1. Язык программирование Си.

2. Этапы получения исполняемого файла из исходного кода. Опции компилятора и компоновщика.

3. Исполняемый файл. Этапы запуска исполняемого файла. Функция main.

4. Переменные, операция присваивания, ввод/вывод значений переменных.

5. Операции и выражения.

6. Оператор-выражение, условный оператор и условная операция, составной оператор, оператор switch.

7. Операторы цикла (while, do-while, for), операция запятая.

8. Операторы break, continue, goto. Пустой оператор.

9. Функции.

10. Автоматизация сборки проекта, утилита make. Сценарий сборки проекта. Простой сценарий сборки. Использование переменных и комментариев. Сборка программы с разными параметрами компиляции.

11. Автоматизация сборки проекта, утилита make. Сценарий сборки проекта.

12. Типы языка Си. Преобразование типов.

13. Статические одномерные массивы.

14. Указатели, базовые операции.

15. Указатели, массивы, адресная арифметика.

16. Указатели, void, указатели на функции (на примере функции qsort).*

17. Динамические одномерные массивы.

18. Указатели и многомерные статические массивы.

19. Массивы переменной длины (c99), их преимущества и недостатки, особенности использования.

20. Динамические многомерные массивы.

21. Строки.

22. Область видимости, время жизни и связывание.

23. Журналирование

24. Классы памяти

25. Стек и куча.

26. Функции с переменным числом параметров.

27. Структуры.

28. Объединения.

29. Динамический расширяемый массив.

30. Линейный односвязный список.

31. Двоичные деревья поиска.

32-33. Директивы препроцессора, макросы. Директивы препроцессора, условная компиляция, операции # и ##.

34. inline-функции.

35. Списки из ядра операционной системы Linux (списки Беркли).

36. Битовые операции. Битовые поля.

37. Неопределенное поведение.

38. Библиотеки.

39. Абстрактный тип данных

40. Дополнение с семинаров Оленева А. А.

1. *Язык программирования Си*

    a. история появления

    В начале 70-х годов прошлого столетия сотрудником Bell Labs Деннисом Ритчи как развитие языка Би. Первоначально был разработан для ОС UNIX. Согласно дизайну языка, его конструкции близко сопоставляются типичным машинным инструкциям. В 1979 году Деннис Ритчи и Брайн Керниган опубликовали первую книгу, описывающую язык Си.
    - эффективен как Assembler, но более понятный и удобный
    - программы должны быть переносимы на уровне исходных текстов

    b. особенности языка

    - сравнительно низкого уровня
    - “компактный” язык с однопроходным компилятором
    - предполагает, что программист знает, что делает

    c. использование

    - Разработка операционных систем (UNIX, Windows, Linux..)
    - СУБД (системы управления базами данных) (MYSQL, Oracle..)
    - Компиляторы и интерпретаторы (Python, Ruby, Eiffel..)
    - Встроенные системы
    - ПО с открытым исходным кодом

2. Этапы получения исполняемого файла из исходного кода. Опции компилятора и компоновщика.

a. препроцессирование

   На данном этапе происходит вырезание комментариев, производятся текстовые замены посредствам директивы define, и включение необходимых файлов при помощи директивы препроцессора include. Файл, получаемый в процессе препроцессирования - единица трансляции.

   cpp -o <name>.i <name>.c (cpp <name>.c > <name>.i)

b. компиляция

   На этапе компиляции происходит трансляция программы, написанной на Си, на язык Assembler.

   gcc -std=c99 -S -mam=intel <name>.i

c. ассемблирование

   Assembler выполняет перевод программы в исполняемый машинный код. В результате работы получается объектный файл в виде блоков машинного кода и данных, с неопределенными адресами ссылок на данные и процедуры в других объектных модулях, список своих процедур и данных.

d. компоновка

   Компоновщик принимает на вход один или несколько объектных файлов и собирает по ним исполняемый. Может извлекать объектные из специальных коллекций - библиотек.

e. POSIX строка запуска компилятора, ключи компилятора и компоновщика: -std, -Wall, -Werror, -pedantic, -c, -o, -E, -S.

   gcc

   i. -std=c99 - указание имя стандарта
   ii. -pedantic (по умолчанию gcc старается собрать (скомпилировать) программу, выбрасывая warnings, если находит ошибки, которые может исправить, без вмешательства программиста). Данный ключ позволяет более тщательно, следуя требованиям коду ANCII и более придирчиво собрать программу.
   iii. -Wall - все предупреждения вывести на экран
   iv. -Werror - все предупреждения интерпретировать, как ошибки

v. -c (--compile) - только компиляция
vi. -o <name>.exe - использовать <name> в качестве имени исполняемого файла
vii. -E
viii. -S - трансляция, программы, написанной на C, на Assembler

1. gcc -E <name>.c > <name.i> - препроцессирование
2. gcc -S <name>.i - трансляция на язык Assembler
3. gcc -c <name>.s - ассемблирование
4. gcc -o <name>.exe <name>.o - компоновка

3. Этапы получения исполняемого файла.

- представление о формате исполняемого файла

    - обработка препроцессором
    - компиляция
    - ассемблирование
    - компоновка

    Исполняемый файл собирается компоновщиком из объектных файлов. Он имеет расширение под DOS/Windows “.exe”. Его структура содержит управляющую информацию для загрузчика и сам загрузочный модуль.

- загрузка файла в память

    На моменте замены директивы препроцессора #include - прогрузятся необходимые библиотеки. С программной стороны загрузка файла происходит при помощи функции FILE* fopen(const char* filename, const char* mode). Если результат работы этой функции вернул NULL - файла не существует, иначе - возвращает указатель, на то место в памяти типа DATA, где этот файл хранится.

- настройка ссылок

    В исполняемом файле содержатся указатели (ссылки) на используемые данные в программе, переменные (stack), библиотеки. Если библиотека статическая, то исполняемый файл будет содержать исходный код функций из используемых библиотек, в случае динамических, лишь указатели на место в памяти, где лежит сама dll или so библиотека.

- планирование процесса

    Исполняемый файл содержит заголовки и инструкции. В заголовках предполагаемые исполнители инструкций, параметры настройки исполнителя и окружения, формат инструкций кода, где исполнитель инструкций - это аппаратно-программный или программный комплекс, способный выполнить код.
    Инструкции представлены в виде машинного кода, содержит вызовы библиотечных функций. После ассемблирования образовался объектный файл, содержащий блоки машинного кода и данных, с неопределенными адресами ссылок на данные и процедуры в других объектных модулях, список своих процедур и данных.
    Компоновщик уже извлекает объектные файлы из специальных коллекций, называемых библиотеками.

- абстрактная память процессора

    - NULL

    - исполняемый файл
        - код
        - данные
        - таблица импорта
    - библиотеки
        - код
        - данные
    - куча
    - стек

- заголовки функций main согласно C99 и аргументы функции
    - int(void) main(void);
    - int(void) main(int argc, char** argv);

- значение, возвращаемое main

  Если программа отработала как положено, то возвращается 0, в противном случае 1. Так же может и не возвращать ничего вовсе, если тип у функции void.

```
@echo off
<name>.exe
if errorlevel 1 goto err
if errorlevel 0 goto ok
goto fin
:err
echo ERROR!
goto fin
:ok
echo OK
:fin
```

4. Переменные, операция присваивания, ввод/вывод значений переменных.

- понятие “переменная”

  Переменная - абстракция ячейки памяти компьютера или совокупность таких ячеек, в зависимости от типа переменной.

- атрибуты переменной

  - имя (идентификатор)
    - строка символов, используемая для идентификации некоторой сущности в программе
  - тип
    - определяет, как хранить переменную, какие значения может она принимать и какие операции можно выполнить над переменной
  - адрес
    - ячейка памяти, с которой связана данная переменная
  - значение
    - содержимое ячейки или ячеек памяти, связанных с данной переменной
  - область видимости
    - часть текста программы, в пределах которой переменная может быть использована
  - время жизни
    - интервал времени выполнения программы, в течении которого существует переменная и выделенная для нее ячейка памяти
- описание переменной в Си

  - указать тип и имя
    - имя не рекомендуется использовать с _ и совпадающее с именами из стандартных заголовочных файлов
    - не должно начинаться с цифры, совпадать с ключевыми словами языка

- операция присваивания и её особенности

  - назначение значения
  - возможно смешение типов, но оно не всегда безопасно
  - определение переменной можно совместить с присваиванием ей начального значения

- printf

  - первый аргумент - строка форматирования, содержащая спецификаторы и esc - последовательности
  - возвращает число (количество) успешно выведенных в stdout символов
  - %[flags][width][.accuracy][size]type

- scanf
    - работа функции управляется строкой форматирования, для каждого переданного спецификатора она пытается выделить данные соответствующего типа во входных данных, остановится на символе, который не относится к очередному вводимому значению
    - возвращает количество успешно считанных символов

5. Операции и выражения.

- операция, операнд, побочный эффект, приоритет, ассоциативность, выражение

    - операция
        - специальный способ записи некоторых действий, конструкция в языке программирования, аналогичная математическим операциям
    - операнд
        - элементы данных, к которым применяют операции
    - побочный эффект
        - при выполнении операций, происходят (помимо вычисления значений) изменения объектов или файлов
    - приоритет
        - определение порядка выполнения операций в иерархическом порядке при вычислении выражений
    - ассоциативность
        - свойство операции, позволяющее восстанавливать последовательность их выполнения при отсутствии явных указаний на очередность при равном приоритете
    - выражение
        - простейшее средство описания действий

- арифметическая операция

    - операции, принимающие в качестве операндов переменные числового типа и возвращающие результат в виде числового значения
        - сложение
        - вычитание
        - умножение
        - деление
        - деление по модулю
        - унарный минус
        - унарный плюс

- составное присваивание

    - операция состоящая из комбинации простого присваивания и одной из арифметических операций

- операции инкремента и декремента

    - постфиксные/префиксные операции увеличения/уменьшения значения переменной. В постфиксной операции значение выражения вычисляется до применения соответствующего оператора, а операции инкремента/декремента

выполняется после вычисления операнда. (c++ - постфиксное инкрементирование переменной c)

- операции сравнения

    - !=, ==, <=, >=, >, <
    - бинарные операторы, имеющие два числовых аргумента, возвращающие логическое значение

- логические операции

    - ! (отрицание - унарный), && (конъюнкция), || (дизъюнкция)
    - в качестве аргументов для всех этих операторов выступают логические литералы (const), логические переменные и выражения, имеющие логическое значение

- порядок вычисления выражений (+логические)

    - () [] . -> - первичные
    - + ~ - ! * & ++ -- sizeof - унарные
    - * / % - мультипликативные
    - + - - аддитивные
    - >> << - сдвиг
    - < > <= >= - отношение
    - & - поразрядное и
    - ^ - поразрядное исключающее и
    - | - поразрядное включающее или
    - && - логическое и
    - || - логическое или
    - ?: - условная
    - = *= /= %= += -= >>= <<= &= |= - простое и составное присваивание
    - , - последовательное вычисление
    - приведение типа имеет тот же порядок, что и унарные операции

6. Оператор выражение, условный оператор, условная операция, составной оператор, switch.

- оператор выражение
    - <выражение>;
    - относится к классу простых операторов языка
    - любое выражение будет оператором, если за ним следует ;
    - формирует основные строительные блоки для операторов и определяют, каким образом программа управляет данными и меняет их
- условный оператор
    - оператор, порождающий ветвление алгоритма
    - если условие истинно, то выполняется этот блок
    - иначе - другой
    - позволяет сделать выбор между двумя альтернативами, проверив значение выражения
- условная операция
    - ?:
    - expr1 ? expr2 : expr3
    - Сначала вычисляется значение выражения expr1, если оно отлично от 0, то вычисляется значение expr2, и его значение становится значением условной операции. Если expr1 равно 0, то значением условной операции становится expr3
- составной оператор
    - заключение нескольких операторов в фигурные скобки, выполнение ряда задач
    - компилятор интерпретирует, как один оператор
- switch
    - switch(выражение)
    - {
        - case const1 : операторы, break;
        - ....
        - default : операторы, break;
    - }
    - управляющее выражение, которое располагается за ключевым словом switch должно быть целочисленным

7. Операторы цикла (while, do-while, for), операция запятая.

- while (выражение) {операторы;}
    - выполнение цикла начинается с вычисления значения выражения, если оно отлично от 0, выполняется тело цикла, после чего значение выражения вычисляется еще раз, процесс продолжается пока значение выражения не станет равно 0
- do-while {операторы;} while (выражение);
    - цикл с постусловием, выполнение начинается с выполнения тела цикла, после чего вычисляется значение выражения, если оно отлично от 0, выполняется тело цикла, после чего значение выражения вычисляется еще раз, процесс продолжается пока значение выражения не станет равно 0
- for (expr1; expr2; expr3); {операторы;}
    - обычно используется для реализации цикла со счетчиком
- выражение1 , выражение2
    - сначала вычисляется выражение1 и его значение отбрасывается, потом выражение2. Значение этого выражения является результатом операции, выражение1 всегда должно содержать побочный эффект, в противном случае от выражение1 не будет никакого толка

8. Операторы break, continue, goto, пустой.

- break
    - используется для принудительного выхода из цикла, выполняется выход из ближайшего цикла или оператора switch
- continue
    - оператор, передающий управление в конец цикла
    - может использоваться только внутри циклов
- goto
    - оператор, который способен передать управление на любой другой оператор, помеченный меткой-идентификатором, расположенной вначале оператора
- ;
    - используется для реализации циклов с пустым телом
    - легко может стать источником ошибки

9. Функции

- подпрограммы, их виды и преимущества использования

    - В си подпрограммами являются функции
    - Может возвращать любое значение
    - Или ничего не возвращать
    - Понятность и простота кода
    - Удобство отладки с помощью реализации модульного тестирования

- общая структура функции

```
// заголовок функции
тип-результата имя-функции (список формальных параметров с их типами)
// тело функции
{
        определения;
        операторы;
}
```

- оператор return

    - завершает выполнение функции и возвращает управление вызывающей стороне
    - использует для возврата значение (иск. void)
    - функция может содержать произвольное число операторов return
    - оператор return может использоваться в функциях типа void, при этом никакое выражение не указывается

- операции вызова функции

    - для вызова функции необходимо указать ее имя, за которым в круглых скобках через запятую перечислить аргументы
    - если функция возвращает значение, ее можно использовать в выражениях.

- передача аргументов в функцию

    - void f(void); - означает, что у функции нет ни одного параметра.
    - все аргументы функции передаются «по значению»
    - параметры можно рассматривать, как локальные переменные

- рекурсия

    - функция называется рекурсивной, если она вызывает саму себя.
    - простейшая форма рекурсии, где рекурсивный вызов расположен в конце функции - хвостовая, действует подобно циклу
        - рекурсивный вызов использует больше памяти

- создает свой набор переменных
- выполняется медленней

10. Автоматизация сборки проекта, утилита make. Сценарий сборки проекта. Простой сценарий сборки. Использование переменных и комментариев. Сборка программы с разными параметрами компиляции.

- автоматизация сборки проекта:
    - основные задачи
        - автоматизация процесса преобразования файлов из одной формы в другую
        - «исходные» данные;
            - make file
            - исходники проекта
    - разновидности утилиты make
        - GNU Make (рассматривается далее)
        - BSD Make
        - Microsoft Make (nmake)
- сценарий сборки проекта:

```
makefile | Makefile
цель: зависимость_1 ... зависимость_n
[tab]команда_1
[tab]команда_2
...
[tab]команда_m
```

- простой сценарий сборки

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

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

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

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

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

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

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

- использование переменных
    - VAR_NAME := value
        - чтобы получить значение переменной, необходимо ее имя заключить в круглые скобки и перед ними поставить символ '$' - $(VAR_NAME)
    - условные конструкции в сценарии сборки

```
ifeq ($(mode), debug)
	# Отладочная сборка
else
endif
```

11. Автоматизация сборки проекта, утилита make. Сценарий сборки проекта. Автоматические переменные. Шаблонные правила.

- автоматизация сборки проекта: основные задачи, «исходные» данные
    - см. вопрос №10 ----->
- разновидности утилиты make
    - см. вопрос №10 ----->
- сценарий сборки проекта: название файла, структура
    - см. вопрос №10 ----->
- автоматические переменные
    - переменные со специальными именами, которые «автоматически» принимают определенные значения перед выполнением описанных в правиле команд
        - "$^" - "список зависимостей"
        - "$@" - "имя цели"
        - "$<" - “первая зависимость”
- шаблонные правила

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

12. Типы языка Си. Преобразование типов.

- понятие «тип данных»
    - внутреннее представление данных в памяти
    - множество значений, которые могут принимать величины этого типа
    - операции и функции, которые можно применять к величинам этого типа
- простые типы:
    - целочисленные
        - В языке Си существует несколько типов целых чисел. Они различаются
            - объемом памяти, отводимым под переменную (диапазоном)
            - возможностью присваивания положительных и отрицательных чисел.
                - short int
                - unsigned short int
                - int
                - unsigned int
                - long int
                - unsigned long int
                - long long int
                - unsigned long long int
    - вещественные
        - язык Си предоставляет три вещественных типа:
            - float
            - double
            - long double
    - символьные
        - char - которому может быть присвоено значение любого ASCII символа
        - 1 байт
    - перечисляемый (enum)
        - тип разработан для переменных, которые принимают небольшое количество значений
    - логический тип (c99)
        - стандарт C99 добавил логический тип _Bool

        - переменные типа _Bool могут принимать только значения 0 и 1.
        - стандарт c99 предоставляет заголовочный файл stdbool.h, который облегчает использование «нового» логического типа.

- составные типы (структурированные)
    - массивы
    - структуры
    - объединения

- оператор typedef

    - позволяет определять имена новых типов.
        typedef тип имя;

    - “+” улучшает читаемость.
    - “+” облегчает внесение изменений.

- операция sizeof

    - возвращает размер переменной или типа в байтах
    - sizeof(выражение)

- неявное и явное преобразование типов.

    - неявные
        ```
        int i = 0;
        i = 3.541 + 3;          // предупреждение компилятора
        printf(“%d”, i);      // 6
        ```
    - явные
        ```
        int i = 0;
        i = (int) 3.541 + 3;
        printf(“%d”, i);      // 6
        ```

13. Статические одномерные массивы

- понятие «массив»

    - последовательность элементов одного и того же типа, расположенных в памяти друг за другом.

- определение переменной-массива, способы инициализации переменной-массива

    - тип элемента может быть любым
    - количество элементов указывается целочисленным константным выражением
    - количество элементов не может быть изменено в ходе выполнения программы

```
#define N 10
...
int a[N];
```

- операция индексации

    - для доступа к элементу массива используется индекс.
    - индексация выполняется с нуля.
    - в качестве индекса может выступать целочисленное выражение
    - Си не предусматривает никаких проверок на выход за пределы массива.

- особенности использования массивов в языке Си

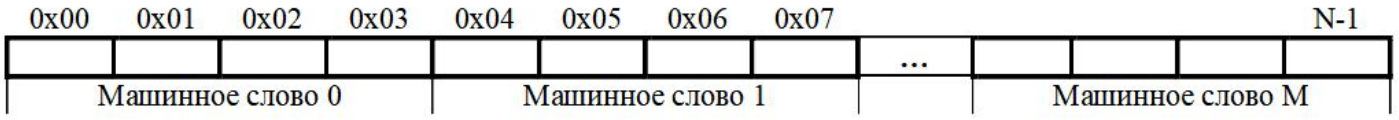
    - компилятор может самостоятельно определить количество элементов в массиве и выделить для них память
    - массив - есть указатель на первый элемент
    - адресная арифметика
    - не предусмотрено проверок на выход за пределы массива
    - индексом может служить целочисленное выражение
    - количество элементов статического массива не может быть изменено в ходе выполнения программы

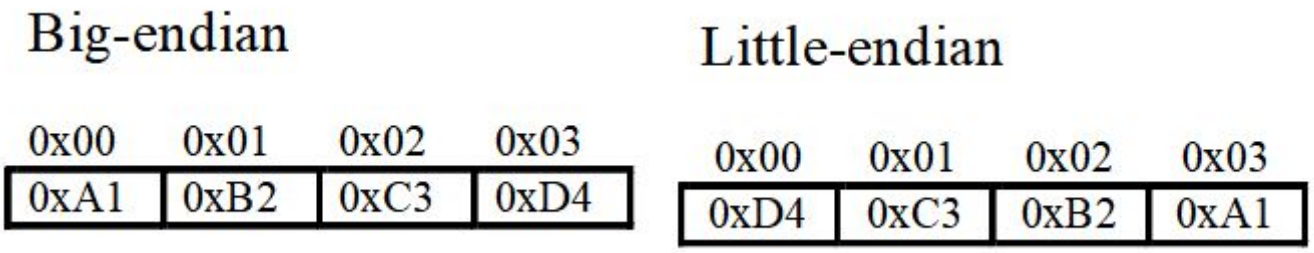
- массивы как параметры функции

    - передается массив в функцию по указателю на начало и конец, указателю на начало и количеству элементов

14. Указатели, базовые операции.

- организация памяти с точки зрения программиста





- понятие «указатель»
    - переменная-указатель – это переменная, которая содержит адрес
    - переменная-указатель описывается как и обычная переменная. Единственное отличие – перед ее именем необходимо указать символ «*»:
- разновидности указателей в языке Си
    - типизированный указатель на данные (тип* имя);
    - бестиповой указатель (void* имя);
    - указатель на функцию.
    -
- использование указателей
    - передача параметров в функцию
        - изменяемые параметры
        - «объемные» параметры
    - обработка областей памяти
        - динамическое выделение памяти
        - ссылочные структуры данных
- определение переменной-указателя
    - инициализация с указанием тип*
- базовые операции над указателями (“&” и “*”)
    - & - операция взятия адреса
    - * - операция разыменования, используется для получения доступа к объекту, на который указывает указатель

- модификатор const и указатель

    - const int *p; - указатель на константу
    - int const *q; - указатель на константу
    - int * const r; - константный указатель

15. Указатели, массивы, адресная арифметика

- понятие «указатель»
    - см. вопрос № 14. ----->
- связь между указателями и статическими массивами
    - результат выражения, состоящего из имени массива, представляет собой адрес области памяти, выделенной под этот массив
- адресная арифметика
    - сложение указателя с числом
        - новый адрес в p = старый адрес из p + n * sizeof(тип)
    - сравнение указателей
        - при сравнении указателей сравниваются адреса.
    - вычитание указателей
        - новый адрес в p = старый адрес из p - m * sizeof(тип)

16. Указатели, void, указатели на функции (на примере функции qsort).*

- понятие «указатель» - см. вопрос №14. ----->

- void*, особенности операций с ним

    - тип указателя void используется, если тип объекта неизвестен.
        - позволяет передавать в функцию указатель на объект любого типа;
        - полезен для ссылки на произвольный участок памяти, независимо от размещенных там объектов
        - нельзя разыменовывать
        - не применима адресная арифметика

- приведение указателей разных типов к void* и обратно

    - присваивание указателя любого другого типа (и наоборот) без явного преобразования типа указателя

- определение указателя на функцию

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

- присваивание значения указателю на функцию

```
result = integrate(0, 3.14, 25, sin);
```

- вызов функции по указателю

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

- использование указателей на функции.

    - в качестве компаратора для обработки данных различных типов в стандартных функциях сортировки

17. Динамические одномерные массивы.

- функции для выделения и освобождения памяти (malloc, calloc, realloc, free)

    - указанные функции не создают переменную, они лишь выделяют область памяти. В качестве результата функции возвращают адрес расположения этой области в памяти компьютера, т.е. указатель.
    - поскольку ни одна из этих функций не знает данные какого типа будут располагаться в выделенном блоке все они возвращают указатель на void.
    - в случае если запрашиваемый блок памяти выделить не удалось, любая из этих функций вернет значение NULL.
    - после использования блока памяти он должен быть освобожден. Сделать это можно с помощью функции free.

    - void* malloc(size_t size);
        - функция malloc выделяет блок памяти указанного размера size. Величина size указывается в байтах.
        - выделенный блок памяти не инициализируется (т.е. содержит «мусор»).
        - для вычисления размера требуемой области памяти необходимо использовать операцию sizeof.
    - void* calloc(size_t nmemb, size_t size);
        - функция calloc выделяет блок памяти для массива из nmemb элементов, каждый из которых имеет размер size байт.
        - выделенная область памяти инициализируется таким образом, чтобы каждый бит имел значение 0
    - void free(void *ptr);
        - функция free освобождает (делает возможным повторное использование) выделенный блок памяти, на который указывает ptr.
        - если значением ptr является нулевой указатель, ничего не происходит.
        - если указатель ptr указывает на блок памяти, который не был получен с помощью одной из функций malloc, calloc или realloc, поведение функции free не определено.
    - void* realloc(void *ptr, size_t size);
        - выделение памяти (как malloc)
        - освобождение памяти аналогично free().
        - перевыделение памяти. В худшем случае:
            - выделить новую область
            - скопировать данные из старой области в новую
            - освободить старую область

- типичные ошибки при работе с динамической памятью

    - запрос 0 байта
        - результат вызова функций malloc, calloc или realloc, когда запрашиваемый размер блока равен 0, зависит от реализации (implementation-defined C99 7.20.3):

        - вернется нулевой указатель;
        - вернется «нормальный» указатель, но его нельзя использоваться для разыменования.
- утечки памяти
- разыменование битого указателя (invalid/wild pointer)
    - попытка разыменовать указатель после вызова free
- двойное освобождение памяти

18. Указатели и многомерные статистические массивы.

- концепция многомерного массива как «массива массивов»
    - расположение строк матрицы в памяти одну за другой вплотную друг к другу
- определение многомерных массивов и инициализация, обработка массивов при помощи указателей
    - простая инициализация [n][m], n - число строк, m- число столбцов.
      n - можно опустить, его значение подставит компилятор при сборке проекта
    - динамическое выделение памяти
        - матрица, как одномерный массив -
            - средство проверки работы с памятью не может отследить выход за пределы строки
            - нужно писать i*m+j, m - число столбцов
            + простота выделения и освобождения памяти
            + возможность использования одномерного массива

```
data = malloc(n * m * sizeof(double));
if (data)
{
   for (int i = 0; i < n; i++)
      for (int j = 0; j < m; j++)
         // Обращение к элементу i, j
         data[i*m+j] = 0.0;
}
```

        - матрица, как массив указателей
            - сложность выделения и освобождения памяти
            - память под матрицу выделена отдельными блоками
            + возможность обмена строки через обмен указателей
            + можно отследить выход за предел строки
            - вход: количество строк (n) и количество столбцов (m)
            - выход: указатель на массив строк матрицы (p)
            - выделить память под массив указателей (p)
            - обработать ошибку выделения памяти
            - в цикле по количеству строк матрицы (0 <= i < n)
                - выделить память под i-ую строку матрицы (q)
                - обработать ошибку выделения памяти
                - p[i]=q
        - матрица, как одномерный массив с указателями
            - сложность начальной инициализации
            - нельзя отследить выход за пределы строки
            + простота выделения и освобождения памяти
            + возможность использовать как одномерный массив
            + перестановка строк через обмен указателей
            - вход: количество строк (n) и количество столбцов (m)

        - выход: указатель на массив строк матрицы (p)
        - выделить память под массив указателей на строки и элементы
        - обработать ошибку выделения памяти
        - в цикле по количеству строк матрицы (0 <= i < n)
            - вычислить адрес i-ой строки матрицы (q)
            - p[i]=q
- передача многомерных массивов в функцию

    - void f(int a[N][M], int n, int m);
    - void f(int a[][M], int n, int m);
    - void f(int (*a)[M], int n, int m);

- const и многомерные массивы

    - формальное объяснение
        - согласно C99 6.7.3 #8 и 6.3.2.3.2 выражение T (*p)[N] не преобразуется неявно в T const (*p)[N].
    - способы борьбы
        - не использовать const
        - использовать явное преобразование типа

19. Массивы переменной длины (c99), их преимущества и недостатки, особенности использования.

- о динамических массивах - см. вопрос № 17 ----->

- создание функции, которая своим параметром принимает количество элементов, возвращает массив

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

char* f(int n)
{
    char arr[n*2];

    for(int i = 0; i<n*2; i++)
    {
        arr[i] = 'a';
    }

    return arr;
}

int main(void)
{
    int n = 0;
    scanf("%d", &n);
    char* a = f();
}
```

- массивы, память под которые можно перевыделить при помощи realloc

    - для уменьшения потерь при распределении памяти (лучше перевыделять сразу крупными блоками, так как перевыделение памяти может сильно замедлить программу)
    - для простоты реализации указатель на выделенную память придется хранить вместе со всей информацией, необходимой для управления динамическим массивом
    - сохраняет средние ожидаемые затраты на копирование элемента
    - программа должна обращаться к элементам массива по индексам, поскольку адрес массива может измениться
    - программа сразу же проверяет код, реализующий выделение памяти, в виду маленького начального размера

- достоинства и недостатки

+ простота использования
+ время доступа к любому элементу постоянно
+ грамотное распределение ресурсов
+ хорошее сочетание с двоичным поиском
- хранение меняющегося набора значений

20. Динамические многомерные массивы

- см. вопрос № 18 - определение многомерных массивов и реализация
- ----->

21. Строки

- понятия «строка» и «строковый литерал»

    - последовательность символов, заканчивающаяся и включающая первый нулевой символ (англ., null character ‘\0’ (символ с кодом 0))
    - последовательность символов, заключенных в двойные кавычки, рассматривается компилятором как массив элементов типа char. Когда компилятор встречает строковый литерал из n символов, он выделяет n+1 байт памяти, которые заполняет символами строкового литерала и завершается нулевым символом.
        - массив, который содержит строковый литерал, существует в течение всего времени выполнения программы.
        - в стандарте сказано, что поведение программы не определено при попытке изменить строковый литерал.
        - обычно строковые литералы хранятся в read only секции.

- определение переменной-строки, инициализация строк

    - char str_1[] = {‘J’,‘u’,‘n’, ‘e’, ‘\0’};
    - char str_2[] = “June”;
    - char str_3[5] = “June”;
    - тип данных, значениями которого является произвольная последовательность (строка) символов алфавита. Каждая переменная такого типа (строковая переменная) может быть представлена фиксированным количеством байтов либо иметь произвольную длину.

- ввод/вывод строк (scanf, gets, fgets, printf, puts)

    - функции scanf и gets небезопасны и недостаточно гибки. Программисты часто реализуют свою собственную функцию для ввода строки, в основе которой лежит посимвольное чтение вводимой строки с помощью функции getchar.

        ```
        #include <stdio.h>
        int getchar(void);
        ```

    - char *fgets(char *s, int size, FILE *stream);
        - прекращает ввод когда (любое из)
            - прочитан символ ‘\n’;
            - достигнут конец файл;
            - прочитано size-1 символов.
        - введенная строка всегда заканчивается нулем.

        ```
        fgets(str, sizeof(str), stdin);
        ```

- функции стандартной библиотеки для работы со строками (strcpy, strlen, strcmp и др.)

```
char* strcpy(char *s1, const char *s2);
char src[] = "Hello!";
```

```
char dst[20];
strcpy(dst, src);
```

- вместо функции strcpy безопаснее использовать функцию strncpy.

```
char* strncpy(char *s1, const char *s2, size_t count);
strncpy(dst, src, sizeof(dst) - 1);
dst[sizeof(dst) - 1] = '\0';

size_t strlen(const char *s);
char dst[20];
size_t len;
strcpy(dst, "Hello!");
len = strlen(dst);      // len = 6, а не 20

char* strcat(char *s1, const char *s2);
char src[] = ", world.";
char dst[20] = "Hello";
strcat(dst, src);
```

Вместо функции strcat безопаснее использовать функцию strncpy.

```
char* strncat(char *s1, const char *s2, size_t count);
int len = (sizeof(des)- 1) - strlen(des);
strncat(des, src, len);

int strcmp(const char *s1, const char *s2);
```

значение < 0, если s1 меньше s2
0, если s1 равна s2
значение > 0, если s1 больше s2
Строки сравниваются в лексикографическом порядке (как в словаре).

```
int strncmp(const char *s1, const char *s2, size_t count);

char* strdup(const char *s);          // НЕ c99
char* strndup(const char *s, size_t count);       // НЕ c99
char* str;
str = strdup("Hello!");
if (str)
{
   ...
   free(str);
}
int sprintf(char *s, const char *format, ...);
// c99
int snprintf(char *s, size_t num, const char *format, ...);

char* strtok(char *string, const char *delim);
char str_test_1[] = "   This   is a,,, test string!!!";
char *pword = strtok(str_test_1, "\n ,.!?");
```

```
while (pword)
{
   printf("[%s]\n", pword);
   pword = strtok(NULL, "\n ,.!?");
}
```

Перевод строки в число

```
#include <stdlib.h>
// Семейство функций (atoi, atof, atoll)
long int atol(const char* str);
// Семейство функций (strtoul, strtoll, ...)
long int strtol(const char* string, char** endptr, int basis);

int read_line(char *s, int n)
{
int ch, i = 0;
while ((ch = getchar()) != '\n' && ch != EOF)
        if (i < n - 1)
                s[i++] = ch;
s[i] = '\0';
return i;
}
```

Семинарское дополнение

Параметры функции: s – массив, в котором сохраняются символы, n – размер этого массива.
Функция возвращает количество символов, сохраненных в массиве.
Символы, которые не помещаются в массив, игнорируются

22. . Область видимости, время жизни и связывание.

- понятия «область видимости», «время жизни» и «связывание» в языке Си

    - область видимости (scope) имени – это часть текста программы, в пределах которой имя может быть использовано
        - блок
        - файл
        - функция
        - прототип функции
    - блок
        - переменная, определенная внутри блока, имеет область видимости в пределах блока.
        - формальные параметры функции имеют в качестве области видимости блок, составляющий тело функции.
    - 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.
    - функция
        - 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.
    - прототип функции
        - область видимости в пределах прототипа функции применяется к именам переменных, которые используются в прототипах функций.
        - область видимости в пределах прототипа функции простирается от точки, в которой объявлена переменная, до конца объявления прототипа.
    - время жизни (storage duration) – это интервал времени выполнения программы, в течение которого «программный объект» существует.
        - глобальное (по стандарту - статическое (англ. static))
        - локальное (по стандарту - автоматическое (англ. automatic))
        - динамическое (по стандарту - выделенное (англ. allocated))
    - связывание (linkadge) определяет область программы (функция, файл, вся программа целиком), в которой «программный объект» может быть доступен другим функциям программы
        - внешнее (external)
        - внутреннее (internal)
        - никакое (none)

- правила перекрытия областей видимости

- переменные, определенные внутри некоторого блока, будут доступны из всех блоков, вложенных в данный
- возможно определить в одном из вложенных блоков переменную с именем, совпадающим с именем одной из "внешних" переменных

- размещение «объектов» в памяти в зависимости от времени жизни

  - переменные
    - глобальные - статические
    - локальные - автоматические
    - динамические - выделенные

  - класс памяти auto
    - применим к переменным, определенным в блоке
    - локальное время жизни, видимость в пределах блока и не имеет связывания
    - любая переменная, объявленная в блоке или заголовке функции
  - класс памяти static
    - может использоваться с любыми переменными независимо от места их расположения
    - для переменной все какого либо блока static изменяет связывание этой переменной на внутреннее
    - для переменной в блоке изменяет время жизни с автоматического на глобальное
    - статическая переменная, определенная вне какого-либо блока имеет глобальное время жизни, область видимости в пределах файла и внутреннее связывание
    - скрывает переменную в файле, в котором она определена
    - статическая переменная, определенная в блоке имеет глобальное время жизни, область видимости в пределах блока и отсутствие связывания
      - такая переменная сохраняет свое значение после выхода из блока.
      - инициализируется только один раз.
      - если функция вызывается рекурсивно, это порождает новый набор локальных переменных, в то время как статическая переменная разделяется между всеми вызовами.
  - класс памяти extern
    - помогает разделить переменную между несколькими файлами
    - используется для переменных, определенных как в блоке, так и вне блока
      - объявлений (extern int number;) может быть сколько угодно.
      - определение (int number;) должно быть только одно.
      - объявления и определение должны быть одинакового типа.
    - глобальное время жизни, файловая область видимости
  - класс памяти register
    - использование класса памяти register – просьба (!) к компилятору разместить переменную не в памяти, а в регистре процессора

        - используется только для переменных, определенных в блоке.
        - задает локальное время жизни, видимость в блоке и отсутствие связывания.
        - обычно не используется.
    - к переменным с классом памяти register нельзя применять операцию получения адреса &

- влияние связывания на объектный и/или исполняемый файл.

    - имена с внешним связыванием доступны во всей программе. Подобные имена «экспортируются» из объектного файла, создаваемого компилятором.
    - имена с внутренним связыванием доступны только в пределах файла, в котором они определены, но могут «разделяться» между всеми функциями этого файла.
    - имена без связывания принадлежат одной функции и не могут разделяться вообще.
    - 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
    - ощутимое значение явления связывания при работе с динамическими библиотеками, используется динамическое связывание функций, строится дерево зависимостей от so/dll файлов

23. Журналирование

- назначение, идеи реализации

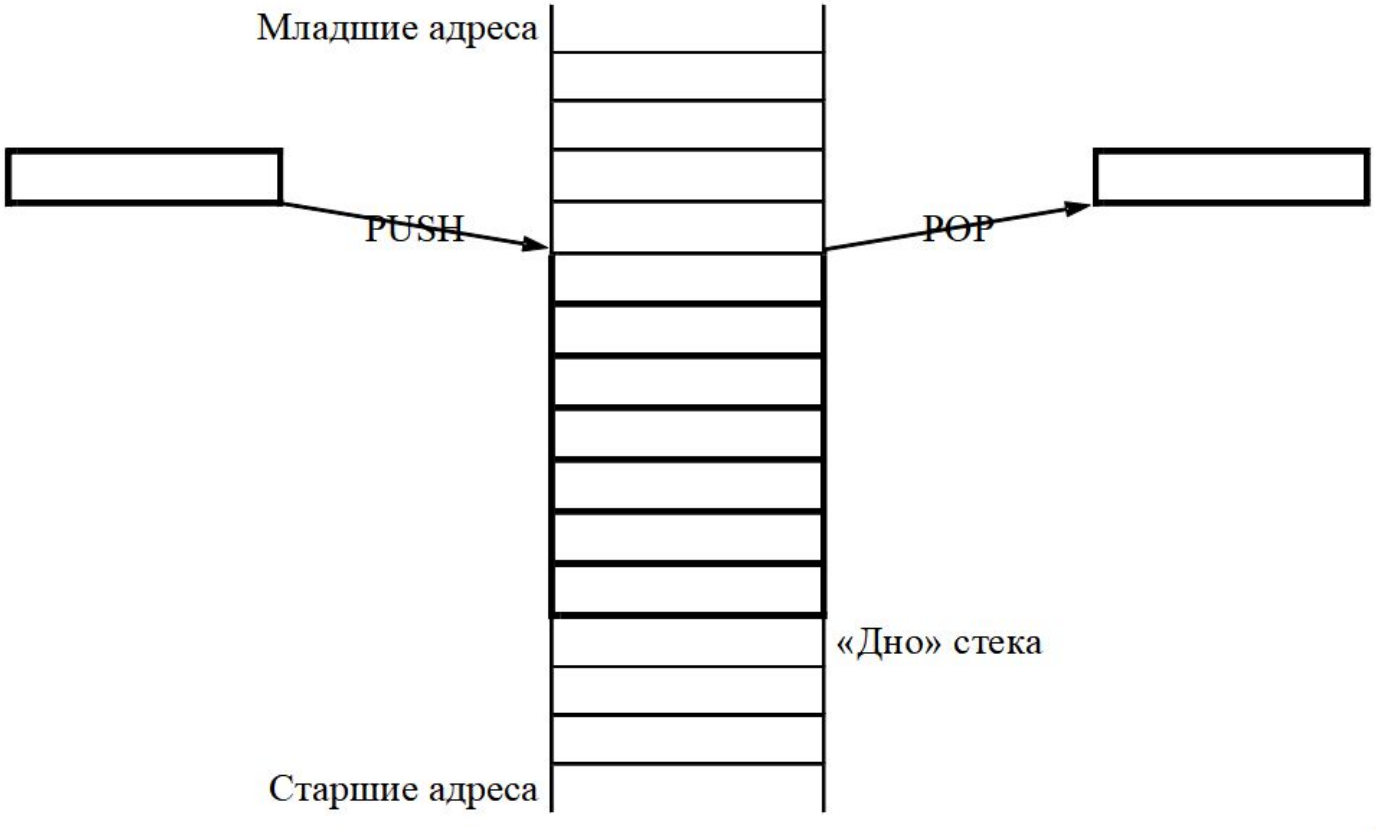
    - процесс записи информации о происходящих с каким-то объектом (или в рамках какого-то процесса) событиях в лог-файл.
    - для эффективного журналирования требуется значительный объём свободного пространства на диске, а в многопоточной среде, где многочисленные потоки записывают свои трассировки, оно увеличивается многократно.
    - кольцевой буфер — это техника журналирования для приложений, при помощи которой важные данные сохраняются в памяти вместо того, чтобы постоянно записываться в файл на диске. Эти данные могут быть сброшены на диск по требованию, например, когда пользователь попросит сделать дамп памяти в файл, программа обнаружит ошибку или программа прервет работу из-за выполнения недопустимой операции или получения недопустимого сигнала. Кольцевой буфер — это выделенный в памяти буфер постоянного размера, используемый приложением для регистрации информации.

24. Классы памяти

- см. вопрос №22 - размещение объектов в памяти
- ----->

25. Стек и куча

- автоматическая память: использование и реализация
    - используется для хранения локальных переменных
    + 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.

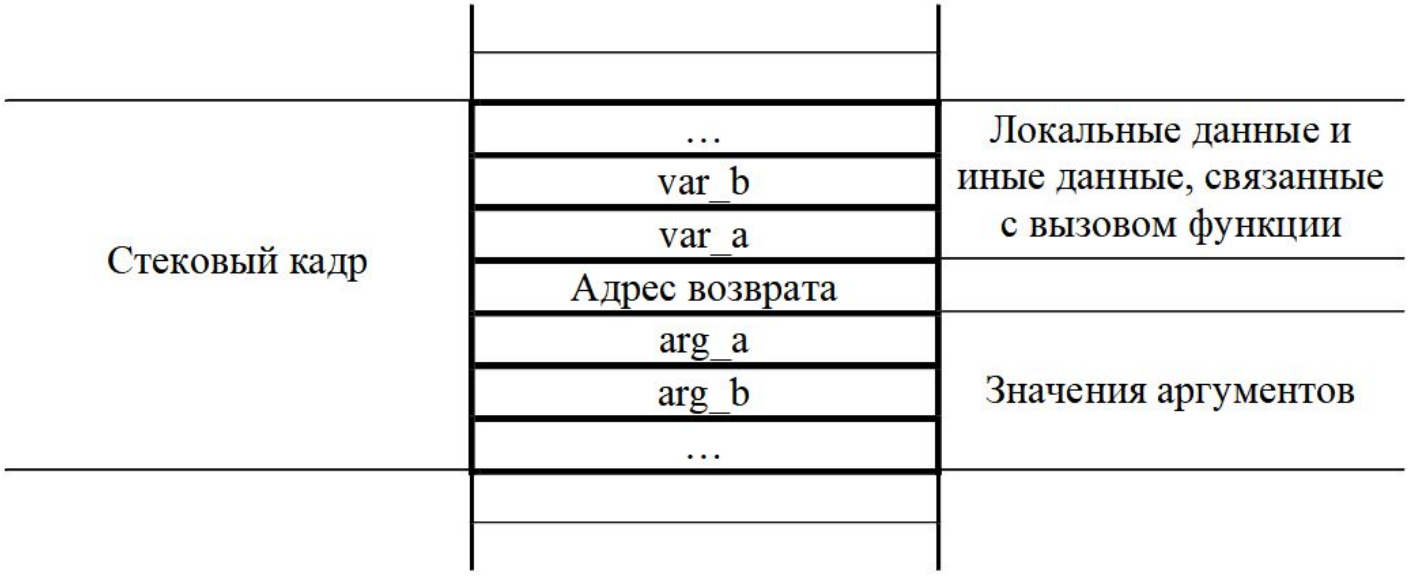


    - используется для
        - вызова функции
        - возврата из функции
        - передачи параметров в функцию
        - выделения и освобождения памяти под локальные переменные

- использование аппаратного стека

    - стековый кадр (фрейм) - механизм передачи аргументов и выделения временной памяти с использованием аппаратного стека.
    - в стековом кадре размещаются:
        - значения фактических аргументов функции;
        - адрес возврата;

- локальные переменные;
- иные данные, связанные с вызовом функции



- ошибки при использовании автоматической памяти

    - возврат указателя на локальную переменную
    - переполнение буфера

- динамическая память: использование и реализация

    - при запуске процесса ОС выделяет память для размещения кучи.
    - куча представляет собой непрерывную область памяти, поделенную на занятые и свободные области (блоки) различного размера.
    - информация о свободных и занятых областях кучи обычно храниться в списках различных форматов
    - для хранения данных используется «куча».
        - создать переменную в «куче» нельзя, но можно выделить память под нее.
        - все «минусы» локальных переменных.
        - ручное управление временем жизни

- идеи реализации функций динамического выделения и освобождения памяти

    - функция malloc выполняет примерно следующие действия:
        - просматривает список занятых/свободных областей памяти, размещенных в куче, в поисках свободной области подходящего размера;
        - если область имеет точно такой размер, как запрашивается, добавляет заданную область в список занятых областей и возвращает указатель на начало области памяти;

    - если область имеет больший размер, она делится на части, одна из которых будет занята (выделена), а другая останется в списке свободных областей;
    - если область не удается найти, у ОС запрашивается очередной большой фрагмент памяти, который подключается к списку, и процесс поиска свободной области продолжается;
    - если по тем или иным причинам выделить память не удалось, сообщает об ошибке (например, malloc возвращает NULL)

- функция free выполняет примерно следующие действия
    - просматривает список занятых/свободных областей памяти, размещенных в куче, в поисках указанной области;
    - удаляет из списка заданную область (или помечает область как свободную);
    - если освобожденная область вплотную граничит со свободной областью с какой-либо из двух сторон, то она сливается с ней в единую область большего размера.

26. Функции с переменным числом параметров

int f(...);

- 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, ...)
- stdarg.h
    - va_list
    - void va_start(va_list arg, last_param)
    - type va_arg(va_list arg, type)
    - void va_end(va_list arg)
- общая процедура создания функции, которая имеет переменное число аргументов, заключается в следующем: функция должна иметь один или более известных параметров. Эти известные параметры следуют перед списком переменных параметров. Самый правый известный параметр называется last_parm. Имя last_parm используется в качестве второго параметра в вызове va_start(). Прежде чем осуществлять доступ к какому-либо из переменных параметров, должен быть инициализирован указатель arg, для чего используется вызов va_start(). После этого параметры возвращаются с помощью вызова функции va_arg() с параметром type, являющимся типом следующего параметра. Наконец, после того, как все параметры прочитаны, перед тем как выйти из функции, необходимо вызвать функцию va_end(), что гарантирует правильное восстановление стека. Если функция va_end() не вызвана, то возникает аварийная ситуация.

27. Структуры

- понятие «структура»
    - представляет собой одну или несколько переменных (возможно разного типа), которые объединены под одним именем.
- определение структурного типа

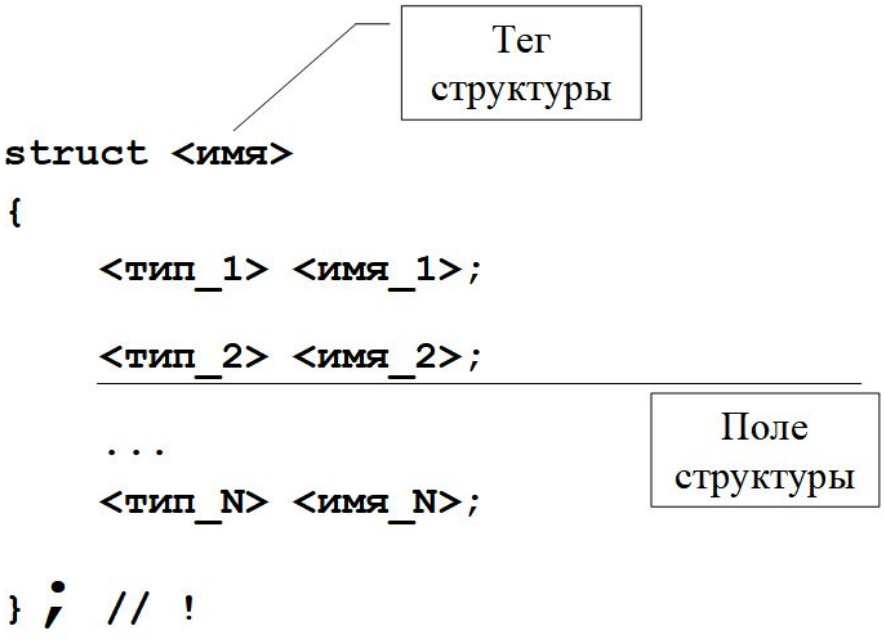
Раздельные определения типа и переменных

```
struct date
{
    int day;
    int month;
    int year;
};

struct date birthday;
struct date exam;
```

Совмещенные определения типа и переменных

```
struct date
{
    int day;
    int month;
    int year;
} birthday, exam;
```

- структура и ее компоненты (тэг, поле)



- Тег - имя, которое располагается за ключевым словом struct, называется тегом структуры.
- используется для краткого обозначения той части объявления, которая заключена в фигурные скобки. Тег может быть опущен (безымянный тип)

- определение переменной-структуры, способы инициализации

    - переменная типа структура - мы как бы создаем новый тип данных, но еще не объявляем переменных этих типов.
        - использование объектов, объединяющих сразу ряд параметров, каждый из которых может фигурировать, как отдельная переменная.
    - переменная-структура - структура содержащая переменное поле (динамический массив, матрица и тд)
        - подобное поле должно быть последним.
        - нельзя создать массив структур с таким полем.
        - структура с таким полем не может использоваться как член в «середине» другой структуры.
        - операция sizeof не учитывает размер этого поля (возможно, за исключением выравнивания).
        - если в этом массиве нет элементов, то обращение к его элементам – неопределенное поведение.
    - для инициализации переменной структурного типа необходимо указать список значений, заключенный в фигурные скобки.
    - значения в списке должны появляться в том же порядке, что и имена полей структуры.
    - если значений меньше, чем полей структуры, оставшиеся поля инициализируются нулями

- операции над структурами

    - доступ к полю структуры осуществляется с помощью операции “.”, а если доступ к самой структуре осуществляется по указателю, то с помощью операции “->”.
    - структурные переменные одного типа можно присваивать друг другу (замечание: у разных безымянных типов тип разный)
    - структуры нельзя сравнивать с помощью “==” и “!=”
    - структуры могут передаваться в функцию как параметры и возвращаться из функции в качестве ее значения

- особенности выделения памяти под структурные переменные

    - память выделяется сплошным куском, выравнивание по самому большому полю до кратного числа байт
        - 1-байтовые поля не выравниваются, 2-байтовые — выравниваются на чётные позиции, 4-байтовые — на позиции кратные четырём и т.д.
    - pragma pack (push, кратность выравнивания)

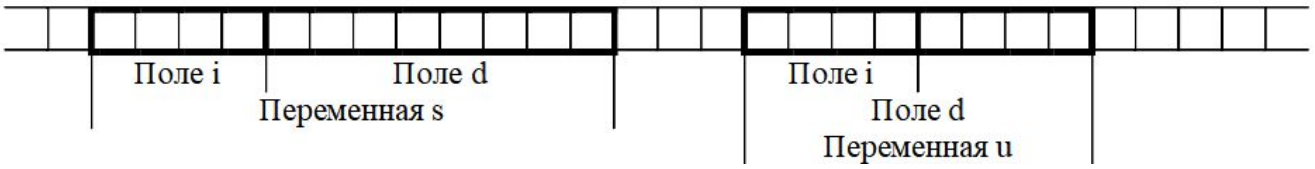
28. Объединения

- понятие «объединение»

    - как и структура, содержит одно или несколько полей возможно разного типа. Однако все поля объединения разделяют одну и ту же область памяти.

```
struct
{
    int i;
    double d;
} s;
```

```
union
{
    int i;
    double d;
} u;
```



- определение переменной-объединения, способы инициализации

```c
struct s_t
{
    int i;
    double d;
};

...

struct s_t s = {1, 5.25};
```

```c
union u_t
{
    int i;
    double d;
};

...

union u_t  u_1 = {1};

// только c99
union u_t  u_2 = { .d = 5.25 };
```

  - присвоение значения одному члену объединения обычно изменит значение других членов.
- использование объединений

  - создание структур данных из разных типов
  - разный взгляд на одни и те же данные (машинно-зависимо)

29. Динамический расширяемый массив

- функция realloc и особенности ее использования

    - см. вопрос № 17 - функции выделения и освобождения памяти
    - ----->

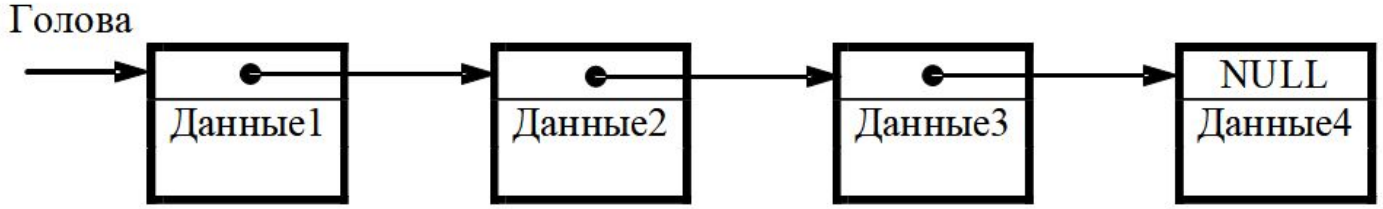
- описание типа

    - для уменьшение потерь при распределении памяти изменение размера должно происходить относительно крупными блоками.
    - для простоты реализации указатель на выделенную память должен храниться вместе со всей информацией, необходимой для управления динамическим массивом.
    - поскольку адрес массива может измениться, программа должна обращаться к элементами массива по индексам.
    - благодаря маленькому начальному размеру массива, программа сразу же «проверяет» код, реализующий выделение памяти

```
int append(struct dyn_array *d, int item)
{
    if (!d->data)
    {
        d->data = malloc(INIT_SIZE * sizeof(int));
        if (!d->data)
            return -1;
        d->allocated = INIT_SIZE;
    }
    else
        if (d->len >= d->allocated)
        {
            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;
}

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

    memmove(d->data + index, d->data + index + 1,
                        (d->len - index - 1) * sizeof(int));
    d->len--;

    return 0;
}
```

30. Линейный односвязный список

- описание типа и основные операции
    - размер списка ограничен лишь свободной памятью
    - удобно хранить списки структур с различными полями
    - списки легко переформировать, изменяя несколько указателей
    - при удалении или вставки элемента в список адрес остальных не меняется



    - добавление элемента в список
    - поиск элемента в списке
    - обработка всех элементов списка
        - печать
        - вставка
        - удаление

- сортировка

31. Двоичные деревья поиска

- описание типа
    - дерево - связный ациклический граф
    - двоичным деревом поиска называют дерево, все вершины которого упорядочены, каждая вершина имеет не более двух потомков (назовём их левым и правым), и все вершины, кроме корня, имеют родителя.
- основные операции
    - добавление элемента в дерево
    - поиск в дереве
    - обход дерева
- рекурсивный и нерекурсивный поиск

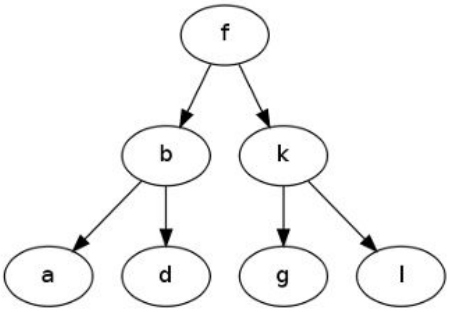
```
struct tree_node* lookup_1(struct tree_node *tree,
                                                const char *name)
{
    int cmp;

    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);
}

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;
}
```

- язык DOT

    - 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.

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

```
// Оформление на странице Trac
{{{
#!graphviz
digraph test_tree {
f -> b;
f -> k;
b -> a;
b -> d;
k -> g;
k -> l;
}
}}}
```



32 - 33. Директивы препроцессора, макросы, условная компиляция, # ##

- классификация директив препроцессора
    - макроопределения #define, #undef
    - директива включения файлов #include
    - директива условной компиляции #if, #ifdef <...>
    - остальные
        - pragma
        - error
        - line
- правила, справедливые для всех директив препроцессора
    - директивы всегда начинаются с символа "#".
    - любое количество пробельных символов может разделять лексемы в директиве.
    - директива заканчивается на символе '\n'.
    - директивы могут появляться в любом месте программы
    - любое количество пробельных символов могут разделять лексемы в директиве.
    - директива заканчивается на символе '\n'
- макросы (простые, с параметрами, с переменным числом параметров, предопределенные)
    - простые макросы используются в качестве
        - имен для числовых, символьных и строковых констант

        - незначительные изменения синтаксиса языка
        - переименования типов
        - управление условной компиляцией
    - макросы с параметрами (список параметров мб пустым)
        - список-замены макроса может содержать другие макросы.
        - препроцессор заменяет только целые лексемы, не их части.
        - определение макроса остается «известным» до конца файла, в котором этот макрос объявляется.
        - макрос не может быть объявлен дважды, если эти объявление не тождественны.
        - макрос может быть «разопределен» с помощью директивы #undef.
    - предопределенные макросы
        - __LINE__ - номер текущей строки (десятичная константа)
        - __FILE__ - имя компилируемого файла
        - __DATE__ - дата компиляции
        - __TIME__ - время компиляции
        - эти идентификаторы нельзя переопределять или отменять директивой undef

- сравнение макросов с параметрами и функций

    + преимущества
        + программа может работать немного быстрее;
        + макросы "универсальны"
    - недостатки
        - скомпилированный код становится больше
        - типы аргументов не проверяются;
        - нельзя объявить указатель на макрос;
        - макрос может вычислять аргументы несколько раз.

- скобки в макросах; создание длинных макросов

    - если список-замены содержит операции, он должен быть заключен в скобки.
    - если у макроса есть параметры, они должны быть заключены в скобки в списке-замены
    - \

- директивы условной компиляции, использование условной компиляции

    - программа, которая должна работать под несколькими операционными системами;
    - программа, которая должна собираться различными компиляторами;
    - начальное значение макросов;
    - временное выключение кода

```
#if defined(OS_WIN)
...
#elif defined(OS_LIN)
...
#elif defined(OS_MAC)
...
#endif
```

```
#ifndef BUF_SIZE
#define BUF_SIZE  256
#endif
```

```
#if 0
for(int i = 0; i < n; i++)
    a[i] = 0.0;
#endif
```

- директива error и pragma
    - #error сообщение
        #if defined(OS_WIN)
        ...
        #elif defined(OS_LIN)
        ...
        #elif defined(OS_MAC)
        ...
        #else
        #error Unsupported OS!
        #endif
    - директива #pragma позволяет добиться от компилятора специфичного поведения
        - once - нестандартная, но широко распространенная препроцессорная директива, разработанная для контроля за тем, чтобы конкретный исходный файл при компиляции подключался строго один раз. То есть, #pragma once применяется для тех же целей, что и include guard, но требует меньше кода и не допускает возможности коллизии имен.
        - pack - запаковать класс,разместить его члены непосредственно друг за другом в памяти, некоторые или все члены могут быть выровнены по границе меньшей, чем выравнивание целевой архитектуры, заданное по умолчанию. Обеспечение контроля на уровне объявления данных
            - при изменении выравнивания структуры она может занимать меньше места в памяти, но возможно снижение

производительности или даже возникновение аппаратного исключения для не выровненного доступа.
- show (необязательно)- отображает текущее байтовое значение выравнивания упаковки. Значение отображается в предупреждении.
- push (необязательно)-помещает текущее значение выравнивания упаковки во внутренний стек компилятора и задает для текущего выравнивания упаковки значение n. Если значение n не указано, текущее значение выравнивания упаковки не помещается в стек.
- pop (необязательно)-удаляет запись из вершины внутреннего стека компилятора.

- операция “#”, операция “##”

    - «Операция» # конвертирует аргумент макроса в строковый литерал.
    - «Операция» ## объединяет две лексемы в одну
        - лексема - группа ассоциированных слов

34. Inline - functions

- пожелание компилятору заменить вызовы функции последовательной вставкой кода самой функции.
- inline-функции по-другому называют встраиваемыми или поставляемыми
- в C99 inline означает, что определение функции предоставляется только для подстановки и где-то в программе должно быть другое такое же определение этой же функции
- extern - чтобы избавиться от unresolved reference или static, убрать ключевое слово inline из определения функции, компилятор умный, сам разберется :)

35. Списки из ядра ОС Linux - списки Беркли

- идеи реализации
    - циклический двусвязный список
        - каждый узел двунаправленного (двусвязного) циклического списка (ДЦС) содержит два поля указателей - на следующий и на предыдущий узлы. Указатель на предыдущий узел корня списка содержит адрес последнего узла. Указатель на следующий узел последнего узла содержит адрес корня списка.
    - интрузивный список
        - это такой список, в котором каждый элемент содержит ссылки на соседей.
    - универсальный список
        - поле data которого может принимать любой тип данных

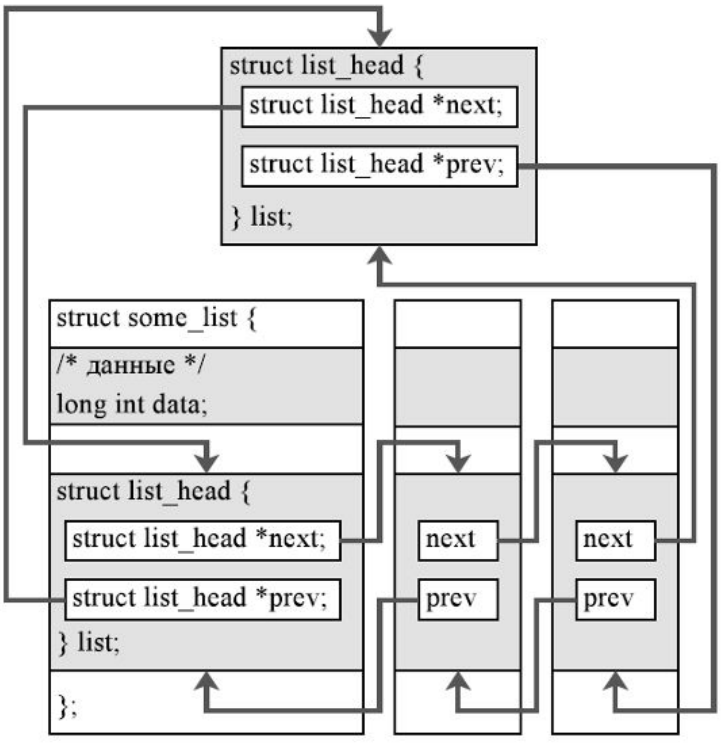
- описание типа
    - список Беркли – это циклический двусвязный список, в основе которого лежит следующая структура:

```
struct list_head
{
        struct list_head *next, *prev;
};
```

    - в отличие от обычных списков, где данные содержатся в элементах списка, структура list_head должна быть частью сами данных

```
struct data
{
        int i;
        struct list_head list;
         ...
};
```

- структуру struct list_head можно поместить в любом месте в определении структуры.
- struct list_head может иметь любое имя.
- в структуре может быть несколько полей типа struct list_head.



- добавление элемента в начало и конец (list_add, list_add_tail)

```
static inline void __list_add(struct list_head *new,
                  struct list_head *prev, struct list_head *next)
{
   next->prev = new;
   new->next = next;
   new->prev = prev;
   prev->next = new;
}

static inline void list_add(struct list_head *new,
                                  struct list_head *head)
{
   __list_add(new, head, head->next);
}

static inline void list_add_tail(struct list_head *new,
                                       struct list_head *head)
{
   __list_add(new, head->prev, head);
}
```

```
#define list_for_each(pos, head) \
   for (pos = (head)->next; pos != (head); pos = pos->next)

#define list_for_each_prev(pos, head) \
   for (pos = (head)->prev; pos != (head); pos = pos->prev)

#define list_for_each_entry(pos, head, member) \
   for (pos = list_entry((head)->next, typeof(*pos), member); \
        &pos->member != (head); \
        pos = list_entry(pos->member.next, typeof(*pos), member))


#define list_for_each_safe(pos, n, head) \
   for (pos = (head)->next, n = pos->next; pos != (head); \
        pos = n, n = pos->next)
```

```
#define list_entry(ptr, type, member) \
  container_of(ptr, type, member)

#define container_of(ptr, type, field_name) ( \
   (type *) ((char *) (ptr) - offsetof(type, field_name)))

#define offsetof(TYPE, MEMBER) \
     ((size_t) &((TYPE *)0)->MEMBER)
```

36. Битовые операции, битовые поля

- битовые операции: сдвиг влево, сдвиг вправо, битовое «НЕ», битовое «И», битовое «исключающее ИЛИ», битовое «ИЛИ» и соответствующие им операции составного присваивания

&	и
|	или
^	исключающее или
~	дополнение
>>	сдвиг вправо
<<	сдвиг влево

- битовые операции применимы только к целочисленным переменным.

- битовые операции обычно выполняют над беззнаковыми целыми, чтобы не было путаницы со знаком

-

<<=	Присваивание со сдвигом влево	X <<= Y
>>=	Присваивание со сдвигом вправо	X >>= Y
&=	Присваивание с побитовым «И»	X &= Y
^=	Присваивание с побитовым исключающим «ИЛИ»	X ^= Y
|=	Присваивание с побитовым «ИЛИ»	X |= Y

~(унар.)	Побитовое «НЕ»	~X
<<	Сдвиг влево	X << Y
>>	Сдвиг вправо	X >> Y
&	Побитовое «И»	X & Y
^	Побитовое исключающее «ИЛИ»	X ^ Y
|	Побитовое «ИЛИ»	X | Y

- использование битовых операций для обработки отдельных битов и последовательностей битов

  - проверка битов

```
unsigned char a    = 0x46;  // 01000110b
unsigned char b    = 0x44;  // 01000100b
unsigned char mask = 0x06;  // 00000110b

printf("a & mask %x, res %d\n", a & mask, (a & mask) == mask);
printf("b & mask %x, res %d\n", b & mask, (b & mask) == mask);
```

  - обнуление битов

```
unsigned char a      = 0x46;  // 01000110b
unsigned char mask_1 = 0xbf;  // 10111111b
unsigned char mask_2 = 0xf9;  // 11111001b

printf("a & mask_1 %x\n", a & mask_1);
printf("a & mask_2 %x\n", a & mask_2);
```

  - установка битов

```
unsigned char a      = 0x40;  // 01000000b
unsigned char mask_1 = 0x06;  // 00000110b
unsigned char mask_2 = 0x44;  // 01000100b

printf("a | mask_1 %x\n", a | mask_1);
printf("a | mask_2 %x\n", a | mask_2);
```

- смена значений битов

```
unsigned char a      = 0x46;  // 01000110b
unsigned char mask_1 = 0x44;  // 01000100b
unsigned char mask_2 = 0xFF;  // 11111111b

printf("a ^ mask_1 %x\n", a ^ mask_1);
printf("a ^ mask_2 %x\n", a ^ mask_2);
```

- побитовый сдвиг вправо

```
unsigned char a = 0xFF;  // 11111111b

printf("a >> 1 = %2x\n", a >> 1);
printf("a >> 4 = %2x\n", a >> 4);
```

- побитовый сдвиг влево

```
unsigned char a = 0x01;  // 00000001b

printf("a << 1 = %2x\n", a << 1);
printf("a << 4 = %2x\n", a << 4);
```

```c
#include <stdio.h>

int main(void)
{
    unsigned char a = 0x01;
    unsigned char b = 0x02;

    if (a && b)
        printf("a && b true\n");
    else
        printf("a && b false\n");

    if ((b && 1) == 1)
        printf("odd\n");
    else
        printf("even\n");

    a = 0;
    if (a && b / a)
        printf("true\n");
    else
        printf("false\n");

    return 0;
}
```

```c
#include <stdio.h>

int main(void)
{
    unsigned char a = 0x01;
    unsigned char b = 0x02;

    if (a & b)
        printf("a & b true\n");
    else
        printf("a & b false\n");

    if ((b & 1) == 1)
        printf("odd\n");
    else
        printf("even\n");

    a = 0;
    if (a & b / a)          // !!!
        printf("true\n");
    else
        printf("false\n");

    return 0;
}
```


- битовые поля:
    - описание
        - особый тип структуры, определяющей, какую длину имеет каждый член в битах.
        - битовые поля должны объявляться как целые, unsigned или signed

```c
struct имя_структуры
{
    тип имя1: длина;
    тип имя2: длина;
    ...
    тип имяN: длина;
};
```

    - использование
        - применяются для максимально полной упаковки информации, если не важна скорость доступа к этой информации
        - увеличение пропускной способности канала при передаче информации по сети
        - для уменьшения размера информации при хранении
    - ограничения использования
        - нельзя получить адрес переменной битового поля
        - переменные битового поля не могут перемещаться в массив
        - нельзя быть уверенным в порядке изменения бит - архитектурно зависим

37. Неопределенное поведение

- особенности вычисления выражений с побочным эффектом

    - модификация данных
    - обращение к переменным, объявленным как volatile
    - вызов системной функции, которая производит побочные эффекты

- понятие «точка следования»

    - это точка в программе, в которой программист знает какие выражения (или подвыражения) уже вычислены, а какие выражения (или подвыражения) еще нет
    - компилятор вычисляет выражения. Выражения будут вычисляться почти в том же порядке, в котором они указаны в исходном коде: сверху вниз и слева направо
        - в конце полного выражения
        - между вычислением левого и правого операндов в операциях && ||
        - между вычислением первого и второго или третьего операндов в тернарной операции
        - перед входом в вызываемую функцию
        - в объявлении с инициализацией на момент завершения вычисления инициализированного выражения

38. Библиотеки

- статические и динамические библиотеки:

    - назначение
        - включают в себя
            - откомпилированный файл самой библиотеки:
            - библиотеки меняются редко
                - нет причин перекомпилировать каждый раз;
                - двоичный код предотвращает доступ к исходному коду
            - код библиотеки помещается в исполняемый файл
    + исполняемый файл включает в себя все необходимое
    + не возникает проблем с использованием не той версии библиотеки
    - размер
    - при обновлении библиотеки программу нужно пересобрать

    - назначение
        - загружаются в приложение во время выполнения, код библиотеки не помещается в исполняемый файл
        + несколько программ могут разделять одну библиотеку
        + меньший размер приложения
        + средство реализации плагинов
        + модернизация библиотеки не требует перекомпиляции программы
        + могут использовать программы на разных языках
        - требуется наличие библиотеки на устройстве
        - версионность библиотек

- статические библиотеки: создание, использование при компоновке

    - сборка библиотеки
        компиляция
        - gcc -std=c99 -Wall -Werror -c arr_lib.c
        упаковка
        - ar rc libarr.a arr_lib.o
        индексирование
        - ranlib libarr.a
        сборка приложения
        - gcc -std=c99 -Wall -Werror main.c libarr.a -o test.exe
        - gcc -std=c99 -Wall -Werror main.c -L. -larr -o test.exe

- динамические библиотеки:

    - сборка библиотеки - динамическая компоновка
        компиляция
        - gcc -std=c99 -Wall -Werror -c arr_lib.c
        компоновка

        - gcc -shared arr_lib.o -Wl,--subsystem,windows -o arr.dll
          сборка приложения
        - gcc -std=c99 -Wall -Werror -c main.c
        - gcc main.o -L. –larr -o test.exe
    - динамическая загрузка
          сборка библиотеки
              компиляция
            - gcc -std=c99 -Wall -Werror -c arr_lib.c
              компоновка
            - gcc -shared arr_lib.o -Wl,--subsystem,windows -o arr.dll
              сборка приложения
            - gcc -std=c99 -Wall -Werror main.c –o test.exe

- особенности использования динамических библиотек с приложением, реализованным на другом (по отношению к библиотеке) языке программирования.

    - ctypes
        - чтобы загрузить библиотеку необходимо создать объект класс CDLL:
              import ctypes
              lib = ctypes.CDLL('example.dll')

        - классы для работы с библиотеками
            - CDLL - return int
            - OleDLL - HRESULT
            - WinDLL - int
        - выбор класса зависит от соглашения о вызовах, которые использует библиотека
        - описание заголовков функций библиотеки с использованием нотаций и типов, известных Python
        - указать аргументы argtypes и restype для правильной конвертации интерпретатором Python
        - для аргументов, использующих указатели, необходимо с помощью описанных в модуле совместимых типов создать объект и передавать именно его
        - avg ожидает получить указатель на массив
        - учитывать преобразование списка, кортежа и тд в массив
    - основная проблема использования - написание большого количества сигнатур для функций, функций-оберток
    - необходимо детально представлять внутреннее устройство типов Python, и то, каким образом они могут быть преобразованы в типы на Си
    - альтернативы
        - Swig
        - Cython

39. Абстрактный тип данных

- понятие «модуль», преимущества модульной организации программы

    - программу удобно рассматривать как набор независимых модулей
    - модуль состоит из двух частей: интерфейса и реализации
    - интерфейс описывает, что модуль делает. Он определяет идентификаторы, типы и подпрограммы, которые будут доступны коду, использующему этот модуль
    - реализация описывает, как модуль выполняет то, что предлагает интерфейс
    - у модуля есть один интерфейс, но реализаций, удовлетворяющих этому интерфейсу, может быть несколько. Часть кода, которая использует модуль, называют клиентом.Клиент должен зависеть только от интерфейса, но не от деталей его реализации.
    + абстракция (когда интерфейсы модулей согласованы, ответственность за реализацию каждого модуля делегируется определенному разработчику)
    + повторное использование (Модуль может быть использован в другой программе)
    + сопровождение (можно заменить реализацию любого модуля, например, для улучшения производительности или переноса программы на другую платформу)

- разновидности модулей

    - набор данных
        - набор связанных переменных и/или констант. В Си модули этого типа часто представляются только заголовочным файлом. (float.h, limits.h.)
    - библиотека
        - набор связанных функций
    - абстрактный объект
        - набор функций, который обрабатывает скрытые данные.
    - абстрактный тип данных
        - интерфейс, который определяет тип данных и операции над этим типом. Тип данных называется абстрактным, потому что интерфейс скрывает детали его представления и реализации.

- организация модуля в языке Си

    - интерфейс описывается в заголовочном файле (*.h).
    - в заголовочном файле описываются макросы, типы, переменные и функции, которые клиент может использовать.
    - клиент импортирует интерфейс с помощью директивы препроцессора include.
    - реализация интерфейса в языке Си представляется одним или несколькими файлами с расширением *.c.
    - реализация определяет переменные и функции, необходимые для обеспечения возможностей, описанных в интерфейсе.

    - реализация обязательно должна включать файл описания интерфейса, чтобы гарантировать согласованность интерфейса и реализации.

- неполный тип в языке Си

    - типы которые описывают объект, но не предоставляют информацию нужную для определения его размера
    - пока тип неполный его использование ограничено.
    - описание неполного типа должно быть закончено где-то в программе.
    + можно
        + определение переменной типа неполный тип
        + передавать эти переменные, как аргументы в функцию
    - нельзя !
        - применять операцию обращения к полю
        - разыменовывать переменные типа неполный тип

- общие вопросы проектирования абстрактного типа данных

    - именование
        - имеет смысл добавлять название АТД в название функций
    - обработка ошибок
        - интерфейс обычно описывает проверяемые ошибки времени выполнения и непроверяемые ошибки времени выполнения и исключения
        - реализация не дает гарантии обнаружения непроверяемых ошибок времени выполнения
        - хороший интерфейс избегает ошибки времени выполнения, но описывает их
        - реализация гарантирует обнаружение проверяемых ошибок времени выполнения и информирование клиентского кода
    - общие АТД
        - стек должен принимать данные любого типа без модификации stack.h
        - программа не может создать два стека с данными разного типа
        - использование void* в качестве типа элемента
        - элементами могут быть динамически выделяемые объекты, но не данные базовых типов
        - стек может содержать указатели на что угодно, очень сложно гарантировать правильность.

Дополнение (с семинаров)

1. Указатель на константу - cost int *
2. Константный указатель - int* const
3. Константный указатель на константу - const int* const
4. Изменение константы

```
int main(void)
{
    const int b = 0xBEEFBEEF;
    int* r = &b;
    *r = 0xDEADDEAD;
}
```

```
int main(void)
{
    int a = 0xDEADDEAD;
    const int b = 0xBEEFBEEF;
    *((&a)+1) = 0xDEADDEAD;
}
```

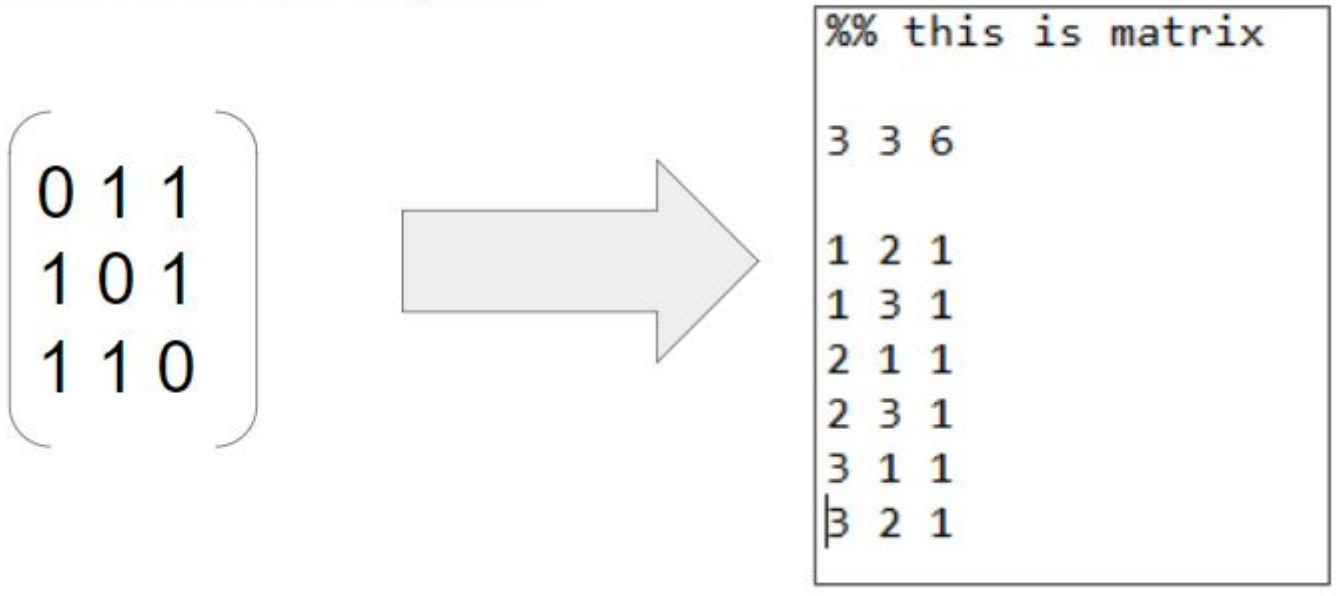
5. Статические массивы - стек, динамические - куча
6. alloca - выделить переданный размер в байтах на куче

`int, char, double, void`	целое, целое, вещественное, неопределенный тип
`[], [][]`	массив типа …, массив массивов типа
`[N]`	массив из N элементов типа...
`(type)`	функция, принимающая аргумент типа type и возвращающая ...
`*`	указатель на ...
`const`	константный (неизменяемы)

Matrix market format

```
%%MatrixMarket matrix coordinate real general
%=================================================================================
% Этот файл  содержит разреженную MxN матрицу с L
% ненулевыми элементами в формате Matrix Market:
%
% +-------------------------------+
% |%%MatrixMarket                 | <--- заголовок
% |%                              | <--+
% |% comments                     |    |-- 0 или больше комментариев
% |%                              | <--+
% |  M  N  L                      | <--- строк, столбцов, элементов
% |  I1  J1  A(I1, J1)            | <--+
% |  I2  J2  A(I2, J2)            |    |
% |  I3  J3  A(I3, J3)            |    |-- L строк
% |       . . .                   |    |
% |  IL JL  A(IL, JL)             | <--+
% +-------------------------------+
%
% Индексы начинаются с 1, т.е. A(1,1) это первый элемент
%%================================================================================
```

MatrixMarket формат

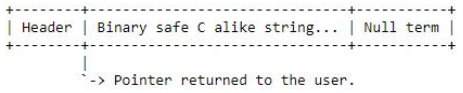


7. Сложные объявления
    a. Читать по часовой стрелке, при этом отправной точкой является идентификатор.
    b. Когда встречается очередной элемент объявления - заменяем его на слово
    c. Скобки “( )” могут использовать для изменения приоритета. Пока внутри скобок “( )” не прочитаны все элементы, “покидать” их нельзя.
8. Структуры и строки
    a. Не известен размер массива => Buffer overflow
    b. Узнать размер массива => \0 => O(n)

c. Функции не работают со строками содержащими \0
d. Неопределенное поведение при передаче NULL
e. Отсутствие операций split, replace, ...
f. aliasing

Структуры и строки. SDS.

- Информация о строке находится в заголовке до(!) возвращаемого указателя
- Совместимы со “стандартными” C строками
- “Простые”, “удобные”, “быстрые”...



Структуры и строки. SDS. Pros & Cons

- Большая часть методов возвращает новый указатель, старый становится мусорным
- Необходимо обновлять все указатели на строку, если она изменяется
+ Можно передавать SDS строки в стандантрые str*() функции
+ Обычная адресная арифметика для доступа к символам
+ Более эффективное выделение и использование памяти

```
bool cycled(struct list *head)
{
        struct list *fast = head;
        struct list *slow = head;

        // v(fast) = 2*v(slow), если fast дошел до конца - цикла нет

        bool flag = false;

        if (head == NULL)
                return false;

        while (true)
        {
                if (fast == NULL)
                        return false;
                if (fast == slow)
                        return true;
                if (flag)
                        slow = slow->next;
                flag = !flag;
                fast = fast->next;
        }
}
```

```
list *tmp = source_list, *prev = NULL;
int i = 0, cyclic = 0;

for(;;) {
  if(tmp == NULL) {

     cyclic = 1;

     break;
  }
  if((i > 0) && (tmp == source_list)) {
     break;
  }
  i++;
  tmp = tmp->next;
}
if(cyclic)
 printf("Nodes number in the cyclic list is %d\n", i);
else
 printf("This list was not cyclic one.\n");
```

Методы queue.h

SLIST_ENTRY(*TYPE***);**
SLIST_HEAD(*HEADNAME*, *TYPE***);**
SLIST_INIT(SLIST_HEAD **head***);*
SLIST_INSERT_AFTER(SLIST_ENTRY **listelm*,
TYPE **elm*, **SLIST_ENTRY** *NAME*);
SLIST_INSERT_HEAD(SLIST_HEAD **head*,
TYPE **elm*, **SLIST_ENTRY** *NAME*);
SLIST_REMOVE(TYPE **elm*, **SLIST_ENTRY** *NAME*);

9. Inline functions - insert manage

- __attribute__((always_inline))
- __attribute__ ((noinline))
- -Winline
- -finline-functions-called-once (01)
- -finline-small-functions (02)
- -finline-functions (03)

Битовые операции. Simhash. Алгоритм.

Simhash - функция локально-чувствительного хэша. Для “похожих” аргументов принимает близкие значения. Используется для вычисления схожести последовательностей

Алгоритм:

1. Выбрать размер хэша L бит (пусть 32)
2. Разбить входное значение на пересекающиеся N-граммы
3. Для каждой N-граммы вычислить значение некоторой hash-функции - H
4. Пусть V[i] == 0, для любого i = [0, L). Если H[i] == 1, то V[i] += 1; если H[i] == 0, то V[i] -= 1
5. V[i] = 0, если V[i] < 0; V[i] = 1, если V[i] > 0.
6. Полученный битовый вектор V1 (|V| = L) от одной последовательности сравнивается с битовым вектором V2 другой последовательности. Если **расстояние Хэмминга** между ними небольшое (D(V1, V2) << L) , последовательности похожи

Определение: Расстояние Хэмминга — число позиций, в которых соответствующие символы двух слов одинаковой длины различны.

- Код Хэмминга —самоконтролирующийся и самокорректирующийся код. Построен применительно к двоичной системе счисления. Позволяет исправлять одиночную ошибку (ошибка в одном бите) и находить двойную.
- Основан на принципе проверки на четность числа единичных символов: к последовательности добавляется такой элемент, чтобы число единичных символов в получившейся последовательности было четным.

$$r_1 = i_1 \oplus i_2 \oplus \ldots \oplus i_k.$$

знак $\oplus$ здесь означает сложение по модулю 2

$$S = i_1 \oplus i_2 \oplus \ldots \oplus i_n \oplus r_1.$$

$S = 0$ — ошибки нет, $S = 1$ однократная ошибка.

10. Универсальная система анализа, трансформации и оптимизации программ, реализующая виртуальную машину с RISC подобными инструкциями - LLMV

- Linker name = <имя_библиотеки>
- Soname это = **lib<имя_библиотеки>.so.<версия>**
- Fully qualified soname = **<полный_путь_до_файла> + soname**
- Real name = **<soname>.<минорная_версия>.<релизная_версия>**

soname и linker name - ссылка на **real name**

Программы внутри себя используют **soname (поле DT_NEEDED)**

soname хранится внутри **.so** файла в поле **DT_SONAME**

Linker name создается установщиком (пакетным менеджером)

Shared objects. Обратная совместимость.

- Изменилось поведение функции
- Изменились экспортируемые данные
- Удалена экспортируемая функция
- Изменилась сигнатура экспортируемой функции

DLL (Dependency) hell

- Проблемы
  - **Множество зависимостей**
  - **Длинные цепочки зависимостей**
  - **Конфликтующие зависимости**
  - **Циклические зависимости**
- Решения
  - **Нумерация версий**
  - **Параллельная установка различных версий ПО**
  - **Хороший пакетный менеджер**
  - **Портативные приложения**

СТРОКИ ->

- strcpy(1, 2) - для копирования содержимого из str2->str1, возвращает указатель на str1, если строки перекрываются, то поведение непредсказуемо
- strncpy(1, 2) - копирует n элементов s2->s1, если n>len(s2) в s1 записывается столько ‘\0’, чтобы общая длина записи была = n
- strlen(s) - считает количество символов строки до встречи ‘\0’, возвращает длину строки
- strcat(s1, s2) - добавляет в s1 s2, символ ‘\0’ помещается в конец объединенных строк, возвращает указатель на массив s1
- strncat(s1, s2, n) - добавляет n символов s1 в s2 + символ конца строки
- strcmp(s1, s2) - сравнивает две строки, начиная с первых символов, сравнение идет поочередно до достижения ‘\0’
- strncmp(s1, s2, n) - сравнивает только n символов s1 с n символами s2
- strdup(s) - дублирует переданную строку, возвращает указатель на копию
- strndup(s, n) - дублирует n символов строки, если n < длины, то копирует n символов, иначе всю строку, возвращает указатель на копию (память под строки выделяется на куче)
- snprintf - контроль длины данных, передается размер буфера, возвращает количество символов, записанных в буфер
- strtok(str, delim) - функция для поиска лексем в строке, последовательными вызовами разбивает строку на лексемы по разделителю, возвращает указатель на первую найденную лексему в строке, если таковых нет, вернет пустой указатель

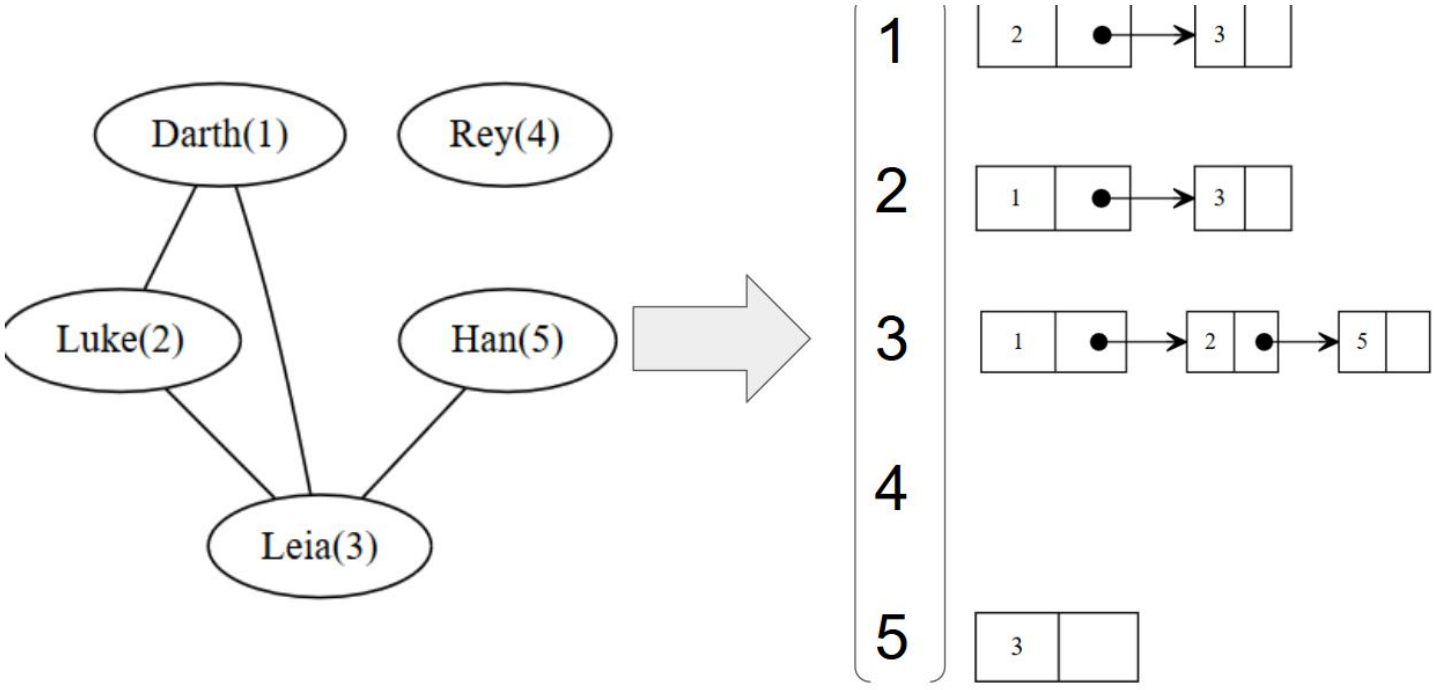
- alloca
    - не стоит освобождать пространство, выделенное под alloca
    - архитектурно-зависима, так как выделяет память на стеке
    - автоматически освобождает кадр стека
    - эффективен

Массивы передаются по указателю, но передать его по значению можно используя структуру-обертку

При переполнении буфера gets начнет записывать данные в чужую память
strcpy(dst, src) - src<dst => src + (dst - len(src))

Структуры передаются по значению

Список смежности вершин графа


Darth(1)
Rey(4)
Luke(2)
Han(5)
Leia(3)
1
2
3
4
5
2
3
1
3
1
2
5
3