Предисловие

В связи с аккредитацией, никаких поблажек на экзамене не будет. На тройку **не** вытягивают. В особом случае (зависит от семинариста) могут быть доставлены дополнительные баллы (до 6).
Бесконечно ходить сдавать не получится: после **первой** несдачи выдаётся направление на пересдачу, после **второй** несдачи — комиссия. Потом Вас имеют право **отчислить**.

Структура билета:

1) Теория — 18 баллов. (вопросы рандомные из разных блоков)

2) Практическая задача (лёгкая) — 6 баллов.

3) Практическая задача (тяжёлая) — 12 баллов.
   • Решается с использованием предыдущей задачи (не всегда).

Чтобы получить минимум за билет (18 баллов): идеально написанная теория + лёгкая задача.

По словам Кострицкого:

1) В практике не будет заданий на библиотеки, графические интерфейсы и списки Беркли.

2) Чем больше будет написано в теории — тем лучше.

3) За ересь в билете экзамен может закончиться очень быстро.

4) Желательно чередовать теорию с практикой. Написали теорию, поделали практику, вернулись к теории.

Дополнение

Если баллов меньше 41 — на экзамен не допущен.

Содержание

1. Модули

1.1. Многофайловый проект. Модуль. Разделы описания и реализации, область видимости. Этапы получения исполняемого файла.

Многофайловый проект — проект, состоящий из некторого связанного между собой набора данных (модулей).

Преимущества модульной организации программы.

1) Программу удобно рассматривать как набор независимых модулей.
2) Модуль состоит из двух частей: интерфейса и реализации.
3) Интерфейс описывает, что модуль делает. Он определяет идентификаторы, типы и подпрограммы, которые будут доступны коду, использующему этот модуль.
4) У модуля есть один интерфейс, но реализаций, удовлетворяющих этому интерфейсу, может быть несколько. Реализация описывает, как модуль выполняет то, что предлагает интерфейс.
5) Часть кода, которая использует модуль, называют клиентом. Клиент должен зависеть только от интерфейса, но не от деталей его реализации.

Так же над многофайловым проектом может работать несколько человек (в отличии от однофайлового), при изменении одного модуля не нужно перекомпилировать все остальные.

Организация в языке С:

1) Интерфейс описывается в заголовочном файле ***.h**.
2) В заголовочном файле описываются макросы, типы, переменные и функции, которые клиент может использовать.
3) Клиент импортирует интерфейс с помощью директивы препроцессора **include**.
4) Реализация интерфейса в языке Си представляется одним или несколькими файлами с расширением ***.c**.
5) Реализация определяет переменные и функции, необходимые для обеспечения возможностей, описанных в интерфейсе.
6) Реализация обязательно должна включать файл описания интерфейса, чтобы гарантировать согласованность интерфейса и реализации

Этапы получения исполняемого файла:

1) Препроцессинг (удаление комментариев, текстовые включения, макроподстановки **define** и т.д)
   `cpp -o main.i main.c`
2) Трансляция: `c99 -S -masm=intel main.i`
3) Ассемблирование: `as -o main.o main.s`
4) Компоновка: `ld -o hello.exe hello.o`

2. Многофайловый проект

ТЕОРИЯ ИДЁТ КАК ОБЯЗАТЕЛЬНЫЙ ДОП ВОПРОС!!!

2.1. Автоматизация сборки проекта на примере утилиты make. Зависимости и правила

make — утилита, автоматизирующая процесс преобразования файлов из одной формы в другую. Чаще всего это компиляция исходного кода в объектные файлы и последующая компоновка в исполняемые файлы или библиотеки. Утилита использует специальные make-файлы, которые состоят из:
• **целей** (то, что данные делают)
• **зависимостей** (то, что необходимо для выполнения правила и получения целей)
• **команд** (выполняющих данные преобразования)
На основе информации о времени последнего изменения каждого файла make определяет и запускает необходимые сценарии. По умолчанию запускается **первая** по счёту цель. По умолчанию будет произведён поиск файла с навзанием **makefile**.
`make [ -f make-файл ] [ цель ] ...`

В общем виде синтаксис make-file можно представить так:

```
<цели>: <реквизиты>
        <команда 1>
        ...
        <команда n>
```

Обязательно сказать, что используются TAB'ы!

Пример **make**-файла для сборки многофалового проекта:

```
# Компилятор
CC = gcc

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

app.exe: main.o hello.o
	$(CC) -o app.exe main.o hello.o

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

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

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

.

3. Динамическая память в C

3.1. Стек как структура данных.

Стек (стопка) — СД, допускающая три операции:

1) **push** — Добавить сверху
2) **peek** — Подсмотреть верхний элемент
3) **pop** — Удалить верхний элемент

Стек устроен достаточно просто — данные обрабатываются в соответствии с принципом «последним пришёл — первым вышел» (LIFO).
По этой причине, для отслеживания содержания стека не нужно сложных управляющих структур – достаточно всего лишь указателя на верхушку стека.
Добавление данных в стек и их удаление – быстрая и четко определенная операция. Также область памяти, как правило, остается в кэше после обработки, что значительно ускоряет доступ

• Операция **peek** по сути содержит в себе **pop**, ведь при просмотре последнего элемента он извлекается из стека.
• Асимптотика: все операции над стеком имеют временную сложность O(1)

3.2. Куча как структура данных.

Куча — СД, включающая в себя набор элементов, обладающих бинарным признаком и допускающая две группы операций:

1) Смена признака
2) Объединение элементов в один (разъединение)

3.3. Разделение памяти программы.

Память, которую использует программа делится на три вида:

1) Статическая память (static memory)
    а) Хранит глобальные переменные и константы
    б) Размер определяется при компиляции
2) Стек (stack)
    а) Хранит локальные переменные, аргументы функций и промежуточные значения вычислений
    б) Размер определяется при запуске программы (обычно выделяется 4 Мб)
3) Куча (heap)
    а) Динамически распределяемая память
    б) ОС выделяет память по частям (по мере необходимости)

в) Размер динамической памяти можно изменять во время работы программы (обращаясь к диспетчеру (менеджеру) памяти).

• Статическая память всегда быстрее динамической (обращение к менеджеру памяти очень затратная операция).
• Менеджер памяти может делить блоки (а может и не делить).
• Возможен отказ в выделении памяти.
• Освободить можно только выделенную память.
• Если нужно n байт и они есть, но не в виде единого целого куска, менеджер может перетасовать память.

3.4. Работа с динамической памятью в C. Функции malloc, calloc, realloc, free.

Все нижеуказанные функции находятся в заголовочном файле **stdlib.h**

1) malloc (`void *malloc(size_t size)`)
   • Выделяет **size** байт. В случае неудачи возвращает **NULL**
   • Выделенная память никаким образом не инициализируется.
   • Тип данных внутри определяется программистом явным приведением типа.

2) calloc (`void *calloc(size_t nmemb, size_t size)`)
   • Выделяет память под **nmemb** элементов размером **size** байт каждый
   • В случае неудачи возвращает **NULL**
   • Выделенная память инициализируется нулями.
   • Замена - **malloc** и **memset**.

3) realloc (`void *realloc(void *ptr, size_t newsize)`)
   • Изменяет величину выделенной памяти, на которую указывает **ptr**, на новую величину, задаваемую параметром **newsize**.
   • В случае неудачи старые данные остаются, возвращает **NULL**
   • Величина **newsize** задается в байтах и может быть больше или меньше оригинала.
   • Возвращается указатель на новый блок памяти (при увеличении размера области старые данные туда копируются).
   • Возможно переразмещение содержимого, полагаться на старый указатель нельзя.
   • Замена - **malloc** и **free**.
   • **realloc(NULL, size)** - аналогично функции **malloc**.

   Achtung! Использование вида **realloc(ptr, 0)** — **НЕ** стандартизированно и **НЕ** является эквивалентным использованию функции **free** (UB)

4) free (`void free(void *ptr)`)
   • Возвращает память, на которую указывает **ptr**, назад в кучу. В результате эта память может быть выделена снова.

   Achtung! Обязательным условием использования функции является то, что освобождаемая память должна была быть предварительно выделена с использованием одной из следующих функций: **malloc**, **realloc** или **calloc**. Использование неверного указателя при вызове этой функции, обычно, ведёт к краху системы.

Также возможны казусы при отправке нулевого указателя. Хотя в документации сказано, что если указатель нулевой никаких действий не происходит.

Всегда стоит сначала проверять значение, возвращаемое функцией и уже потом продолжать работу (блок обработки ошибок работы с памятью). Типичные ошибки:

1) Двойное освобождение (double free)

2) Дикий указатель (wild pointer) - указатель на объект, которого не существует, или который был потерян в процессе выполнения программы.

3) data = realloc(data, size * sizeof(int))

4) Утечка памяти - процесс в программе, когда выделенная память не была освобождена функцией **free**.

3.5. Одномерные динамические массивы элементов простых типов или статических структур данных.

```
#define N 2
typedef struct {
    int a;
    int b;
    } info_t;
/* --- */
int *data;
info_t *struct_data;

// Error handling
data = (int *)malloc(N * sizeof(int *));
struct_data = (struct_data *)malloc(N * sizeof(info_t));

free(simple_data), free(struct_data);
```

3.6. Одномерные динамические массивы указателей на одномерные динамические массивы элементов простых типов или статических структур данных

По своей структуре являются массивами, состоящими из элементов, каждый из которых является также массивом.

```
#define N 2
#define M 3
typedef struct {
    int a;
    int b;
    } info_t;
/* --- */
int **data;
info_t **struct_data;
```

```
10
11 // Error handling
12 data = malloc(N * sizeof(int *));
13 for (int i = 0; i < N; i++)
14     data[i] = malloc(M * sizeof(int));
15
16 struct_data = malloc(N * sizeof(info_t *));
17 for (int i = 0; i < N; i++)
18     struct_data[i] = malloc(M * sizeof(info_t));
19
20 for (int i = 0; i < m; i++)
21     if (data[i])
22         free(data[i]);
23 free(data);
24
25 for (int i = 0; i < N; i++)
26     free()struct_data[i]);
27 free(struct_data);
```

3.7. Записи с указателями на динамическую память

```
1  typedef struct {
2      char *surname;
3      int age;
4  } person_t;
5  /* --- */
6  person_t human;
7  human.surname = NULL;
8  /* --- */
9  free(human.surname);
10 free(person_t);
```

В этом случае, данные хранятся в какой-то области памяти, а в структуре лишь хранится указатель на данную область.
Если мы меняем метами две структуры, можно не освобождать и выделять заново память, а поменять местами указатели.

3.8. Рекурсивная деструкция

Всегда нужно исходить из правила вложенности. Начиная с самой глубокой точки освобождать внутренние объекты постепенно "поднимаясь" вверх. Иначе - утечка памяти (memmory leak), так как мы теряем доступ к указателю.

4. Классы памяти в C

4.1. Область видимости и время жизни.

Область видимости (scope) — это часть текста программы, в пределах которой переменная может быть использована. Выделяют следующие:

1) Блок
2) Файл
3) Функция
4) Прототип функции (сигнатура + тип возврата)

Время жизни (storage duration) — это интервал времени выполнения программы, в течение которого «программный объект» существует. Выделяют:

1) Глобальное (статическое (англ. static))
2) Локальное (автоматическое (англ. automatic))
3) Динамическое (выделенное (англ. allocated))

4.2. Ключевое слово auto.

Класс памяти. Применимо только к переменным, определенным в блоке.
• Переменная, принадлежащая к классу auto, имеет локальное время жизни, видимость в пределах блока.
• По умолчанию любая переменная, объявленная в блоке или в заголовке функции, относится к классу **auto**.

4.3. Ключевое слово extern.

Класс памяти. Используется для переменных определенных как в блоке, так и вне блока. Помогает разделить переменную между несколькими файлами.

```
1 // file1_.c                      | // file_1.c
2 int number;                      | int number;
3                                  |
4 // file_2.c                      | // file_2.c
5 number = 5;                      | extern int number = 5;
6 ...                              | ...
7                                  |
8 // error: 'number' undeclared    | // OK
```

Переменная такого типа имеет глобальное время жизни и файловая область видимости.

Правила использования:

1) Объявлений (extern int number) может быть сколько угодно.
2) Определение (int number) должно быть **только** одно.
3) Объявления и определение должны быть **одинакового** типа.

4.4. Ключевое слово static.

Класс памяти.
У **static** время жизни расширено за пределами области видимости, область видимости - **ОДИН** файл, если объявлено вне блока, иначе - (область видимости) блок.

```
1 static int i;
2 void foo(void)
3 {
4     i = 1;
5 }
6 void bar(void)
7 {
8     i = 5;
9 }
```

1) Такая переменная **сохраняет** свое значение после выхода из блока.

2) Инициализируется **только** один раз.

4.5. Ключевое слово register

Класс памяти. Не путать с const (!). Просьба (!) к компилятору разместить переменную не в памяти, а в регистре процессора.

1) Используется только для переменных, определенных в блоке.

2) Задает локальное время жизни и видимость в блоке.

3) К переменным с классом памяти register **нельзя** применять операцию получения адреса &

5. Матрицы.

5.1. Размещение матриц по строкам и столбцам.

В зависимости от нужд можно хранить матрицу по столбцам или по строкам. По умолчанию в языке **C** матрица в памяти будет расположена по строкам. В **Fortran** наоборот — по столбцам. Важно понимать когда и зачем используются различные методы хранения. Можно ещё заикнуться про кеширование, если осведомленны в данной области.

(используем другие способы из-за алгоритмической сложности, если спросит)

5.2. Хранение матрицы в виде указателя на массив указателей на массивы-строки.

```
1 int **allocate_matrix(int n, int m)
2 {
3     double **int = (double**)malloc(n * sizeof(int*));
4 
5     for (int i = 0; i < n; i++)
6         data[i] = (double*)malloc(m * sizeof(int));
7 
8     return data;
9 }
```

Преимущества:

1) Возможность обмена строки через обмен указателей.

2) Возможно отследить выход за пределы строки.

Недостатки:

1) Сложность выделения и освобождения памяти.

2) Память под матрицу "не лежит"одним куском.

5.3. Хранение матрицы в виде указателя на массив указателей на массивы-столбцы.

Аналогично предыдущему пункту (столбцы и строки меняются местами).

```
1 int **allocate_matrix(int n, int m)
2 {
3     double **int = (double**)malloc(m * sizeof(int*));
4 
5     for (int i = 0; i < m; i++)
6         data[i] = (double*)malloc(n * sizeof(int));
7 
8     return data;
9 }
```

5.4. Хранение матрицы в виде указателя на массив указателей на подмассивы-строки внутри единого блока данных.

```
1  void allocate_matrix(const int n, const int m, int ***row_adr_arr, int **matrix)
2  {
3      int *tmp_matrix;
4      int **tmp_adr;
5
6      tmp_matrix = (int *) malloc(n * m * sizeof(int));
7      tmp_adr = (int **) malloc(n * sizeof(int *));
8
9      for (int *cur = tmp_matrix, i = 0; i < n; i++)
10     {
11         tmp_adr[i] = cur;
12         cur += m;
13     }
14
15     *row_adr_arr = tmp_adr;
16     *matrix = tmp_matrix;
17 }
```

Преимущества:

1) Простота выделения и освобождения памяти.

2) Перестановка строк через обмен указателей.

Недостатки:

1) Расходуется больше ресурсов.

2) СПРП не может отследить выход за пределы строки.

5.5. Хранение матрицы в виде указателя на массив указателей на подмассивы-столбцы внутри единого блока данных.

Аналогично предыдущему пункту. Только теперь у нас в блоке лежат не строки а столбцы.

```
1  void allocate_matrix(const int n, const int m, int ***row_adr_arr, int **matrix)
2  {
3      int *tmp_matrix;
4      int **tmp_adr;
5
6      tmp_matrix = (int *) malloc(n * m * sizeof(int));
7      tmp_adr = (int **) malloc(m * sizeof(int *));
8
9      for (int *cur = tmp_matrix, i = 0; i < m; i++)
10     {
11         tmp_adr[i] = cur;
12         cur += n;
13     }
14
15     *row_adr_arr = tmp_adr;
```

```
16      *matrix = tmp_matrix;
17  }
```

5.6. Хранение матрицы в виде указателя на объединённый массив элементов и указателей на массивы-строки.

```
1   int** allocate_matrix_solid(int n, int m)
2   {
3       int **data = malloc(n * sizeof(int*) +
4           n * m * sizeof(int));
5
6       for (int i = 0; i < n; i++)
7           data[i] = (int*)(data + n * sizeof(int*)
8               + i * m * sizeof(int));
9
10      return data;
11  }
```

Преимущества:

1) Простота выделения и освобождения памяти.
2) Возможность использовать как одномерный массив.
3) Перестановка строк через обмен указателей.

Недостатки:

1) Сложность начальной инициализации.
2) СПРП не может отследить выход за пределы строки.

5.7. Хранение матрицы в виде указателя на объединённый массив элементов и указателей на массивы-столбцы.

Аналогично предыдущему пункту.

```
1   int** allocate_matrix_solid(int n, int m)
2   {
3       int **data = malloc(m * sizeof(int*) +
4           n * m * sizeof(int));
5
6       for (int i = 0; i < m; i++)
7           data[i] = (int*)(data + m * sizeof(int*)
8           + i * n * sizeof(int));
9
10      return data;
11  }
```

5.8. Хранение матрицы в виде одномерного массив единым блоком (пусть будет)

```
int *data;
int n = 3, m = 2;
data = malloc(n * m * sizeof(int));
if (data)
{
    for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
            data[i * m + j] = 0.0;
}
```

Преимущества:

1) Простота выделения и освобождения памяти.

2) Возможность использовать как одномерный массив.

Недостатки:

1) Средство проверки работы с памятью (СПРП), например Doctor Memory, не может отследить выход за пределы строки.

6. Элементы динамического программирования

6.1. Рекурсивная подпрограмма.

Рекурсивной называется функция, которая вызывает сама себя.
Возможность, иногда единственная, выразить итеративный процесс без возможности выражения итеративного цикла.

1) Рекурсия должна иметь базу - набор условий, определяющий окончание рекурсии.
2) Все вызовы попадают в стек вызовов (call stack).
3) Стек вызовов может быть переполнен!

6.2. Хвостовая рекурсия.

Частный случай рекурсии при котором рекурсивный вызов является последней инструкцией перед точкой выхода.
Хвостовая рекурсия может быть "оптимизирована"самим транслятором - преобразована в итеративный пороцесс (а может и нет).

6.3. Мемоизация.

ЗДЕСЬ ОГРОМНЫЙ ПРОСТОР ДЛЯ ДЕЙСТВИЙ. ЧЕМ БОЛЬШЕ НАПИШЕТЕ ТЕМ ЛУЧШЕ (КЕШ С ОКНОМ, ДИНАМИЧЕСКИЙ КЕШ И Т.Д.)

Мемоизация (оно же кеширование) — сохранение результатов выполнения функции, для предотвращения повторных выполнений.

6.4. Рекурсия, хвостовая рекурсия и мемоизация на примере реализации функции факториала.

Факториал обычной рекурсией.

```
1 int fact_bad(int n)
2 {
3     if (n == 0)
4         return 1;
5
6     return n * fact_bad(n - 1);
7 }
```

Факториал с хвостовой рекурсией. Пишем "обёртку"для нашей функции.

```
1 int fact_good(int n, int a)
2 {
3     if (n == 0)
4         return a;
5
6     return fact_good(n - 1, n * a);
7 }
```

```
8
9  int fact_wrap(int n)
10 {
11     return fact_good(n, 1);
12 }
```

Для применения мемоизации заведём отдельный массив, в который будем помещать уже высчитанный результат.

```
1  #define MAX_NUM 10
2  static int results[MAX_NUM] = { 0 };
3
4  int fact_mem(int n)
5  {
6      if (!results[n])
7      {
8          if (n == 0)
9              return 1;
10
11         results[n] = fact_mem(n - 1) * n;
12     }
13
14     return results[n];
15 }
```

6.5. Рекурсия, хвостовая рекурсия и мемоизация на примере реализации функции Фибоначчи

Фибоначчи обычной рекурсией.

```
1  int fib_bad(int n)
2  {
3      if (n <= 1)
4          return n;
5      return fib_bad(n - 1) + fib_bad(n - 2);
6  }
```

Фибоначчи хвостовой рекурсией. Аналогично пишем "обёртку".

```
1  int fib_good(int n, int a, int b)
2  {
3      if (n == 0)
4          return a;
5      if (n == 1)
6          return b;
7
8      return fib_good(n - 1, b, a + b);
9  }
10
11 int fib_wrap(int n)
12 {
13     return fib_good(n, 0, 1)
```

```
14 }
```

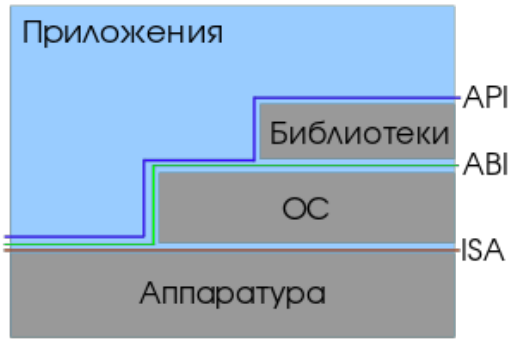
Для применения мемоизации заведём отдельный массив, в который будем помещать уже высчитанный результат. Так же нам понадобиться функция инициализации т.к 0 является числом фибоначчи.

```
1  #define MAX_NUM 40
2  static int results[MAX_NUM];
3
4  void init(void)
5  {
6      for(int i = 0; i < MAX_NUM; i++)
7          results[i] = -1;
8  }
9
10 int fib_mem(int n)
11 {
12     if (results[n] == -1)
13     {
14         if(n <= 1)
15             results[n] = n;
16         else
17             results[n] = fib_mem(n-1) + fib_mem(n-2);
18     }
19
20     return results[n];
21 }
```

7. Подпрограммы.

7.1. Соглашение о вызовах, единство бинарного интерфейса.

Способ передачи аргумента называется соглашением о вызовах и является частью реализации, а не стандарта.



1) Если значение простое (до 4-х байт) — выравнится до 4-х байт и возвращается через **eax** (регистр общего пользования).
2) Если значение до 8 байт — возвращается через 2 регистра **edx** и **eax**.
3) Если больше 8 байт — возвращается адрес.

7.2. Соглашение cdecl.

1) Возврат как выше.
2) Передача аргументов через стек **справа налево**.
3) Ответственность за освобождение стека лежит на вызывающей стороне. (программист)

7.3. Соглашение pascal.

1) Возврат через аргумент.
2) Передача аргументов через стек **слева направо**.
3) Ответственность за освобождение стека лежит на самой функции.
4) Передача по ссылке только через адрес.

7.4. Соглашение stdcall.

1) Аргументы функции передаются через стек **справа налево**.
2) Ответственность за освобождение стека лежит на самой функции.

7.5. Соглашение fastcall.

1) Первые два аргумента функции передаются через регистры, остальные через стек **справа налево**.

2) Аргументы передаются до 12 байт через 3 регистра, иначе - 3 регистра + стек. (в регистры **слева направо**, в стек **справа налево**)

3) Возрат осуществляется через регистры.

4) Ответственность за освобождение стека лежит на самой функции.

7.6. Реализация подпрограммы с переменным числом аргументов с использованием стека.

Такая реализация **ЗАВИСИТ** от архитектуры и транслятора.

```
1  int sum(int cnt, ...)
2  {
3      int sum = 0;
4
5      if (cnt <= 0)
6          return sum;
7
8      int *ap = &cnt;
9      int val;
10
11
12     for (int i = 0; i < cnt; i++)
13     {
14         ap += sizeof(int);
15         val = *ap;
16         sum += val;
17     }
18
19     return sum;
20 }
```

7.7. Реализация подпрограммы с переменным числом аргументов с использованием va_args.

```
1  int sum(int cnt, ...)
2  {
3      int sum = 0;
4
5      if (cnt <= 0)
6          return sum;
7
8      va_list argv;
9      va_start(argv, cnt);
10
```

```
11      for (int i = 0; i < cnt; i++)
12          sum += va_arg(argv, int);
13
14      va_end(argv);
15
16      return sum;
17  }
```

8. Абстракция данных

8.1. Указатель на void.

void* - указатель, способный представлять адреса любых объектов. Особенности:

1) Тип указателя **void** используется, если тип объекта неизвестен.

2) Ответственность за валидность содержимого и правильное приведения типов - на программисте.

3) Позволяет передавать в функцию указатель на объект любого типа.

4) Полезен для ссылки на произвольный участок памяти, независимо от размещенных там объектов.

5) В языке С допускается присваивание указателя типа **void*** указателю любого другого типа (и наоборот) без явного преобразования типа указателя.

```
1     double d = 5.0;
2     double *pd = &d;
3     void *pv = pd;
4     pd = pv;
```

6) Указателя типа **void*** нельзя разыменовывать.

7) К указателям типа **void*** не применима адресная арифметика.

8) Размер зависит от разрядности ОС.

8.2. Указатель на функцию. Реализация сортировки с передаваемым указателем на функцию-компаратор.

В ПРАКТИЧЕСКОМ ЗАДАНИИ ИСПОЛЬЗОВАТЬ ВСТРОЕННУЮ СОРТИРОВКУ НЕЛЬЗЯ!!!

Определение указателя на функцию:

```
1 double trapezium(double a, double b, int n, double (*func)(double))
2 double rezult = trapezium(0, 3.14, 25, sin);
```

Вызов функции по указателю:

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

Использование указателей на функции. Пусть необходимо упорядочить массив целых чисел по возрастанию.

```
1 void qsort(void *base, size_t nmemb, size_t size,
2 int (*compar)(const void*, const void*));
3
4 int compare_int(const void* p, const void* q)
5 {
6     const int *a = p;
```

```
7       const int *b = q;
8       return *(int*)p - *(int*)q;
9   }
10
11  int a[10];
12  /* --- */
13
14  qsort(a, sizeof(a) / sizeof(a[0]), sizeof(a[0]),compare_int)
```

8.3. Абстрактный тип данных. Определение применимости реализации к определённым типам и структурам данных, расширение, сужение области применимости.

Абстрактный тип данных — это интерфейс, который определяет тип данных и операции над этим типом.
Тип данных называется абстрактным, потому что интерфейс скрывает детали его представления и реализации.
Возьмём, к примеру, реализацию стека. У нас есть **stack.h** с базовым набором функций: stack_create, stack_destroy, stack_is_empty.
Интерфейс это своего рода контракт:

1) Интерфейс обычно описывает проверяемые ошибки времени выполнения и непроверяемые ошибки времени выполнения и исключения.

2) Реализация не гарантирует обнаружение непроверяемых ошибок времени выполнения. Хороший интерфейс избегает таких ошибок, но должен описать их.

3) Реализация гарантирует обнаружение проверяемых ошибок времени выполнение и информирование клиентского кода.

4) Используются имена функций, которые подходят для многих АТД (create, destroy, is_empty). Если в программе будет использоваться несколько разных АТД, это может привести к конфликту. Поэтому приходится добавлять название АТД в название функций (приписка **stack**).

5) Хотелось бы чтобы стек мог «принимать» данные любого типа без модификации файла **stack.h**.

6) Программа не может создать два стека с данными разного типа.
   Решение – использовать **void*** как тип элемента, НО:
   • элементами могут быть динамически выделяемые объекты, но не данные базовых типов int, double.
   • стек может содержать указатели на что угодно, очень сложно гарантировать правильность.

Про сужение области:

Допустим у вас есть что-то похожее на **printf**. Вы выводите на экран спецификатор `%d`. При этом ваша функция принимает в качестве аргументов **void***. Но так как Вы выводите `%d` ваша область сужается до целых, независимо от того, что Ваша функция принимает универсальный **void***.

Про расширение:

Для вышеуказанного примера, можно спецификатор вынести в `#define`, чтобы пользователь сам определял, что ему нужно.

8.4. Смещение, макрос offsetof. Абстрактные типы данных, построенные на смещениях. Списки в стиле Беркли

Макрос **offsetof** (size_t offsetof(type, member)) — возвращает смещение поля **member** от начала структуры **type**. Смещение элемента не всегда является суммой размеров предыдущих элементов так как размеры полей, составляющих структуру, могут значительно изменяться в зависимости от реализаций, а компиляторы могут добавлять различное количество дополнительных байт между полями.

Если **member** не выровнен по границе байта (т.е., если это битовое поле), то компилятор вернёт ошибку.

Списки в стиле Беркли. АТД на смещении. Является встраиваемым циклическим двусвязным списком, в основе которого лежит 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 самих данных

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

Структуру **struct list_head** можно поместить в любом месте в определении структуры. **struct list_head** может иметь любое имя. В структуре может быть несколько полей типа **struct list_head**

Для обращения к списку нужно посчитать количество бит до структуры **list_head** `int offset = (int) (&((struct s*) 0)->i)`

Код функций и макросов:

```
static inline void list_add(struct list_head *new,

struct list_head *head)
{
__list_add(new, head, head->next);
}

static inline void list_add_tail(struct list_head *new,

struct list_head *head)
{
__list_add(new, head->prev, head);
}

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

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

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

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

```
29
30 #define list_entry(ptr, type, member) \
31 container_of(ptr, type, member)
32
33 #define container_of(ptr, type, field_name) ( \
34 (type *) ((char *) (ptr) - offsetof(type, field_name)))
35
36 #define offsetof(TYPE, MEMBER) \
37 ((size_t) &((TYPE *)0)->MEMBER)
```

9. Библиотеки

ОТЛИЧАЮТСЯ ОТ ПРИЛОЖЕНИЙ ОТСУТСТВИЕМ ТОЧКИ ВХОДА!!!

9.1. Статические библиотеки.

Связываются с программой в момент компоновки. Код библиотеки помещается в исполняемый файл.
«+»
• Исполняемый файл включает в себя все необходимое.
• Не возникает проблем с использованием не той версии библиотеки.
«-»
• ‘Размер‘.
• При обновлении библиотеки программу нужно пересобрать.
Сборка библиотеки:

1) Компиляция:
   `gcc -std=c99 -Wall -Werror -c arr_lib.c`
   Unix:
   `gcc -std=c99 -Wall -Werror -c -fPIC arr_lib.c`

2) Упаковка:
   `ar rc libarr.a arr_lib.o`

3) Индексирование:
   `ranlib libarr.a`

4) Сборка приложения:
   `gcc -std=c99 -Wall -Werror main.c libarr.a -o test.exe`
   `gcc -std=c99 -Wall -Werror main.c -L.-larr -o test.exe`

9.2. Динамические библиотеки

Подпрограммы из библиотеки загружаются в приложение во время выполнения. Код библиотеки не помещается в исполняемый файл.
«+»
• Несколько программ могут «разделять» одну библиотеку.
• Меньший размер приложения (по сравнению с приложением со статической библиотекой).
• Средство реализации плагинов.
• Модернизация библиотеки не требует перекомпиляции программы.
• Могут использовать программы на разных языках.
«-»
• Требуется наличие библиотеки на компьютере.
• Версионность библиотек.

Использование динамической библиотеки (динамическая компоновка):

1) Компиляция:
   `gcc -std=c99 -Wall -Werror -c arr_lib.c`

Unix:
```
gcc -std=c99 -Wall -Werror -c -fPIC arr_lib.c
```

2) Компоновка:
```
gcc -shared arr_lib.o -Wl,-subsystem,windows -o arr.dll
```
Unix:
```
gcc -shared -o libarrf.so list_lib.o
```

3) Сборка приложения:
```
gcc -std=c99 -Wall -Werror -c main.c
gcc main.o -L.-larr -o test.exe
```
Unix:
```
gcc -o test.exe main.c -L.-larrf -Wl,-rpath,.
```

Использование динамической библиотеки (динамическая загрузка):

1) Компиляция:
```
gcc -std=c99 -Wall -Werror -c arr_lib.c
```
Unix:
```
gcc -std=c99 -Wall -Werror -c -fPIC arr_lib.c
```

2) Компоновка:
```
gcc -shared arr_lib.o -Wl,-subsystem,windows -o arr.dll
```
Unix:
```
gcc -shared -o libarrf.so list_lib.o
```

3) Сборка приложения:
```
gcc -std=c99 -Wall -Werror main.c -o test.exe
```
Unix:
```
gcc main.o -o dynamic_load.exe -ldl
```

9.3. Особенности взаимодействия библиотек и внешних ресурсов ОБЯЗАТЕЛЬНО ДОЛЖНО СОБЛЮДАТЬСЯ ABI

Внешние ресурсы по отношению к программе: динамическая память и файлы.
Между загрузкой до выгрузки нужно всегда освободить динамические ресурсы.
Если с помощью библиотеки были выделены динамические ресурсы (или был открыт ФАЙЛ), то они должны быть также освобождены библиотекой. Потому что нам **НЕ** известно, будут ли у нас конфликты при использовании разных менеджеров памяти. (Если пишем ‘allocate‘ в библиотеке, так же пишем ‘free‘)

Fun fact: в отличии от **Windows** на **Linux** доступны все функции библиотеки, а не только специально "выгруженные".

10. Структуры даных, построенные на узлах

Тех кто вовремя закрыл ТИСД напрягать по спискам и очередям сильно не будут. Однако важно знать базовые операции добавления/удаления/вставки и ТРАСИРОВКУ с СОРТИРОВКОЙ.

10.1. Узел как структура данных.

СД, содержащая полезную нагрузку и указатель куда-нибудь (необязательно на следующий такой же узел).

```
1 typedef struct Node
2 {
3     int data;
4     struct Node *next;
5 } node_t;
```

Размер такой СД не фиксирован и упирается лишь в лимиты железа.

10.2. Односвязный линейный список.

Описан выше. Хранит полезную нагрузку и адрес на следующий узел.
Первый узел — голова (head), последний узел — хвост (tail) указывает в NULL.
По такому списку можно перемещаться только в одном направлении: из головы в хвост.
При удалении или вставке узла в список адрес отсальных узлов не меняется

Основные операции:

1) Добавление элемента в список.
2) Поиск элемента в списке.
3) Удаление элемента.
4) Обработка всех элементов списка (печать, сортировка и т.д.).

Добавление элемента в список:

```
1 node_t *add(node_t *list, int n)
2 {
3     node_t *tmp = malloc(sizeof(node_t));
4     tmp->data = n;
5     tmp->next = NULL;
6 
7     list->next = tmp;
8     return list;
9 };
```

Удаление элемента из списка:

```
1 node_t *delete(node_t *list)
2 {
3     node_t *tmp;
4     tmp = list->next;
5     free(list);
```

```
6       return tmp;
7   }
```

Любая обработка целого списка происходит путем обхода всех его элементов:

```
1   void traversal(node_t *list)
2   {
3       node_t *current = list;
4       for ( ; current; current = current->next);
5   }
```

10.3. Односвязный кольцевой список.

Всё тоже самое, только хвост указывает на голову, а не в NULL.

10.4. Двусвязный линейный список.

Представляется следующим образом:

```
1   typedef struct Node
2   {
3       int field;
4       struct Node *next;
5       struct Node *prev;
6   } node_t;
```

Теперь по списку можно перемещаться в оба направления.
Инициализация списка:

```
1   node_t *init(int n)
2   {
3       node_t *list;
4
5       list = malloc(sizeof(node_t));
6       list->field = a;
7       list->next = NULL;
8       list->prev = NULL;
9       return(list);
10  }
```

Добавление узла:

```
1   node_t *addelem(node_t *list, int number)
2   {
3       node_t *temp, *p;
4       temp = malloc(sizeof(node_t));
5       p = list->next;
6       list->next = temp;
7       temp->field = number;
8       temp->next = p;
9       temp->prev = list;
10      if (p != NULL)
11      p->prev = temp;
```

```
12      return(temp);
13  }
```

Удаление узла:

```
1   node_t deletelem(node_t *list)
2   {
3       node_t *prev, *next;
4       prev = list->prev;
5       next = list->next;
6       if (prev != NULL)
7       prev->next = list->next;
8       if (next != NULL)
9       next->prev = list->prev;
10      free(list);
11      return(prev);
12  }
```

10.5. Двусвязный кольцевой список.

Тоже самое, только предыдущий элемент с головы указывает в хвост, а следующий с хвоста указывает в голову.

Алгоритм сортировки (успешно принят)

```
1  void list_bubble_sort(node_t **head)
2  {
3      int done = 0;
4
5      if (*head == NULL || (*head)->next == NULL)
6          return;
7
8      while (!done)
9      {
10         node_t **pv = head;
11         node_t *cur = *head;
12         node_t *nxt = (*head)->next;
13
14         done = 1;
15
16         while (nxt)
17         {
18             if (cur->data > nxt->data)
19             {
20                 cur->next = nxt->next;
21                 nxt->next = cur;
22                 *pv = nxt;
23
24                 done = 0;
25             }
26             pv = &cur->next;
27             cur = nxt;
28             nxt = nxt->next;
29         }
30     }
31 }
```

10.6. Реализация стека и очереди на базе списков.

ВАЖНО ЗНАТЬ ДЕТАЛИ РЕАЛИЗАЦИИ СТЕКА И ОЧЕРЕДИ НА МАССИВЕ!!!

Для стека:

Пишутся стандартные **push** и **pop** (не забудте проверить пуст/полон ли стек).

Для очереди

Аналогично стеку **push** и **pop**.
Вариант реализации очереди: создаются два стека **s1** и **s2**. В стек **s1** добавляются элементы. Потом из стека **s1** извлекаются элементы и помещаются в в **s2**. При этом стек **s1** освобождается. Реализовать так же функцию, которая проверяет, пуст ли стек или полон.

Асимптотика очереди (отчёт ТИСД)

Размер заранее известен? — массив (лучше скорость), иначе — список (лучше память).

11. Остальное

11.1. Встраиваемые функции. Ключевое слово inline.

Рекомендация компилятору копировать код функции непосредственно в код программы по месту вызова, а не создавать функцию в памяти. Он может подсчитать встраивание нецелесообразным и просто проигнорировать модификатор inline и трактовать функцию как обычную. Существует __attribute__((always_inline)), чтобы очень явно намекнуть на счёт встраивомости.

• **inline** стирает область видимости (допускается существование функций с одинаковым именем (плохая практика))
• **inline** функция может быть рекурсивной (вырастает сильно объём кода).
Плюсы и минусы:

1) Внедрение кода функции в код программы поможет избежать использования лишних инструкций (связанных с вызовом функции и возвратом из нее), соответственно повысит скорость, однако слишком частое использование встраиваемых функций, к тому же больших, приведет к разрастанию кода.

2) Тоже самое касательно размера исполняемого файла. Может увеличиться из-за частых встраиваний, а может наоборот уменьшиться за счёт грамотной оптимизации. (Маленькие функции проще встраивать, а не строить для них отдельно конструкции входа/выхода).

11.2. Побитовые операции.

Нужно добавить примеры на целых числах и работает ли с действительными (нет?)

Операция	Название	Нотация	Класс	Приоритет	Ассоциат.
~ (унар.)	Побитовое «НЕ»	~x	Префиксные	15	Справа налево
<<	Сдвиг влево	x << y	Инфиксные	11	Слева направо
>>	Сдвиг вправо	x >> y			
&	Побитовое «И»	x & y	Инфиксные	8	Слева направо
^	Побитовое исключающее «ИЛИ»	x ^ y	Инфиксные	7	Слева направо
|	Побитовое «ИЛИ»	x | y	Инфиксные	6	Слева направо

11.3. Операции сдвига.

Можно сказать, что операции умножения/деления происходит быстрее (целые числа).

`<<=`	Присваивание со сдвигом влево	`x <<= y`	Инфиксные	2	Справа налево
`>>=`	Присваивание со сдвигом вправо	`x >>= y`			
`&=`	Присваивание с побитовым «И»	`x &= y`			
`^=`	Присваивание с побитовым исключающим «ИЛИ»	`x ^= y`			
`|=`	Присваивание с побитовым «ИЛИ»	`x |= y`			

11.4. Неопределённое поведение.

Точки следования (sequence points) - это некие точки в программе, где состояние реальной программы полностью соответствует состоянию абстрактной машины, описанной в Стандарте. С помощью точек следования стандарт объясняет, что может, а чего не может делать компилятор и что нам нужно сделать, чтобы написать корректный код. В каждой точке следования все побочные эффекты кода, который уже выполнен, уже случились, а побочные эффекты для кода, который еще не был выполнен, еще не случились. Не вдаваясь в детали: запись значения в переменную при присваивании есть пример побочного эффекта.

Классика: `i = i++ + ++i` или выход за границы массива.

11.5. Поведение, определяемое реализацией.

Это когда программа корректно работает при определённых условиях (версия компилятора, ОС, железо и т.д). Не считается ошибкой, как в случае UB Пример: `int *a = malloc(0)`

11.6. Директивы препроцессора.

ВАЖНО УКАЗАТЬ ПРО ВОЗМОЖНОСТЬ () ПРИНИМАТЬ АРГУМЕНТЫ (СОЗДАНИЕ МАКРОСОВ)

1) `#include <файл>` — включает целиком файл.
   Если имя файла указано в <> — поиск файла будет произведён в системных каталогах, если в "" — сначала в текущем каталоге.

2) `#define <идентификатор> <текст>`
   Используется для задания констант, ключевых слов, операторов и выражений, используемых в программе. Если нет <текста>, значит поднимает флаг.

3) `#undef` — отменяет определение, введённое директивой `#define`

4) `#ifndef, #ifdef, #else, #elif` — директивы условной компиляции.

5) `#pragma` — "просьба"к компилятору выполнить определенные действия. Чаще всего:
`#pragma pack` - выравнивает пол внутри записи
`#pragma once` - контролирует за тем, чтобы конкретный файл при компиляции подключался строго один раз.

12. Элементы событийного программировани (доп вопросы).

12.1. Мастер-цикл.

Цикл, ожидающий поступление события. Данный цикл находится на самом высоком уровне потока управления в рамках программы.

12.2. Графический интерфейс на основе событийной парадигмы. Основные компоненты графического интерфейса.

Виджеты - это кнопки, формы, метки и т.д
(добавить что являеттся мастером-контроллером)
К элементам графического интерфейса можно прикреплять события. Действие на кнопку и.т.д. Каждому элементу присуще состояние. Инициализирует событие компонент действия. Основные компоненты

1) Текстовое поле (uiEntry)
   - Вывод/Ввод: Да/Да
   - Действие: Редко

2) Метка (uiLabel)
   - Вывод/Ввод: Да/Нет
   - Действие: Редко

3) Многострочный текст (uiMuiltilineEntry)
   - Вывод/Ввод: Да/Да
   - Действие: Редко

4) Таблица (uiTable)
   - Вывод/Ввод: Да/Косвенно
   - Действие: Часто

5) Прогресс-бар (uiProgressBar)
   - Вывод/Ввод: Да/Нет
   - Действие: Нет

6) Полоса прокрутки (uiScrollBar)
   - Вывод/Ввод: Редко/Да
   - Действие: Часто

7) Флажок (uiCheckBox)
   - Вывод/Ввод: Нет/Да
   - Действие: Часто

8) Переключатель (uiRadioButtom)
   - Вывод/Ввод: Нет/Да
   - Действие: Часто

9) Кнопка (uiButton)
   - Вывод/Ввод: Нет/Да
   - Действие: Часто

10) Графический блок (uiArea)
    - Вывод/Ввод: Да/Нет(Редко)
    - Действие: Да (Редко)

12.3. Псевдопараллелизм. Таймеры.

Есть квант времени (минимальная неделимая единица). Идёт разбитие действия на некие кванты времени. Несколько событий разбиваются на равные части, и сначала выполняется "первая часть"одного события, потом "первая часть"второго и т.д, пока не выполнятся все части.

Таймер выполняет прикреплённое к нему событие с определённой задержкой. У таймера приоритет меньше чем у мастер-цикла (программы), по этому если программой выполняется какое-то событие таймер не сможет запустить в обработку прикреплённое к нему событие.

13. Известные билеты

Билет №4

1) Одномерные массивы указателей на одномерные массивы с простыми или статическими структурами.
2) Добавление элементов и удаление двусвязного списка (CDIO).
3) Сортировка двусвязного списка (через перестановку адресов узлов).

Билет №5

1) Запись с динамическим полем. Рекурсивная деструкция.
2) Добавление элементов и удаление односвязного списка (CDIO).
3) Сортировка односвязного списка (через перестановку адресов узлов).

Билет №12

1) Функции с переменным числом аргументов. Реализация на стеке.
2) Транспонировать прямоугольную матрицу (хранится как обычно).
3) Возвести квадратную матрицу в неотрицательную степень.

Билет №16

1) Макрос **offsetof**. Списки Беркли. АТД.
2) Создать матрицу на основе...
3) Удалить столбец по индексу.

Билет №28

1) Реализация очереди на списке.
2) Транспонировать прямоугольную матрицу (хранится как обычно).
3) Возвести квадратную матрицу в неотрицательную степень.

Билет №30

1) Односвязные кольцевые списки.
2) Создать матрицу на основе массива указателей на столбцы.
3) Удалить строку по индексу.