1. Указатель на void. Стандартные функции обработки областей памяти.

- Для чего void * используется в Си (обработка областей памяти и передача в функцию чего угодно)
- memcpy, memmove, memset, memcmp

Используется:

- полезен для **ссылки на произвольный участок памяти**, независимо от размещенного там объекта;

  ```
  void *memcpy(void *dst, const void *src, size_t n);
  ```

  Например, мы реализуем функцию, которая *копирует одну область памяти в другую*. Нас интересует только то, **где эта память размещается** и размер хранящихся данных.

- позволяет передавать в функцию **указатель на объект любого типа**

```
void qsort(void *first, size_t number, size_t size, int
(*comparator)(const void *, const void *));
```

Особенности использования:

- допускается **присваивание указателя** типа **void указателю любого другого типа** без явного преобразования типа указателя.

```
double a = 1.0;
double *p_a = &a;
void *p_aa = p_a;
p_a = p_aa;
```

- указатель **нельзя разыменовывать**, так как неизвестен размер того типа, на который указывает указатель.

```
printf("%lf\n", *p_aa); //ERROR
```

- к указателям типа void **не применима адресная арифметика** (по тем же соображениям, что и пункт выше)

```
p_aa++; //ERROR
```

Функции обработки областей памяти:

*все функции определены в *string.h*

прототип функции	описание
void *memcpy(void *restrict dst, const void *restrict src, size_t count)	Копирует `count` байтов блока памяти, на который ссылается `src`, во второй блок памяти, на который ссылается указатель `dst` Поведение **не определено:** - **блоки** памяти **перекрываются** - недействительные указатели Возвращает указатель на `dst`
void *memmove(void *dst, const void *src, size_t count)	Копирует `count` байтов блока памяти, на который ссылается `src`, во второй блок памяти, на который ссылается указатель `dst`. Копирование происходит через промежуточный буфер, что **позволяет использовать функцию при перекрытии областей памяти** Поведение **не определено:** - недействительные указатели

	Возвращает указатель на `dst`
`int memcmp(const void *s_1, const void *s_2, size_t count)`	Сравнивает первые `count` байтов блока памяти указателя `s_1` `c` первыми `count` байтов блока памяти указателя `s_2` Поведение **не определено:** - недействительные указатели Возвращает: - **0:** Содержимое обоих блоков памяти **равны** - **> 0:** Первый блок памяти **больше,** чем второй - **< 0:** Первый блок памяти **меньше,** чем второй
`void *memset(void *dst, int c, size_t n)`	Заполняет `count` байт по адресу dst. Код заполняемого символа `-` `c` Возвращает указатель на `dst`

2. Функции динамического выделения памяти.

- malloc, calloc, realloc, free: основной алгоритм использования, особенности работы
- Стоит ли использовать явное приведение типа и почему
- Выделение 0 байт памяти
- Обработка NULL, возвращенного функцией

Особенности:

- все функции находятся в *stdlib.h*
- функции не создают переменную, они лишь выделяют область памяти.

  В качестве результата функции **возвращают** адрес расположения этой области в памяти компьютера, те **указатель**
- тип возвращаемого указателя – **void**
- в случае ошибки при выделении памяти возвращают значение **NULL**
- после использования необходимо освободить память с помощью **free**

прототип функции	принцип работы
`void *malloc(size_t size);`	- выделяет блок памяти, указанный в байтах - блок памяти **не инициализируется**

	- для вычисления необходимого размера памяти используют операцию `sizeof`
`void *calloc(size_t nmemb, size_t size);`	- выделяет блок памяти для массива из `nmeb` элементов, размер каждого из которых size - **инициализирует** блок памяти **0**
`void free(void *ptr);`	- **освобождает** ранее выделенный блок памяти, на который указывает указатель - если `*ptr == NULL`, то ничего не происходит - если указатель **не был получен** с помощью malloc, calloc, realloc, то **поведение не определено**
`void *realloc(void *ptr, size_t size);`	● `ptr == NULL && size != 0`: выделение памяти, как malloc ● `ptr != NULL && size == 0`: освобождение памяти, как free ● `ptr != NULL && size != 0`: перевыделение памяти. В худшем случае: - выделить новую область - скопировать данные из старой в новую - освободить старую область

<u>+/- явного приведения:</u>

+	-
- компиляция с помощью **C++** компилятора - у malloc до ANSI был прототип `char *malloc` - дополнительная проверка аргументов	- начиная с ANSI приведение не нужно - может скрыть ошибку, если не подключить *stdlib.h* - в случае изменения типа указателя придется менять и тип в приведении

<u>Что будет, если выделить 0 байт памяти?</u>

Результат вызова функций `malloc, calloc и realloc`, когда запрашиваемый размер блока равен 0, зависит от реализации компилятора:

- вернется нулевой указатель;
- вернется *«нормальный»* указатель, но его нельзя использовать для разыменования;

Поэтому перед вызовом этих функций необходимо убедиться, что размер блока не равен **0!**

P. S. В случае **realloc**, при вызове с 0 блока памяти, будет работать как **free**

Обработка NULL, возвращенного функцией:

- Отладчик: **valgrind, Dr. Memory**
- Возвращение ошибки
- Ошибка сегментации **(segfault)**
- Аварийное завершение **(abort)** – идея Кернигана и Ритчи
- Восстановление(recovery) – **xmalloc** из git

3. Выделение памяти под динамический массив. Типичные ошибки при работе с динамической памятью

- функция, возвращающая динамический массив как результат и через параметр

	как возвращаемое значение	как параметр функции
Прототип	`int *create_array(FILE *f, size_t *n);`	`int create_array(FILE *f, size_t *n, int **arr);`

Вызов	`int *arr;`
	`size_t n;`
	`arr = create_array(f, &n);`
		`int *arr, rc;`
		`size_t n;`
		`rc = create_array(f, &n, &arr);`

Типичные ошибки при работе с динамической памятью:

- **неверный расчет** кол-ва выделяемой памяти

`struct date *p = NULL`	`int n = 5;`

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

```
int *arr = NULL;

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

- **отсутствие проверки** после выделения памяти
- утечки памяти

```
int *p = NULL;

p = malloc(sizeof(int)); // так как может быть
NULL

if (p)

{

      *p = 5;

      p = malloc(sizeof(int));

      …
```

- **wild pointer:** использование неинициализированного указателя

```
int *p;

if (scanf(“%d”, p) == 1)

{

      …
```

- **dangling pointer:** использование указателя сразу же после освобождения

```
free(p);

*p = 7;

printf(“%d\n”, *p);
```

- **изменение указателя,** который вернула функция выделения памяти

```
int *ptr = malloc(sizeof(int));
ptr++;
printf(“%d\n”, *ptr);
```

- **двойное освобождение** памяти
- **освобождение не выделенной** памяти или **не динамической** памяти
- выход за границы динамического массива

4. Указатели на функцию. Функция qsort.

- Для чего используются
  - callback
    - для обработки данных
    - для передачи сигнала (событийное программирование)
  - таблица переходов
  - динамическое связывание
- Как описываются указатели на функцию, инициализируются, как выполняется вызов функции по указателю, особенности
- qsort

Для чего используются:

- **функция обратного вызова (callback)** - передача исполняемого кода в качестве одного из параметров другого кода. Обратный вызов позволяет в функции исполнять код, который задаётся в аргументах при её вызове

```
void populate_array(int *array, size_t size, int (*get_next_value)(void))
{
	for (size_t i=0; i<array; i++)
		array[i] = get_next_value();
}
int get_next_value(void)
{
	return rand();
}
```

```
int main(void)

{

	int array[10];

	populate_array(array, 10, get_next_value);

	...

}
```

- **таблица переходов (jump table)** - это метод передачи управления программой (ветвления) другой части программы (или другой программе, которая могла быть динамически загружена) с использованием таблицы инструкций перехода

```
typedef void (*Handler)(void); // Указатель на функцию-обработчик

void func3 (void) { printf( "3\n" ); } // функции
void func2 (void) { printf( "2\n" ); }
void func1 (void) { printf( "1\n" ); }
void func0 (void) { printf( "0\n" ); }

Handler jump_table[4] = {func0, func1, func2, func3};

int main (int argc, char **argv) {
	int value;

	/* Convert first argument to 0-3 integer (Hash) */
	value = atoi(argv[1]) % 4;
	if (value < 0) {
	    value *= -1;
	}

	// Вызов соответствующей функции
	jump_table[value]();
}
```

- **динамическое связывание (blinding)** - подключение к программе предварительно откомпилированного и хранящегося в виде отдельного т. н. «файла динамической библиотеки» модуля, выполняемое с помощью специальной команды в ходе работы основной программы.

Описание указателя на функцию:

<возвращаемый тип>(*<имя>)(<тип аргументов>)

```
typedef <возвращаемый тип>(*<имя_t>)(<тип аргументов>)
```

- Объявление указателя на функцию

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

- Получение адреса функции

  ```
  result = trapezium(1.0, 2.0, 2, &sin /* sin */)
  ```

- Вызов функции по указателю

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

Особенности использования:

- выражение из имени функции неявно **преобразуется в указатель на функцию**

  ```
  int add(int a, int b);

  ...

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

- операция **"&"** для функции **возвращает указатель на функцию**, но это лишняя операция.

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

- операция **"*"** для указателя на функцию **возвращает саму функцию**, которая неявно преобразуется в указатель на функцию

  ```
  int (*p3)(int, int) = *add;

  int (*p4)(int, int) = *****add;
  ```

- указатели на функции **можно сравнивать**

  ```
  if (p1 == add)

       printf("p1 points to add\n");
  ```

- указатель на функцию может быть типом возвращаемого значения функции
- при применении адресной арифметики, указатель может указывать уже не на функцию, а в другое место памяти

qsort:

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

- ***base -** указатель на первый элемент сортируемого массива
- **nmemb -** кол-во элементов в массиве
- **size -** размер одного элемента
- **compare -** функция сравнения

Пусть необходимо упорядочить массив целых чисел:

```
int compare(const void *p, const void *q)
{
      return *(int *)p - *(int *)q;
}
…
int a[10];
…
qsort(a, sizeof(a) / sizeof(a[0]), sizeof(a[0]), compare);
```

qsort сортирует массив, на который указывает параметр base, используя *quicksort – алгоритм сортировки Хоара*

5. Утилита make. Назначение. Простой сценарий сборки

- Для чего предназначена
- Идеи в основе утилиты make: какие исходные данные нужны для того, чтобы make выполнял свои функции.
- Разновидности make
- Подробно про правила (составные части, что для чего используется, виды правил бывают, какие составные части могут отсутствовать)
- Простой сценарий сборки и как работает make на нем

Make – утилита, **предназначенная** для автоматизации преобразования файлов из одной формы в другую.

Правила преобразования задаются **в скрипте** с именем makefile, который должен находиться в корне рабочей директории проекта.

Утилита make использует **информацию из make-файла и время последнего изменения каждого файла** для того, чтобы решить, какие файлы нужно обновить

Принцип работы:

Необходимо создать так называемый **сценарий сборки проекта** (make-файл)**.** Он описывает

- **отношения между файлами**
- содержит **команды для обновления каждого файла**

Сам скрипт состоит из набора правил, которые в свою очередь описываются:

- *целями* (то, что нужно делать)
- *зависимостями* (то, из чего нужно собрать)
- *командами* (то, как нужно собрать)

```
цель: зависимости
      команда 1
      …
      команда n
{исходники} ---> [трансляция] ---> {объектники}
{объектники} ---> [линковка] ---> {исполняемые файлы}
```

Разновидности make:

- **GNU make**
- **BSD make (pmake)** - по функциональности примерно соответствует GNU make
- **nmake (Microsoft make)** - работает под Windows, мало функционален, только базовые принципы make.

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

Про правила (подробно):

Простой make-файл состоит из правил:

```
цель: зависимости
      команда 1
      …
      команда n
```

Обычно **цель (target)** представляет имя файла, который мы собираем

Например. исполняемый или объектный.

Абстрактная цель (phony target) – это цель, которая не является на самом деле именем файла. Это просто название некоторой последовательности команд (пр. all, clean).

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

Зависимости – это файлы, из которых формируется цель. Правило так же может и не иметь зависимости:

Например, clean

```
clean : (нет зависимостей)
    rm *.o *.exe
```

Команда – это действие, выполняемое утилитой. Цель может иметь несколько команд

Правило (rule) описывает, когда и каким образом следует обновлять файлы, указанные в нем в качестве создания или обновления цели.

```
clean :
    rm *.o *.exe
    echo “dir is cleaned!”
```

Правило также может **описывать** каким образом должно выполняться то или иное действие

Простой сценарий и пример работы make на нем:

```
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
```

Первый запуск:

1. **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
```

2. Для выполнения этого правила необходимо сначала **обработать зависимости**
3. **make** ищет правило для создания файла hello.o. Файл **hello.o** отсутствует, файлы **hello.c** и **hello.h** существуют. Следовательно, правило для создания **hello.o** может быть выполнено

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

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

4. Аналогично обрабатываются зависимости **bye.o и main.o.**
5. Все зависимости получены, теперь правило для построения **greeting.exe** может быть выполнено

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

hello.c был изменен

Второй запуск: 1 - 3 по аналогии

4. Файлы **hello.o, hello.c и hello.h** существуют, но время изменения **hello.o** меньше времени изменения **hello.c**. Придется пересоздать файл **hello.o**
5. Аналогично обрабатываются зависимости **bye.o и main.o**, но эти файлы были изменены позже соответствующих си-файлов, т.е. ничего делать не нужно.
6. Все зависимости получены. Время изменения **greeting.exe** меньше времени изменения **hello.o.** Придется пересоздать **greeting.exe**

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

6. Утилита make. Назначение. Переменные. Шаблонные правила

- Для чего предназначена
- Идеи в основе утилиты make: какие исходные данные нужны для того, чтобы make выполнял свои функции.
- Разновидности make
- Кратко про правила
- Описание переменных, для чего используются, примеры
- Неявные переменные и правила, что это, зачем
- Автоматические переменные и шаблонные правила

Make – утилита, **предназначенная** для автоматизации преобразования файлов из одной формы в другую.

Правила преобразования задаются **в скрипте** с именем makefile, который должен находиться в корне рабочей директории проекта.

Утилита make использует **информацию из make-файла и время последнего изменения каждого файла** для того, чтобы решить, какие файлы нужно обновить

Принцип работы:

Необходимо создать так называемый **сценарий сборки проекта** (make-файл)**.** Он описывает

- **отношения между файлами**
- содержит **команды для обновления каждого файла**

Про правила (кратко):

Сам скрипт состоит из набора правил, которые в свою очередь описываются:

- *целями* (то, что нужно делать)
- *зависимостями* (то, из чего нужно собрать)
- *командами* (то, как нужно собрать)

```
цель: зависимости
      команда 1
      …
      команда n
{исходники} ---> [трансляция] ---> {объектники}
{объектники} ---> [линковка] ---> {исполняемые файлы}
```

Разновидности make:

- **GNU make**
- **BSD make (pmake)** - по функциональности примерно соответствует GNU make
- **nmake (Microsoft make)** - работает под Windows, мало функционален, только базовые принципы make.

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

Про переменные:

- Строки, начинающиеся с # - *комментарии*
- *Определение* переменной: **VAR_NAME := value**
- *Получение значения* переменной: **$(VAR_NAME)**

Пример:

```
CC := gcc

CFLAGS := -std=c99 -Wall -Wpedantic -Werror

OBJS := hello.o bye.o

app.exe: $(OBJS) main.o

      $(CC) -o app.exe $(OBJS) main.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 bye.h hello.h

    $(CC) $(CFLAGS) -c main.c


clean:

      rm *.o *.exe
```

Неявные правила и переменные:

Неявные правила (implicit rules) указывают make на некоторые *"стандартные"* приемы обработки файлов, дабы пользователь мог использовать их, не занимаясь каждый раз **детальным описанием способа обработки.**

- **-p** – **показывает** неявные правила и переменные
- **-r** – **запрещает** использовать неявные правила

```
OBJS := hello.o bye.o


app.exe: $(OBJS) main.o
```

```
	$(CC) -o app.exe $(OBJS) main.o


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

Так, например, имеется неявное правило для компиляции исходных файлов на языке Си. Вопрос о запуске тех или иных правил решается исходя из имен обрабатываемых файлов. Как правило, например, при компиляции программ на Си, из "входного" файла с расширением `.c' получается файл с расширением `.o'. Таким образом, при наличии подобной комбинации расширений файлов, make может применить к ним неявное правило для компиляции Си программ.

Автоматические переменные:

- это переменные со специальными именами, которые **автоматически принимают определенные значения** перед выполнением описанных в правиле команд

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

```
CC := gcc
CFLAGS := -std=c99 -Wall -Wpedantic -Werror
OBJS := hello.o bye.o
app.exe: $(OBJS) main.o
	$(CC) -o $@ $^


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


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

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


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

Шаблонные правила:

Шаблонные правила (pattern rules) позволяют указать правило преобразования одних файлов в другие **на основании зависимостей между их именами.**

```
%.расш_файлов_целей: %.расш_файлов_зав
	команда 1
	…
	команда n
```

Пример:

```
CC := gcc
CFLAGS := -std=c99 -Wall -Wpedantic -Werror
OBJS := hello.o bye.o


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


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


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

! % - в точности один, любая непустая строка !

Символ `%' в **зависимости** шаблонного правила означает **ту же основу**, которая соответствует символу `%' в имени **цели**. Для того, чтобы шаблонное правило могло быть применено к рассматриваемому файлу, имя этого файла должно подходить под шаблон цели, а из шаблонов зависимостей должны получиться имена файлов, которые существуют или могут быть получены. Эти файлы станут зависимостями рассматриваемого целевого файла.

7. Утилита make. Назначение. Условные конструкции. Анализ зависимостей

- Для чего предназначена
- Идеи в основе утилиты make: какие исходные данные нужны для того, чтобы make выполнял свои функции.
- Разновидности make
- Кратко про правила
- Условные конструкции
    - непосредственно условные конструкции
    - переменные, зависящие от цели
- Анализ зависимостей
    - Ручной
    - “Все .c файлы зависят от всех .h файлов”
    - Привлечение компилятора

Make – утилита, **предназначенная** для автоматизации преобразования файлов из одной формы в другую.

Правила преобразования задаются **в скрипте** с именем makefile, который должен находиться в корне рабочей директории проекта.

Утилита make использует **информацию из make-файла и время последнего изменения каждого файла** для того, чтобы решить, какие файлы нужно обновить

Принцип работы:

Необходимо создать так называемый **сценарий сборки проекта** (make-файл)**.** Он описывает

- **отношения между файлами**
- содержит **команды для обновления каждого файла**

Про правила (кратко):

Сам скрипт состоит из набора правил, которые в свою очередь описываются:

- *целями* (то, что нужно делать)
- *зависимостями* (то, из чего нужно собрать)

- *командами* (то, как нужно собрать)

```
цель: зависимости
      команда 1
      …
      команда n
{исходники} ---> [трансляция] ---> {объектники}
{объектники} ---> [линковка] ---> {исполняемые файлы}
```

Разновидности make:

- **GNU make**
- **BSD make (pmake)** - по функциональности примерно соответствует GNU make
- **nmake (Microsoft make)** - работает под Windows, мало функционален, только базовые принципы make.

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

Условная конструкция:

Условная конструкция (conditional) заставляет make обрабатывать или игнорировать часть make-файла в зависимости от значения некоторых переменных.

В этой условной конструкции используется три директивы: **ifeq, else** и **endif.**

условная-директива
фрагмент-для-выполненного-условия
ifeq
	условная-директива
	фрагмент-для-выполненного-условия
	else
	фрагмент-для-невыполненного-условия
	endif

ifeq ($(mode), release) CFLAGS += -g0	

```
endif


ifeq ($(mode), debug):
      CFLAGS += -DNDEBUG -g3
endif
```

Переменные, зависящие от цели:

```
debug : CFLAGS += -g3
release : CFLAGS += -DNDEBUG –g0
```

Анализ зависимостей:

1. Ручное перечисление *.h файлов

   Плохой подход, так как невозможно проследить всю цепочку зависимостей заголовочных файлов. К тому же, можно забыть указать необходимые файлы.

```
CC := gcc
CFLAGS := -std=c99 -Wall -Wpedantic -Werror
OBJS := hello.o bye.o
app.exe: $(OBJS) main.o
      $(CC) -o $@ $^

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

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

main.o : main.c bye.h hello.h
```

```
	$(CC) $(CFLAGS) -c $<


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

2. Подключение всех *.h файлов

```
CC := gcc
CFLAGS := -std=c99 -Wall -Wpedantic -Werror
OBJS := hello.o bye.o


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


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


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

3. Автоматическая генерация зависимостей

Для каждого исходного файла **`имя.c'** имеется make-файл **`имя.d'**, в котором перечислен список файлов, от которых зависит объектный файл **`имя.o'**. При таком подходе, новые списки зависимостей могут строиться только для тех исходных файлов, которые действительно были модифицированы.

```
CC := gcc
CFLAGS := -std=c99 -Wall -Wpedantic -Werror
OBJS := hello.o bye.o
SRCS := $(wildcard *.c) # all *.c files
```

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

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

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

include $(SRCS: .c = .d)

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

Суть автоматической генерации зависимостей – в получении универсального make-файла, который можно будет использовать в разных проектах

-M: Для каждого файла с исходным текстом препроцессор будет выдавать на стандартный вывод список зависимостей в виде правила для программы make. В список зависимостей попадает сам исходный файл, а также все файлы, включаемые с помощью директив ***#include <имя_файла>*** **и** ***#include "имя_файла"***. После запуска препроцессора компилятор останавливает работу, и генерации объектных файлов не происходит.

После того как файлы зависимостей сформированы, нужно сделать их доступными утилите **make**. Этого можно добиться с помощью директивы **include**.

8. Динамические матрицы. Представление в виде одномерного массива и в виде массива указателей на строки. Анализ преимуществ и недостатков.

- Сравнить представления друг с другом; в идеале написать табличку с критериями сравнения.

В качестве критериев можно взять(имхо идк):

- относительная сложность начальной инициализации
- относительная сложность выделения, освобождения памяти
- использование как одномерного массива
- отслеживание выхода за границы массива
- обмен строк между собой через указатель (хз натянуто, но пусть будет)

Одномерный массив

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

int main()
{
    size_t n = 10, m = 10;
    double *mat = malloc(n * m * sizeof(double));
    if (!mat)
        return NULL;

    for (size_t i = 0; i < n; ++i)
    {
        for (size_t j = 0; j < m; ++j)
        {
            mat[i * m + j] = 0;
        }
    }

    free(mat);

    return 0;
}
```

Преимущества	Недостатки
• Простота выделения и освобождения памяти • Возможность использовать как одномерный массив	• Отладчик может не отследить выход за пределы массива • Сложная индексация

Массив указателей на строки (столбцы)

```
double **allocate_matrix(size_t n, size_t m)
{
    double **data = calloc(n, sizeof(double *));
    if (!data)
        return NULL;

    for (size_t i = 0; i < n; ++i)
```

```
    {
        data[i] = malloc(m * sizeof(double));
        if (!data[i])
        {
            free_matrix(data, n); // можно обойтись без calloc и вместо
n передать i
            return NULL;
        }
    }
}

void free_matrix(double **data, size_t n)
{
    for (size_t i = 0; i < n; ++i)
        free(data[i]);
    free(data);
}
```

Преимущества	Недостатки
• Отладчик может отследить выход за пределы массива • Возможность обмена строк через указатели	• Сложность выделения и освобождения памяти • Память под матрицу не лежит одной областью

Отдельное выделение под массив указателей

```
double **allocate_matrix(size_t n, size_t 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 (size_t i = 0; i < n; ++i)
        ptrs[i] = data + i * m;

    return ptrs;
}

void free_matrix(double **ptrs)
{
```

```
    free(ptrs[0]);
    free(ptrs)
}
```

Преимущества	Недостатки
• Простота выделения и освобождения памяти • Возможность обмена строк через указатели • Возможность использовать как одномерный массив	• Сложность начальной инициализации • Отладчик не может отследить выход за пределы массива

Массив указателей и массив данных в одной области

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

    for (size_t i = 0; i < n; ++i)
    {
        data[i] = (double *)((char *)data + n * sizeof(double *) + \
                                                i * m * sizeof(double));
    }
    return data;
}

free(mat);
```

Преимущества	Недостатки
• Простота выделения и освобождения памяти • Возможность использовать как одномерный массив • Перестановка строк через обмен указателей	• Сложность начальной инициализации • Отладчик не может отследить выход за пределы массива

9. -//-
10. -//-
11. -//-
12. -//-
13. -//-

14. Чтение сложных объявлений

[]	Массив типа
[N]	Массив из N элементов типа
(type)	Функция, принимающая аргумент типа type и возвращающая
*****	Указатель на

Декодирование изнутри. Начало от идентификатора
Предпочтение [], (), а не *
`*name[] – массив типа …`
`*name() – функция, принимающая …`
Невозможно создать массив функций
`int a[10](int)`
Функция не может возвращать функцию
`int g(int)(int);`
Функция не может вернуть массив
`int f(int)[]`

15. Строки в динамической памяти. Функции POSIX и расширения GNU, возвращающие такие строки

- Строки в динамической памяти ничем не отличаются от строк в статической памяти
- Обратить внимание на особенности выделения строки: `(size + 1) * sizeof(char),` так как размер символа может разниться из-за широких символов в разных ОС.
- Feature Test Macro

При компиляции следует указывать стандарт gnu99, используя линукс.
char *strdup(const char *) - возвращает копию указанной строки, стандарт POSIX
char *strndup(const char *, size_t n) - аналог strdup, но копирует не более n элементов
Для getline следует использовать Feature Test Macro - позволяет библиотеке соответствовать нескольким стандартам. _GNU_SOURCE/_POSIX_C_SOURCE/в stdio.h
ssize_t getline(char **lineptr, size_t *n, FILE *stream) - чтение строки. Возвращает количество считанных символов (-1 в случае ошибки). *lineptr - либо NULL, либо дин. строка. Стандарт POSIX.
Функция sprintf не является ни частью стандартной библиотеки, ни частью POSIX, является расширением библиотеки GNU.
int snprintf(char *restrict buf, size_t num, const char *restrict format, ...); - вывод строки формата в динамическую строку. Если передать NULL и 0, то функция вернет предположительную длину строки. Сохраняется максимум num - 1 символов.
int asprintf(char **strp, const char *fmt, ...) - самостоятельно выделяет память под строку в strp. Возвращает количество записанных символов.

Макросы тестирования свойств позволяют программисту контролировать какие определения будут доступны из системных заголовочных файлов при компиляции программы.

16. Особенности использования структур с полями-указателями

- Операция = для структур одного типа – по сути побитовое копирования области памяти одной переменной в область памяти другой. К чему плохому или хорошему может привести наличие поля-указателя в структуре
- Поверхностное и глубокое копирование
- Рекурсивное освобождение памяти для структуры с динамическими полями

В си определена операция присваивания для структур одного типа. Она фактически эквивалентна копированию одной области памяти переменной в другую.
Однако в случае, если структуры содержат указатели, следует быть внимательным - так как копируется только сам указатель, а не его содержимое. Это явление называется ***поверхностное копирование.***
Это явление может привести к ошибкам с двойным освобождением и потерей области памяти одной структуры в случае присваивания.
В случае ***глубокого копирования*** происходит копия не только полей структуры, но и их содержимых. Для этой реализации требуются отдельные функции для выделения и освобождения памяти.
Для освобождения памяти из структуры, содержащей указатели на динамически выделенную память, следует освободить сначала адреса по указателям, а затем саму структуру.

17. Структуры переменного размера

- Что это
- Примеры, где с такими структурами можно столкнуться на практике
- Реализация в рамках Си
- Flexible Array Member
- FAM до c99
- Желательно: выделение памяти под такую структуру, сохранение в файл, чтение из файла.

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

Структуры переменного размера - это структуры, у которых может меняться размер занимаемой области памяти.

Пример : TLV кодирование - Type Len Value. Используется в семействе протоколов TCP/IP, спецификации PC/SC (смарт карты, usb ключи), ASN.1 нотация в кодировании (сертификаты, подписанное сообщение, защищенное сообщение).

Реализация Flexible array member (поле гибкий массив):

1. Подобное поле должно быть последним в структуре;
2. Нельзя создать массив таких структур;
3. Такая структура не может быть НЕ последним полем в другой структуре;
4. Операция sizeof не учитывает размер поля;
5. В случае обращения к массиву, в котором нет элементов - неопределенное поведение.

Для создания такой структуры нужно выделить память под структуру и массив из n элементов.

ДО С99:

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

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;
}
```

"unwarranted chumminess with the C impleme
(c) Dennis Ritchie

ПОСЛЕ С99:

```
struct s *arr_create(int n, const double *d)
{
   struct s *answer = NULL;
    if (n > 1)
        answer = calloc(sizeof(struct s) + n * sizeof(double));
    if (answer)
    {
        answer->n = n;
        memmove(answer->d, d, n * sizeof(double));
    }

    return answer;
}
```

До С99 в поле гибкий массив указывали размер массива ‘1’. При создании проверялась требуемая длина массива - если больше 1, то выделяли память с помощью calloc.
https://wiki.c2.com/?StructHack

Преимуществом является атомарность данных, экономия данных, разовое выделение/освобождение памяти и легкое копирование структур.
Для работы с файлом потребуется fwrite fread. Для чтения - сначала прочитать постоянную часть (станет известна длина) - прочитать оставшиеся данные/создать новую структуру и прочитать полностью.

До си99

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

void print_s(const struct s *elem)
{
    printf("n = %d\n", elem->n);

    for (int i = 0; i < elem->n; i++)
        printf("%4.2f ", elem->d[i]);

    printf("\n\n");
}
```

```
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;
}
```

```
int save_s(FILE *f, const struct s *elem)
{
    int size = sizeof(*elem) + (elem->n > 1 ?
(elem->n - 1) * sizeof(double) : 0);
    int rc = fwrite(elem, 1, size, f);

    if (size != rc)
```

После си99

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

void print_s(const struct s *elem)
{
    printf("n = %d\n", elem->n);

    for (int i = 0; i < elem->n; i++)
        printf("%4.2f ", elem->d[i]);

    printf("\n\n");
}
```

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

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

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

    return elem;
}
```

```
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;
}
```

```
        return 1;

    return 0;
}
```

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

    size = fread(&part_s, 1, sizeof(part_s),
f);
    if (size != sizeof(part_s))
        return NULL;

    elem = calloc(sizeof(part_s) + (part_s.n >
1 ? (part_s.n - 1) * sizeof(double) : 0), 1);
    if (!elem)
        return NULL;

    memmove(elem, &part_s, sizeof(part_s));

    if (elem->n > 1)
    {
        size = fread(elem->d + 1,
sizeof(double), elem->n - 1, f);
        if (size != elem->n - 1)
        {
            free(elem);

            return NULL;
        }
    }

    return elem;
}
```

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

    size = fread(&part_s, 1, sizeof(part_s),
f);
    if (size != sizeof(part_s))
        return NULL;

    elem = malloc(sizeof(part_s) + part_s.n *
sizeof(double));
    /*
    // Чтобы успокоить valgrind
    elem = calloc(sizeof(part_s) + part_s.n *
sizeof(double), 1);
    */
    if (!elem)
        return NULL;

    memmove(elem, &part_s, sizeof(part_s));

    size = fread(elem->d, sizeof(double),
elem->n, f);
    if (size != elem->n)
    {
        free(elem);

        return NULL;
    }

    return elem;
}
```

18. Динамически расширяемый массив

- Что такое массив
- +/- этой СД
- Особенности:
    - Память выделяется относительно крупными блоками
    - Обычно такой массив представляется структурой с несколько большим набором данных
- Добавление, удаление элемента
- Особенности использования
    - При работе с таким массивом используются индексы, а не указатели на элементы. Почему так?

Массив - это последовательность элементов одного и того же типа и размера, расположенных друг за другом. Линейность, Однородность, Произвольный доступ.

Преимущества и недостатки массива объясняются стратегией выделения памяти: память под все элементы выделяется в одном блоке.
“+” Минимальные накладные расходы.
“+” Константное время доступа к элементу.
“–” Хранение меняющегося набора значений.

```
struct arr_t
{
    int len;                // Количество элементов в массиве
    int allocated;     // Количество выделенных элементов массива
    int *data;           // Указатель на массив
}
```

Добавление элемента (можно сразу realloc):

```
int arr_add(arr_t *array, int n)
{
    if (!array->data)
    {
        array->data = malloc(INIT_SIZE * sizeof(int));
        if (!array->data)
            return ERR:
        array->allocate = INIT_SIZE;
    }
    else
    {
        if (array->len >= array->allocate)
        {
            int *p = realloc(array->data, STEP * array->allocate * sizeof(int));
            if (!p)
                return ERR;
            array->data = p;
            array->allocate *= STEP;
        }
    }
    array->data[array->len++] = n;

    return OK;
}
```

Удаление элемента (можно циклом):

```
int arr_remove(arr_t *array, int ind)
{
    if (!array->data && array->len >= ind)
        return ERR;

    memove(array->data[ind], array->data[ind + 1], (array->len - ind - 1) * sizeof(int));
    array->len--;
    return OK;
}
```

Память под динамический массив выделяется блоками (с каким-то шагом) для оптимизации выделения памяти.
Для динамически расширяемого массива следует использовать индексы, а не указатели, так как массив при добавлении может поменять адрес.

В случае динамически расширяемого массива есть несколько минусов - ручное освобождение памяти, хранение значений количества элементов массива и размера области под массив.

19. Линейный односвязный список. Добавление элемента, удаление элемента.

- Определение списка
- Список vs массив
- Описание узла
- Удаление памяти из-под всего списка
- Если будет время: порассуждать о возможных улучшениях:
  - Универсализация (void *)
  - …
- Рассказ по теме

Линейный односвязный список - это структура данных, которая хранит данные и ссылку на следующий узел. Единицу структуры называют узлом.
Основные операции: добавление в начало/конец, поиск, вставка перед/после, удаление.

	Массив	Список
Доступ/Поиск	По индексу/бинарный	Последовательный
Добавление/Удаление	Медленное из-за смещения	Легко и быстро
Хранение в памяти	В одной области	В разных областях
Размер	Указан во время объявления	Ограничен памятью

Описание структуры:

```
struct node_t
{
    int data;
    struct node_t *next;
}
```

Удаление памяти из под всего списка происходит с помощью дополнительного указателя на узел, предшествующий данному.
Добавление в начало

```
struct person_t* list_add_front(struct person_t *head, struct person_t
*pers)
{
    pers->next = head;
    return pers;
}
```

Удаление - поиск требуемого узла и хранение его родителя. Когда найден - замена указателей и освобождение памяти.

20. Линейный односвязный список. Вставка элемента, удаление элемента.

- Определение списка
- Список vs массив
- Описание узла
- Удаление памяти из-под всего списка
- Если будет время: порассуждать о возможных улучшениях:
  - Универсализация (void *)
  - …
- Рассказ по теме

См. 19

Вставка до и после элемента (по значению)

Вставка **до** значения:

```
struct node_t* list_insert_before(struct node_t *head, struct node_t *tmp,
int data)
{
    struct node_t *prev = NULL, *cur = head;

    while (cur && (cur->data != data))
    {
        prev = cur;
        cur = cur->next;
    }

    if (prev && cur)
    {
        prev->next = tmp;
        tmp->next = cur;
    }
    else if (cur == head && head)
    {
       tmp->next = head;
       head = tmp;
    }

    return head;
}
```

Вставка **после** значения:

```
struct node_t* list_insert_after(struct node_t *head, struct node_t *tmp,
int data)
{
    struct node_t *cur = head;

    while (cur && (cur->data != data))
        cur = cur->next;

    if (cur)
    {
        tmp->next = cur->next;
        cur->next = tmp;
    }

    return head;
}
```

Удаление из-под **всего списка:**

```
void list_clear(struct node_t **head)
{
    struct node_t *tmp = *head;

    while (*head)
    {
        (*head) = (*head)->next;

        free(tmp);
        tmp = NULL;
        tmp = (*head);
    }
}
```

Универсализация:

- **Добавление указателя на конец списка**
- **Представление элемента списка**
  - Универсальный элемент *(void*)*
- **Двусвязные списки**
  - Требует больше ресурсов.
  - Поиск последнего и удаление текущего – операции порядка O(1)

21. Линейный односвязный список. Обход.

- Определение списка
- Список vs массив
- Описание узла
- Удаление памяти из-под всего списка
- Если будет время: порассуждать о возможных улучшениях:
  - Универсализация (void *)
  - …
- Рассказ по теме

См. 19

Обход с помощью функции:

void list_apply(struct person_t *head, void (*f)(struct person*, void*), void *arg)

```
void list_apply(struct person_t *head, void(*f)(struct person *, void *), void *arg)
{
    for(;head;head=head->next)
    {
        f(head, arg);
    }
}
```

22. Двоичное дерево поиска. Добавление элемента

- Определение, отличие от обычного дерева, отношение порядка
- Описание типа
- Освобождение памяти из-под всего дерева
- Рассказ по теме

Дерево - это связный ациклический граф.

ДДП - 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, поиск, обход, удаление.

```
struct node_t
{
    int data;
    struct node_t *left, *right;
}
```

Освобождение памяти из-под всего дерева осуществляется при помощи постфиксного обхода.

Добавление элемента (если элемент уже есть, то сами эту ситуацию обрабатывайте как хотите):

struct node_t *insert(struct node_t *head, struct node_t *ins);

```
struct node_t *insert(struct node_t *head, struct node_t *ins)
{
    if (!head)
        return ins;

    int cmp = head->data - ins->data;
    if (cmp == 0)
        assert(0);
    else if (cmp < 0)
        head->left = insert(head->left, ins);
    else
        head->right = insert(head->right, ins);

    return head;
}
```

23. Двоичное дерево поиска. Поиск элемента

- Определение, отличие от обычного дерева, отношение порядка
- Описание типа
- Освобождение памяти из-под всего дерева
- Рассказ по теме

См. 22

Поиск с помощью функции (либо рекурсивно, либо циклом):
struct node_t* find(struct node_t *head, int val);

```
struct node_t *find(struct node_t *head, int val)
{
    int cmp;
    while (head)
    {
        cmp = head->data - val;
        if (cmp == 0)
            return head;
        else if (cmp < 0)
            head = head->left;
        else
            head = head->right;
    }
    return NULL;
}
```

```
struct node_t *find(struct node_t *head, int val)
{
    if (!head)
        return NULL;

    int cmp = head->data - val;
    if (cmp == 0)
        return head;
```

```
    else if (cmp < 0)
        return find(head->left, val);
    else
        return find(head->right, val);
}
```

24. Двоичное дерево поиска. Обход

- Определение, отличие от обычного дерева, отношение порядка
- Описание типа
- Освобождение памяти из-под всего дерева
- Рассказ по теме
- Можно рассказать про язык DOT

См. 22

Обход с помощью функции:
void node_view(struct node_t *head, void (*f)(struct node_t *, void*), void*);
Префиксный - f view view
Инфиксный - view f view
Постфиксный - view view f

```
void node_view(struct node_t *head, void (*f)(struct node_t *, void *), void *param)
{
    if (!head)
        return;

    f(head, param);
    node_view(head->left, f, param);
    node_view(head->right, f, param);
}
```

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**.

25. Двоичное дерево поиска. Удаление элемента

- Определение, отличие от обычного дерева, отношение порядка
- Описание типа
- Освобождение памяти из-под всего дерева
- Рассказ по теме

См. 22

Удаление элемента (нужно будет запоминать предка вершины):
node_t *node_del(node_t *head, int del);

Рассматривается 3 случая:

1. Если у узла нет потомков, то удаляется и меняется указатель предка;
2. Если у узла 1 потомок, то удаляется и меняется указатель предка на потомка;
3. Если у узла 2 потомка, то идет поиск минимального в правом поддереве, меняются значения узлов для удаления и найденного, после чего найденный удаляется.

Реализация удаления ПО ИНДЕКСУ:

```
tree_node_t *find_right_min_tree(tree_node_t *tree)
{
    if (!(tree->left->left))
    {
        tree_node_t *tmp_tree = tree->left;


        tree->left = NULL;

        return tmp_tree;
    }

    tree_node_t *new_tree = find_right_min_tree(tree->left);


    return new_tree;
}


void destroy_node(void *node, void *trash)
{
    if (!trash)
        free(node);
}

tree_node_t *del_tree_node(tree_node_t *tree, int *ind)
{
    if (!tree)
        return NULL;

    if (tree->index == *ind)
    {
```

```
        *ind = -1;

        if (!(tree->left) && !(tree->right))
        {
            destroy_node((void *) tree, NULL);
            return NULL;
        }

        if (tree->left && !(tree->right))
        {
            tree_node_t *tmp_node = tree->left;


            destroy_node((void *) tree, NULL);
            return tmp_node;
        }

        if (tree->right && !(tree->left))
        {
            tree_node_t *tmp_node = tree->right;


            destroy_node((void *) tree, NULL);
            return tmp_node;
        }

        if (!(tree->right->left))
        {
            tree_node_t *tmp_node = tree->right;


            tree->right->left = tree->left;
            destroy_node((void *) tree, NULL);
            return tmp_node;
        }

        tree_node_t *new_tree = find_right_min_tree(tree->right);


        new_tree->left = tree->left;
        new_tree->right = tree->right;
```

```
        destroy_node((void *) tree, NULL);
        return new_tree;
    }

    if (tree->left && *ind > 0)
        tree->left = del_tree_node(tree->left, ind);
    if (tree->right && *ind > 0)
        tree->right = del_tree_node(tree->right, ind);

    return tree;
}
```

26. Понятие связывания

- Что такое связывание
- Какие есть виды связывания
- Как влияет на свойства объектного и исполняемого файла
- Упомянуть static
- Как работает компоновщик
- Пример, таблица имен, какие переменные куда попадают

Связывание - определяет область программы, в рамках которой будет доступен “программный объект” другим функциям.
Виды связывания - внешнее, внутреннее и никакое.
Объектный файл содержит машинный код и информацию о переменных и функциях, которые определены или понадобятся для работы.
Имена со внешним связыванием попадают в таблицу символов .symtabl

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, «заглавные» - внешние (глобальные).

```
#include <stdio.h>

int a = 10;           // D
static int b = 5;     // d
int c;                // B
static int d;         // b
extern int e;         // U
const char f = 'e'; // R
void foo_boo(void); // U
void foo(void)        // T
{
    foo_boo();
}
static void boo(void)     // t
{
    e = 5;
}
```

27. Классы памяти. Общие понятия. Классы памяти auto и static для переменных

- Свойства (область видимости, время жизни, связывание) переменных по умолчанию.
- До какой-то степени этими свойствами можно управлять с помощью классов памяти
- Перечисление классов памяти, особенности их использования

Управлять временем жизни, областью видимости и связыванием переменной (до определенной степени) можно с помощью так называемых классов памяти.
Существует 4 класса памяти: auto static extern register
По умолчанию:

```
int a; // Файловая ОВ, глобальное время жизни, внешнее связывание

{
   int b;  // Блочная ОВ, локальное ВЖ, отсутствие связывания
}
```

auto - применим только к блочным переменным. Имеет блочную ОВ, локальное ВЖ, отсутствие связывания. По умолчанию все переменные в блоке или заголовке функции имеют это связывание.

static - применим к любым переменным. Меняет связывание на внутреннее, время жизни на глобальное.
Глобальная переменная с классом памяти static имеет внутреннее связывание, файловую область видимости и глобальное время жизни. Это скрывает переменную в файле.

28. Классы памяти. Общие понятия. Классы памяти extern и register для переменных

- Свойства (область видимости, время жизни, связывание) переменных по умолчанию.
- До какой-то степени этими свойствами можно управлять с помощью классов памяти
- Перечисление классов памяти, особенности их использования

См. 27
extern - применим к любым переменным. Меняет время жизни на глобальное, связывание на связывание родителя. Помогает “экспортировать” переменные из других файлов. В случае, когда в файле объявлена переменная сначала extern, потом она же static, будет ошибка, так как в таком случается будет меняться связывание, что противоречит стандарту.
register - применим к локальным переменным. Это просьба компилятора поместить эту переменную в регистр процессора для быстрой работы с ней. Для нее нельзя применить операцию адресации.

29. Классы памяти. Общие понятия. Классы памяти для функций

- Свойства (область видимости, время жизни, связывание) функций по умолчанию.
- До какой-то степени этими свойствами можно управлять с помощью классов памяти
- Перечисление классов памяти, особенности их использования

См. 27
У функции по умолчанию (аккуратнее с ОВ и ВЖ, я не уверен в корректности таких выражений к функции))) :)

```
void foo(void);  // Файловая ОВ, Глобальное ВЖ, Внешнее связывание
extern void boo(void); // Файловая ОВ, Глобальное ВЖ, Внешнее связывание
static void too(void); // Файловая ОВ, Глобальное ВЖ, Внутреннее связывание
```

По умолчанию у функции класс памяти **extern**.
Static помогает скрыть функцию и не дать возможность ее использования из других файлов. (инкапсуляция, можно повторно использовать имя)

30. Глобальные переменные. Журналирование.

- Особенности использование глобальных переменных. +/-. Когда можно использовать глобальные переменные
- 3 подхода к журналированию

+:

1. Доступны из любой части программы
2. Глобальное время жизни

-:

1. При изменении переменной сложно отследить ошибку
2. Требует проверки правильности во всех функциях, где используется
3. Функции нельзя использовать в других программах

Журналирование – это процесс записи информации о происходящих с каким-то объектом (или в рамках какого-то процесса) событиях в **«журнал»** (например, в файл).

Файловую переменную для журнала определяют **глобальной** и объявляют во всех файлах реализации проекта. Это позволяет вызывать функции записи в файл для журналирования отовсюду.

Для записи в журнал можно написать собственную функцию, где переменная-указатель на файл с журналом мб как глобальной, так и статической переменной.

1. **<u>Использование глобальной переменной</u>**

Заводим **глобальную переменную** и объявляем ее в **заголовочном файле**. Дальше 2 функции – **создание файла и его закрытие**.

Из плюсов: журналирование из любого места программы.

Минусы: если мы испортим глобальную переменную – сломаем журналирование

log.h
```c
#ifndef __LOG_H__
#define __LOG_H__

#include <stdio.h>

extern FILE *flog;

int log_init(const char *name);

void log_close(void);

#endif // __LOG_H__
```

log.c
```c
#include "log.h"

FILE *flog;

int log_init(const char *name)
{
    flog = fopen(name, "w");

    return (!flog) ? 1 : 0;
}

void log_close(void)
{
    fclose(flog);
}
```

main.c
```c
#include "log.h"

void func_1(void)
{
    fprintf(flog, "func_1\n");
}

void func_2(void)
{
    fprintf(flog, "func_2\n");
}

int main(void)
{
    if (log_init("test.log"))
    {
        printf("kek\n");
        return -1;
    }

    func_1();
    func_2();

    log_close();

    return 0;
}
```

2. **<u>Сокрытие</u>**

Переменная по-прежнему **глобальная. log_get** создает копию, но еще не защищает – можно вызывать и закрывать файл

log.h
```c
#ifndef __LOG_H__
#define __LOG_H__

#include <stdio.h>

int log_init(const char *name);

FILE *log_get(void);

void log_close(void);

#endif // __LOG_H__
```

log.c
```c
#include "log.h"

static FILE *flog;

int log_init(const char *name)
{
    flog = fopen(name, "w");

    return (!flog) ? 1 : 0;
}

FILE *log_get(void)
{
    return flog;
}

void log_close(void)
{
    fclose(flog);
}
```

main.c
```c
#include "log.h"

void func_1(void)
{
    fprintf(log_get(), "func_1\n");
}

void func_2(void)
{
    fprintf(log_get(), "func_2\n");
}

int main(void)
{
    if (log_init("test.log"))
    {
        printf("kek\n");
        return -1;
    }

    func_1();
    func_2();

    log_close();

    return 0;
}
```

3. **<u>Идеальный вариант</u>** - **не log_get**, а функция записи в журнал log_message, которая выводит только строку.
   Минус: перегружает, так как нужно сначала сформировать строку. Но можно реализовать с **переменным числом параметров**

log.h
```c
#ifndef __LOG_H__
#define __LOG_H__

#include <stdio.h>
#include <stdarg.h>

int log_init(const char *name);

FILE *log_message(const char *format, ...);

void log_close(void);

#endif // __LOG_H__
```

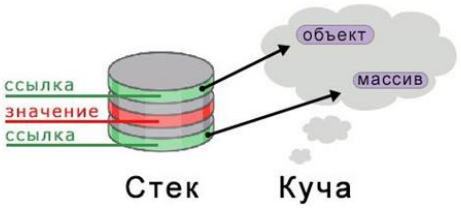
log.c
```c
#include "log.h"

static FILE *flog;

int log_init(const char *name)
{
    flog = fopen(name, "w");

    return (!flog) ? 1 : 0;
}

FILE *log_message(const char *format, ...)
{
    va_list args;
    va_start(args, format);
    vfprintf(flog, format, args);
    va_end(args);
}

void log_close(void)
{
    fclose(flog);
}
```

main.c
```c
#include "log.h"

void func_1(void)
{
    log_message("%s", "func_1\n");
}

void func_2(void)
{
    log_message("%s", "func_2\n");
}

int main(void)
{
    if (log_init("test.log"))
    {
        printf("kek\n");
        return -1;
    }

    func_1();
    func_2();

    log_close();

    return 0;
}
```

31. Куча в программе на Си. Алгоритмы работы функций malloc и free.

- Что такое куча, откуда взялся этот термин
- Свойства памяти, которая выделяется в куче
- В идеале написать код с комментариями, но можно подробно описать алгоритм
- Особенности реализации
    - выравнивание
    - фрагментация

Куча – хранилище памяти, расположенное в ОЗУ. Допускает динамическое выделение памяти и является, по сути простым складом для переменных.



Первый раз термин Куча употребил Дональд Кнут, сказав, что несколько авторов в начале 1975 года ссылались на **Пул свободной памяти** как на **Кучу.** Название работы и имена авторов остались никому неизвестны, поэтому пришли к следующему выводу:

“Термин Куча возник как противопоставление термину Стек”

Свойства памяти:

- Выделяется по крайней мере указанное количество байт (меньше нельзя, больше можно)
- Указатель возвращенный malloc указывает на выделенную область
- Ни один другой вызов malloc не может выделить эту область или ее часть, если она не была освобождена с помощью free.

код TODO, а может и в ass его

Структура блока памяти

```
struct block_t
{
    size_t *size;
    int free;
    struct block_t *next
};
```

Алгоритм malloc:

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

Алгоритм free:

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

`merge_blocks` ничего не принимает в качестве параметров.

Особенности реализации:

1. Фрагментация – состояние, в котором доступная память разбивается на небольшие несвязанные блоки. При этом, даже если суммарный объем памяти достаточен для выполнения запроса, выделение памяти может завершиться сбоем
2. Выравнивание

```
union block_t
{
    struct
    {
        size_t size;
        int free;
        union block_t *next;
    } block;
    size_t align_x;
}
```

Для хранения произвольных объектов блок должен быть правильно выровнен.

Если элемент самого требовательного типа можно поместить по некоторому адресу, то любые другие элементы тоже можно вместить туда.

32. Массивы переменного размера. alloca

c99: массивы переменной длины – VLA.

```
int a[n] // n - переменная
```

- Длина такого массива вычисляется во время выполнения программы, а не во время компиляции
  (по стандарту требует производить инициализацию во время

компиляции, а в VLA она происходит во время работы, поэтому инициализировать VLA по стандарту нельзя (-pedantic забанит)).

- Память под элементы массива выделяется на стеке
- VLA нельзя инициализировать при определении
- VLA мб многомерными
- Для VLA справедлива адресная арифметика
- VLA облегчают описание заголовков функций, которые обрабатывают массивы (проверка правильности аргументов не производится)

```
void foo(int n, int m, int a[n][m]) // a[n][m] не мб в начале
```

```
void *alloca(size_t size);
```

Выделяет область памяти size на стеке. Область памяти освобождается автоматически, когда вызвавшая alloca функция возвращает управление вызывающей стороне. При переполнении стека поведение программы не определено.

```
void foo(int size) {
    ...
    while(b){
        char tmp[size];
        ...
    }
```

```
void foo(int size) {
    ...
    while(b){
        char* tmp = alloca(size);
        ...
    }
```

При использовании VLA в теле цикла, массив будет каждую итерацию разрушаться и заново создаваться. При использовании alloca() в теле цикла, память под массив после итерации **не будет освобождаться**! Это произойдет только после выхода из функции foo().

33. Функции с переменным числом параметров

- идея реализации таких функций
- так делать нельзя!!!
- каким образом с помощью стандартной библиотеки реализуются такие функции

```
double avg(int n, ...)
{
    int *p_i = &n;
    double *p_d = (double *)(p_i + 1);
    double sum = 0.;

    if (!n)
        return 0;
    for (int i = 0; i < n; ++i, p_d++)
        sum += *p_d;
    return sum/n;
}
```

```
int main(void)
{
    double res = avg(4, 1.0, 2.0, 3.0, 4.0);
    ...
}
```

Такой реализацией пользоваться нежелательно, т.к. соглашение о вызове хоть и регламентирует некоторые моменты, например порядок передачи аргументов (справа налево) или “расширение” некоторых маленьких типов, но все же множество других оставляет на разработчиков компиляторов.

Для реализации функций с переменным числом параметров используется библиотека `stdarg.h`

```
#include <stdarg.h>

double avg(int n, ...)
{
    va_list ap;
    double sum = 0, num;
    if (!n)
        return 0.;

    va_start(ap, n);

    for (int i = 0; i < n; ++i)
    {
        num = va_arg(ap, double);
        sum += num;
    }
    va_end(ap);
    return sum/n;
}
...
```

34. Препроцессор. Общие понятия. Директива #include. Простые макросы. Предопределенные макросы.

- Что такое препроцессор
- Какие основные действия он выполняет с программой
- На какие группы можно разделить директивы препроцессора
- Какие правила справедливы для всех директив

Препроцессор – первая утилита, с которой сталкивается программа на пути получения исполняемого файла.

Действия:

1. Удаление комментариев
2. Включение файлов `#include`
3. Обработка директив условной компиляции и макроопределений (т.е. текстовая замена)

Директивы препроцессора:

1. Макроопределения `#define, #undef`
2. Включения файлов `#include`
3. Условной компиляции `#if, #ifdef, #else, #endif` и др.
4. Остальные #pragma, `#error, #line` и др.

Общие правила:

1. Директивы начинаются с # и заканчиваются `\n`.
2. `\` используется для многострочных директив
3. Лексемы могут разделяться сколь угодно большим количеством пробелов.
4. Могут быть определены в любом месте программы

Простые макросы

Макрос формата `#define идентификатор список-замены`

```
#define PI 3.14
#define ABO “BA”
#define end }
```

В случае обнаружения макроса препроцессор производит список-замены.

Используются:

1. В качестве имен для числовых, символьных и строковых констант;
2. Незначительного изменения синтаксиса языка (лучше не использовать);
3. Переименования типов (лучше не использовать);
4. Управления условной компиляции.

Предопределенные макросы:

Полезны для журналирования, генерации сообщений об ошибках с указанием имени файла, строки при отладке кода

- **`__LINE__`** - номер текущей строки (десятичная константа)
- **`__FILE__`** - имя компилируемого файла

Мб использованы для предоставления информации о времени компиляции

- **`__DATE__`** - дата компиляции
- **`__TIME__`** - время компиляции

Эти идентификаторы нельзя переопределять или отменять директивой undef.

- **__func__** - имя функции как строка (GCC only, C99 и не макрос)

35. Препроцессор. Макросы с параметрами

- Что такое препроцессор
- Какие основные действия он выполняет с программой
- На какие группы можно разделить директивы препроцессора
- Какие правила справедливы для всех директив
- Рассказ по теме
- Макросы с параметрами vs функции
- Макросы с переменным числом параметров
- Написание длинных макросов
- Примеры

п. 34

Макросы vs функции

Преимущества	**Недостатки**
программа может работать немного быстрее	Скомпилированный код становится больше
макросы “универсальны” `#define MAX(x,y) (x)>(y)?(x):(y)`	Типы аргументов не проверяются
	Нельзя объявить указатель на макрос
	Макрос может вычислять аргументы несколько раз `n = MAX(i++,j);`

Макросы с переменным числом параметров

```
#define DBGPRINT(fmt, ...) printf(fmt, __VA_ARGS__)
```

Написание длинных макросов

Способ	Реализация	+/-

1	`#define ECHO(s) {gets(s);puts(s);}`
	
	`if (echo_flag)`
	`    ECHO(s)`
	`else`
	`    gets(s);`
		Привычность синтаксиса для описания макроса

		При просмотре самой программы будет казаться, что правила языка не соблюдены
2	`#define ECHO(s) (gets(s),puts(s))`	К постановке ; в программе не возникает вопросов

	`ECHO(s);`	Невозможность использования операторов (if-else, for, while, …)

3	`#define ECHO(s) \`
	`do \`
	`{ \`
	`  gets(s); \`
	`  puts(s); \`
	`} while(0) \`
		К постановке ; в программе не возникает вопросов

		Оператор do-while используется лишь для группировки выражений

36. Препроцессор. Общие понятия. Директивы условной компиляции. Директивы #error и #pragma

- Что такое препроцессор
- Какие основные действия он выполняет с программой
- На какие группы можно разделить директивы препроцессора
- Какие правила справедливы для всех директив
- #ifdef vs #if (чек файлик в примерах к лекции)
- Рассказ по теме
- Примеры

п. 34

if	ifdef

`#if value`
`// код, который выполнится, если value -- истина`
`#elif value1`
`// код, который выполнится, если value1 -- истина`
`#else`
`// код, который выполнится в противном случае`
`#endif`
	`#ifdef NAME_TOKEN`
	`// код, который выполнится, если NAME_TOKEN определен`
	`#else`
	`// код, который выполнится, если NAME_TOKEN не определен`
	`#endif`

value – значение (выражения) могут быть вложенные директивы #elif	NAME_TOKEN – символическая константа или макрос, определенный через #define

`#error “Сообщение об ошибке”`

Позволяет отображать в списке ошибок компиляции сообщение, если возникла соответствующая ошибка

```
#ifndef SIZE
#error “SIZE is not defined”
#endif
```

Директива **#pragma** позволяет добиться от компилятора специфичного поведения.

#pragma once – альтернатива include guard.

Дополнительные преимущества: она короче, и из-за того, что компилятор сам отвечает за обработку #pragma once и программисту нет необходимости создавать новые имена (например, __LIST_H__), риск коллизии имен исключается. Однако поддерживается не всеми компиляторами, хотя широко распространена.

list.h	table.h	main.c

`#pragma once`
 	
`typedef struct`
`{`
`  int data;`
`  node_t *next;`
`} node_t;`
	`#include "list.h"`	`#include "list.h"`
		`#include "table.h"`

#pragma pack – изменение выравнивания структур

```
// push -- сообщение компилятору сохранить текущее выравнивание
// 1 -- задаем нужное выравнивание
// в данном случае выравнивание отсутствует, поля располагаются вплотную
#pragma pack(push, 1)
struct s
{
    char c;
    int i;
    double d;
};
// восстанавливаем ранее сохраненное выравнивание
#pragma pack(pop)
```

37. Препроцессор. Общие понятия. Операции # и

- Что такое препроцессор
- Какие основные действия он выполняет с программой
- На какие группы можно разделить директивы препроцессора
- Какие правила справедливы для всех директив
- Рассказ по теме
- Примеры
- Особенности использования

п. 34

Операция # конвертирует аргумент макроса в строковый литерал

```
#define PRINT_INT(x) printf(#x " = %d\n", x)

PRINT_INT(i/j); // printf("i/j" " = %d", i/j);
```

Операция ## объединяет две лексемы в одну

```
#define GENERAL_MAX(type)       \
type type##_max(type x, type y) \
{                               \
...
```

Особенности:

1. Аргументы подставляются в список замены уже «раскрытыми», если к ним не применяются операции # или ##.
2. После того, как все аргументы были «раскрыты» или выполнены операции # или ##, результат просматривается препроцессором еще раз. Если результат работы препроцессора содержит имя исходного макроса, оно не заменяется.

и # сразу превращают то что было передано в текст и в код, даже если это имя макроса. Таким образом, следующий пример сработает не так, как нами задумывается:

```
#include <stdio.h>

#define STR(x) #x

#define NAME Bob

int main(void)
{
    puts(STR(NAME));
    return 0;
}
```

Макрос раскроется так:

```
puts("NAME");
```

Чтобы все-таки получить желаемое, нужно вспомнить вышеуказанные свойства, а именно: аргументы подставляются в список уже раскрытыми, если к ним не применяется # или ##. Т.е. нужно написать небольшую обертку чтобы к моменту действия операции #x попали уже раскрытые макросы

```
#define STR_HELPER(x) #x
#define STR(x) STR_HELPER(x)
```

Это сначала раскроется как `STR_HELPER(Bob)` а затем уже как "Bob"

Аналогично для ##

```
#include <stdio.h>

#define AVG(x) (((max_##x) - (min_##x)) / (x##_count))

#define TIME time

int main(void)
{
    double max_v = 5, min_v = 0, v_count = 10;
    double max_time = 10, min_time = 5, time_count = 100;

    printf("avg(v) %f\n", AVG(v));

    printf("avg(TIME) %f\n", AVG(TIME));

    return 0;
}
```

Это сделает неправильные выражения в которых будет не `time`, а `TIME`.

```
#include <stdio.h>

#define GLUE_HELPER(x, y) x##y
#define GLUE(x, y) GLUE_HELPER(x, y)

#define AVG(x) (((GLUE(max_, x)) - (GLUE(min_,x))) / (GLUE(x,_count)))

#define TIME time

int main(void)
{
    double max_v = 5, min_v = 0, v_count = 10;
    double max_time = 10, min_time = 5, time_count = 100;

    printf("avg(v) %f\n", AVG(v));

    printf("avg(TIME) %f\n", AVG(TIME));

    return 0;
}
```

А вот тут уже все правильно сработает

38. Встраиваемые функции

- c99: inline
- Почему появилось
- Способы описания встраиваемых функций

`inline` – пожелание компилятору заменить вызовы функции последовательной вставкой кода самой функции

В с99 `inline` означает, что определение функции предоставляется только для подстановки и где-то в программе должно быть другое такое же определение этой же функции.

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

До стандарта **С99**, если функция имела класс памяти `static` в заголовочном файле, то она заменялась в файле на свое тело. Это привело к проблеме большого увеличения размера исполняемого файла. Поэтому было добавлено ключевое слово **`inline`**, которое уже и советует компилятору заменить вызов функции на ее тело.

Способы описания:

1. Использовать ключевое слово static
   `static inline add(a, b) {return a + b;}`
   Недостаток: необходимо дублировать функцию по всей программе, если надо
2. Убрать inline
   Недостаток: Компилятор может не выполнить встраивание
3. Добавить не-inline определение где-нибудь в программе.
   Недостаток: возможны 2 разных определения, необходимость дублирования функции.
4. С99: добавить файл реализации с таким объявлением
   `extern inline int add(int a, int b);`
   Недостаток: **та хз)))))**

39. Библиотеки. Статические библиотеки.

ЧТО ТАКОЕ БИБЛИОТЕКА?

Библиотека – сборник подпрограмм и объектов, используемых для разработки ПО.

Библиотека включает в себя

– заголовочный файл;

– откомпилированный файл самой библиотеки:

• библиотеки меняются редко – нет причин перекомпилировать каждый раз;

• двоичный код предотвращает доступ к исходному коду.

Библиотеки делятся на

– статические;

– динамические.

Распространяются в виде откомпилированных объектных файлов, объединенных в библиотеки. Библиотеки могут включать в себя какие-то функции (например, математические) и/или типы данных (va_list в stdarg.h).

У каждой библиотеки должен быть свой заголовочный файл, в котором должны быть описаны прототипы (объявления) всех функций, содержащихся в этой библиотеке. С помощью заголовочных файлов вы "сообщаете" вашему программному коду, какие библиотечные функции есть и как их использовать.

Статические библиотеки связываются с программой в момент компоновки.

Код библиотеки помещается в исполняемый файл.

	ПЛЮСЫ И МИНУСЫ СТАТИЧЕСКИХ БИБЛИОТЕК
+	– Исполняемый файл включает в себя все необходимое. – Не возникает проблем с использованием не той версии библиотеки.
-	– «Размер» если несколько приложений используют одну и ту же библиотеку, то код библиотеки в нескольких экземплярах присутствует на компьютере – При обновлении библиотеки программу нужно пересобрать.

СБОРКА СТАТИЧЕСКИХ БИБЛИОТЕК

Одинаково для Windows и GNU/Linux

компиляция:

gcc -std=c99 -Wall -Werror -c name_lib.c

упаковка:

ar rc libname.a name_lib.o

индексирование (необязательно):

ranlib libname.a

Упаковка – получение архива из объектных файлов.

c – создать файл библиотеки, если он не существует (create)
r – если уже есть файл с библиотекой и ОФ, то функции будут помещаться, если библиотека содержит их более старые версии. (replace)

Индексирование – если библиотека состоит из большого числа объектных файлов и функций, то, чтобы ускорить работу компоновщика, рекомендуется создать индекс. Это делается с помощью утилиты ranlib, которая модифицирует сам файл библиотеки libname.a

Функции таких библиотек никаким специальным образом не “оформляются” (статическая библиотека – архив объектных файлов)

СБОРКА ПРИЛОЖЕНИЙ С ИСПОЛЬЗОВАНИЕМ СТАТИЧЕСКИХ БИБЛИОТЕК

gcc -std=c99 -Wall -Werror main.c libname.a -o app.exe

или

gcc -std=c99 -Wall -Werror main.c -L. -lname -o app.exe

-L - задает папки, в которых компоновщик будет искать библиотеки (аналог I для заголовочников при препроцессировании) . (точка)– текущая директория

Исходный код приложений, которые используют такие библиотеки, также никаким специальным образом не “оформляется”

40. Библиотеки. Динамические библиотеки. Динамическая компоновка

ЧТО ТАКОЕ БИБЛИОТЕКА?

Библиотека – сборник подпрограмм и объектов, используемых для разработки ПО.

Библиотека включает в себя

– заголовочный файл;

– откомпилированный файл самой библиотеки:

• библиотеки меняются редко – нет причин перекомпилировать каждый раз;

• двоичный код предотвращает доступ к исходному коду.

Библиотеки делятся на

– статические;

– динамические.

Распространяются в виде откомпилированных объектных файлов, объединенных в библиотеки. Библиотеки могут включать в себя какие-то функции (например, математические) и/или типы данных (va_list в stdarg.h).

У каждой библиотеки должен быть свой заголовочный файл, в котором должны быть описаны прототипы (объявления) всех функций, содержащихся в этой библиотеке. С помощью заголовочных файлов вы "сообщаете" вашему программному коду, какие библиотечные функции есть и как их использовать.

Динамическая компоновка – это делегирование части функции по загрузке библиотеки и поиску нужных функций в этой библиотеке компоновщику

При использовании динамических библиотек подпрограммы из библиотеки загружаются в приложение во время выполнения. Код библиотеки не помещается в исполняемый файл.

	ПЛЮСЫ И МИНУСЫ ДИНАМИЧЕСКИХ БИБЛИОТЕК
+	– Несколько программ могут «разделять» одну библиотеку. – Меньший размер приложения (по сравнению с приложением со статической библиотекой). – Модернизация библиотеки не требует перекомпиляции программы. – Могут использовать программы на разных языках.
-	– Требуется наличие библиотеки на компьютере. – Версионность библиотек (если поменялся интерфейс)

СБОРКА ДИНАМИЧЕСКИХ БИБЛИОТЕК

Windows	**Linux**
`.dll – dynamic linked library`	`.so`
`-shared` – собираем именно динамическую библиотеку	

компиляция:
`- gcc -std=c99 -Wall -Werror -c name_lib.c`
компоновка:
`gcc -shared name_lib.o -Wl, --subsystem,windows -o name.dll`
	компиляция:
	`- gcc -std=c99 -Wall -Werror -fPIC -c name_lib.c`
	компоновка:
	`gcc -o libname.so -shared name_lib.o`
	`export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:.`

ОФОРМЛЕНИЕ ФУНКЦИЙ ДИНАМИЧЕСКИХ БИБЛИОТЕК

Windows	**Linux**
Изменяются заголовки функций в .h и .c файлах. Функции, которые доступны из библиотеки `__declspec(dllexport)` Функции, которые недоступны из библиотеки – **не помечаются** Указывается соглашение о вызове, по умолчанию – `__cdecl`. Мы его пишем для подстраховки	Не нужно

Пример

```
// При сборке DLL определяем макрос NAME_EXPORTS
#ifdef NAME_EXPORTS
#define NAME_DLL __declspec(dllexport)
#else
#define NAME_DLL __declspec(dllimport)
#endif

#define NAME_DECL __cdecl

NAME_DLL void NAME_DECL foo(type args);
```

ОФОРМЛЕНИЕ ФУНКЦИЙ ПРИЛОЖЕНИЙ ПРИ ИСПОЛЬЗОВАНИИ ДИНАМИЧЕСКИХ БИБЛИОТЕК

Windows	Linux
Компоновщик создает таблицу импорта для функций, которые нужно взять из библиотек. Компоновщик должен понять, что данная функция берется из библиотеки. Для этого используется атрибут dllimport с помощью конструкции __declspec в качестве ее аргумента `__declspec(dllimport)`)	Не нужно

ПЛЮСЫ И МИНУСЫ СПОСОБОВ КОМПОНОВКИ ПРИЛОЖЕНИЯ

	Динамическая загрузка	Динамическая компоновка
+	- Функции используются по мере необходимости	- Linux: не нужно изменять код библиотеки - Подключение библиотеки происходит автоматически
-	- Управление загрузкой/выгрузкой функций вручную - Изменение кода программы	- Windows: приходится вносить изменения в код библиотеки - Не все функции могут быть нужны

СБОРКА ПРИЛОЖЕНИЙ ПРИ ДИНАМИЧЕСКОЙ КОМПОНОВКЕ:

```
gcc -std=c99 -Wall -Werror -c main.c
```

```
gcc main.o -L. –larr -o test.exe
```

41. Библиотеки. Динамические библиотеки. Динамическая загрузка

ЧТО ТАКОЕ БИБЛИОТЕКА?

Библиотека – сборник подпрограмм и объектов, используемых для разработки ПО.

Библиотека включает в себя

– заголовочный файл;

– откомпилированный файл самой библиотеки:

• библиотеки меняются редко – нет причин перекомпилировать каждый раз;

• двоичный код предотвращает доступ к исходному коду.

Библиотеки делятся на

– статические;

– динамические.

Распространяются в виде откомпилированных объектных файлов, объединенных в библиотеки. Библиотеки могут включать в себя какие-то функции (например, математические) и/или типы данных (va_list в stdarg.h).

У каждой библиотеки должен быть свой заголовочный файл, в котором должны быть описаны прототипы (объявления) всех функций, содержащихся в этой библиотеке. С помощью заголовочных файлов вы "сообщаете" вашему программному коду, какие библиотечные функции есть и как их использовать.

Динамическая загрузка – осуществление поиска и загрузки функций динамической библиотеки самостоятельно, без участия компоновщика, с использованием интерфейса, который предоставляет ОС.

Windows	Linux
`windows.h`	`dlfcn.h`
`HMODULE LoadLibrary(LPCSTR);` – возвращает дескриптор библиотеки	`void* dlopen(const char *file, int mode);`

`FARPROC GetProcAddress(HMODULE, LPCSTR);` – получение адреса функции	`void* dlsym(void *restrict handle, const char *restrict name);`
`FreeLibrary(HMODULE);`	`int dlclose(void *handle);`

При использовании динамических библиотек подпрограммы из библиотеки загружаются в приложение во время выполнения. Код библиотеки не помещается в исполняемый файл.

	ПЛЮСЫ И МИНУСЫ ДИНАМИЧЕСКИХ БИБЛИОТЕК
+	– Несколько программ могут «разделять» одну библиотеку. – Меньший размер приложения (по сравнению с приложением со статической библиотекой). – Модернизация библиотеки не требует перекомпиляции программы. – Могут использовать программы на разных языках.
-	– Требуется наличие библиотеки на компьютере. – Версионность библиотек (если поменялся интерфейс)

СБОРКА ДИНАМИЧЕСКИХ БИБЛИОТЕК (ДИНАМИЧЕСКАЯ КОМПОНОВКА)

Windows	**Linux**
.dll – dynamic linked library	.so
-shared – собираем именно динамическую библиотеку	
компиляция: - gcc -std=c99 -Wall -Werror -c name_lib.c компоновка: gcc -shared name_lib.o -Wl, --subsystem,windows -o name.dll	компиляция: - gcc -std=c99 -Wall -Werror -fPIC -c name_lib.c компоновка: gcc -o libname.so -shared name_lib.o export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:.

ОФОРМЛЕНИЕ ФУНКЦИЙ ДИНАМИЧЕСКИХ БИБЛИОТЕК

Windows	Linux
Изменяются заголовки функций в .h и .c файлах. Функции, которые доступны из библиотеки __declspec(dllexport) Функции, которые недоступны из библиотеки – **не помечаются** Указывается соглашение о вызове, по умолчанию – __cdecl. Мы его пишем для подстраховки	Не нужно

Пример

```
// При сборке DLL определяем макрос NAME_EXPORTS
#ifdef NAME_EXPORTS
#define NAME_DLL __declspec(dllexport)
#else
#define NAME_DLL __declspec(dllimport)
#endif

#define NAME_DECL __cdecl

NAME_DLL void NAME_DECL foo(type args);
```

ОФОРМЛЕНИЕ ФУНКЦИЙ ПРИЛОЖЕНИЙ ПРИ ИСПОЛЬЗОВАНИИ ДИНАМИЧЕСКИХ БИБЛИОТЕК

Windows	Linux
Компоновщик создает таблицу импорта для функций, которые нужно взять из библиотек. Компоновщик должен понять, что данная функция берется из библиотеки. Для этого используется атрибут dllimport с помощью конструкции __declspec в качестве ее аргумента __declspec(dllimport))	Не нужно

ПЛЮСЫ И МИНУСЫ СПОСОБОВ КОМПОНОВКИ ПРИЛОЖЕНИЯ

	Динамическая загрузка	Динамическая компоновка
+	- Функции используются по мере необходимости	- Linux: не нужно изменять код библиотеки

		- Подключение библиотеки происходит автоматически
-	- Управление загрузкой/выгрузкой функций вручную - Изменение кода программы	- Windows: приходится вносить изменения в код библиотеки - Не все функции могут быть нужны

СБОРКА ПРИЛОЖЕНИЙ ПРИ ДИНАМИЧЕСКОЙ ЗАГРУЗКЕ:

gcc -std=c99 -Wall -Werror main.c –o test.exe -ldl

42. Библиотеки. Динамические библиотека на Си. Приложение на Python

- Что такое библиотека
- В какой форме распространяются и почему
- Виды библиотек их +/-
- Оформление отличается в windows и linux
- Рассказ по теме

Все кроме рассказа по теме см в 41.

Чтобы загрузить библиотеку в программу на Python необходимо создать объект класс CDLL:

```
import ctypes
lib = ctypes.CDLL('example.dll')
```

Классов для работы с библиотеками в модуле ctypes несколько:

- CDLL (cdecl и возвращаемое значение int);
- OleDLL (stdcall и возвращаемое значение HRESULT);
- WinDLL (stdcall и возвращаемое значение int).

Класс выбирается в зависимости от соглашения о вызовах, которое использует библиотека.

После загрузки библиотеки необходимо описать заголовки функций библиотеки, используя нотацию и типы известные Python.

```
# int add(int, int)
add = lib.add
add.argtypes = (ctypes.c_int, ctypes.c_int)
add.restype = ctypes.c_int
```

Чтобы интерпретатор Python смог правильно конвертировать аргументы, вызвать функцию add и вернуть результат ее работы, необходимо указать атрибуты argtypes и restype.

Целые числа в Python «неизменяемые» объекты. Попытка их изменить вызовет исключение. Поэтому для аргументов, которые «используют» указатель, необходимо с помощью описанных в модуле ctypes совместимых типов создать объект и передать именно его.

```
def divide(x, y):
  rem = ctypes.c_int()
  quot = _divide(x, y, rem)
  return quot, rem.value
```

Функция avg ожидает получить указатель на массив. Необходимо понять, какой тип данных Python будет использоваться (список, кортеж и т.п.) и как он преобразуется в массив.

```
# void avg(double*, int)
_avg = lib.avg
_avg.argtypes = (ctypes.POINTER(ctypes.c_double), \
                                          ctypes.c_int)
_avg.restype = ctypes.c_double

def avg(nums):
  src_len = len(nums)
  src = (ctypes.c_double * src_len)(*nums)
  return _avg(src, src_len)
```

ctypes: итоги:

- Основная проблема использования этого модуля с большими библиотеками – написание большого количества сигнатур для функций и, в зависимости от сложности функций, функций-оберток.
- Необходимо детально представлять внутреннее устройство типов Python и то, каким образом они могут быть преобразованы в типы Си.
- Альтернативные подходы – использование Swig или Cython.

Обычно функции модуля расширения имеют следующий вид

```
static PyObject* py_func(PyObject* self, PyObject* args)
{
    ...
}
```

- PyObject – это тип данных Си, представляющий любой объект Python.

- Функция модуля расширения получает кортеж таких объектов (args) и возвращает новый Python объект в качестве результата.
- Аргумент self не используется в простых функциях.
- Функция PyArg_ParseTuple используется для конвертирования переменных из представления Python в представление Си.
- На вход эта функция принимает строку форматирования, которая описывает тип требуемого значения, и адреса переменных, в которые будут помещены значения.
- В ходе конвертации функция PyArg_ParseTuple выполняет различные проверки. Если что-то пошло не так, функция возвращает NULL.

```
int a, b;
if (!PyArg_ParseTuple(args,"ii", &a, &b))
    return NULL;
```

- Функция Py_BuildValue используется для создания объектов Python из типов данных Си. Эта функция также получает строку форматирования с описанием желаемого типа.

```
int a, b, c;
if (!PyArg_ParseTuple(args,"ii", &a, &b))
   return NULL;
c = add(a, b);
return Py_BuildValue("i", c);
```

- Ближе к концу модуля расширения располагаются таблица методов модуля PyMethodDef и структура PyModuleDef, которая описывает модуль в целом.
- В таблице PyMethodDef перечисляются
  - Си функции;
  - имена, используемые в Python;
  - флаги, используемые при вызове функции,
  - строки документации.

- Структура PyModuleDef используется для загрузки модуля.
- В самом конце модуля располагается функция инициализации модуля, которая практически всегда одинакова, за исключением своего имени.

Для компиляции модуля используется Python-скрипт setup.py. Компиляция выполняется с помощью команды:

```
python setup.py build_ext --inplace
```

43. Неопределенное поведение

Определение:
В результате вычисления значения выражения порождаются не только новые данные, но происходит также изменение состояния среды выполнения.

Виды:

- Модификация данных.
- Обращение к переменным, объявленным как volatile.
- Вызов системной функции, которая производит побочные эффекты (например, файловый ввод или вывод).
- Вызов функций, выполняющих любое из вышеперечисленных действий.

Выражения в СИ:

Выражения будут вычисляться почти в том же порядке, в котором они указаны в исходном коде: сверху вниз и слева направо.

Однако компилятор может выполнять действия в разной последовательности в целях оптимизации, кроме случаев, в которых явно гарантируется порядок вычисления

Пример неопределенного поведения:

```
i = 1;
x[i] = i++ + 1;

// Способ 1 (справа налево)
(i++ + 1) => 2, i = 2, x[2] = 2

// Способ 2 (слева направо)
x[1] = (i++ + 1) = 2, i = 2
```

Точки следования:

Порядок определен у :

- Выражений с &&, || .
- У выражений в которых используется “,”
- В тернарных операторах

Операция	Название	Нотация	Класс	Приоритет	Ассоциат.
!	Не	!x	Префиксные	15	Справа налево
&&	И	x && y	Инфиксные	5	Слева направо
||	Или	x || y		4	Слева направо

при && и || следовательно слева направо!!!

Точка следования – это точка в программе, в которой программист знает какие выражения (или подвыражения) уже вычислены, а какие выражения (или подвыражения) еще нет.

Виды точек следования в C99:

- Между вычислением левого и правого операндов в операциях &&, || и "," .

```
*p++ != 0 && *q++ != 0
```

- Между вычислением первого и второго или третьего операндов в тернарной операции.

```
a = (*p++) ? (*p++) : 0;
```

- В конце полного выражения.

```
a = b;
if ()
switch ()
while ()
do{} while()
for ( x; y; z)
return x
```

- Перед входом в вызываемую функцию.
  - Порядок, в котором вычисляются аргументы не определен, но эта точка следования гарантирует, что все ее побочные эффекты проявятся на момент входа в функцию.
- В объявлении с инициализацией на момент завершения вычисления инициализирующего значения.

```
int a = (1 + i++);
```

Виды неопределенного поведения:

- **Unspecified behavior.**

Стандарт предлагает несколько вариантов на выбор. Компилятор может реализовать любой вариант. При этом на вход компилятора подается корректная программа.

Например: все аргументы функции должны быть вычислены до вызова функции, но они могут быть вычислены в любом порядке.

- **Implementation-defined behavior.**

Похоже на неспецифицированное (unspescified) поведение, но в документации к компилятору должно быть указано, какое именно поведение реализовано.

Например: результат x % y, где x и y целые, а y отрицательное, может быть как положительным, так и отрицательным.

- **Undefined behavior**

Такое поведение возникает как следствие неправильно написанной программы или некорректных данных. Стандарт ничего не гарантирует, может случиться все, что угодно.

Почему неопределенное поведение есть в СИ:

- освободить разработчиков компиляторов от необходимости обнаруживать ошибки, которые трудно диагностировать.
- избежать предпочтения одной стратегии реализации другой.
- отметить области языка для расширения языка (language extension).

Способы борьбы с ним:

- Включайте все предупреждения компилятора, внимательно читайте их.
- Используйте возможности компилятора (-ftrapv).
- Используйте несколько компиляторов.
- Используйте статические анализаторы кода (например, clang).
- Используйте инструменты такие, как valgrind, Doctor Memory и др.
- Используйте утверждения.

  - -ftrapv
  - This option generates traps for signed overflow on addition, subtraction, multiplication operations.
  - Эта опция генерирует ловушки для переполнения со знаком при операциях сложения, вычитания, умножения.

Примеры неопределенного поведения:

- Использование неинициализированных переменных.
- Переполнение знаковых целых типов.
- Выход за границы массива.
- Использование «диких» указателей.
- …

См. приложение J стандартна (стр. 490 - 502)

Пример поведения, зависящего от реализации:

Результат x % y, где x и y целые, а y отрицательное, может быть как положительным, так и отрицательным. Размер int может быть от 2 до 4 байт, в зависимости от реализации или настроек компилятора.

Пример неспецифицированного поведения:

Все аргументы функции должны быть вычислены до вызова функции, но они могут быть вычислены в любом порядке.

Результат округления, когда значение выходит за пределы диапазона

Порядок двух элементов, сравниваемых как равные в массиве, отсортированном функцией qsort (неустойчивая сортировка)

Самый опасный вид неопределенного поведения – Undefined behavior

44. АТД. Понятие модуля. Разновидности модулей. Абстрактный объект. Стек целых чисел

Программу можно рассматривать как набор независимых модулей. Выделяют две его части: интерфейс и реализация.
Интерфейс описывает возможности модуля. Реализация описывает, как модуль делает то, что интерфейс предлагает.
Выделяют 4 разновидности модуля:

1. Абстрактный тип данных;
2. Абстрактный объект;
3. Набор данных;
4. Библиотека.

Абстрактный тип данных - это интерфейс, который определяет тип данных и операции над этим типом. Абстрактным является потому, что интерфейс скрывает детали реализации типа.
Абстрактный объект - набор функций, которые обрабатывают скрытые данные.

stack.h	stack.c
… #include <stdbool.h>	#include <stdlib.h> #include <assert.h> #include "stack.h"

`void make_empty(void);`
`bool is_empty(void);`
`bool is_full(void);`
`int push(int i);`
`int pop(int *i);`
	`struct stack_type`
	`{`
	`  int* content;`
	`  size_t top;`
	`  size_t size;`
	
	`};`
	
	`struct stack_type stack;`
	
	… Реализация функций с глобальной переменной stack…

45. АТД. Понятие модуля. Разновидности модулей. Абстрактный тип данных. Стек целых чисел

см. предыдущий пункт

stack.h	stack.c

…
`typedef struct stack_type* stack_t;`
 	
`stack_t create(void);`
`void destroy(stack_t s);`
`void make_empty(stack_t s);`
`bool is_empty(const stack_t s);`
`bool is_full(const stack_t s);`
`int push(stack_t s, int i);`
`int pop(stack_t s, int *i);`
	`#include <stdlib.h>`
	`#include <assert.h>`
	`#include "stack.h"`
	
	`struct stack_type`
	`{`
	`  int* content;`
	`  size_t top;`
	`  size_t size;`
	
	`};`
	
	… Реализация функций с параметрами функций…

46. Списки ядра Linux. Идея. Основные моменты использования.

- Идея реализации универсального списка в ядре Linux
- Как описывается голова списка
- Добавление
- Обход
- Удаление
- Освобождение

Списки Беркли: идея

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;
    ...
};
```


Списки Беркли: описание

```
#include "list.h"

struct data
{
    int num;
    struct list_head list;
};
```

Следует отметить следующее:

- 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.

Список Беркли – циклический двусвязный список, в основе которого лежит такая структура:

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

`list_head` сама является частью данных.

Мб в любом месте в определении СД.

Может иметь любое имя

В СД может быть несколько полей типа `struct list_head`

Способы описания:

1.
```
LIST_HEAD(num_list); // передаем имя переменной, которую хотим
создать

#define LIST_HEAD_INIT(name) { &(name), &(name) }
#define LIST_HEAD(name) \
    struct list_head *name = LIST_HEAD_INIT(name)
```
2.
```
struct data num_list;
INIT_LIST_HEAD(&num_list.list);

static inline void INIT_LIST_HEAD(struct list_head *list)
{
    list->next = list;
    list->prev = list;
}
```

Добавление:

```
struct data *item;
LIST_HEAD(num_list);

item = malloc(sizeof(*item))
if (!item)
    ...

*item = 5;
INIT_LIST_HEAD(&item->list);
list_add(&item->list, &num_list); // добавление в начало
list_add_tail(&item->list, &num_list); // добавление в конец
```

Обход:

1. Обычный обход, внутри которого данные придется доставать с помощью list_entry

```
struct list_head *iter;
```

```
list_for_each(iter, &num_list)
{
    // list_entry(элемент, сама структура, имя поля)
    item = list_entry(iter, struct data, list)
    ...
}

list_for_each_prev(iter, list_head) // в обратном направлении
```

2. Обычный обход, внутри которого не нужно будет использовать list_entry

```
list_for_each_entry(item, &num_list, list)
{
    ...
}

list_for_each_entry_prev()
```

3. Обход с копированием

```
list_for_each_safe(iter, iter_safe, &num_list)
{
    ...
}
```

Удаление элемента:

```
tmp = list_entry(num_list.next, struct data, list);
list_del(num_list.next);
free(tmp);
```

Освобождение списка

```
list_for_each_safe(iter, iter_safe, &num_list)
{
    item = list_entry(iter, struct data, list);
    list_del(iter);
    free(item);
}
```

47. Списки ядра Linux. Идея. Основные моменты реализации.

- Идея реализации универсального списка в ядре Linux
- Основное внимание уделяем макросам container_of и offsetof

Список Беркли – циклический двусвязный список, в основе которого лежит такая структура:

```
struct list_head
```

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

list_head сама является частью данных.

Мб в любом месте в определении СД.

Может иметь любое имя

В СД может быть несколько полей типа struct list_head

Идея offsetof

Идея заключается в том, чтобы узнать смещение заданного поля. То есть предполагаем, что адрес структуры располагается по нулевому адресу. Тогда, расстояние до поля данной структуры легко вычисляется путем преобразования его к целому числу:

```
#define offsetof(TYPE, MEMBER) ((size_t) &((TYPE *)0)->MEMBER)
int offset = (int) (&((struct s*)0)->i)
```

```
((struct s*)0) // эта строчка говорит компилятору, что по адресу 0 располагается структура, и мы получаем указатель на нее

((struct s*)0)->i // компилятор думает, что это поле расположено по адресу 0 + смещение i

&((struct s*)0)->i // вычисляем смещение i в структуре s

(size_t) &((struct s*)0)->i
```

Идея container_of

Данный макрос используется для получения указателя на структуру по заданному полю этой структуры. Он помогает получить доступ к данным, например, в списках Беркли:

```
#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))
```