресами возврата. (с этим начинают бороться - заводить 2 стека: 1 для параметров, 2 – для адресов возврата)

**Что такое соглашение о вызове?**

Соглашение о вызове – договоренности по типу: когда одна функция вызывает другую, нужно договориться, как одна функция другой будет передавать параметры, и как возвращаться значения.

*Соглашение о вызове:*



- расположение входных данных;
- порядок передачи параметров;
- какая из сторон очищает стек;
- etc

### Какое соглашение о вызове используется в языке Си? В чем оно заключается?

*cdecl*

- аргументы передаются через стек, справа налево;
- очистку стека производит вызывающая сторона;
- результат функции возвращается через регистр EAX (но возвращаемый объект может быть очень большой и не помещаться в регистр, тогда появляются разные детали - нас детали не особо волнуют в этом курсе)

### Что такое переполнение буфера? Чем оно опасно?

Переполнение буфера - явление, возникающее, когда запись данных происходит за пределами выделенного буфера памяти.

Опасно, например, тем, что память под локальные переменные выделяется на стеке. В стеке есть кадр функции. Переполнив локальную переменную, есть вероятность затереть адрес возврата функции ("невеселые результаты").

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

int main(int argc, char **argv)
{
    char str[16];

    if (argc < 2)
        return 1;

    sprintf(str, "Hello, %s!", argv[1]);
    printf("%s (%d)\n", str, strlen(str));
    return 0;
}
```

### Почему нельзя из функции возвращать указатель на локальную переменную, определенную в этой функции?

Потому что переменная будет уничтожена после выхода из функции.

Пример:

```
#include <stdio.h>
char* make_greeting(const char *name)
{
    char str[64];

    snprintf(str, sizeof(str),
```



```
        "Hello, %s!", name);

    return str;
}

int main(void)
{
    char *msg =
make_greeting("Petya");

    printf("%s\n",
msg);

    return 0;
}
```

Либо программа упадет сразу, либо вернет неадекватный результат, либо будет работать на одной машине, а на другой - нет.

### **Для чего в программе используется куча?**

#### **Происхождение термина «куча».**

Существует структура данных – куча. Она не имеет никакого отношения к нашей куче. Термин куча ввел Кнут (1975). Куча стал использоваться как противопоставление термину стек. LIFO – стек, стопка тарелок. Куча предметов без дисциплины.

#### **Свойства области памяти, которая выделяется динамически.**

В языке си нельзя динамически создать переменную. Можно динамически выделить память и с этой областью памяти связать указатель.

Для хранения данных используется «куча». Создать переменную в «куче» нельзя, но можно выделить память под нее.

+

Все «минусы» локальных переменных:

- Время жизни области не связано с блоком, в котором она выделена (выделить можно в одной функции, освободить - в другой)
- Размер области памяти не обязательно должен быть известен на этапе компиляции
- Размер динамически выделяемых областей памяти может быть на порядок больше, чем размер переменных, которые можно создать на стеке

-



- Ручное управление временем жизни.

### Как организована куча?

“Куча предметов без дисциплины”

## 16. (126-132) Выделение динамической памяти

### Алгоритм работы функции `malloc`.

*Выделение области памяти (`malloc`)*

- Просмотреть список занятых/свободных областей памяти в поисках свободной области подходящего размера.
- Если область имеет точно такой размер, как запрашивается, пометить найденную область как занятую и вернуть указатель на начало области памяти.
- Если область имеет больший размер, разделить ее на части, одна из которых будет занята (выделена), а другая останется в свободной.
- Если область не найдена, вернуть нулевой указатель.

*Свойства функции (`malloc`)*

- `malloc` выделяет по крайней мере указанное количество байт (меньше нельзя, больше можно).
- Указатель, возвращенный `malloc`, указывает на выделенную область (т.е. область, в которую программа может писать и из которой может читать данные).
- Ни один другой вызов `malloc` не может выделить эту область или ее часть, если только она не была освобождена с помощью `free`.

Нужна метаинформация - информация о выделенных и свободных областях памяти. Для этого используется список, в узле которого хранятся размер, состояние и указатель на следующую область.

### Алгоритм работы функции `free`.

*Освобождение области памяти (`free`)*

- Просмотреть список занятых/свободных областей памяти в поисках указанной области.
- Пометить найденную область как свободную.
- Если освобожденная область вплотную граничит со свободной областью с какой-либо из двух сторон, то объединить их в единую область большего размера.

```
void my_free(void *ptr)
{
    if (my_heap <= (char*) ptr && (char*)ptr <= my_heap + sizeof(my_heap))
    {
        struct block_t *cur = ptr;

        --cur;
        cur->free = 1;

        merge_blocks();
    }
}
```



```
    }  
    else  
        printf("Wrong pointer\n");  
}
```

**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.**

17. (132-140) Массивы переменной длины

pass

18. (141-142) alloca

pass

19. (143-150) Функции с переменным числом параметров

**Можно ли реализовать в языке Си функцию со следующим прототипом `int f( . . . )` ? Почему?**

Для передачи параметров в функцию используется стек. Если `int f(...)`, мы не сможем добраться до параметров функции. Поэтому нужен хотя бы 1 явный параметр `int f(int k, ...)` для того, чтобы узнать адрес, по которому располагается первый и остальные параметры.

- Во время компиляции компилятору не известны ни количество параметров, ни их типы.
- Во время компиляции компилятор не выполняет никаких проверок.

Но список параметров функции с переменным числом аргументов совсем пустым быть не может.

```
int f(int k, ...);
```

**Покажите идею реализации функций с переменным числом параметров.**



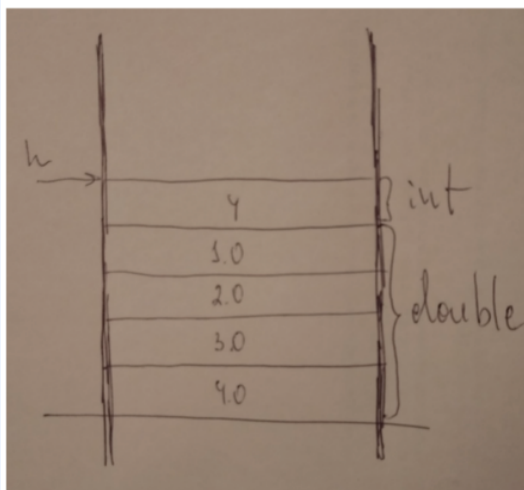
```
#include <stdio.h>

double avg(int n, ...)
{
    ...
}

int main(void)
{
    double a =
        avg(4, 1.0, 2.0, 3.0, 4.0);

    printf("a = %5.2f\n", a);

    return 0;
}
```



**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?**

Заголовочный файл *stdarg.h*

Тип *va\_list* – переменная этого типа связывается со списком с переменным числом параметров.

Макросы:

- `void va_start(va_list argptr, last_param)` – инициализация переменной
- `type va_arg(va_list argptr, type)` – получение очередного значения типа `type` из списка с переменным числом параметров
- `void va_end(va_list argptr)` – окончание работы с переменной типа `va_list`, когда в ней нет необходимости

**Какая особенность языка Си упрощает реализацию функций (с точки зрения компилятора) с переменным числом параметров?**

Компилятор не отслеживает количество и типы приходящих значений?

**Почему при вызове `va_arg(argp, short int)` (или `va_arg(argp, float)`) выдается предупреждение?**

«Повышение типа аргумента по умолчанию» - согласно Хабру (default argument promotion). Это значит, что если у аргумента тип `char`, `short` (со знаком или без) или `float`, то к соответствующим параметрам надо обращаться как к `int`, `int` (со знаком или без) или `double`. Иначе — неопределенное поведение [C11].





**Какая "опасность" существует при использовании функций с переменным числом параметров?**

Зависит от компилятора и платформы.

Ответственность за количество аргументов, переданных в функцию, лежит на программисте.

Также если в функции ожидаются, например, переменные типа double, а пришел int, но компилятор не сможет это отследить.

**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.)**

20. (151-169, 173, 174) Препроцессор

pass

21. (171, 172) inline функции

pass

22. (175-191) Библиотеки

pass

23. (192-194) Python + Си

pass

24. (195-201) Побочные эффекты

pass

25. (202-208) Неопределенное поведение

pass

26. (209-212, 217) Модуль

pass

27. (213-223) Абстрактный тип данных

pass

