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

– *понятие «указатель»;*

Указатель – это объект, содержащий адрес объекта или функции, либо выражение, обозначающее адрес объекта или функции.

– *void*, особенности операций с ним;*

Указатель типа void используется, если тип объекта неизвестен. Он:
1) позволяет передавать в функцию указатель на объект любого типа;
2) полезен для ссылки на произвольный участок памяти, независимо от размещенных там объектов.

Особенности операций с ним:
1)Указателя типа void* нельзя разыменовывать.
2) К указателям типа void* не применима адресная арифметика.

– *приведение указателей разных типов к void* и обратно;*

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

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

– *определение указателя на функцию;*

возвращаемый_тип (*имя_указателя_на_функцию) (параметры);

```
double(*func)(double);
```

– *присваивание значения указателю на функцию;*

```
int moo(int a);
int (*fcnPtr3)(int) = moo;
// ок, т.к. возвращаемый тип и аргументы совпадают
```

– *вызов функции по указателю;*

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

– использование указателей на функции.

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

2. Динамические одномерные массивы.

– Функции для выделения и освобождения памяти (malloc, calloc, realloc, free). Порядок работы и особенности использования этих функций.

Для выделения памяти необходимо вызвать одну из трех функций (C99 7.20.3), объявленных в заголовочном файле stdlib.h:

1) malloc (выделяет блок памяти и не инициализирует его);

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

2) calloc (выделяет блок памяти и заполнят его нулями);

```
void* calloc(size_t nmemb, size_t size);
```

3) realloc (перевыделяет предварительно выделенный блок памяти).

```
void* realloc(void *ptr, size_t size);
```

Функция *free (C99 7.20.3.2)* освобождает (делает возможным повторное использование) ранее выделенный блок памяти, на который указывает ptr.

– Способы возврата динамического массива из функции.

1) Возврат указателя на начало массива из функции при помощи return

```
double* get_array_1(int *n)
{
    *n = 0;
    int nmemb = 5; //определить кол-во элементов
    double *p = NULL; // выделить память
    if (nmemb)
    {
        p = malloc(nmemb * sizeof(double));
        if (p)
            *n = nmemb;
    }
    return p;
    // Возвращает NULL, если произошла ошибка
}
```

```
double *p_1;
int n_1;

p_1 = get_array_1(&n_1);

free(p_1);
```

```
double *p_2;
int n_2, rc;

rc = get_array_2(&p_2, &n_2);

free(p_2);
```

2) Передать в функцию адрес указателя на начало массива, вернуть код ошибки при помощи return.

```
int get_array_2(double **data, int *n)
{
    int rc = 0;
    *n = 0;
    *data = NULL;
    int nmemb = 5; // определить кол-во элементов
    if (!rc && nmemb) // выделить память
    {
        *data = malloc(nmemb * sizeof(double));
        if (*data)
            *n = nmemb;
        else
            rc = -1; // ошибка выделения памяти
    }
    return rc;
}
```

– Типичные ошибки при работе с динамической памятью (утечка памяти, «дикий» указатель, двойное освобождение).

1) Утечка памяти - выделили, но не освободили.
2) wild pointer - пытаемся разыменовать указатель, память под который не была выделена, либо в него не был записан никакой адрес, либо память, на которую он указывал была освобождена.
3) Double free.

3. Указатели и многомерные статические массивы.

– концепция многомерного массива как «массива массивов»;

– определение многомерных массивов;

Количество размерностей массива практически не ограничено.

int a[3][2];

Компилятор Си располагает строки матрицы a в памяти одну за другой вплотную друг к другу.

– инициализация многомерных массивов;

```
int a[3][3] =
{
    {1, 2, 3},
    {4, 5}
};
```

```
int d[][2] = { {1, 2} };
// c99
int c[2][2] = {[0][0] = 1, [1][1] = 1};
```

– ***«слои», составляющие многомерные массивы;***

«Компоненты многомерного массива»

```
int a[2][3][5];
```

a – массив из двух элементов типа “int [3][5]”

```
int (*p)[3][5] = a;
```

a[i] – массив из трех элементов типа “int [5]” (i ∈[0, 1])

```
int (*q)[5] = a[i];
```

a[i][j] – массив из пяти элементов типа “int” (i ∈[0, 1], j ∈ [0, 1, 2])

```
int *r = a[i][j];
```

a[i][j][k] – элемент типа “int” (i ∈ [0, 1], j ∈ [0, 1, 2], k ∈ [0, 1, 2, 3, 4])

```
int s = a[i][j][k];
```

– ***обработка многомерных массивов с помощью указателей;***

```
// адресная арифметика
*(*(a + i) + j)
// различные комбинации
(*(a + i))[j]
*(a[i] + j)
*(&a[0][0] + 4 * i + j)
```

Иногда удобно многомерный массив рассматривать как одномерный.

```
#define N 2
#define M 5
...
int a[N][M];
int *p;
...
for (p = &a[0][0]; p <= &a[N-1][M-1]; p++)
    *p = 0;
```

1)Обработка строки матрицы (обнуление i-ой строки)

```
// указатель на начало i-ой строки
int *p = &a[i][0];
&a[i][0] => &(*(a[i] + 0)) => &(*a[i]) => a[i]
```

Т.е. выражение a[i] – это адрес начала i-ой строки.

```
// обнуление i-ой строки
for (p = a[i]; p < a[i] + M; p++)
    *p = 0;
```

2)Обработка столбца матрицы (обнуление j-го столбца)

```
// указатель на строку (строка – это массив из M элементов)
int (*q)[M];
```

Выражение q++ смещает указатель на следующую строку

Выражение (*q)[j] возвращает значение в j-ом столбце строки, на которую указывает q.

```
for (q = a; q < a + N; q++)
    (*q)[j] = 0;
```

– ***передача многомерных массивов в функцию;***

Пусть определена матрица int a[N][M];

Для ее обработки могут быть использованы функции со следующими прототипами:

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

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

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

– ***const и многомерные массивы.***

Формальное объяснение

Согласно C99 выражение T (*p)[N] не преобразуется неявно в T const (*p)[N].

Способы борьбы: 1)не использовать const; 2) использовать явное преобразование типа (**f((const int (*)[M]) a, 2, 3);**)

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

В C99 внутри функции или блока можно задавать размер массива с помощью выражений, содержащих переменные.

```
#include <stdio.h>
int main(void)
{
    int n;
    printf("n: ");
    scanf("%d", &n);
    int a[n];
    for (int i = 0; i < n; i++)
        a[i] = i;
    return 0;
```

```
}
```

- Длина такого массива вычисляется во время выполнения программы, а не во время компиляции.
- Память под элементы массива выделяется на стеке.
- Массивы переменного размера нельзя инициализировать при определении.
- Массивы переменной длины могут быть многомерными.
- Адресная арифметика справедлива для массивов переменной длины.
- Массивы переменной длины облегчают описание заголовков функций, которые обрабатывают массивы.

5. Динамические многомерные массивы.

Способы выделения памяти для динамических матриц: идеи, реализации, анализ преимуществ и недостатков.

1) Матрица как одномерный массив

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

Преимущества:

-Простота выделения и освобождения памяти.

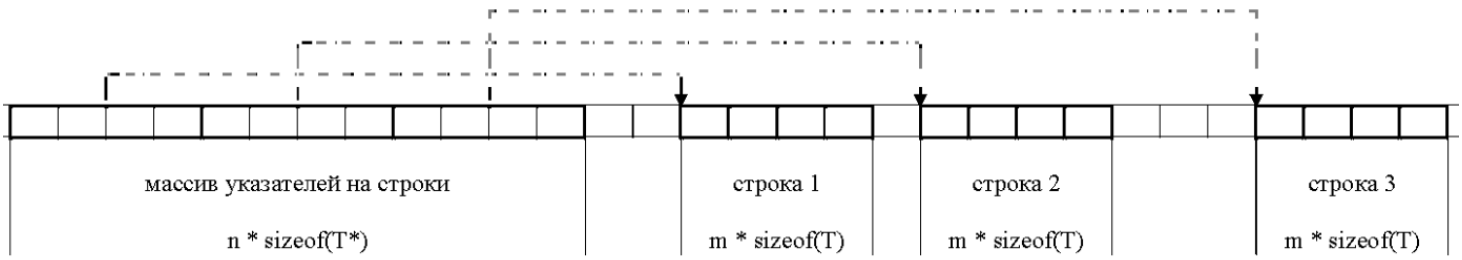
-Возможность использовать как одномерный массив.

Недостатки:

-Средство проверки работы с памятью (СПРП) не может отследить выход за пределы строки.

-Нужно писать i * m + j, где m – число столбцов.

2)Матрица как массив указателей



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

    return data;
}
```

Преимущества:

-Возможность обмена строки через обмен указателей.

-СПРП может отследить выход за пределы строки.

Недостатки:

-Сложность выделения и освобождения памяти.

-Память под матрицу "не лежит" одним куском.

3) Объединение подходов(1)

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

    free(ptrs);
}
```

Преимущества:

-Относительная простота выделения и освобождения памяти.

-Возможность использовать как одномерный массив.

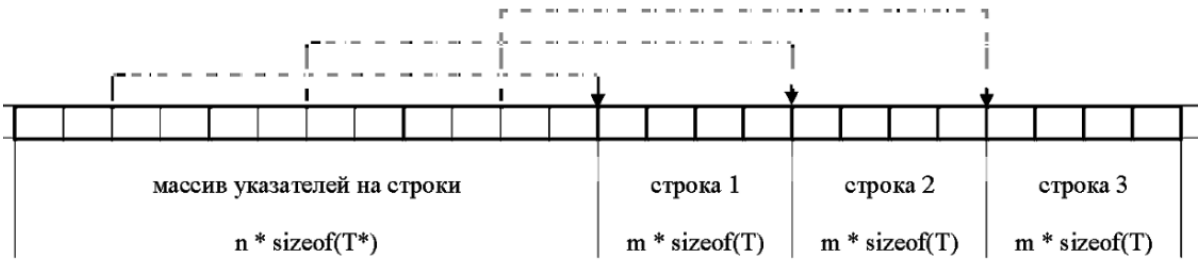
-Перестановка строк через обмен указателей.

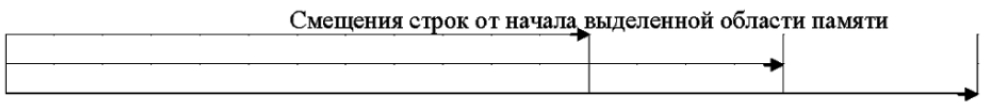
Недостатки:

-Относительная сложность начальной инициализации.

-СПРП не может отследить выход за пределы строки.

4) Объединение подходов(2)





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

Преимущества:

-Простота выделения и освобождения памяти.

-Возможность использовать как одномерный массив.

-Перестановка строк через обмен указателей.

Недостатки:

-Сложность начальной инициализации.

-СПРП не может отследить выход за пределы строки.

6. Строки. План ответа:

– ***понятия «строка» и «строковый литерал»;***

Строка – это последовательность символов, заканчивающаяся и включающая первый нулевой символ (англ., null character ‘\0’ (символ с кодом 0)).

Строковый литерал – последовательность символов, заключенных в двойные кавычки.

```
// массив символов
char str_arr[] = "June";
// указатель на строковый литерал
char *str_ptr = "June";
```

– ***определение переменной-строки, инициализация строк;***

```
#define STR_LEN 80
...
char str[STR_LEN+1];    // !
```

Строка - это массив, поэтому к ней применима операция индексации.

Инициализация:

char str_1[] = {‘J’,‘u’,‘n’, ‘e’, ‘\0’};

char str_2[] = “June”;

char str_3[5] = “June”;

– ***ввод/вывод строк (scanf, gets, fgets, printf, puts);***

printf(«%s», str); <=> puts(s);

scanf("%s", str);

// Через scanf нельзя ввести строку с пробелами!

gets(str);

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

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

char *fgets(char *s, int size, FILE *stream);

Прекращает ввод когда (любое из)

-прочитан символ ‘\n’;

-достигнут конец файл;

-прочитано size-1 символов.

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

```
char* strcpy(char *s1, const char *s2);
char* strncpy(char *s1, const char *s2, size_t n);
size_t strlen(const char *s);
char* strcat(char *s1, const char *s2); //Объединение строк
char* strncat(char *s1, const char *s2, size_t n);
int strcmp(const char *s1,const char *s2);
int strncmp(const char *s1,const char *s2, size_t n);
char* strdup(const char *s); //выделяет память, копирует
char* strndup(const char *s, size_t count); // строку s
int sprintf(char *s, const char *format, …);
int snprintf(char *s, size_t n, const char *format, …);
char* strtok(char *string, const char *delim);
long int atol(const char* str); //str->int
long int strtol(const char* string,char** endptr, int
basis);
```

– ***идиомы обработки строк.***

Выход за пределы выделенного буфера строки – крайне нежелательный эффект, не отслеживается компилятором и ложится на плечи программиста. Терминирующий нуль при преобразованиях строк необходимо сдвигать, а также обеспечивать его нахождение в буфере. Пробег по строке с помощью указателей также должен быть аккуратным, использование вне пределов буфера чревато последствиями.

7.Область видимости, время жизни и связывание.

– ***понятия «область видимости», «время жизни» и «связывание»;***

Область видимости (scope) имени – это часть текста программы, в пределах которой имя может быть использовано(блок;файл;функция;прототип функции)

1)Область видимости: блок

блоком считается последовательность объявлений, определений и операторов, заключенная в фигурные скобки. (составной оператор; определение функции.) Переменная, определенная внутри блока, имеет область видимости в пределах блока. Формальные параметры функции имеют в качестве области видимости блок, составляющий тело функции.

2)Область видимости: файл

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

3)Область видимости: функция

Метки - это единственные идентификаторы, область действия которых - функция.

Метки видны из любого места функции, в которой они описаны.

В пределах функции имена меток должны быть уникальными.

4)Область видимости: прототип функции

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

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

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

```
int f(int, double);         // ок
int f(int i, double i);     // ошибка компиляции
```

Время жизни (storage duration) – это интервал времени выполнения программы, в течение которого «программный объект» существует.(глоб-е, лок-е, дин-е)

1)Глобальное время жизни

Если «программный объект» имеет глобальное время жизни, он существует на протяжении выполнения всей программы. Примерами таких «программных объектов» могут быть функции и переменные, определенные вне каких либо функций.

2)Локальное время жизни

Локальным временем жизни обладают «программные объекты», область видимости которых ограничена блоком.Такие объекты создаются при каждом входе в блок, где они определяются. Они уничтожаются при выходе из этого «родительского» блока.Примерами таких переменных являются локальные переменные и параметры функции.

3) Динамическое время жизни

Время жизни «выделенных» объектов длится с момента выделения памяти и заканчивается в момент ее освобождения. В Си нет переменных, обладающих динамическим временем жизни. Динамическое выделение выполняется программистом «вручную» с помощью соответствующих функций. Единственные способ «добраться» до выделенной динамической памяти – использование указателей.

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

1)Внешнее связывание

Имена с внешним связыванием доступны во всей программе. Подобные имена «экспортируются» из объектного файла, создаваемого компилятором.

2)Внутреннее связывание

Имена с внутренним связыванием доступны только в пределах файла, в котором они определены, но могут «разделяться» между всеми функциями этого файла.

3)Без связывания

Имена без связывания принадлежат одной функции и не могут разделяться вообще.

- ***правила перекрытия областей видимости;***

1)Переменные, определенные внутри некоторого блока, будут доступны из всех блоков, вложенных в данный. 2)Возможно определить в одном из вложенных блоков переменную с именем, совпадающим с именем одной из "внешних" переменных.

- ***размещение «объектов» в памяти в зависимости от времени жизни;***
- ***влияние связывания на объектный и/или исполняемый файл..***

8.Журналирование

Назначение, идеи реализации (глобальная файловая переменная; статическая файловая переменная; функции с переменным числом параметров).

Журналирование – процесс записи информации о происходящих с каким-то объектом (или в рамках какого-то процесса) событиях в журнал (например, в файл). Этот процесс часто называют также аудитом.

Идеи реализации:

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 собственные функции, где переменная-указатель на файл с журналом может быть как глобальной, так и статической переменной

Журналирование можно реализовать при помощи ф-й с переменным числом параметров

```
// log.c
#include <stdio.h>
#include <stdarg.h>
static FILE* flog;
int log_init(const char*name)
{
     flog = fopen(name, "w");
     if(!flog)
          return 1;
     return 0;
}
void 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);
}
```

```
// log.h
#ifndef __LOG__H__
#define __LOG__H__
#include <stdio.h>
int log_init(const char* name);
void log_message(const char*
format, ...);
void log_close(void);
#endif // __LOG__H__
```

9. Классы памяти

Управлять временем жизни, областью видимости и связыванием переменной (до определенной степени) можно с помощью так называемых классов памяти. В языке Си существует четыре класса памяти:

-auto;

Применим только к переменным, определенным в блоке.

```
int main(void)
{
    auto int i;
```

Переменная, принадлежащая к классу auto, имеет локальное время жизни, видимость в пределах блока, не имеет связывания.

По умолчанию любая переменная, объявленная в блоке или в заголовке функции, относится к классу автоматической памяти.

-static;

Класс памяти static может использоваться с любыми переменными независимо от места их расположения.

-Для переменной вне какого-либо блока, static изменяет связывание этой переменной на внутреннее.

-Для переменной в блоке, static изменяет время жизни с автоматического на глобальное.

1)Статическая переменная, определенная вне какого-либо блока, имеет глобальное время жизни, область видимости в пределах файла и внутреннее связывание.

```
static int i;
void f1(void)
{
    i = 1;
}
void f2(void)
{
    i = 5;
}
```

Этот класс памяти скрывает переменную в файле, в котором она определена

2)*Статическая переменная, определенная в блоке, имеет *глобальное время жизни, область видимости в пределах блока и отсутствие связывания.

```
void f(void)
{
```

```
    static int j;
    ...
}
```
-Такая переменная сохраняет свое значение после выхода из блока.
-Инициализируется только один раз.
-Если функция вызывается рекурсивно, это порождает новый набор локальных переменных, в то время как статическая переменная разделяется между всеми вызовами.

-extern;

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

```
extern int i;
```

Глобальное время жизни, файловая область видимости, связывание непонятное

```
{
    extern int i;
    ...
```

Глобальное время жизни, видимость в блоке, связывание непонятное
Связывание определяется по определению переменной

- Объявлений (extern int number;) может быть сколько угодно.
- Определение (int number;) должно быть только одно.
- Объявления и определение должны быть одинакового типа.

-register.

Использование класса памяти register – просьба (!) к компилятору разместить переменную не в памяти, а в регистре процессора.
-Используется только для переменных, определенных в блоке.
-Задает локальное время жизни, видимость в блоке и отсутствие связывания.
-Обычно не используется.
К переменным с классом памяти register нельзя применять операцию получения адреса &.

10.Стек и куча.

Стек — это область оперативной памяти, которая создаётся для каждого потока. Он работает в порядке LIFO (Last In, First Out).
Куча — это хранилище памяти, также расположенное в ОЗУ, которое допускает динамическое выделение памяти и не работает по принципу стека: это просто склад для ваших переменных.

– ***автоматическая память: использование и реализация;***

Для хранения локальных переменных используется так называемая автоматическая память. См локальные переменные в 8

Организация автоматической памяти

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

1. **Вызов main**
2. Создание e
3. **Вызов f_2**
4. Создание c
5. Создание d
6. **Вызов f_1**
7. Создание a
8. Создание b
9. **Завершение f_1**
10. Разрушение b
11. Разрушение a
12. **Завершение f_2**
13. Разрушение d
14. Разрушение c
15. **Завершение main**
16. Разрушение e

3

– ***использование аппаратного стека (вызов функции, возврат управления из функции, передача параметров, локальные переменные, кадр стека);***

Аппаратный стек используется для:

1)вызова функции(call name)

-поместить в стек адрес команды, следующей за командой call

-передать управление по адресу метки name

2)возврата из функции (ret)

-извлечь из стека адрес возврата address

-передать управление на адрес address

3) передачи параметров в функцию

соглашение о вызове:

- расположение входных данных;
- порядок передачи параметров;

- какая из сторон очищает стек;
- etc

cdecl:

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

4) выделения и освобождения памяти под локальные переменные

– ***ошибки при использовании автоматической памяти;***

возврат указателя на локальную переменную; переполнение буфера

– ***динамическая память: использование и реализация;***

- При запуске процесса ОС выделяет память для размещения кучи.
- Куча представляет собой непрерывную область памяти, поделённую на занятые и свободные области (блоки) различного размера.
- Информация о свободных и занятых областях кучи обычно храниться в списках различных форматов.

Функция malloc выполняет примерно следующие действия:

◦ просматривает список занятых/свободных областей памяти, размещённых в куче, в поисках свободной области подходящего размера;
◦ если область имеет точно такой размер, как запрашивается, добавляет найденную область в список занятых областей и возвращает указатель на начало области памяти;
◦ если область имеет больший размер, она делится на части, одна из которых будет занята (выделена), а другая останется в списке свободных областей;
◦ если область не удается найти, у ОС запрашивается очередной большой фрагмент памяти, который подключается к списку, и процесс поиска свободной области продолжается;
◦ если по тем или иным причинам выделить память не удалось, сообщает об ошибке (например, malloc возвращает NULL).

Функция free выполняет примерно следующие действия:

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

11.Функции с переменным числом параметров.

Идея реализации, использование стандартной библиотеки

```
int f(...);
```

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

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

```
int f(int k, …);
```

```
#include <stdio.h>

double avg(int n, ...)
{
    int *p_i = &n;
    double *p_d =
                (double*) (p_i+1);
    double sum = 0.0;

    if (!n)
        return 0;

    for (int i = 0; i < n;
                        i++, p_d++)
        sum += *p_d;

    return sum / n;
}
```

```
double avg(double a, ...)
{
    int n = 0;
    double *p_d = &a;
    double sum = 0.0;

    while (*p_d)
    {
        sum += *p_d;
        n++;

        p_d++;
    }

    if (!n)
        return 0;

    return sum / n;
}
```

stdarg.h

- va_list
- void va_start(va_list argptr, last_param)
- type va_arg(va_list argptr, type)
- void va_end(va_list argptr)

```
#include <stdarg.h>
#include <stdio.h>
double avg(double a, ...)
{
    va_list vl;
    int n = 0;
    double num, sum = 0.0;

    va_start(vl, a);
    num = a;

    while (num)
    {
        sum += num;
        n++;
        num = va_arg(vl, double);
    }

    va_end(vl);

    if(!n)
        return 0;
    return sum / n;
}
```

12.Структуры.

– ***понятие «структура»;***

Структура представляет собой одну или несколько переменных (возможно разного типа), которые объединены под одним именем. Структуры помогают в организации сложных данных, потому что позволяют описывать множество логически связанных между собой отдельных элементов как единое целое.

```
struct <имя> // тег
{
    <тип_1> <имя_1>;
    ...                 //поле
    <тип_N> <имя_N>;
};
```

– ***определение структурного типа;***

Раздельное и совместное определения типа переменных.

– ***структура и ее компоненты (тег, поле);***

Имя, которое располагается за ключевым словом struct, называется ***тегом*** *структуры.*

◦ Используется для краткого обозначения той части объявления, которая заключена в фигурные скобки.
◦ Тег может быть опущен (безымянный тип)
◦ Тег структуры не распознается без ключевого слова struct. Благодаря этому тег не конфликтует с другими именами в программе.

Перечисленные в структуре переменные называются ***полями*** *структуры.*

◦ Поля структуры располагаются в памяти в порядке описания.
◦ С целью оптимизации доступа компилятор может располагать поля в памяти не одно за другим, а по адресам кратным, например, размеру поля.
◦ Адрес первого поля совпадает с адресом переменной структурного типа.
◦ Поля структуры могут иметь любой тип, кроме типа этой же структуры, но могут быть указателями на него.

– ***определение переменной-структуры, способы инициализации переменной-структуры;***

- Для инициализации переменной структурного типа необходимо указать список значений, заключенный в фигурные скобки.
- Значения в списке должны появляться в том же порядке, что и имена полей структуры.
- Если значений меньше, чем полей структуры, оставшиеся поля инициализируются нулями.

– ***операции над структурами;***

Доступ к полю структуры осуществляется с помощью операции “.”, а если доступ к самой структуре осуществляется по указателю, то с помощью операции “->”.

- Структурные переменные одного типа можно присваивать друг другу
- Структуры нельзя сравнивать с помощью “==” и “!=”.
- Структуры могут передаваться в функцию как параметры и возвращаться из функции в качестве ее значения.

– ***особенности выделения памяти под структурные переменные;***

– ***структуры с полем переменной длины (flexible array member, C99).***

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

- Подобное поле должно быть последним.
- Нельзя создать массив структур с таким полем.

- Структура с таким полем не может использоваться как член в «середине» другой структуры.
- Операция sizeof не учитывает размер этого поля (возможно, за исключением выравнивания).
- Если в этом массиве нет элементов, то обращение к его элементам – неопределенное поведение

13.Объединения.

– ***понятие «объединение»;***

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

– ***определение переменной-объединения, способы инициализации переменной-объединения;***

```
union u_t
{
    int i;
    double d;
};
...
union u_t  u_1 = {1};
// только c99
union u_t  u_2 = { .d = 5.25 };
```

– ***особенности выделения памяти под объединения.***

– ***использование объединений.***

- Экономия места.
- Создание структур данных из разных типов.
- Разный взгляд на одни и те же данные (машинно-зависимо).

14. Динамический расширяемый массив.

– ***функция realloc и особенности ее использования;***

```
void* realloc(void *ptr, size_t size);
```

ptr == NULL && size != 0

Выделение памяти (как malloc)

ptr != NULL && size == 0

Освобождение памяти аналогично free(). Результат можно (но не обязательно!) передать во free().

ptr != NULL && size != 0

Перевыделение памяти.

– *описание типа;*

```
struct dyn_array
{
    int len;
    int allocated;
    int step;
    int *data;
};
#define INIT_SIZE 1
void init_dyn_array(struct dyn_array *d)
{
    d->len = 0;
    d->allocated = 0;
    d->step = 2;
    d->data = NULL;
}
```

– *добавление и удаление элементов.*

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

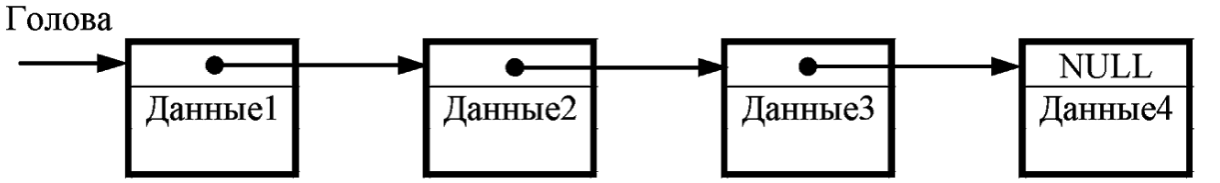
```
int delete(struct dyn_array *d, int index)
{
    if (index < 0 || index >= d->len)
        return -1;

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

    return 0;
}
```

15. Линейный односвязный список. План ответа:

– *описание типа;*




```
struct person
{
    char *name;
    int born_year;

    struct person *next;
};

struct person* create_person(char *name, int born_year)
{
    struct person *pers = malloc(sizeof(struct person));
    if (pers)
    {
        pers->name = name;
        pers->born_year = born_year;
        pers->next = NULL;
    }
    return pers;
}
```

– *основные операции (добавление элемента в начало/конец списка, удаление элемента, обход списка, освобождение памяти из-под списка).*

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

struct person* add_end(struct person *head,
                       struct person *pers)
{
    struct person *cur = head;

    if (!head)
        return pers;
    for ( ; cur->next; cur = cur->next)
        ;
    cur->next = pers;
    return head;
}

struct person* lookup(struct person *head,
                      const char *name)
{
    for ( ; head; head = head->next)
        if (strcmp(head->name, name) == 0)
            return head;

    return NULL;
}

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

head: список
f: указатель на функцию, которая применяется к каждому элементу списка
arg: аргумент функции f

```
void free_all(struct person *head)
{
    struct person *next;

    for ( ; head; head = next)
    {
        next = head->next;
        free(head);
    }
}
```

16. Двоичные деревья поиска. План ответа:

– ***описание типа;***

Дерево - это связный ациклический граф.

Двоичным деревом поиска называют дерево, все вершины которого упорядочены, каждая вершина имеет не более двух потомков (назовём их левым и правым), и все вершины, кроме корня, имеют родителя.

```
struct tree_node
{
    const char *name;

    // меньшие
    struct tree_node *left;
    // большие
    struct tree_node *right;
};

struct tree_node* create_node(const char *name)
{
    struct tree_node *node = malloc(sizeof(struct
tree_node));
    if (node)
    {
        node->name = name;
        node->left = NULL;
        node->right = NULL;
    }

    return node;
}
```

```
struct tree_node* insert(struct tree_node *tree,
                                        struct tree_node *node)
{
    int cmp;

    if (tree == NULL)
        return node;

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

    return tree;
}
```

– *основные операции;*

```
struct tree_node* insert(struct tree_node *tree,
                                        struct tree_node *node)
{
    int cmp;

    if (tree == NULL)
        return node;

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

    return tree;
}
```

– *рекурсивный и нерекурсивный поиск;*

```
struct tree_node* lookup_2(struct tree_node *tree,
                                        const char *name)
{
    int cmp;
    while (tree != NULL)
    {
        cmp = strcmp(name, tree->name);
        if (cmp == 0)
            return tree;
        else if (cmp < 0)
            tree = tree->left;
        else
            tree = tree->right;
    }
    return NULL;
}

struct tree_node* lookup_1(struct tree_node *tree,
                                          const char *name)
{
    int cmp;
    if (tree == NULL)
        return NULL;
    cmp = strcmp(name, tree->name);
    if (cmp == 0)
        return tree;
    else if (cmp < 0)
        return lookup_1(tree->left, name);
    else
        return lookup_1(tree->right, name);
}
```

— обход дерева

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

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

– ***язык DOT.***

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

```
digraph test_tree {
f -> b;
f -> k;
b -> a;
b -> d;
k -> g;
k -> l;
}
```

17.Директивы препроцессора, макросы.

– ***классификация директив препроцессора;***

1) Макроопределения
#define, #undef
2) Директива включения файлов
#include
3) Директивы условной компиляции
#if, #ifdef, #endif и др.

– ***правила, справедливые для всех директив препроцессора;***

1) Директивы всегда начинаются с символа "#".
2) Любое количество пробельных символов может разделять лексемы в директиве.

3) Директива заканчивается на символе '\n'.
4) Директивы могут появляться в любом месте программы.

– ***макросы (простые, с параметрами, с переменным числом параметров, предопределенные);***

1) Простые макросы

#define идентификатор список-замены

Используются

-в качестве имен для числовых, символьных и строковых констант.

-для незначительного изменения синтаксиса языка

-Переименования типов.

-Управления условной компиляцией

2) Макросы с параметрами

#define идентификатор(x1, x2, ..., xn) список-замены

Пример: #define MAX(x, y) ((x) > (y) ? (x) : (y))

3) Макросы с переменным числом параметров

Пример:

```
#ifndef NDEBUG
#define DBG_PRINT(s, ...) printf(s, __VA_ARGS__)
#else
#define DBG_PRINT(s, ...) ((void) 0)
#endif
```

4) Предопределенные макросы

__LINE__ - номер текущей строки (десятичная константа)
__FILE__ - имя компилируемого файла
__DATE__ - дата компиляции
__TIME__ - время компиляции итд

Эти идентификаторы нельзя переопределять или отменять директивой undef.

– сравнение макросов с параметрами и функций;

Преимущества

- программа может работать немного быстрее;
- макросы "универсальны".

Недостатки

- скомпилированный код становится больше;
- типы аргументов не проверяются;
- нельзя объявить указатель на макрос;
- макрос может вычислять аргументы несколько раз

– ***скобки в макросах;***

1) Если список-замены содержит операции, он должен быть заключен в скобки.
2) Если у макроса есть параметры, они должны быть заключены в скобки в списке-замены.

```
#define TWO_PI  2 * 3.14
f = 360.0 / TWO_PI;
// f = 360.0 / 2 * 3.14;
#define SCALE(x)  (x * 10)
j = SCALE(i + 1);
// j = (i + 1 * 10);
```

– ***создание длинных макросов;***

Пример:

```
#define ECHO(s) \
do              \
{               \
    gets(s);    \
    puts(s);    \
}               \
while(0)        \
```

18.Директивы препроцессора, условная компиляция, операции # и ##.

– ***классификация директив препроцессора;***

См 17

– ***правила, справедливые для всех директив препроцессора;***

см 17

– ***директивы условной компиляции, использование условной компиляции;***

Использование условной компиляции:

1) программа, которая должна работать под несколькими операционными системами;
2) программа, которая должна собираться различными компиляторами;
3) начальное значение макросов;
4) временное выключение кода.

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

```
#endif

#ifndef BUF_SIZE
#define BUF_SIZE    256
#endif

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

– ***директива #if vs директива #ifdef;***

См. выше

– ***директива error;***

```
#error сообщение
```

– ***операция “#”,***

Операция» # конвертирует аргумент макроса в строковый литерал.

– ***операция “##”;***

Операция» ## объединяет две лексемы в одну.

– ***директива pragma (once, pack).***

Директива #pragma позволяет добиться от компилятора специфичного поведения.

#pragma once — нестандартная, но широко распространенная препроцессорная директива, разработанная для контроля за тем, чтобы конкретный исходный файл при компиляции подключался строго один раз.

При использовании **#pragma pack** можно для !конкретных выбранных! структур указать свое выравнивание.

19. inline-функции.

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

```
inline double average(double a, double b)
{
    return (a + b) / 2;
}
```

inline-функции по-другому называют встраиваемыми или подставляемыми.

В C99 inline означает, что определение функции предоставляется только для подстановки и где-то в программе должно быть другое такое же определение этой же функции.

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

int main(void)
{
    int i = add(4, 5);

    return i;
}
// main.c:(.text+0x1e): undefined reference to `add'
// collect2.exe: error: ld returned 1 exit status
```

Способы исправления проблемы «unresolved reference»

1) Использовать ключевое слово static

```
static inline int add(int a, int b) {return a + b;}
int main(void)
{
    int i = add(4, 5);
    return i;
}
```

2)Убрать ключевое слово inline из определения функции.

```
int add(int a, int b) {return a + b;}
int main(void)
{
    int i = add(4, 5);
    return i;
}
```

3)Добавить еще одно такое же не-inline определение функции где-нибудь в программе.

4)C99: добавить файл реализации вот с таким объявлением

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

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

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

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

В отличии 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;
    ...
};
```

```
#include "list.h"

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

Следует отметить следующее:
1) Структуру struct list_head можно поместить в любом месте в определении структуры.
2) struct list_head может иметь любое имя.
3) В структуре может быть несколько полей типа struct list_head.

Создание head:

```
#define LIST_HEAD_INIT(name) { &(name), &(name) }

#define LIST_HEAD(name) \
    struct list_head name = LIST_HEAD_INIT(name)

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

Добавление:

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

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

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

Обход:

```
#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)
```

– ***идеи реализации (циклический двусвязный список, интрузивный список, универсальный список);***
– ***описание типа;***
– ***добавление элемента в начало и конец (list_add, list_add_tail), итерирование по списку (list_for_each, list_for_each_entry), освобождение памяти (list_for_each_safe);***

См выше

– ***особенности реализации макроса list_entry.***

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

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

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

21. Битовые операции. Битовые поля.

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

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

Битовые операции – логические операции, проводимые над каждым битом фигурирующих операндов. Операнды при этом имеют целый тип и одинаковый размер.

– ***использование битовых операций для обработки отдельных битов и последовательностей битов;***

2.1. & -- поразрядная конъюнкция.

На каждом бите проходит конъюнкция/логическое И.

2.2. | -- поразрядная дизъюнкция.

На каждом бите происходит дизъюнкция/логическое ИЛИ

2.3. ^ -- поразрядная симметрическая разность.
На каждом бите операндов происходит симметрическая разность/сумма по модулю 2/логическое ИСКЛЮЧАЮЩЕЕ ИЛИ/АНТИЭКВИВАЛЕНТНОСТЬ.
2.4. ~ -- поразрядная инверсия/логическое НЕ.
Каждый бит операнда изменяет своё значение. При этом результат – обратный код числа op.
2.5. Логические сдвиги числа на некоторое число бит.
Сдвиг влево.
Синтаксис: op<<n_bit.
Иначе говоря, Все биты сдвигаются на n_bit позиций влево, освободившиеся биты заполнятся нулями, а выдвинутые биты уничтожаются.

Сдвиг вправо.
Синтаксис: op>>n_bit.
Все биты сдвигаются на n_bit позиций вправо, освободившиеся биты заполнятся нулями, а выдвинутые биты уничтожаются.
– ***различие между битовыми и логическими операциями;***
При проведении логических операций все ненулевые числа (как бы) приводятся к числу (-1), представимому всеми единицами в двоичной записи. Поэтому 00100100&10010010 == 00000000 != (некая единица) == 00100100&&10010010.
– ***битовые поля: описание, использование, ограничения использования.***
Битовое поле - особый тип структуры, определяющей, какую длину имеет каждый член в битах.
Стандартный вид объявления битовых полей следующий:

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

Битовые поля должны объявляться как целые, unsigned или signed.

22. Неопределенное поведение.

– ***особенности вычисления выражений с побочным эффектом;***

Побочные эффекты:

1) Модификация данных.

2) Обращение к переменным, объявленным как volatile.

3) Вызов системной функции, которая производит побочные эффекты (например, файловый ввод или вывод).

4)Вызов функций, выполняющих любое из вышеперечисленных действий.

– ***понятие «точка следования»;***

Компилятор вычисляет выражения. Выражения будут вычисляться почти в том же порядке, в котором они указаны в исходном коде: сверху вниз и 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.

– ***расположение «точек следования»;***

Определены следующие точки следования:

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

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

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

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

3)В конце полного выражения.

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

4)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.

5)В объявлении с инициализацией на момент завершения вычисления инициализирующего значения.

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

– ***«виды» неопределенного поведения;***

1)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.

2) 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

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

– ***преимущества и недостатки неопределенного поведения;***

Неопределенное поведение нужно, чтобы

1)освободить разработчиков компиляторов от необходимости обнаруживать ошибки, которые трудно диагностировать.

2)избежать предпочтения одной стратегии реализации другой.

3)отметить области языка для расширения языка (language extension).

– ***способы борьбы с неопределенным поведением.***

1)Включайте все предупреждения компилятора, внимательно читайте их.

2)Используйте возможности компилятора (-ftrapv).

3)Используйте несколько компиляторов.

4)Используйте статические анализаторы кода (например, clang).

5)Используйте инструменты такие как valgrind, Doctor Memory и др.

6)Используйте утверждения.

23.Библиотеки.

– ***Статические и динамические библиотеки: назначение, особенности использования, анализ преимуществ и недостатков.***

Библиотека включает в себя

-заголовочный файл;

-откомпилированный файл самой библиотеки:

Библиотеки делятся на статические и динамические.

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.

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.

Способы использования динамических библиотек:

динамическая компоновка;
динамическая загрузка.

– ***Статические библиотеки: создание, использование при компоновке.***

Использование статической библиотеки

Сборка библиотеки

– компиляция

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

– упаковка

```
ar rc libarr.a arr_lib.o
```

– индексирование

```
ranlib libarr.a
```

Сборка приложения

```
gcc -std=c99 -Wall -Werror main.c libarr.a -o test.exe
```

или

```
gcc -std=c99 -Wall -Werror main.c -L. -larr -o test.exe
```

– ***Динамические библиотеки: создание (экспорта и импорта функций), динамическая компоновка и динамическая загрузка.***

Использование динамической библиотеки

(динамическая компоновка)

Сборка библиотеки

- компиляция

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

- компоновка

```
gcc -shared arr_lib.o -Wl,--subsystem,windows -o arr.dll
```

Сборка приложения

```
gcc -std=c99 -Wall -Werror -c main.c
gcc main.o -L. -larr -o test.exe
```

Использование динамической библиотеки

(динамическая загрузка)

Сборка библиотеки

- компиляция

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

- компоновка

```
gcc -shared arr_lib.o -Wl,--subsystem,windows -o arr.dll
```

Сборка приложения

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

– ***Особенности использования динамических библиотек с приложением, реализованным на другом (по отношению к библиотеке) языке программирования.***

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 внутренне устройство типов Python и то, каким образом они могут быть преобразованы в типы Си.

24. Абстрактный тип данных. План ответа:

– ***Понятие «модуль», преимущества модульной организации программы.***

Программу удобно рассматривать как набор независимых модулей.

Модуль состоит из двух частей: интерфейса и реализации.

Интерфейс описывает, что модуль делает. Он определяет идентификаторы, типы и подпрограммы, которые будут доступны коду, использующему этот модуль.

Реализация описывает, как модуль выполняет то, что предлагает интерфейс.

У модуля есть один интерфейс, но реализаций, удовлетворяющих этому интерфейсу, может быть несколько.

Часть кода, которая использует модуль, называют клиентом.

Клиент должен зависеть только от интерфейса, но не от деталей его реализации.

Преимущества модулей:

Абстракция (как средство борьбы со сложностью)

Когда интерфейсы модулей согласованы, ответственность за реализацию каждого модуля делегируется определенному разработчику.

Повторное использование

Модуль может быть использован в другой программе.

Сопровождение

Можно заменить реализацию любого модуля, например, для улучшения производительности или переноса программы на другую платформу.

– ***Разновидности модулей.***

В языке Си интерфейс описывается в заголовочном файле (*.h).

В заголовочном файле описываются макросы, типы, переменные и функции, которые клиент может использовать. Клиент импортирует интерфейс с помощью директивы препроцессора include.

Реализация интерфейса в языке Си представляется одним или несколькими файлами с расширением *.c.
Реализация определяет переменные и функции, необходимые для обеспечения возможностей, описанных в интерфейсе.
Реализация обязательно должна включать файл описания интерфейса, чтобы гарантировать согласованность интерфейса и реализации.

Типы модулей:

Набор данных

Набор связанных переменных и/или констант. В Си модули этого типа часто представляются только заголовочным файлом. (float.h, limits.h.)

Библиотека

Набор связанных функций.

Абстрактный объект

Набор функций, который обрабатывает скрытые данные.

Абстрактный тип данных

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.

– ***Организация модуля в языке Си. Неполный тип в языке Си.***

выше

Неполный тип:

Стандарт Си описывает неполные типы как «типы которые описывают объект, но не предоставляют информацию нужную для определения его размера».

struct t;

Пока тип неполный его использование ограничено.

Описание неполного типа должно быть закончено где-то в программе.

Допустимо определять указатель на неполный тип

typedef struct t *T;

Можно

определять переменные типа T;

передавать эти переменные как аргументы в функцию.

Нельзя

применять операцию обращения к полю (->);

разыменовывать переменные типа T.

– ***Общие вопросы проектирования абстрактного типа данных.***

25. Чтение сложных объявлений.

(замена элементов объявления фразами)

[]	массив типа …
[N]	массив из N элементов типа…
(type)	функция, принимающая аргумент типа type и возвращающая …
*	указатель на …

«Декодирование» объявления выполняется «изнутри наружу». При этом отправной точкой является идентификатор.
Когда сталкиваетесь с выбором, отдавайте предпочтение «[]» и «()», а не «*», т.е.
*name[] – «массив типа», не «указатель на»
*name() – «функция, принимающая», не «указатель на»
При этом «()» могут использоваться для изменения приоритета.
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)[];
```

У массива только левая лексема [] может быть пустой.
Тип void ограниченный.

```
void x;       // ошибка
void x[5];    // ошибка
```