

1. Указатель на void. Стандартные функции обработки областей памяти

Существует особый тип указателей – **указатели типа void** или пустые указатели. Эти указатели используются в том случае, когда тип переменной не известен.

Так как void не имеет типа, то к нему не применима операция разадресации (взятие содержимого) и адресная арифметика, так как неизвестно представление данных. Тем не менее, если мы работаем с указателем типа void, то нам доступны операции сравнения.

Тип указателя void (*обобщенный указатель*, англ. *generic pointer*) используется, если тип объекта неизвестен:

- полезен для ссылки на произвольный участок памяти, независимо от размещенных там объектов;
- позволяет передавать в функцию указатель на объект любого типа.

- 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;
```

- Указатель типа void нельзя разыменовывать.
- К указателям типа void не применима адресная арифметика.



memmove – копирование массивов (в том числе пересекающихся).

Синтаксис:

```
#include <string.h>
void *memmove (void *destination, const void *source, size_t n);
```

Аргументы:

destination – указатель на массив в который будут скопированы данные.
source – указатель на массив источник копируемых данных.
n – количество байт для копирования.

Возвращаемое значение:

Функция возвращает указатель на массив, в который скопированы данные.



Описание:

Функция memmove копирует n байт из массива (области памяти), на который указывает аргумент source, в массив (область памяти), на который указывает аргумент destination. При этом массивы (области памяти) могут пересекаться.

```
#include < stdio.h > //для printf
#include < string.h > //для memmove

int main (void)
{
    // Исходный массив данных
    unsigned char src[10] = "1234567890";

    // Вывод массива src на консоль
    printf ("src old: %s\n", src);

    // Копируем 3 байт
    memmove (&src[4], &src[3], 3);

    // Вывод массива src на консоль
    printf ("src new: %s\n", src);

    return 0;
}
```

Результат:

Вывод в консоль:

```
src old: 1234567890
src new: 1234456890
```

memcpу – копирование непересекающихся массивов.

Синтаксис:

```
#include < string.h >
void *memcpу (void *destination, const void *source, size_t n);
```

Аргументы:

destination – указатель на массив в который будут скопированы данные.
source – указатель на массив источник копируемых данных.
n – количество байт для копирования.

Возвращаемое значение:

Функция возвращает указатель на массив, в который скопированы данные.

Описание:

Функция memcpу копирует n байт из массива (области памяти), на который указывает аргумент source, в массив (область памяти), на который указывает аргумент destination. Если массивы перекрываются, результат копирования будет не определен.

Пример:

В примере создается массив src, содержащий строку «123456», и пустой массив dst. Затем из массива src копируется 6 байт в массив dst и массив dst выводится на консоль.

```
#include < stdio.h > //для printf
#include < string.h > //для memcpу

int main (void)
{
    // Массив источник данных
    unsigned char src[10] = "123456";

    // Массив приемник данных
    unsigned char dst[10] = "";

    // Копируем 6 байт из массива src в массив dst
    memcpу (dst, src, 6);

    // Вывод массива dst на консоль
    printf ("dst: %s\n", dst);

    return 0;
}
```

Результат:

Вывод в консоль:

```
dst: 123456
```

memcmp – сравнение массивов.

Синтаксис:

```
#include < string.h >
memcmp (const void *arr1, const void *arr2, size_t n);
```

Аргументы:

arr1, arr2 – указатели на сравниваемые массивы.
n – размер сравниваемой части массива в байтах.

Возвращаемое значение:

0 – если сравниваемые части массивов идентичны.



Положительное число, если при сравнении массивов встретился отличный байт и байт из массива, на который указывает аргумент arr1, больше байта из массива, на который указывает аргумент arr2.

Отрицательное число, если при сравнении массивов встретился отличный байт и байт из массива, на который указывает аргумент arr1, меньше байта из массива, на который указывает аргумент arr2.

Описание:

Функция memcmp побайтно сравнивает два массива (области памяти), на которые указывают аргументы arr1 и arr2. Каждый байт массива определяется типом unsigned char. Сравнение продолжается, пока не будут проверено n байт или пока не встретятся отличающиеся байты.

<pre>#include < stdio.h > //Для printf #include < string.h > //Для memcmp int main (void) { // Исходные массивы unsigned char src[15] = "1234567890"; unsigned char dst[15] = "1234567890"; // Сравниваем первые 10 байт массивов // и результат выводим на экран if (memcmp (src, dst, 10) == 0) puts ("Области памяти идентичные."); else puts ("Области памяти отличаются."); return 0; }</pre>	<p>Результат: Вывод в консоль:</p> <div style="border: 1px solid black; padding: 5px;"><p>mc - /programs</p><p>File Edit View Terminal Tabs Help</p><p>Области памяти идентичные.</p></div>
---	--

memset – заполнения массива указанными символами.

Синтаксис:

```
#include < string.h >
void *memset (void *destination, int c, size_t n);
```

Аргументы:

destination – указатель на заполняемый массив
c – код символа для заполнения
n – размер заполняемой части массива в байтах

Возвращаемое значение:

Функция возвращает указатель на заполняемый массив.



Описание:

Функция memset заполняет первые n байт области памяти, на которую указывает аргумент destination, символом, код которого указывается аргументом c.

Пример:

В примере создается массив src, содержащий строку «123456789», затем первые 10 байт этого массива заполняются символом ‘1’ и массив src выводится на консоль.

```

#include <stdio.h> //Для printf
#include <string.h> //Для memset

int main (void)
{
    // Исходный массив
    unsigned char src[15] = "1234567890";

    // Заполняем массив символом '1'
    memset (src, '1', 10);

    // Вывод массива src на консоль
    printf ("src: %s\n", src);

    return 0;
}

```

2. Функции динамического выделения памяти (malloc, calloc, realloc, free)

- рассказать про malloc calloc free
- рассказать про realloc отдельно (типичные ошибки, правильное использование функции)
- выделение 0 байт памяти, необходимость использования приведения типа

Особенности malloc, calloc, realloc (1)

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

malloc (1)



```
#include <stdlib.h>
void* malloc(size_t size);
```

- Функция malloc (C99 7.20.3.3) выделяет блок памяти указанного размера size. Величина size указывается в байтах.
- Выделенный блок памяти не инициализируется (т.е. содержит «мусор»).
- Для вычисления размера требуемой области памяти необходимо использовать операцию sizeof.

malloc и явное приведение типа

```
a = (int*) malloc(n * sizeof(int));
```

Преимущества явного приведения типа:

- компиляции с помощью C++ компилятора;
- у функции malloc до стандарта ANSI C был другой прототип (char* malloc(size_t size));
- дополнительная «проверка» аргументов разработчиком.

Недостатки явного приведения типа:

- начиная с ANSI C приведение не нужно;
- может скрыть ошибку, если забыли подключить stdlib.h;
- в случае изменения типа указателя придется менять тип в приведении.

calloc (1)

```
#include <stdlib.h>
void* calloc(size_t nmemb, size_t size);
```

- Функция `calloc` (C99 7.20.3.1) выделяет блок памяти для массива из `nmemb` элементов, каждый из которых имеет размер `size` байт.
- Выделенная область памяти инициализируется таким образом, чтобы каждый бит имел значение 0.



Что будет, если запросить 0 байт?

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

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

ПОЭТОМУ перед вызовом этих функций нужно убедиться, что запрашиваемый размер блока не равен нулю.

- Логические ошибки (продолжение)
 - Изменение указателя, который вернула функция выделения памяти.
 - Двойное освобождение памяти.
 - Освобождение невыделенной или нединамической памяти.
 - Выход за границы динамического массива.
 - И многое другое ☺

```
#include <stdlib.h>
void free(void *ptr);
```

- Функция `free` (C99 7.20.3.2) освобождает (делает возможным повторное использование) ранее выделенный блок памяти, на который указывает `ptr`.
- Если значением `ptr` является нулевой указатель, ничего не происходит.
- Если указатель `ptr` указывает на блок памяти, который не был получен с помощью одной из функций `malloc`, `calloc` или `realloc`, поведение функции `free` не определено.

Типичные ошибки (1)

- Неверный расчет количества выделяемой памяти.
- Отсутствие проверки успешности выделения памяти
- Утечки памяти
- Логические ошибки
 - Wild (англ., дикий) pointer: использование непропонициализированного указателя.
 - Dangling (англ., висящий) pointer: использование указателя сразу после освобождения памяти.



realloc

```
#include <stdlib.h>
void* realloc(void *ptr, size_t size); // C99 7.20.3.4
```

- `ptr == NULL && size != 0`
Выделение памяти (как `malloc`)
- `ptr != NULL && size == 0`
Освобождение памяти (как `free`).
- `ptr != NULL && size != 0`
Перевыделение памяти. В худшем случае:
 - выделить новую область
 - скопировать данные из старой области в новую
 - освободить старую область



Типичная ошибка вызова `realloc`

Неправильно

```
// pbuf и n имеют корректные значения
pbuf = realloc(pbuf, 2 * n);
```

Что будет, если `realloc` вернет `NULL`?

Правильно

```
void *ptmp = realloc(pbuf, 2 * n);
if (ptmp)
    pbuf = ptmp;
else
    // обработка ошибочной ситуации
```

Функция `realloc` выполняет перераспределение блоков памяти. Размер блока памяти, на который ссылается параметр `ptrmem` изменяется на `size` байтов. Блок памяти может уменьшаться или увеличиваться в размере.

Эта функция может перемещать блок памяти на новое место, в этом случае функция возвращает указатель на новое место в памяти. Содержание блока памяти сохраняется даже если новый блок имеет меньший размер, чем старый. Отбрасываются только те данные, которые не поместились в новый блок. Если новое значение `size` больше старого, то содержимое вновь выделенной памяти будет неопределенным.

В случае, если `ptrmem` равен `NULL`, функция ведет себя именно так, как функция `malloc`, т. е. выделяет память и возвращает указатель на этот участок памяти.

В случае, если `size` равен 0, ранее выделенная память будет освобождена, как если бы была вызвана функция `free`, и возвращается нулевой указатель.

Особенности `malloc`, `calloc`, `realloc`

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

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

- 2 способа выделения памяти под массив
- Классификация ошибок, примеры ошибок
- Подходы к обработке ситуации отсутствия динамической памяти (`NULL` результат `malloc`)
- **Как вернуть из функции динамический массив? - ВАЖНО**

Как возвращаемое значение	Как параметр функции
<code>int* create_array(FILE *f, int *n);</code> прототип <code>int *arr, n; arr = create_array(f, &n);</code> вызов	<code>int create_array(FILE *f, int **arr, int *n)</code> <code>int *arr, n, rc; rc = create_array(f, &arr, &n);</code>

Ошибка сегментации (Segmentation fault)

Если процесс попытается использовать "чужую" память (что в защищном режиме работы процессора в принципе невозможно из-за механизма виртуальной адресации), обратившись по некоторому случайному адресу, операционная система аварийно завершит процесс с выводом предупреждения пользователю.

Пример

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

void foo(int *pointer)
{
    *pointer = 0; //потенциальный Segmentation fault
}

int main()
{
    int *p;
    int x;
    *NULL = 10; //совсем очевидный Segmentation fault
    *p = 10; //достаточно очевидный Segmentation fault
    foo(NULL); //скрытый Segmentation fault
    scanf("%d", x); //скрытый и очень популярный у новичков на Си Segmentation fault

    return 0;
}
```

Утечка памяти (Memory leak)



Если процесс попросил у ОС память, а затем про нее забыл и более не использует, это называется утечкой памяти.

Утечки памяти не являются критической ошибкой и в небольшом масштабе допустимы, если процесс работает очень недолго (секунды). Однако при разработке сколько-нибудь масштабируемого и выполняющегося продолжительное время приложения, допущение даже маленьких утечек памяти — серьезная ошибка.

```

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

void swap_arrays(int *A, int *B, size_t N)
{
    int * tmp = (int *) malloc(sizeof(int)*N); //временный массив
    for(size_t i = 0; i < N; i++)
        tmp [i] = A[i];
    for(size_t i = 0; i < N; i++)
        A[i] = B[i];
    for(size_t i = 0; i < N; i++)
        B[i] = tmp [i];
    //выходя из функции, забыли освободить память временного массива
}

int main()
{
    int A[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int B[10] = {10, 9, 8, 7, 6, 5, 4, 3, 2, 1};
    swap_arrays(A, B, 10); //функция swap_arrays() имеет утечку памяти

    int *p;
    for(int i = 0; i < 10; i++) {
        p = (int *)malloc(sizeof(int)); //выделение памяти в цикле 10 раз
        *p = 0;
    }
    free(p); //а освобождение вне цикла - однократное. Утечка!
    return 0;
}

```

- Логические ошибки (продолжение)

- Изменение указателя, который вернула функция выделения памяти.
- Двойное освобождение памяти.
- Освобождение невыделенной или нединамической памяти.
- Выход за границы динамического массива.
- И многое другое ☺

Проверять с помощью IF какой указатель вернула нам функцию malloc, calloc, realloc

Подходы к обработке ситуации отсутствия памяти (англ., OOM)

- Возвращение ошибки (англ., return failure) – Подход, который используем мы
- Ошибка сегментации (англ., segfault) – Обратная сторона - проблемы с безопасностью
- Аварийное завершение (англ., abort) – Идея принадлежит Кернигану и Ритчи (xmalloc)
- Восстановление (англ., recovery) – xmalloc из git



`xmalloc()` является нестандартной функцией, которая имеет девиз *добиться успеха или умереть*. Если он не сможет выделить память, он завершит вашу программу и напечатает сообщение об ошибке в `stderr`.

само выделение ничем не отличается; только поведение в случае, когда никакая память не может быть выделена, отличается.



Керниган, Ритчи

```
#include <stdio.h>
extern char *malloc ();
void *
xmalloc (size)
```

```
unsigned size;
{
    void *new_mem = (void *) malloc (size);
    if (new_mem == NULL)
    {
        fprintf (stderr, "fatal: memory exhausted (xmalloc of %u bytes).\n", size);
        exit (-1);
    }
    return new_mem;
}
```

```

Одна из реализаций xmalloc из git – пытается спастиесь
static void *do_xmalloc(size_t size, int gentle)
{
    void *ret;
    if (memory_limit_check(size, gentle))
        return NULL;
    ret = malloc(size);
    if (!ret && !size)
        ret = malloc(1); // если пытались выделить 0 байт
    if (!ret) { // если просто не получилось выделить
        try_to_free_routine(size);
        ret = malloc(size);
        if (!ret && !size)
            ret = malloc(1);
        if (!ret) {
            if (!gentle)
                die("Out of memory, malloc failed (tried to allocate %lu bytes)",
                    (unsigned long)size);
            else {
                error("Out of memory, malloc failed (tried to allocate %lu bytes)",
                    (unsigned long)size);
                return NULL;
            }
        }
    }
#endif XMALLOC_POISON
    memset(ret, 0xA5, size);
#endif
    return ret;
}

```

4. Указатель на функцию. Функция qsort.

- Для чего нужны указатели на функцию, примеры (желательно целостный пример)
- Описание указателя на функцию, присваивание, использование функции по указателю
- Адресная арифметика с указателем на функцию
- Примеры использования функции qsort

Синтаксис объявления указателей на функцию

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

Указатель на функцию

- Объявление указателя на функцию

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



- Получение адреса функции

```
result = trapezium(0, 3.14, 25, &sin /* sin */);
```

- Вызов функции по указателю

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

Использование указателей на функции (1)

С помощью указателей на функции в языке Си реализуются

- функции обратного вызова (англ., callback);
- таблицы переходов (англ., jump table);
- динамическое связывание (англ., binding).

Использование указателей на функции (2)

Callback (англ., функция обратного вызова) - передача исполняемого кода в качестве одного из параметров другого кода. [\[wiki\]](#)

Функция обратного вызова - это "действие", передаваемое в функцию в качестве аргумента, которое обычно используется

- для обработки данных внутри функции (map);
- для того, чтобы «связываться» с тем, кто вызвал функции, при наступлении какого-то события.



В языке программирования С функция тоже имеет адрес и может иметь указатель. Указатель на функцию представляет собой выражение или переменную, которые используются для представления адреса функции. Указатель на функцию содержит адрес первого байта в памяти, по которому располагается выполняемый код функции.

Самым распространенным указателем на функцию является ее имя. С помощью имени функции мы можем вызывать ее и получать результат ее работы.

- Согласно C99 6.7.5.3 #8, выражение из имени функции неявно преобразуется в указатель на функцию.

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

- Операция "&" для функции возвращает указатель на функцию, но из-за 6.7.5.3 #8 это лишняя операция.

```
int (*p2)(int, int) = &add;
```

- Операция "*" для указателя на функцию возвращает саму функцию, (которая неявно преобразуется в указатель на функцию).

```
int (*p3)(int, int) = *add;
int (*p4)(int, int) = *****add;
int (*p5)(int, int) = get_action_1('+');
int (*p6)(int, int) = get_action_2('+');
```



- Указатели на функции можно сравнивать

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

- Указатель на функцию может быть типом возвращаемого значения функции

```
int (*get_action(char ch))(int, int);
// typedef приходит на помощь :)
typedef int (*ptr_action_t)(int, int);
ptr_action_t get_action(char ch);
```

при применении адресной арифметики, указатель может указывать уже не на функцию, а в другое место памяти при передачи указателя на функцию в виде func + 1 (2, 3, 4///) код может сработать, а может и нет

```
1 #include <conio.h>
2 #include <stdio.h>
3
4 int dble(int a) {
5     return 2*a;
6 }
7
8 int deleteEven(int a) {
9     if (a % 2 == 0) {
10         return 0;
11     } else {
12         return a;
13     }
14 }
15
16 //Функция принимает массив, его размер и указатель на функцию,
17 //которая далее применяется ко всем элементам массива
18 void map(int *arr, unsigned size, int (*fun)(int)) {
19     unsigned i;
20     for (i = 0; i < size; i++) {
21         arr[i] = fun(arr[i]);
22     }
23 }
24
25 void main () {
26     int a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
27     unsigned i;
28     map(a, 10, deleteEven);
29     for (i = 0; i < 10; i++) {
30         printf("%d ", a[i]);
31     }
32     map(a, 10, dble);
33     printf("\n");
34     for (i = 0; i < 10; i++) {
35         printf("%d ", a[i]);
36     }
37     getch();
38 }
```

qsort (stdlib.h)

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



Пусть необходимо упорядочить массив целых чисел по возрастанию.

```
int compare_int(const void* p, const void* q)
{
    const int *a = p;
    const int *b = q;
    return *a - *b; // return *(int*)p - *(int*)q;
}
...
int a[10];
...
qsort(a, sizeof(a) / sizeof(a[0]), sizeof(a[0]),
       compare_int);
```

5. Утилита make. Простой сценарий сборки

- Для чего нужна утилита make, что поступает на вход, какая идея лежит в основе утилиты
- Разновидности утилиты, что под собой представляет сценарий сборки (переменные и правила)
- Про переменные **коротко**, про правила **подробно**
- **Правила:** из каких частей состоит, что за назначения у правил в зависимости от их составных частей
- Пример сценария сборки, на примере рассказать, как утилита make руководствуясь сценарием выполняет сборку
- **Все что нужно утилите make для работы – сценарий сборки и время последней модификации файлов**

Многофайловый проект

Компиляция

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

Компоновка

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

Почему плохо делать так?

```
gcc -std=c99 -Wall -Werror *.c -o app.exe
```

make — утилита, автоматизирующая процесс преобразования файлов из одной формы в другую.

- GNU Make (рассматривается далее)
- BSD Make
- Microsoft Make (nmake)



Принципы работы

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

- отношения между файлами программы;
- содержит команды для обновления каждого файла.

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

<h2>Сценарий сборки проекта</h2> <p>цель: зависимость_1 ... зависимость_n</p> <p>[tab]команда_1 [tab]команда_2 ... [tab]команда_m</p> <p>что создать/сделать: из чего создать как создать/что сделать</p>	<h2>Простой сценарий сборки</h2> <pre> greeting.exe : hello.o bye.o main.o gcc -o greeting.exe hello.o bye.o main.o test_greeting.exe : hello.o bye.o test.o gcc -o test_greeting.exe hello.o bye.o test.o hello.o : hello.c hello.h gcc -std=c99 -Wall -Werror -pedantic -c hello.c bye.o : bye.c bye.h gcc -std=c99 -Wall -Werror -pedantic -c bye.c main.o : main.c hello.h bye.h gcc -std=c99 -Wall -Werror -pedantic -c main.c test.o : test.c hello.h bye.h gcc -std=c99 -Wall -Werror -pedantic -c test.c clean : rm *.o *.exe </pre>
<h2>Алгоритм работы make (1)</h2> <p>Первый запуск make</p> <ul style="list-style-type: none"> make читает сценарий сборки и начинает выполнять первое правило <pre>greeting.exe : hello.o bye.o main.o gcc -o greeting.exe hello.o bye.o main.o</pre> <ul style="list-style-type: none"> Для выполнения этого правила необходимо сначала обработать зависимости <pre>hello.o bye.o main.o</pre> <ul style="list-style-type: none"> make ищет правило для создания файла hello.o <pre>hello.o : hello.c hello.h gcc -std=c99 -Wall -Werror -pedantic -c hello.c</pre>	<h2>Алгоритм работы make (2)</h2> <p>Первый запуск make</p> <ul style="list-style-type: none"> Файл hello.o отсутствует, файлы hello.c и hello.h существуют. Следовательно, правило для создания hello.o может быть выполнено <pre>gcc -std=c99 -Wall -Werror -pedantic -c hello.c</pre> <ul style="list-style-type: none"> Аналогично обрабатываются зависимости bye.o и main.o. Все зависимости получены, теперь правило для построения greeting.exe может быть выполнено <pre>gcc -o greeting.exe hello.o bye.o main.o</pre>
<h2>Алгоритм работы make (3)</h2> <p>Второй запуск make (hello.c был изменен)</p> <ul style="list-style-type: none"> make читает сценарий сборки и начинает выполнять первое правило <pre>greeting.exe : hello.o bye.o main.o gcc -o greeting.exe hello.o bye.o main.o</pre> <ul style="list-style-type: none"> Для выполнения этого правила необходимо сначала обработать зависимости <pre>hello.o bye.o main.o</pre> <ul style="list-style-type: none"> make ищет правило для создания файла hello.o <pre>hello.o : hello.c hello.h gcc -std=c99 -Wall -Werror -pedantic -c hello.c</pre>	<h2>Алгоритм работы make (4)</h2> <p>Второй запуск make (hello.c был изменен)</p> <ul style="list-style-type: none"> Файлы hello.o, hello.c и hello.h существуют, но время изменения hello.o меньше времени изменения hello.c. Придется пересоздать файл hello.o <pre>gcc -std=c99 -Wall -Werror -pedantic -c hello.c</pre> <ul style="list-style-type: none"> Аналогично обрабатываются зависимости bye.o и main.o, но эти файлы были изменены позже соответствующих си-файлов, т.е. ничего делать не нужно.

Алгоритм работы make (5)

Второй запуск make (hello.c был изменен)

- Все зависимости получены. Время изменения greeting.exe меньше времени изменения hello.o.
Придется пересоздать greeting.exe

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



Ключи запуска make

- Ключ «-f» используется для указания имени файла сценария сборки
`make -f makefile_2`
- Ключ «-B» используется для безусловного выполнения правил
`make -B`
- Ключ «-n» используется для вывода команд без их выполнения
`make -n`
- Ключ «-i» используется для игнорирования ошибок при выполнении команд
`make -i`

13

Переменные необязательны, но позволяют упростить сценарий сборки

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

Правило – совокупность цели, её зависимостей и команд для выполнения данной цели.

Правило будет выполнено, если время создания файла цели было раньше времени модификации файлов её зависимостей.

Неявные правила и переменные

```
# Общие объектные файлы
OBJS := hello.o bye.o

greeting.exe : $(OBJS) main.o
    $(CC) -o greeting.exe $(OBJS) main.o

test_greeting.exe : $(OBJS) test.o
    $(CC) -o test_greeting.exe $(OBJS) test.o

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

Ключ «-r» показывает неявные правила и переменные. Ключ «-g» запрещает использовать неявные правила.

18

Неявные правила make представляют собой автоматические правила, которые используются для создания файлов. Они определяются на основе расширения файлов или изменения времени их модификации. Например, если в Makefile присутствует правило для создания файла .o из файла .c, make будет автоматически выполнять это правило при необходимости, не требуя явного указания этого правила в Makefile.

Шаблонные правила (1)

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

Шаблонные правила (2)

```
# Компилятор
CC := gcc

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

# Общие объектные файлы
OBJS := hello.o bye.o

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

test_greeting.exe : $(OBJS) test.o
    $(CC) $^ -o $@

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

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

6. Утилита make. Назначение, переменные, шаблонные правила

- Для чего нужна утилита make, что поступает на вход, какая идея лежит в основе утилиты
- Разновидности утилиты, что под собой представляет сценарий сборки (переменные и правила)
- Про переменные **подробно**, про правила **коротко**
- **Переменные:** обычные переменные, неявные переменные и правила, автоматические переменные, шаблонные правила (везде примеры)

Многофайловый проект

Компиляция

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

Компоновка



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

Почему плохо делать так?

```
gcc -std=c99 -Wall -Werror *.c -o app.exe
```

make — утилита, автоматизирующая процесс преобразования файлов из одной формы в другую.

- GNU Make (рассматривается далее)
- BSD Make
- Microsoft Make (nmake)

Принципы работы

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

- отношения между файлами программы;
- содержит команды для обновления каждого файла.

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

<h2>Сценарий сборки проекта</h2> <p>цель: зависимость_1 ... зависимость_n [tab]команда_1 [tab]команда_2 ... [tab]команда_m</p> <p>что создать/сделать: из чего создать как создать/что сделать</p>	<h2>Простой сценарий сборки</h2> <pre> greeting.exe : hello.o bye.o main.o gcc -o greeting.exe hello.o bye.o main.o test_greeting.exe : hello.o bye.o test.o gcc -o test_greeting.exe hello.o bye.o test.o hello.o : hello.c hello.h gcc -std=c99 -Wall -Werror -pedantic -c hello.c bye.o : bye.c bye.h gcc -std=c99 -Wall -Werror -pedantic -c bye.c main.o : main.c hello.h bye.h gcc -std=c99 -Wall -Werror -pedantic -c main.c test.o : test.c hello.h bye.h gcc -std=c99 -Wall -Werror -pedantic -c test.c clean : rm *.o *.exe </pre>
<h2>Алгоритм работы make (1)</h2> <p>Первый запуск make</p> <ul style="list-style-type: none"> make читает сценарий сборки и начинает выполнять первое правило <pre> greeting.exe : hello.o bye.o main.o gcc -o greeting.exe hello.o bye.o main.o </pre> Для выполнения этого правила необходимо сначала обработать зависимости <pre> hello.o bye.o main.o </pre> make ищет правило для создания файла hello.o <pre> hello.o : hello.c hello.h gcc -std=c99 -Wall -Werror -pedantic -c hello.c </pre> 	<h2>Алгоритм работы make (2)</h2> <p>Первый запуск make</p> <ul style="list-style-type: none"> Файл hello.o отсутствует, файлы hello.c и hello.h существуют. Следовательно, правило для создания hello.o может быть выполнено <pre> gcc -std=c99 -Wall -Werror -pedantic -c hello.c </pre> Аналогично обрабатываются зависимости bye.o и main.o. Все зависимости получены, теперь правило для построения greeting.exe может быть выполнено <pre> gcc -o greeting.exe hello.o bye.o main.o </pre>
<h2>Алгоритм работы make (3)</h2> <p>Второй запуск make (hello.c был изменен)</p> <ul style="list-style-type: none"> make читает сценарий сборки и начинает выполнять первое правило <pre> greeting.exe : hello.o bye.o main.o gcc -o greeting.exe hello.o bye.o main.o </pre> Для выполнения этого правила необходимо сначала обработать зависимости <pre> hello.o bye.o main.o </pre> make ищет правило для создания файла hello.o <pre> hello.o : hello.c hello.h gcc -std=c99 -Wall -Werror -pedantic -c hello.c </pre> 	<h2>Алгоритм работы make (4)</h2> <p>Второй запуск make (hello.c был изменен)</p> <ul style="list-style-type: none"> Файлы hello.o, hello.c и hello.h существуют, но время изменения hello.o меньше времени изменения hello.c. Придется пересоздать файл hello.o <pre> gcc -std=c99 -Wall -Werror -pedantic -c hello.c </pre> Аналогично обрабатываются зависимости bye.o и main.o, но эти файлы были изменены позже соответствующих си-файлов, т.е. ничего делать не нужно.

Алгоритм работы make (5)

Второй запуск make (hello.c был изменен)

- Все зависимости получены. Время изменения greeting.exe меньше времени изменения hello.o.
- Придется пересоздать greeting.exe

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

Ключи запуска make

- Ключ «-f» используется для указания имени файла сценария сборки
`make -f makefile_2`
- Ключ «-B» используется для безусловного выполнения правил
`make -B`
- Ключ «-n» используется для вывода команд без их выполнения
`make -n`
- Ключ «-i» используется для игнорирования ошибок при выполнении команд
`make -i`

13

Переменные необязательны, но позволяют упростить сценарий сборки

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

Правило – совокупность цели, её зависимостей и команд для выполнения данной цели.

Правило будет выполнено, если время создания файла цели было раньше времени модификации файлов её зависимостей

Использование переменных и комментариев (1)



Строки, которые начинаются с символа '#', являются комментариями.

Определить переменную в make-файле можно следующим образом:

```
VAR_NAME := value
```

Чтобы получить значение переменной, необходимо ее имя заключить в круглые скобки и перед ними поставить символ '\$'.

```
$ (VAR_NAME)
```

Использование переменных и комментариев (2)

```
# Компилятор
CC := gcc

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

# Общие объектные файлы
OBJS := hello.o bye.o

greeting.exe : $(OBJS) main.o
    $(CC) -o greeting.exe $(OBJS) main.o

test_greeting.exe : $(OBJS) test.o
    $(CC) -o test_greeting.exe $(OBJS) test.o

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

..

Автоматические переменные (1)

Автоматические переменные - это переменные со специальными именами, которые «автоматически» принимают определенные значения перед выполнением описанных в правиле команд.

- Переменная "\$^" означает "список зависимостей".
- Переменная "\$@" означает "имя цели".
- Переменная "\$<" является просто первой зависимостью.
- ...



Автоматические переменные (2)

Было

```
greeting.exe : $(OBJS) main.o  
$(CC) -o greeting.exe $(OBJS) main.o
```

Стало

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

Было

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

Стало

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

Шаблонные правила (1)

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



Шаблонные правила (2)

```
# Компилятор  
CC := gcc  
  
# Опции компиляции  
CFLAGS := -std=c99 -Wall -Werror -pedantic  
  
# Общие объектные файлы  
OBJS := hello.o bye.o  
  
greeting.exe : $(OBJS) main.o  
$(CC) $^ -o $@  
  
test_greeting.exe : $(OBJS) test.o  
$(CC) $^ -o $@  
  
%.o : %.c %.h  
$(CC) $(CFLAGS) -c $<  
  
.PHONY : clean  
clean :  
$(RM) *.o *.exe
```

Примеры неявных переменных – RM, CC, CFLAGS, PC (для Паскаля)

7. Утилита make. Назначение, условные конструкции, анализ зависимостей

- Для чего нужна утилита make, что поступает на вход, какая идея лежит в основе утилиты
- Разновидности утилиты, что под собой представляет сценарий сборки (переменные и правила)
- Про переменные коротко, про правила коротко
- **Про условные конструкции:** про самих себя, про переменные зависящие от цели, варианты подхода к анализу зависимостей (вручную, любой си файл от всех заголовочных, работу переложить на компилятор)

Многофайловый проект

Компиляция

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

Компоновка

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

Почему плохо делать так?

```
gcc -std=c99 -Wall -Werror *.c -o app.exe
```

<p>make — утилита, автоматизирующая процесс преобразования файлов из одной формы в другую.</p> <ul style="list-style-type: none"> – GNU Make (рассматривается далее) – BSD Make – Microsoft Make (nmake) 	<h2>Принципы работы</h2> <p>Необходимо создать так называемый <i>сценарий сборки проекта</i> (make-файл). Этот файл описывает</p> <ul style="list-style-type: none"> – отношения между файлами программы; – содержит команды для обновления каждого файла. <p>Утилита make использует информацию из make-файла и время последнего изменения каждого файла для того, чтобы решить, какие файлы нужно обновить.</p>
<h3>Сценарий сборки проекта</h3> <p>цель: зависимость_1 ... зависимость_n</p> <p>[tab]команда_1 [tab]команда_2 ... [tab]команда_m</p> <p>что создать/сделать: из чего создать как создать/что сделать</p>	<h3>Простой сценарий сборки</h3> <pre> greeting.exe : hello.o bye.o main.o gcc -o greeting.exe hello.o bye.o main.o test_greeting.exe : hello.o bye.o test.o gcc -o test_greeting.exe hello.o bye.o test.o hello.o : hello.c hello.h gcc -std=c99 -Wall -Werror -pedantic -c hello.c bye.o : bye.c bye.h gcc -std=c99 -Wall -Werror -pedantic -c bye.c main.o : main.c hello.h bye.h gcc -std=c99 -Wall -Werror -pedantic -c main.c test.o : test.c hello.h bye.h gcc -std=c99 -Wall -Werror -pedantic -c test.c clean : rm *.o *.exe </pre>
<h3>Алгоритм работы make (1)</h3> <p>Первый запуск make</p> <ul style="list-style-type: none"> • make читает сценарий сборки и начинает выполнять первое правило <pre>greeting.exe : hello.o bye.o main.o gcc -o greeting.exe hello.o bye.o main.o</pre> <ul style="list-style-type: none"> • Для выполнения этого правила необходимо сначала обработать зависимости <pre>hello.o bye.o main.o</pre> <ul style="list-style-type: none"> • make ищет правило для создания файла hello.o <pre>hello.o : hello.c hello.h gcc -std=c99 -Wall -Werror -pedantic -c hello.c</pre>	<h3>Алгоритм работы make (2)</h3> <p>Первый запуск make</p> <ul style="list-style-type: none"> • Файл hello.o отсутствует, файлы hello.c и hello.h существуют. Следовательно, правило для создания hello.o может быть выполнено <pre>gcc -std=c99 -Wall -Werror -pedantic -c hello.c</pre> <ul style="list-style-type: none"> • Аналогично обрабатываются зависимости bye.o и main.o. <ul style="list-style-type: none"> • Все зависимости получены, теперь правило для построения greeting.exe может быть выполнено <pre>gcc -o greeting.exe hello.o bye.o main.o</pre>

<h3>Алгоритм работы make (3)</h3> <p>Второй запуск make (hello.c был изменен)</p> <ul style="list-style-type: none"> make читает сценарий сборки и начинает выполнять первое правило <pre>greeting.exe : hello.o bye.o main.o gcc -o greeting.exe hello.o bye.o main.o</pre> <ul style="list-style-type: none"> Для выполнения этого правила необходимо сначала обработать зависимости <pre>hello.o bye.o main.o</pre> <ul style="list-style-type: none"> make ищет правило для создания файла hello.o <pre>hello.o : hello.c hello.h gcc -std=c99 -Wall -Werror -pedantic -c hello.c</pre>	<h3>Алгоритм работы make (4)</h3> <p>Второй запуск make (hello.c был изменен)</p> <ul style="list-style-type: none"> Файлы hello.o, hello.c и hello.h существуют, но время изменения hello.o меньше времени изменения hello.c. Придется пересоздать файл hello.o <pre>gcc -std=c99 -Wall -Werror -pedantic -c hello.c</pre> <ul style="list-style-type: none"> Аналогично обрабатываются зависимости bye.o и main.o, но эти файлы были изменены позже соответствующих си-файлов, т.е. ничего делать не нужно.
<h3>Алгоритм работы make (5)</h3> <p>Второй запуск make (hello.c был изменен)</p> <ul style="list-style-type: none"> Все зависимости получены. Время изменения greeting.exe меньше времени изменения hello.o. Придется пересоздать greeting.exe <pre>gcc -o greeting.exe hello.o bye.o main.o</pre>	<h3>Ключи запуска make</h3> <ul style="list-style-type: none"> Ключ «-f» используется для указания имени файла сценария сборки <pre>make -f makefile_2</pre> <ul style="list-style-type: none"> Ключ «-B» используется для безусловного выполнения правил <pre>make -B</pre> <ul style="list-style-type: none"> Ключ «-n» используется для вывода команд без их выполнения <pre>make -n</pre> <ul style="list-style-type: none"> Ключ «-i» используется для игнорирования ошибок при выполнении команд <pre>make -i</pre>

Переменные необязательны, но позволяют упростить сценарий сборки

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

Правило – совокупность цели, её зависимостей и команд для выполнения данной цели.

Правило будет выполнено, если время создания файла цели было раньше времени модификации файлов её зависимостей

Сборка программы с разными параметрами компиляции (1)

```
# Компилятор
CC := gcc

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

# Общие объектные файлы
OBJS := hello.o bye.o

ifeq ($(mode), debug)
    # Отладочная сборка: добавим генерацию отладочной информации
    CFLAGS += -g3
endif

ifeq ($(mode), release)
```

Присваивание переменных, зависящих от цели (1)

```
# Компилятор
CC := gcc

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

# Общие объектные файлы
OBJS := hello.o bye.o

debug : CFLAGS += -g3
debug : greeting.exe

release : CFLAGS += -DNDEBUG -g0
release : greeting.exe
```

Генерация зависимостей (1)

```
# Компилятор
CC := gcc

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

# Общие объектные файлы
OBJS := hello.o bye.o

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

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

Сборка программы с разными параметрами компиляции (2)

```
# Финальная сборка: исключим отладочную информацию и
# утверждения (asserts)
CFLAGS += -DNDEBUG -g0
endif

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

test_greeting.exe : $(OBJS) test.o
    $(CC) $^ -o $@

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

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

Присваивание переменных, зависящих от цели (2)

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

test_greeting.exe : $(OBJS) test.o
    $(CC) $^ -o $@

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

.PHONY : clean debug release
clean :
    $(RM) *.* *.exe
```



Генерация зависимостей (2)

```
test_greeting.exe : $(OBJS) test.o
    $(CC) $^ -o $@

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

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

# $(SRCS:.c=.d) – заменяет в переменной SRCS имена файлов с
# c расширением "c" на имена с расширением "d"
include $(SRCS:.c=.d)

.PHONY : clean
clean :
    $(RM) *.* *.exe *.d
```

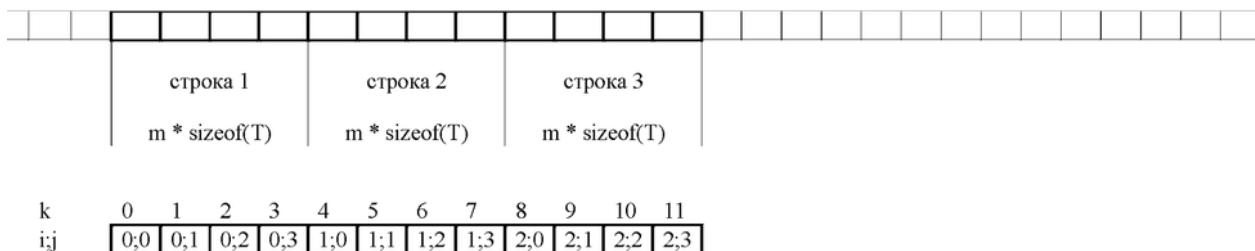
...

8-13. Динамические матрицы. Анализ преимуществ и недостатков различных представлений

- Описать как представлена матрица в памяти
 - Сравнить различные представления матриц с помощью таблицы

Матрица как одномерный массив

$n = 3$ – количество строк
 $m = 4$ – количество столбцов
 T – тип элементов матрицы



$$a[i][j] \Leftrightarrow a[k], k = i * m + j$$

```
double *data;
size_t n = 3, m = 2;

// Выделение памяти под "матрицу"
data = malloc(n * m * sizeof(double));
if (data)
{
    // Работа с "матрицей"
    for (size_t i = 0; i < n; i++)
        for (size_t j = 0; j < m; j++)
            // Обращение к элементу i, j
            data[i * m + j] = 0.0;

    // Освобождение памяти
    free(data);
}
```

```
// Освобождение памяти  
free(data);
```

Матрица как одномерный массив

Преимущества:

- Простота выделения и освобождения памяти.
- Возможность использовать как одномерный массив.

Недостатки:

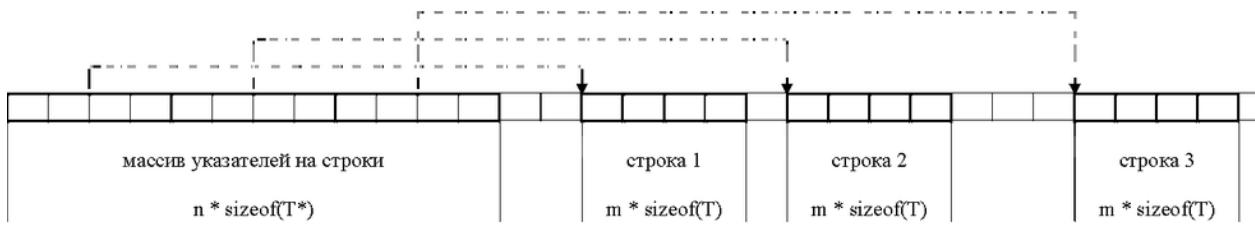
- Отладчик использования памяти (например, valgrind) не может отследить выход за пределы строки.
- Нужно писать $i * m + j$, где m – число столбцов.

Матрица как массив указателей

$n = 3$ – количество строк

$m = 4$ – количество столбцов

T – тип элементов матрицы



```
void free_matrix(double **data, size_t n);

double** allocate_matrix(size_t n, size_t m)
{
    double **data = calloc(n, sizeof(double *));
    if (!data)
        return NULL;
    for (size_t i = 0; i < n; i++)
    {
        data[i] = malloc(m * sizeof(double));
        if (!data[i])
        {
            free_matrix(data, n);
            return NULL;
        }
    }
    return data;
}

void free_matrix(double **data, int n)
{
    for (int i = 0; i < n; i++)
        // free можно передать NULL
        free(data[i]);

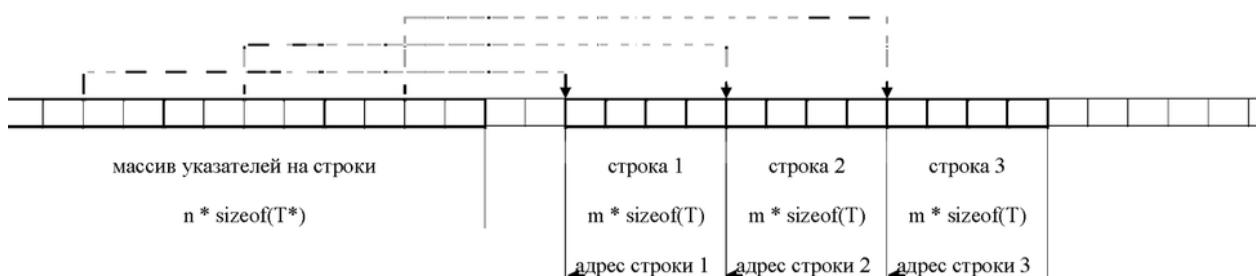
    free(data);
}
```

Матрица как массив указателей

- Преимущества:
 - Возможность обмена строки через обмен указателей.
 - Отладчик использования памяти может отследить выход за пределы строки.
- Недостатки:
 - Сложность выделения и освобождения памяти.
 - Память под матрицу "не лежит" одной областью.

Объединение подходов (1)

$n = 3$ – количество строк
 $m = 4$ – количество столбцов
 T – тип элементов матрицы



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

void free_matrix(double **ptrs)
{
    free(ptrs[0]);

    free(ptrs);
}
```

ВНИМАНИЕ

Здесь скрывается потенциальная ошибка.

ОШИБКА: указатель нулевой (`ptrs[0]`) может указывать не на начало области памяти выделенную под все матрицу..... надо как-то запоминать указатель на область памяти под матрицу

Объединение подходов (1)

Преимущества:



- Относительная простота выделения и освобождения памяти.
- Возможность использовать как одномерный массив.
- Перестановка строк через обмен указателей. (Возможна ошибка, см. слайд 19.)

Недостатки:

- Относительная сложность начальной инициализации.
- Отладчик использования памяти не может отследить выход за пределы строки.

Объединение подходов (2)

$n = 3$ – количество строк
 $m = 4$ – количество столбцов
T – тип элементов матрицы



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

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

free(указатель на матрицу)

Объединение подходов (2)

Преимущества:

- Простота выделения и освобождения памяти.
- Возможность использовать как одномерный массив.
- Перестановка строк через обмен указателей.

Недостатки:

- Сложность начальной инициализации.
- Отладчик использования памяти не может отследить выход за пределы строки.

	Легкость выделения и освобождения	Отладчик использования памяти не может отследить выход за пределы строки.
Матрица 1		
Матрица 2		



14. Чтение сложных объявлений

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 ...

Чтение сложных объявлений (правила)

- «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.

Чтение сложных объявлений (примеры)

```

1. int *(*x[10])(void);
2. char *(*(**foo[])[8])();
3. void (*signal(int, void (*fp)(int)))(int);

```

X - это массив из 10-ти элементов типа указатель на функцию, которая имеет не получает и возвращает указатель на int

char *(*(** foo [iC 8]) ()) [] ;
 foo - это массив типа массив из 8-ми элементов типа указатель на указатель на
других 8-ми элементов, которые все указывают на массив типа указатель на
 char

signal — функция, принимающая int и указатель на функцию, принимающую int, возвращающую void, возвращающая указатель на функцию, принимающую int и возвращающую void.

15. Строки в динамической памяти. Функции POSIX и расширения GNU.

- Наверное, стоит рассказать как создать динамическую строку (И.В. про это не сказал)
- Рассказать про strdup, strndup, getline и sprintf
- Придумать как реализовать эти функции самостоятельно
- Рассказать про Feature Test Macro

Строки и динамическая память

```

// Тут нужны include-ы
#define NAME "Bauman Moscow State Technical University"

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

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

  return 0;
}
  
```



Строки и динамическая память

```

// Тут нужны include-ы
// Для компиляции -std=gnu99
#define NAME "Bauman Moscow State Technical University"

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

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

  return 0;
}
  
```

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

Функции strdup, strndup, getline содержатся в библиотеке string.h, sprint
содержится в stdio.h. Функции strdup strndup getline доступны в
стандарте gnu99, sprintf доступна в c99

strdup – дублирование строк с выделением памяти под новую строку.

Синтаксис:

```
#include <string.h>
char *strdup(const char *str);
```



Аргументы:

str – указатель на дублируемую строку.

Возвращаемое значение:

NULL – если не удалось выделить память под новую строку или скопировать строку на которую указывает аргумент str.

Указатель на дублирующую строку.

Описание:

Функция strdup дублирует строку, на которую указывает аргумент str. Память под дубликат строки выделяется с помощью функции malloc, и по окончанию работы с дубликатом должна быть очищена с помощью функции free.

```
#include <stdio.h> // Для printf
#include <string.h> // Для strdup
#include <stdlib.h> // Для free

int main (void)
{
    // Исходная строка
    char str [11] = "0123456789";
    // Переменная, в которую будет помещен указатель на дубликат строки
    char *istr;

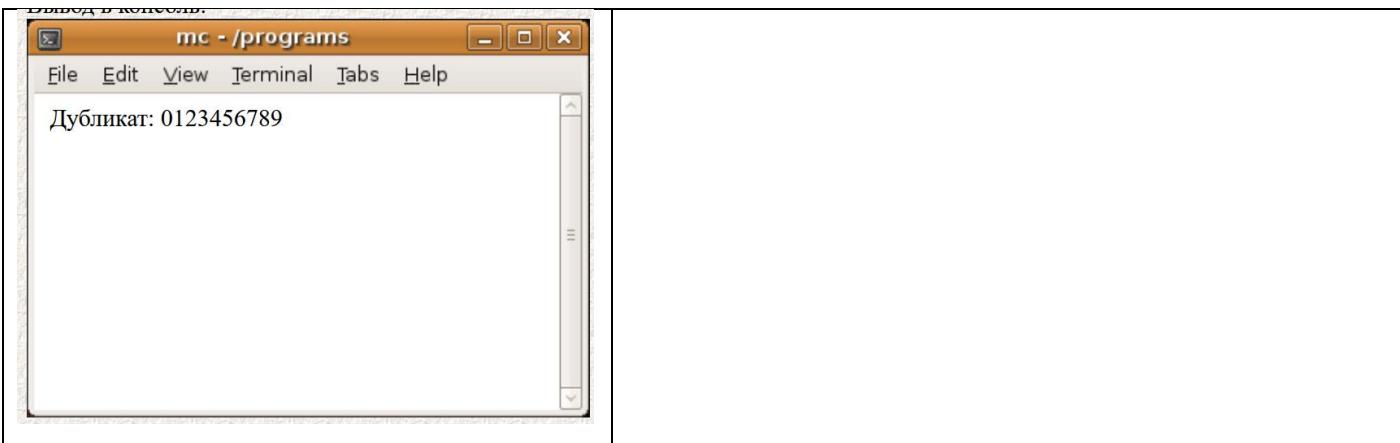
    // Дублирование строки
    istr = strdup (str);

    // Вывод дубликата на консоль
    printf ("Дубликат: %s\n", istr);

    // Очищаем память, выделенную под дубликат
    free (istr);

    return 0;
}
```

```
#include <stdlib.h>
#include <string.h>
char* strdup(const char* src) {
    size_t len = strlen(src) + 1;
    char* dst = malloc(len);
    if (dst == NULL) {
        return NULL;
    }
    memcpy(dst, src, len);
    return dst;
}
```



strup – дублирование строк с ограничением длины и выделением памяти под новую строку.

Синтаксис:

```
#include < string.h >
char *strup(const char *str, size_t n);
```

Аргументы:

str – указатель на дублируемую строку.
n – ограничение длины дублируемой строки.

Возвращаемое значение:

NULL – если не удалось выделить память под новую строку или скопировать строку на которую указывает аргумент str.
Указатель на дублирующую строку.

Описание:

Функция strup дублирует строку, на которую указывает аргумент str. При этом вводится ограничение на максимальную длину дублируемой строки. Если строка короче чем n байт, то дублируется вся строка. Если строка длиннее, чем n байт, то продублировано будет только n байт.

Память под дубликат строки выделяется с помощью функции malloc, и по окончанию работы с дубликатом должна быть очищена с помощью функции free.

```
#include < stdio.h > // для printf
#include < string.h > // для strup
#include < stdlib.h > // для free

int main (void)
{
    // Исходная строка
    char str [11] = "0123456789";
    // переменная, в которую будет помещен указатель на дубликат строки
    char *istr;

    // Дублирование строки
    istr = strup(str,5);

    // Вывод дубликата на консоль
    printf ("Дубликат: %s\n", istr);

    // Очищаем память, выделенную под дубликат
    free (istr);

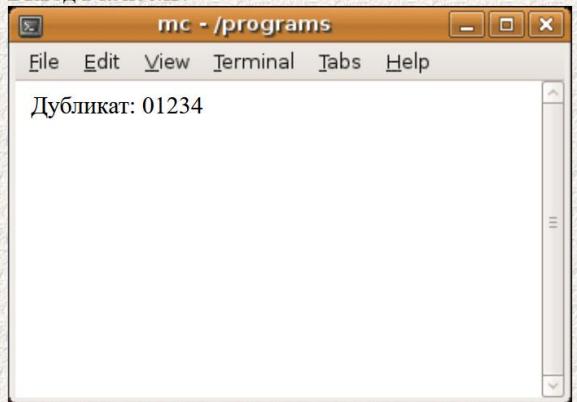
    return 0;
}
```

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

char* strup(const char* src,
size_t n) {
    size_t len = strlen(src);
    if (len > n) {
        len = n;
    }
    char* dst = malloc(len + 1);
```

Результат:

Вывод в консоль:



mc - /programs

File Edit View Terminal Tabs Help

Дубликат: 01234

```
if (dst == NULL)
{
    return NULL;
}
memcpuy(dst, src, len);
dst[len] = '\0';
return dst;
```

ssize_t getline(char **lineptr, size_t *n, FILE *stream);

Функция **getline()** считывает целую строку из *stream*, сохраняет адрес буфера с текстом в **lineptr*. Буфер завершается null и включает символ новой строки, если был найден разделитель для новой строки.

Если **lineptr* равно NULL и **n* равно 0 перед вызовом, то **getline()** выделит буфер для хранения строки. Этот буфер должен быть высвобожден программой пользователя, даже если **getline()** завершилась с ошибкой.

Как альтернатива, перед вызовом **getline()**, **lineptr* может содержать указатель на буфер, выделенный с помощью **malloc** (3) размером **n* байтов. Если буфер недостаточно велик для размещения строки, то **getline()** изменяет размер буфера с помощью **realloc** (3), обновляя **lineptr* и **n* при необходимости.

В любом случае при успешном выполнении вызова **lineptr* и **n* будут содержать правильный адрес буфера и его размер, соответственно.

ВОЗВАЩАЕМОЕ ЗНАЧЕНИЕ

При успешном выполнении **getline()** и **getdelim()** возвращают количество считанных символов, включая символ разделителя, но не включая завершающий байт null ('\0'). Это значение может использоваться для обработки встроенных байтов null при чтении строки.

Обе функции возвращают -1 при ошибках чтения строки (включая условие достижения конца файла). При возникновении ошибки в *errno* сохраняется её значение.

```

#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
int
main(void)
{
    FILE *stream;
    char *line = NULL;
    size_t len = 0;
    ssize_t read;
    stream = fopen("/etc/motd", "r");
    if (stream == NULL)
        exit(EXIT_FAILURE);
    while ((read = getline(&line, &len, stream)) != -1) {
        printf("Получена строка длиной %zu :\n", read);
        printf("%s", line);
    }
    free(line);
    fclose(stream);
    exit(EXIT_SUCCESS);
}

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

// ...

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

    free(line);
    fclose(f);
}

```

```

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

ssize_t getline(char** lineptr,
size_t* n, FILE* stream) {
    if (lineptr == NULL || n == NULL
    || stream == NULL) {
        return -1;
    }

    size_t buffer_size = *n;
    char* buffer = *lineptr;

    int c;
    size_t i = 0;
    while ((c = getc(stream)) != EOF) {
        if (i >= buffer_size - 1) {
            buffer_size *= 2;
            char* new_buffer =
realloc(buffer, buffer_size);
            if (new_buffer == NULL)
{
                return -1;
            }
            buffer = new_buffer;
        }
        buffer[i++] = (char) c;
        if (c == '\n') {
            break;
        }
    }

    *n = buffer_size;
}
```

```

        *lineptr = buffer;
        buffer[i] = '\0';

        if (c == EOF && i == 0) {
            return -1;
        }

        return (ssize_t) i;
    }
}

```

int sprintf(char *buf, const char *format, arg-list)

Прототип:
stdio.h



Описание:

Функция sprintf() идентична printf(), за исключением того, что вывод производится в массив, указанный аргументом buf.

Возвращаемая величина равна количеству символов, действительно занесенных в массив.

После выполнения следующего фрагмента программы строка str содержит one 2 3:
`char str[80];
sprintf (str, "%s %d %c", "one", 2, '3');`

11 /p

What does "#define _GNU_SOURCE" imply?

Asked 12 years, 9 months ago Modified 2 years, 4 months ago Viewed 106k times

Today I had to use the `basename()` function, and the [man 3 basename](#) ([here](#)) gave me some strange message:

205

Notes

There are two different versions of `basename()` - the **POSIX** version described above, and the **GNU version**, which one gets after

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

I'm wondering what this `#define _GNU_SOURCE` means: is it *tainting* the code I write with a GNU-related license? Or is it simply used to tell the compiler something like "Well, I know, this set of functions is not POSIX, thus not portable, but I'd like to use it anyway".

If so, why not give people different headers, instead of having to define some obscure macro to get one function implementation or the other?

Something also bugs me: how does the compiler know which function implementation to link with the executable? Does it use this `#define` as well?

Anybody have some pointers to give me?

230

Defining `_GNU_SOURCE` has nothing to do with license and everything to do with writing (non-)portable code. If you define `_GNU_SOURCE`, you will get:

1. access to lots of nonstandard GNU/Linux extension functions
2. access to traditional functions which were omitted from the POSIX standard (often for good reason, such as being replaced with better alternatives, or being tied to particular legacy implementations)
3. access to low-level functions that cannot be portable, but that you sometimes need for implementing system utilities like `mount`, `ifconfig`, etc.
4. broken behavior for lots of POSIX-specified functions, where the GNU folks disagreed with the standards committee on how the functions should behave and decided to do their own thing.

As long as you're aware of these things, it should not be a problem to define `_GNU_SOURCE`, but you should avoid defining it and instead define `_POSIX_C_SOURCE=200809L` or `_XOPEN_SOURCE=700` when possible to ensure that your programs are portable.

In particular, the things from `_GNU_SOURCE` that you should *never* use are #2 and #4 above.

Что подразумевает "#define _GNU_SOURCE"?

Задано 12 лет 9 месяцев назад Изменено 2 года 4 месяца назад Просмотрено 106 тысяч раз

Сегодня мне пришлось использовать `basename()` функцию, и [man 3 basename](#) ([здесь](#)) выдала мне какое-то странное сообщение:

205

Примечания

Существует две разные версии `basename()` - версия **POSIX**, описанная выше, и версия **GNU**, которую можно получить после

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

Мне интересно, что это `#define _GNU_SOURCE` означает: это *портит* код, который я пишу с лицензией, связанной с GNU? Или это просто используется, чтобы сообщить компилятору что-то вроде "Ну, я знаю, этот набор функций не является POSIX, следовательно, не переносим, но я бы все равно хотел его использовать".

Если да, то почему бы не дать людям разные заголовки вместо того, чтобы определять какой-то непонятный макрос, чтобы получить ту или иную реализацию функции?

Меня тоже кое-что беспокоит: как компилятор узнает, какую реализацию функции связать с исполняемым файлом? Использует ли он это `#define` также?

У кого-нибудь есть несколько советов, которые можно мне дать?

230

Определение `_GNU_SOURCE` не имеет ничего общего с лицензией, а все связано с написанием (не) переносимого кода. Если вы определите `_GNU_SOURCE`, вы получите:

1. доступ ко множеству нестандартных функций расширения GNU / Linux
2. доступ к традиционным функциям, которые были исключены из стандарта POSIX (часто по уважительной причине, например, заменены лучшими альтернативами или привязаны к определенным устаревшим реализациям)
3. доступ к низкоуровневым функциям, которые не могут быть переносимыми, но которые иногда необходимы для реализации системных утилит, таких как `mount`, `ifconfig` и т.д.
4. нарушенное поведение множества функций, определенных в POSIX, когда сотрудники GNU не согласились с комитетом по стандартам относительно того, как должны вести себя функции, и решили действовать по-своему.

Пока вы знаете об этих вещах, определение не должно быть проблемой `_GNU_SOURCE`, но вам следует избегать его определения и вместо этого определять `_POSIX_C_SOURCE=200809L` или `_XOPEN_SOURCE=700`, когда это возможно, чтобы гарантировать переносимость ваших программ.

В частности, вещи из, `_GNU_SOURCE` которые вы никогда не должны использовать, - это # 2 и # 4 выше.

https://translated.turbopages.org/proxy_u/en-ru.ru.72c121d5-6596ba58-f592b6dc-74722d776562/https/stackoverflow.com/questions/5582211/what-does-define-gnu-source-imply

[feature_test_macros\(7\) — Linux manual page](#)

[NAME](#) | [DESCRIPTION](#) | [STANDARDS](#) | [HISTORY](#) | [NOTES](#) | [EXAMPLES](#) | [SEE ALSO](#)

Search online pages

feature..._macros(7) Miscellaneous Information Manual feature..._macros(7)

NAME [top](#)
feature_test_macros - feature test macros

DESCRIPTION [top](#)
Feature test macros allow the programmer to control the definitions that are exposed by system header files when a program is compiled.

NOTE: In order to be effective, a feature test macro *must be defined before including any header files*. This can be done either in the compilation command (`cc -DMACRO=value`) or by defining the macro within the source code before including any headers. The requirement that the macro must be defined before including any header file exists because header files may freely include one another. Thus, for example, in the following lines, defining the `_GNU_SOURCE` macro may have no effect because the header `<abc.h>` itself includes `<xyz.h>` (POSIX explicitly allows this):

```
#include <abc.h>
#define _GNU_SOURCE
#include <xyz.h>
```

Some feature test macros are useful for creating portable applications, by preventing nonstandard definitions from being exposed. Other macros can be used to expose nonstandard definitions that are not exposed by default.

The precise effects of each of the feature test macros described below can be ascertained by inspecting the `<features.h>` header file. **Note:** applications do *not* need to directly include `<features.h>`; indeed, doing so is actively discouraged. See NOTES.

feature_test_macros (7) — страница руководства по Linux

[Имя](#) | [Описание](#) | [СТАНДАРТЫ](#) | [история](#) | [Примечания](#) | [ПРИМЕРЫ](#) | [СМОТРЕТЬ ТАКЖЕ](#)

Search online pages

особенность ..._macros(7) Руководство по различной информации особенность ..._macros(7)

НАЗВАНИЕ [вверху](#)
feature_test_macros - макросы для тестирования функций

ОПИСАНИЕ [вверху](#)
Макросы тестирования свойств позволяют программисту контролировать определения, которые подвергаются системных заголовочных файлов при компиляции программы.

ПРИМЕЧАНИЕ: Для обеспечения эффективности, макрос тестирования функции *должен быть определен перед включением любых заголовочных файлов*. Это может быть сделано в сборник команде (УК `-DMACRO=значение`) или определив макрос в исходном коде до включения любых заголовков. Требование о том, что макрос должен быть определен до включения заголовочного файла существует, потому что заголовочные файлы могут свободно включать друг друга. Таким образом, например, в следующих строках определение макроса `_GNU_SOURCE` может не иметь эффекта, поскольку заголовок `<abc.h>` сам включает `<xyz.h>` (POSIX явно разрешает это):

```
#включить <abc.h>
#определить _GNU_SOURCE
#включить <xyz.h>
```

Некоторые макросы тестирования функций полезны для создания переносимых приложений предотвращая отображение нестандартных определений. Другие макросы можно использовать для отображения нестандартных определений, которые не отображаются по умолчанию.

Точные эффекты каждого из макросов тестирования функций, описанных ниже, можно определить, просмотрев `<features.h>` файл заголовка. **Примечание:** приложением *не* нужно включать напрямую `<features.h>`; на самом деле, это активно не рекомендуется. Смотрите ПРИМЕЧАНИЯ.

Смысл в том, что функции в различных стандартах либо могут иметь разную реализацию, либо могут вообще отсутствовать. Feature Test Macro позволяет явно указать какую реализацию функции программист (не) хочет использовать.

ChatGPT:

Feature Test Macro (FTM) — это макрос в языке программирования C или C++, который проверяет наличие конкретной функциональности или возможностей в текущей среде выполнения программы.

FTM применяется, чтобы определить, поддерживает ли компилятор или окружение определенные функции, константы или структуры данных, прежде чем использовать их в коде. Это позволяет разрабатывать переносимый код, который будет работать на разных платформах и с разными версиями компиляторов.

FTM очень полезны при разработке кросс-платформенных приложений или при использовании функциональности, которая может быть доступна только в некоторых версиях операционных систем или компиляторов.

16. Особенности использования структур с полями указателями.

- Для структурного типа существует операция присваивания, реализуется она с помощью побитового копирования.
Соответственно если внутри структуры есть поле указатель, то это приводит к определенным особенностям такого копирования
(привести пример)
- Глубокое и поверхностное копирование: что это, в каких случаях использовать то или иное (нельзя сказать, что какое-то лучше или хуже)
- Рекурсивное освобождение памяти (освобождение памяти из-под внутренних полей структуры, затем из-под самой структуры)

Структуры с полями-указателями

В Си определена операция присваивания для структурных переменных одного типа. Эта операция фактически эквивалента копированию области памяти, занимаемой одной переменной, в область памяти, которую занимает другая.

При этом реализуется стратегия так называемого «поверхностного копирования» (англ., *shallow coping*), при котором копируется содержимое структурной переменной, но не копируется то, на что могут ссылать поля структуры.

Структуры с полями-указателями

Иногда стратегия «поверхностного копирования» может приводить к ошибкам.

До присваивания

```
C (gcc 4.8, C11)
(known limitations)

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

После присваивания

```
C (gcc 4.8, C11)
(known limitations)

1 #include <string.h>
2 #include <stdlib.h>
3
4 int main(void)
5 {
6     struct book_t
7     {
8         char *title;
9         int year;
10    };
11    a = { 0 }, b = { 0 };
12
13    a.title = strdup("Book a");
14    a.year = 2000;
15
16    b.title = strdup("Book b");
17    b.year = 2005;
18
19    a = b;
20
21    free(a.title);
22    free(b.title);
23 }
```

Структуры с полями-указателями

Стратегия так называемого «глубокого копирования» (англ., *deep coping*) подразумевает создание копий объектов, на которые ссылаются поля структуры.

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

        return 0;
    }

    return 1;
}
```

10

Структуры с полями-указателями

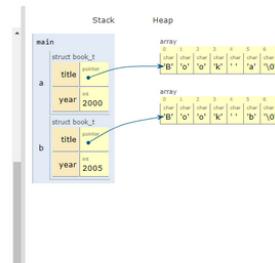
Стратегия «глубокого копирования».

До копирования

```
C (gcc 4.8, C11)
(Known limitations)

23 dst->year = src->year;
24
25 return 0;
26 }

27 int main(void)
28 {
29     struct book_t a = { 0 }, b = { 0 };
30     a.title = strdup("Book a");
31     a.year = 2000;
32
33     b.title = strdup("Book b");
34     b.year = 2005;
35
36     book_copy(&b, &a);
37
38     free(a.title);
39     free(b.title);
40 }
```

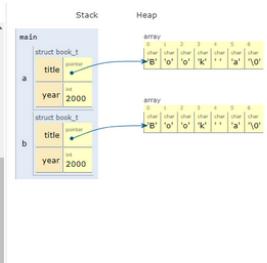


После копирования

```
C (gcc 4.8, C11)
(Known limitations)

23 dst->year = src->year;
24
25 return 0;
26 }

27 int main(void)
28 {
29     struct book_t a = { 0 }, b = { 0 };
30     a.title = strdup("Book a");
31     a.year = 2000;
32
33     b.title = strdup("Book b");
34     b.year = 2005;
35
36     book_copy(&b, &a);
37
38     free(a.title);
39     free(b.title);
40 }
```



Структуры с полями-указателями

```
struct book_t* book_create(const char *title, int year)
{
    struct book_t *pbook = malloc(sizeof(struct book_t));

    if (pbook)
    {
        pbook->title = strdup(title);
        if (pbook->title)
            pbook->year = year;
        else
        {
            free(pbook);
            pbook = NULL;
        }
    }

    return pbook;
}
```

```
    struct book_t *pbook = NULL;

    pbook = book_create("Book a", 2000);
    if (pbook)
    {
        // Работа с книгой

        // Корректно ли так освобождать память?
        free(pbook);
    }
```

12

17. Структуры переменного размера

- Где на практике такие структуры встречаются?
- Flexible Array Member:** особенности поля, особенности структур, где это поле описываются, как обходились до создания FAM в C99
- Сравнить структуру с FAM и с указателем



Структуры переменного размера

TLV (Type (или Tag) Length Value) - схема кодирования произвольных данных в некоторых телекоммуникационных протоколах.



Type – описание назначения данных.

Length – размер данных (обычно в байтах).

Value – данные.

Первые два поля имеют фиксированный размер.

Структуры переменного размера

TLV кодирование используется в:

- семействе протоколов TCP/IP
- спецификация PC/SC (smart cards)
- ASN.1
- ...

Структуры переменного размера

Преимущества TLV кодирования:

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

Flexible array member (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.



Flexible array member (C99)

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

    struct s *elem = malloc(sizeof(struct s) + n * sizeof(double));

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

    return elem;
}
```

Flexible array member до C99

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

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

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

    return elem;
}
```

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

7

Flexible array member vs pointer field

- Экономия памяти.
- Локальность данных (data locality).
- Атомарность выделения памяти.
- Не требует «глубокого» копирования и освобождения.



18. Динамически расширяемый массив

- Определение массива
- Тип данных, с помощью которого определяется динамически расширяемый массив
- Объяснить почему память нужно выделять относительно крупными блоками
- Рассмотреть добавление элемента в массив, освобождение памяти, особенности использования такого массива

Массив — последовательность элементов одного типа, расположенных в памяти друг за другом.



Ошибки при использовании realloc

Неправильно

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

Правильно

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

3

Ошибки при использовании realloc

```
int* select_positive(const int *a, int n, int *k)
{
    int m = 0;
    int *p = NULL;

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

    *k = m;
    return p;
}
```

Динамически расширяемые массивы

- Для уменьшения потерь при распределении памяти изменение размера должно происходить относительно крупными блоками.
- Для простоты реализации указатель на выделенную память должен храниться вместе со всей информацией, необходимой для управления динамическим массивом.

Динамически расширяемый массив

```
struct dyn_array_t
{
    int             *data;
    size_t          len;
    size_t          allocated;
};

#define DA_INIT_SIZE     1
#define DA_STEP          2

void da_init(struct dyn_array_t *parr)
{
    parr->data = NULL;
    parr->len = 0;
    parr->allocated = 0;
}
```

Добавление элемента

```
int da_append(struct dyn_array_t *parr, int item)
{
    if (!parr->data)
    {
        parr->data = malloc(DA_INIT_SIZE * sizeof(parr->data[0]));
        if (!parr->data)
            return DA_ERR_MEM;
        parr->allocated = DA_INIT_SIZE;
    }
    else
        if (parr->len >= parr->allocated)
    {
        void *tmp = realloc(parr->data, parr->allocated *
                            DA_STEP * sizeof(parr->data[0]));
        if (!tmp)
            return DA_ERR_MEM;
        parr->data = tmp;
        parr->allocated *= DA_STEP;
    }
    parr->data[parr->len] = item;
    parr->len++;
    return DA_OK;
}
```

7

Удаление элемента

```
int da_delete(struct dyn_array_t *parr, size_t index)
{
    if (index >= parr->len)
        return DA_ERR_RANGE;

    memmove(parr->data + index, parr->data + index + 1,
            (parr->len - index - 1) * sizeof(parr->data[0]));
    parr->len--;
    return DA_OK;
}
```

Удаление элемента: на что обратить внимание

- Важен ли порядок элементов в массиве?
 - Нет: на место удаляемого записать последний.
 - Да: сдвинуть элементы за удаляемым вперед.
- for, memcpу или memmove?
 - for
 - memcpу НЕЛЬЗЯ (как и strcpy), memmove надежнее.
- А нужно ли удалять элементы?

Достоинства и недостатки массивов

«+»

- Простота использования.
- Константное время доступа к любому элементу.
- Не тратят лишние ресурсы.
- Хорошо сочетаются с двоичным поиском.

«-»

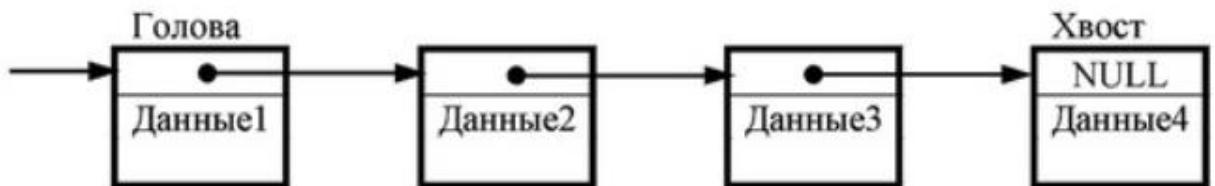
- Хранение меняющегося набора значений.

19-21. Линейный односвязный список. Добавление элемента, удаление элемента, вставка, обход

- Определение линейного односвязного списка, определение узла
- Сравнение линейного односвязного списка и массива
- Функции (добавление в начало и конец, вставка перед после)
- Выделение и освобождение памяти

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

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



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

Узел – единица хранения данных, несущая в себе ссылки на связанные с ней узлы.

Узел обычно состоит из двух частей

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

Основное преимущество связных списков перед массивами заключается в возможности эффективного изменения расположения элементов.

За эту гибкость приходиться жертвовать скоростью доступа к произвольному элементу списка, поскольку единственный способ получения элемента состоит в отслеживании связей от начала списка.



Основа для сравнения	массив	Связанный список
основной	Это последовательный набор фиксированного количества элементов данных.	Это упорядоченный набор, содержащий переменное количество элементов данных.
Размер	Указано во время декларации.	Не нужно указывать; расти и сжиматься во время исполнения.
Распределение памяти	Расположение элемента выделяется во время компиляции.	Положение элемента назначается во время выполнения.
Порядок элементов	Хранится последовательно	Хранится в случайному порядке
Доступ к элементу	Прямой или случайный доступ, т. Е. Указание индекса массива или индекса.	Последовательный доступ, т. Е. Traverse, начиная с первого узла в списке по указателю.
Вставка и удаление элемента	Медленно, поскольку требуется смещение.	Легче, быстрее и эффективнее.
поиск	Бинарный поиск и линейный поиск	линейный поиск
Требуется память	Меньше	Больше
Использование памяти	неэффективный	эффективное

```
typedef struct node node_t;

struct node
{
    void *data;
    struct node *next;
}
```

Создание	Добавление в конец
<pre>node_t* create_node(void *data) { node_t* node = (node_t*) malloc(sizeof(node_t)); if (node) { node->data = data; node->next = NULL; } return node; }</pre>	<pre>void add_node(node_t** list, node_t* new_node) { if (*list == NULL) *list = new_node; else { node_t* current = *list; while (current->next != NULL) current = current->next; current->next = new_node; } }</pre>
Удаление	Вставка перед другим узлом
<pre>void delete_node(node_t** list, node_t* delete_node) { if (*list == NULL delete_node == NULL) return; // Если удаляемый узел - первый узел if (*list == delete_node) { *list = delete_node->next; free(delete_node); return; } }</pre>	<pre>void insert_node(node_t* list, node_t* new_node) { if (list == NULL new_node == NULL) return; node_t* current = list; node_t* previous = NULL; while (current != NULL) { if (current == insert_before) { if (previous == NULL) *list = new_node; else previous->next = new_node; new_node->next = current; return; } previous = current; current = current->next; } }</pre>

<pre> } // Поиск удаляемого узла node_t* current = *list; while (current->next != NULL && current->next != delete_node) current = current->next; // Удаление узла из списка if (current->next == delete_node) { current->next = delete_node->next; free(delete_node); } } </pre>	<pre> { new_node->next = current; list = new_node; } else { new_node->next = current; previous->next = new_node; } return; } previous = current; current = current->next; } } </pre>
<p>Обход</p> <pre> void traverse_list(node_t* list, void (*modify_node)(node_t*)) { node_t* current = list; while (current != NULL) { modify_node(current); current = current->next; } } </pre>	<p>Освобождение</p> <pre> void free_list(node_t* list) { node_t* current = list; while (current != NULL) { node_t* next = current->next; free(current); current = next; } } </pre>

22-25. Двоичное дерево поиска. Добавление элемента, поиск элемента, обход, удаление элемента

- Определение двоичного дерева поиска
- В чем отличие ДДП от обычного дерева
- Описание узла дерева (выделение освобождение памяти)
- Добавление и поиск – рекурсивная и итерационная реализация
- Обход с указателем на функцию
- Доп. баллы: язык DOT

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;
}

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



Добавление рекурсивное

```
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;
}
```



Добавление итерационное

```
void add_node(struct tree_node **root,
              const char *name) {

    struct tree_node *new_node = (struct
        tree_node*) malloc(sizeof(struct
        tree_node));

    new_node->name = name;
    new_node->left = NULL;
    new_node->right = NULL;

    if (*root == NULL) {
        *root = new_node;
        return;
    }

    struct tree_node *current = *root;
    struct tree_node *parent = NULL;

    while (current != NULL) {
        parent = current;
        if (strcmp(name, current->name) <
            0) {
            current = current->left;
        }
        else {
            current = current->right;
        }
    }
}
```

```

        if (strcmp(name, parent->name) < 0)
            parent->left = new_node;
        }
        else {
            parent->right = new_node;
        }
    }
}

```

Поиск рекурсивный

```

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

```

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);
}

```

```

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;
}

```

Удаление

```

void delete_node(struct tree_node
**root, const char *name)
{
    // проверяем, не пустое ли дерево
    if (*root == NULL)
    {
        printf("Дерево пустое.\n");
        return;
    }
}

```

```

struct tree_node *parent = NULL;
struct tree_node *current = *root;

```

```

// ищем узел с заданным именем
while (current != NULL &&
strcmp(current->name, name) != 0)
{
    parent = current;

    if (strcmp(name, current->name) <
0)
    {
        current = current->left;
    }
    else
    {
        current = current->right;
    }
}

// если не нашли узел с заданным
именем
if (current == NULL)
{
    printf("Узел с именем %s не
найден.\n", name);
    return;
}

// случай 1: узел без потомков
(листовой узел)

```

```

if (current->left == NULL &&
    current->right == NULL)
{
    // проверяем, является ли узел
    // корнем дерева
    if (parent == NULL)
    {
        *root = NULL;
    }
    else if (current == parent->left)
    {
        parent->left = NULL;
    }
    else
    {
        parent->right = NULL;
    }

    free(current);
    return;
}

// случай 2: узел имеет только
// одного потомка
if (current->left == NULL || current-
>right == NULL)
{
    struct tree_node *child = NULL;
}

```

```
if (current->left != NULL)
{
    child = current->left;
}
else
{
    child = current->right;
}

// проверяем, является ли узел
корнем дерева
if (parent == NULL)
{
    *root = child;
}
else if (current == parent->left)
{
    parent->left = child;
}
else
{
    parent->right = child;
}

free(current);
return;
}
```

```
// случай 3: узел имеет двух
ПОТОМКОВ

// находим наименьший элемент в
правом поддереве

struct tree_node *successor = current-
>right;

struct tree_node *successor_parent =
current;

while (successor->left != NULL)
{
    successor_parent = successor;
    successor = successor->left;
}

// копируем содержимое
наименьшего элемента в текущий узел
current->name = successor->name;

// удаление наименьшего элемента в
правом поддереве
if (successor_parent->left ==
successor)
{
    successor_parent->left = successor-
>right;
}
else
{
```

```

successor_parent->right =
successor->right;
}

free(successor);
}

```

26-27. Куча в программе на Си. Алгоритм работы функции malloc и free,



пример реализации

- Происхождение термина куча (необязательно)
- Для чего нужна куча, преимущества и недостатки динамической памяти
- Гарантии относительно выделенного блока памяти даются программисту
- Алгоритм работы функции malloc, пример реализации / Алгоритм работы функции free, пример реализации
- Что такое дефрагментация?

Куча как структура данных – не просто куча. Термин возник из противопоставления стека как структуры данных, которая реализует определенный алгоритм обслуживания, и кучи, которая представляет просто набор чего-то (в нашем случае – областей памяти).

Происхождение термина «куча»

Согласно Дональду Кнуту, «Several authors began about 1975 to call the pool of available memory a "heap."».



В стеке элементы расположены один над другим.



В куче нет определенного порядка в расположении элементов.

Некоторые авторы начали около 1975 года называть пул доступной памяти "кучей".

- + память можно выделить сколько нужно
 - + можно увеличить в процессе реализации
 - + память ограничена только размером оперативной памяти
-
- вся ответственность на программиста ложится
 - возможно потери памяти (утечки,...)
 - возможная фрагментация памяти
 - выравнивание(((



Для хранения данных используется «куча».

Создать переменную в «куче» нельзя, но можно выделить память под нее.

“+”

Все «минусы» локальных переменных.

“_”

Ручное управление временем жизни.



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

Реализация malloc/free

- Для моделирования области памяти, используемой под кучу, воспользуемся одномерным массивом.



- Пусть программист уже выделил 100 байт и хочет выделить еще 32 байта.



- Нельзя использовать 100 байт, которые уже были выделены, и еще не были освобождены.
- Начиная с какого места можно выделять память?
- Как найти нужный блок после выделения (например, чтобы освободить)?

Необходимо вести учет выделенных и свободных областей, но где хранить эти данные?

- Мы не можем воспользоваться malloc, потому что сами реализуем эту функцию :(
- Но мы можем выделить область чуть больше, чем нужно, и в ее начале расположить необходимые данные.



Какие сведения об области нам нужны?

- Размер.
- Состояния (выделена/свободна).
- Где находится следующая область?

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

```

#define MY_HEAP_SIZE 1000000

// пространство под "кучу"
static char my_heap[MY_HEAP_SIZE];

// список свободных/занятых областей
static struct block_t *free_list = (struct block_t*) my_heap;

// начальная инициализация списка свободных/занятых областей
static void initialize(void)
{
    free_list->size = sizeof(my_heap) - sizeof(struct block_t);
    free_list->free = 1;
    free_list->next = NULL;
}

```

Выделение области памяти (malloc)

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

```

void* my_malloc(size_t size)
{
    struct block_t *cur;
    void *result;

    if (!free_list->size)
        initialize();

    cur = free_list;
    while (cur && (cur->free == 0 || cur->size < size))
        cur = cur->next;

    if (!cur)
    {
        result = NULL;
    }
    else
    {
        printf("Out of memory\n");
        if (cur->size == size)
        {
            cur->free = 0;
            result = (void*) (++cur);
        }
        else
        {
            split_block(cur, size);
            result = (void*) (++cur);
        }
    }
}

return result;
}

```

```

static void split_block(struct block_t *block, size_t size)
{
    size_t rest = block->size - size;

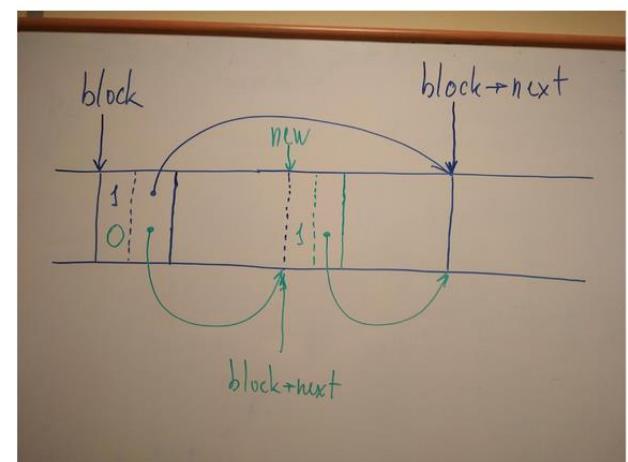
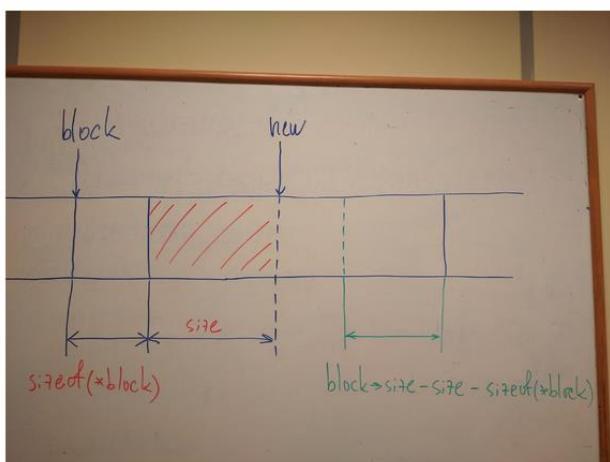
    if (rest > sizeof(struct block_t))
    {
        struct block_t *new = (void*)((char*)block + size + sizeof(struct block_t));

        new->size = block->size - size - sizeof(struct block_t);
        new->free = 1;
        new->next = block->next;

        block->size = size;
        block->free = 0;
        block->next = new;
    }
    else
        block->free = 0;
}

```

12



Освобождение области памяти (free)

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

```

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

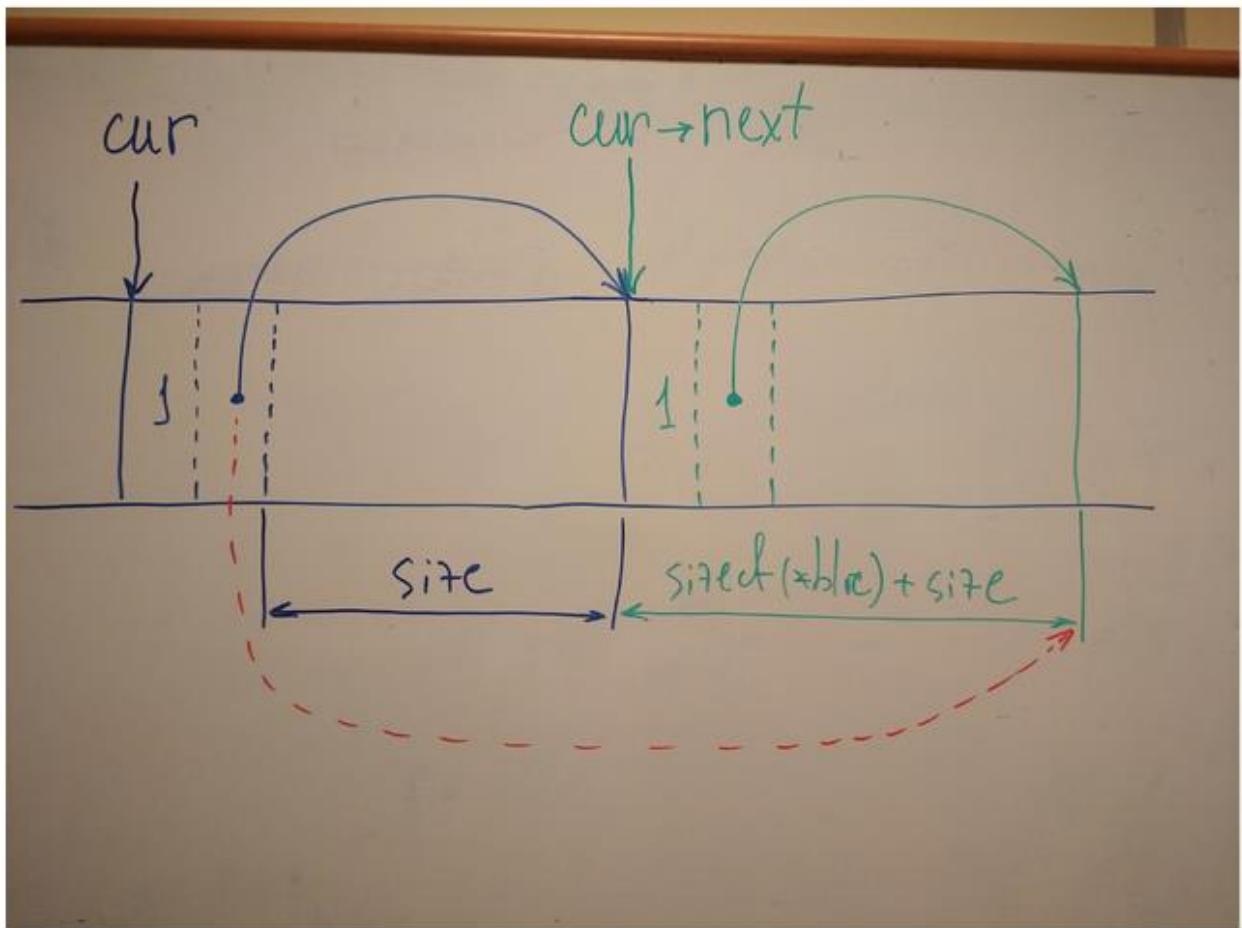
        --cur;
        cur->free = 1;

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

static void merge_blocks(void)
{
    struct block_t *cur = free_list;

    while (cur && cur->next != NULL)
    {
        if (cur->free && cur->next->free)
        {
            cur->size += cur->next->size + sizeof(struct block_t);
            cur->next = cur->next->next;
        }
        else
            cur = cur->next;
    }
}

```



- Выравнивание.
- Фрагментация.
- Возможность увеличения области, отведенной под кучу.

Выравнивание данных

По Кернигану, Ритчи

Для хранения произвольных объектов блок должен быть правильно выровнен. В каждой системе есть самый «требовательный» тип данных – если элемент этого типа можно поместить по некоторому адресу, то любые другие элементы тоже можно поместить туда.

```
// По Кернигану, Ритчи
typedef long align_t;

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

    align_t x;
};


```

Запрашиваемый размер области обычно округляется до размера кратного размеру заголовка.

```
n_blocks = (size + sizeof(union block_t) - 1) /  
           sizeof(union block_t) + 1;
```

```
alloc_size = n_blocks* sizeof(sizeof(union block_t));
```

Фрагментация – чередование участков памяти при последовательных запросах на выделение и освобождение памяти. «Занятые» участки чередуются со «свободными» - однако последние могут быть недостаточно большими для того, чтобы сохранить в них нужное данное.

Фрагментация



Размер «кучи» 1000 байт. 600 байт занято. Пользователю нужно выделить область в 400 байт :(



Дефрагментация в куче выполняется для устранения фрагментации и улучшения производительности выделения и освобождения блоков памяти. В процессе дефрагментации происходит перераспределение блоков памяти таким образом, чтобы создать большие непрерывные свободные блоки и уменьшить фрагментацию.

28. Куча в программе на Си. Проблемы выравнивания выделенной области памяти. Пример реализации

- Происхождение термина куча (необязательно)
- Для чего нужна куча, преимущества и недостатки динамической памяти

- Гарантии относительно выделенного блока памяти даются программисту
- Выравнивание памяти

Выравнивание данных

По Кернигану, Ритчи

Для хранения произвольных объектов блок должен быть правильно выровнен. В каждой системе есть самый «требовательный» тип данных – если элемент этого типа можно поместить по некоторому адресу, то любые другие элементы тоже можно поместить туда.

- 1) Определяем самый требовательный тип (теперь уже long long)
- 2) Заводим объединение – структура будет выровнена по максимальному элементу, и поле align_t- объединение будет выровнено поциальному адресу – самого требовательного типа.

-
- 3) Чтобы все были выровнены одинаковыми, размер выделенной памяти делают кратной размеру метаинформации – размер области в блоках – затем в байтах, но правильно

Реализация malloc/free: недоделки и т.п.



// По Кернигану, Ритчи <pre>typedef long align_t; union block_t { struct { size_t size; int free; union block_t *next; } block; align_t x; };</pre>	Запрашиваемый размер области обычно округляется до размера кратного размеру заголовка. <pre>n_blocks = (size + sizeof(union block_t) - 1) / sizeof(union block_t) + 1; alloc_size = n_blocks * sizeof(sizeof(union block_t));</pre>
--	--

Выравнивание alignment — размещение значений в памяти по адресам, кратным некоторому целому числу, большему единицы.

Причина, по которой существует такое понятие как выравнивание, заключается в том, что процессорам проще оперировать выровненными значениями.

Естественное выравнивание natural alignment — выравнивание значений встроенных типов (как правило, поддерживаемых процессором непосредственно) по адресам, кратным размеру этого типа. Например, 4-байтные целые размещаются по адресам, кратным четырём (0, 4, 8, 12, ...), а 8-байтные значения типа double размещаются по адресам, кратным восьми (0, 8, 16, 24, ...).

29. Массив переменной длины. Функция alloca.

В стандарте C99 появилась возможность создавать массивы, размер которых не известен на момент компиляции. Массивы произвольной длины нельзя инициализировать при создании.

The screenshot shows a code editor with a dark theme. The code is as follows:

```
3  
4     int main()  
5     {  
6         printf("va_list %d\n\n", (int)sizeof(va_list));  
7         int n = 5;  
8         int a[n];  
9  
10        for (int i = 0; i < n; ++i)  
11        {  
12            a[i] = i + 1;  
13        }  
14        for (int i = 0; i < n; ++i)  
15        {  
16            printf("%d ", a[i]);  
17        }  
18        printf("\n\n");  
19        n = 10;  
20        for (int i = 0; i < n; ++i)  
21        {  
22            printf("%d ", a[i]);  
23        }  
24        printf("\n\n");  
25    }
```

Below the code, there are tabs for PROBLEMS, OUTPUT, DEBUG CONSOLE, and TERMINAL. The TERMINAL tab is selected. The output window shows:

```
va_list 24  
1 2 3 4 5  
1 2 3 4 5 8 1431654857 21845 1431658368 21845
```

Размер статического массива задается константой, значение которой должно быть известно на этапе компиляции.
Размер массива переменной длины можно задавать с помощью выражения (то есть известно во время выполнения) – массив на стеке, размер которого задается переменной во время выполнения

Массивы произвольной длины нельзя инициализировать при создании. ААА статические можно

размер статики должен знать компилятор
а размер VLA компилятор может и не знать

Variable Length Array



- 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.

Операцию `sizeof`: до введения массива переменной длины она выполнялась только во время компиляции, то есть в исполняемом файле ее было невозможно увидеть. С появлением же эту операцию пришлось реализовывать, и она выполняется в runtime.

alloca	alloca
<pre>#include <alloca.h> void* alloca(size_t size);</pre> <p>Функция <code>alloca</code> выделяет область памяти, размером <code>size</code> байт, на стеке. Функция возвращает указатель на начало выделенной области. Эта область автоматически освобождается, когда функция, которая вызвала <code>alloca</code>, возвращает управления вызывающей стороне.</p> <p>Если выделение вызывает переполнение стека, поведение программы не определено.</p>	<pre>#include <alloca.h> #include <stdio.h> int main(void) { int n; printf("n: "); scanf("%d", &n); int *a = alloca(n * sizeof(int)); for (int i = 0; i < n; i++) a[i] = i; } for (int i = 0; i < n; i++) printf("%d ", a[i]); return 0;</pre>



alloca

“+”

- Выделение происходит быстро.
 - Выделенная область освобождается автоматически.
- “-”
- Функция *нестандартная*.
 - Серьезные ограничения по размеру области.

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

```
void foo(int size) {  
    ...  
    while(b){  
        char* tmp = alloca(size);  
        ...  
    }  
} 6
```

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

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

VLA, когда тело цикла закончится, массив разрушится
(переменная вышла из области видимости)

выделяется, но память останется до конца работы
функции и легко получить переполнение стека

30. Функции с переменным числом параметров

- Идея реализации таких функций (с адресами), сказать, что так делать нельзя
- Рассказать, как нужно делать (stdarg.h, соответствующие макросы)
(примеры)
- Как написать функцию аналогичную функции printf с помощью
стандартной библиотеки (журналирование?)

Функции с переменным числом параметров

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

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

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

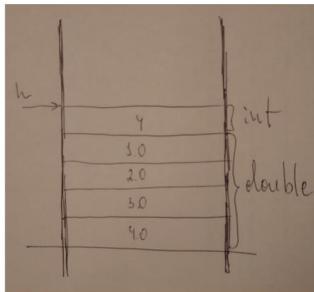
```
int f(int k, ...);
```

Функции с переменным числом параметров

```
#include <stdio.h>

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

int main(void)
{
    double a =
        avg(4, 1.0, 2.0, 3.0, 4.0);
    printf("a = %5.2f\n", a);
    return 0;
}
```

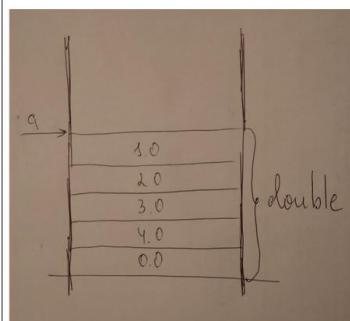


Функции с переменным числом параметров

```
#include <stdio.h>

double avg(double a, ...)
{
    ...
}

int main(void)
{
    double a =
        avg(1.0, 2.0, 3.0,
            4.0, 0.0);
    printf("a = %5.2f\n", a);
    return 0;
}
```



Функции с переменным числом параметров

Напишем функцию, вычисляющую среднее арифметическое своих аргументов.

Проблемы:

1. Как определить адрес параметров в стеке?
2. Как перебирать параметры?
3. Как закончить перебор?

Функции с переменным числом параметров

```
#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;
}
```

```
int main(void)
{
    double a =
        avg(4, 1.0, 2.0, 3.0, 4.0);
    printf("a = %5.2f\n", a);
    return 0;
}
```

5

Функции с переменным числом параметров

Функции с переменным числом параметров

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

#define EPS 1.0e-7

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

    while (fabs(*p_d) > EPS)
    {
        sum += *p_d;
        n++;
        p_d++;
    }
}
```

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

int main(void)
{
    double a =
        avg(1.0, 2.0, 3.0,
            4.0, 0.0);
    printf("a = %5.2f\n", a);
    return 0;
}
```

7

```

ну тут везде ужас будет....
a = 1573514575233966972237195359288760380972472279913449082235021778809018772470
04529276433664423640707602727010215573117396501427584766005437943602912193338752
04194488715466703621652715358524150160500708221284734838069061675201868577084940
956179195900624474542167813474963248474347677001464342763983916236800.00
irina@irina-X541UVK:~/cprog/github/exam/sizee$ ./app.exe
0.000000
4
0.000000
0.000000
-0.000000
0.000000
a = -0.00
irina@irina-X541UVK:~/cprog/github/exam/sizee$ ./app.exe
0.000000
4
0.000000
0.000000
269262843599374796973932544.000000
0.000000
a = 67315710899843699243483136.00

```

```

#include <stdio.h>

void print_ch(int n, ...)
{
    int *p_i = &n;
    char *p_c = (char*) (p_i+1);

    for (int i = 0; i < n; i++, p_c++)
        printf("%c %d\n", *p_c, (int) *p_c);
}

int main(void)
{
    print_ch(5, 'a', 'b', 'c', 'd', 'e');
    return 0;
}

```

Объяснение:

- 1) `Sizeof('a')=sizeof(int)` из-за неявного расширения целочисленных типов – если тип меньше `int`, то в выражениях он расширяется до типа `int` из-за машинного слова – единицы памяти, которой оперирует большинство команд, чтобы все эффективно работало. И когда мы передаем все в функцию, каждый символ расширяется до целого, и в стек помещаются целые числа.
- 2) Команда `push`, с помощью которой все помещается в стек также оперирует машинным словом.
- 3) В программе с `double` возможно из-за выравнивания

Вывод: эта идея хороша только как идея. Так писать вообще нельзя. Все сильно завязано на платформу и компилятор. Более того – что делать, если типы разные?

Стандартный способ работы с параметрами функций с переменным числом параметров

`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)`

Функции с переменным числом параметров

```

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

double avg(int n, ...)
{
    va_list vl;
    double sum = 0, num;

    if (!n)
        return 0.0;

    va_start(vl, n);

    for (int i = 0; i < n; i++)
    {
        num = va_arg(vl, double);
        printf("%f\n", num);
        sum += num;
    }

    va_end(vl);
    return sum /n;
}

int main(void)
{
    double a =
        avg(4, 1.0, 2.0, 3.0, 4.0);
    printf("a = %5.2f\n", a);
    return 0;
}

```

10



Функции с переменным числом параметров

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

#define EPS 1.0e-7

double avg(double a, ...)
{
    va_list vl;
    int n = 0;
    double num, sum = 0.0;

    va_start(vl, a);
    num = a;

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

    va_end(vl);

    if(!n)
        return 0;

    return sum / n;
}

int main(void)
{
    double a =
        avg(1.0, 2.0, 3.0,
            4.0, 0.0);

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

    return 0;
}
```

11

Общая процедура создания функции, которая имеет переменное число аргументов, заключается в следующем:
функция должна иметь один или более известных параметров. Эти известные параметры следуют перед списком переменных параметров.

Самый правый известный параметр называется `last_parm`.

Имя `last_parm` используется в качестве второго параметра в вызове `va_start()`. Прежде чем осуществлять доступ к какому-либо из переменных параметров, должен быть инициализирован указатель `argptr`, для чего используется вызов `va_start()`.

После этого параметры возвращаются с помощью вызова функции `va_arg()` с параметром `type`, являющимся типом следующего параметра.

Наконец, после того, как все параметры прочитаны, перед тем как выйти из функции, необходимо вызвать функцию `va_end()`, что гарантирует правильное восстановление стека. Если функция `va_end()` не вызвана, то возникает аварийная ситуация.



```

// 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__

```

13

Функции vprintf(), vfprintf() и vsprintf() функционально эквивалентны функциям printf(), fprintf() и sprintf() соответственно. Различие состоит лишь в том, что список аргументов заменен на указатель на список аргументов. Этот указатель должен иметь тип va_list, определенный в stdarg.h.

31-34. Препроцессор. Общие понятия. Директива include, простые макросы, предопределенные макросы / Макросы с параметрами / Директивы условной компиляции, директивы error и pragma / Операции # и

- Что такое препроцессор? В какой момент времени он имеет дело с программой, какие функции выполняет
- Работа препроцессора управляется директивами: на какие группы можно разделить директивы, какие правила справедливы для всех директив
- Директива include: чем использование двойных кавычек отличается от использования угловых скобок

- **Простые макросы:** как такие макросы обрабатываются препроцессором (примеры), для чего они используются
- Какие предопределенные макросы Вам известны, для чего используют
- **Макросы с параметрами:** как обрабатываются, сравнить их с функциями, упомянуть про макросы с переменным числом параметров, объяснить почему у макросов с параметрами нужно использовать круглые скобки (примеры), подходы к написанию длинных макросов
- **Условная компиляция:** для чего используются директивы условной компиляции, какие виды бывают (if vs ifdef), error, pragma
- **# и ##:** что это, примеры использования, особенности работы препроцессора при раскрытии макросов



Препроцессор лучше всего рассматривать как отдельную программу, которая выполняется перед компиляцией. При запуске программы, препроцессор просматривает код сверху вниз, файл за файлом, в поиске директив. **Директивы** — это специальные команды, которые начинаются с символа # и НЕ заканчиваются точкой с запятой. Есть несколько типов директив, которые мы рассмотрим ниже.

Препроцессор — это [компьютерная программа](#), принимающая данные на входе и выдающая данные, предназначенные для входа другой программы (например, [компилятора](#)). О данных на выходе препроцессора говорят, что они находятся в **препроцессированной** форме, пригодной для обработки последующими программами (компилятором).

4 основные задачи препроцессора:

- 1) Удаление комментариев
- 2) Включение файлов (include)
- 3) Текстовая замена (define)
- 4) Условная компиляция (например, include guard)



Макроопределения (макросы) – делятся на: простые макросы, макрос с параметрами, макросы с переменным числом параметров, предопределенные макросы

- #define, #undef

Директива включения файлов

- #include

Директивы условной компиляции

- #if, #ifndef, #ifdef, #endif и др.

Остальные директивы (#pragma, #error, #line и др.) используются реже.

Правила, справедливые для всех директив

- Директивы всегда начинаются с символа "#".
- Любое количество пробельных символов может разделять лексемы в директиве.
- Директива заканчивается на символе '\n'.
- Директивы могут появляться в любом месте программы.

Правила, справедливые для всех директив (пояснения)

- Любое количество пробельных символов могут разделять лексемы в директиве.

```
# define N 1000
```

- Директива заканчивается на символе '\n'.

```
#define DISK_CAPACITY (SIDES *  
    TRACKS_PER_SIDE *  
        SECTORS_PER_TRACK *  
            BYTES_PER_SECTOR)
```

Простые макросы

```
#define идентификатор список-замены
```

```
#define PI 3.14  
#define EOS '\0'  
#define MEM_ERR "Memory allocation error."
```

Используются:

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

Простые макросы (object-like macro), так как их использование похоже на использование обычных переменных

Препроцессор, когда просматривает текст программы, находит идентификатор, за который отвечает макрос и просто подставляет вместо него список замены.

Предопределенные макросы

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

- и др.

Эти идентификаторы нельзя переопределять или отменять директивой undef.

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

1 и 2 используются при журналировании

Журналирование в программировании - это процесс записи и анализа событий и

Простые макросы

Окончание предыдущего слайда.

- Незначительного изменения синтаксиса языка.

```
#define BEGIN {  
#define END }  
#define INF_LOOP for( ; ; )
```

- Переименования типов.

```
#define BOOL int
```

- Управления условной компиляцией.

Отличие между использованием двойных кавычек и угловых скобок в директиве include заключается в порядке поиска файла:

1. Двойные кавычки ("") - при использовании двойных кавычек компилятор будет сначала искать указанный файл в текущем каталоге проекта. Если файл не будет найден, компилятор перейдет к следующему шагу поиска.

<p>действий, происходящих в программе, с целью отслеживания и отладки ее работы</p>	<p>2. Угловые скобки (<>) - при использовании угловых скобок компилятор будет искать указанный файл в системных или стандартных каталогах компилятора. Такие каталоги могут содержать стандартные библиотеки или другие системные файлы. Если файл не будет найден в системных каталогах, компилятор выдаст ошибку.</p>
<h3>Макросы с параметрами</h3> <pre>#define идентификатор(x1, x2, ..., xn) список-замены – Не должно быть пробела между именем макроса и (. – Список параметров может быть пустым. #define MAX(x, y) ((x) > (y) ? (x) : (y)) #define IS_EVEN(x) ((x) % 2 == 0) Где-то в программе</pre>  <pre>i = MAX(j + k, m - n); // i = ((j + k) > (m - n) ? (j + k) : (m - n)); if (IS_EVEN(i)) // if (((i) % 2 == 0)) i++;</pre>	<h3>Макросы с переменным числом параметров (C99)</h3> <pre>#ifndef NDEBUG #define DBG_PRINT(s, ...) printf(s, __VA_ARGS__) #else #define DBG_PRINT(s, ...) ((void) 0) #endif</pre>
<p>Макросы с параметрами в языке программирования обычно требуют круглые скобки для указания границы параметров. Это необходимо для интерпретации правильно заданных параметров макроса.</p> <p>Например, предположим, что у нас есть макрос с одним параметром, который удваивает значение, переданное в него:</p> <pre>```c #define DOUBLE(x) ((x) * 2) ``` </pre>	<h3>Создание длинных макросов</h3> <pre>// 1 #define ECHO(s) {gets(s); puts(s);} if (echo_flag) ECHO(str); else gets(str); // 2 #define ECHO(s) (gets(s), puts(s)) ECHO(str); #define ECHO(s) \ do \ { \ gets(s); \ puts(s); \ } \ while(0)</pre>

Если мы вызываем этот макрос без круглых скобок, он будет работать неправильно:

```
```c
int a = 5;
int b = DOUBLE(a + 1); // Результат будет 7,
а не 12 как ожидается
```

```

В этом примере, без круглых скобок, макрос будет интерпретировать выражение `a + 1 * 2` как `a + (1 * 2)`, что даст 7 вместо ожидаемого результата 12. С использованием круглых скобок, макрос будет интерпретировать выражение правильно как `(a + 1) * 2`.

Условная компиляция

Использование условной компиляции:

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

Условная компиляция

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

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

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

Директива `error` - это инструкция, которая используется в языках программирования, чтобы вызвать ошибку компиляции или выполнения. Когда компилятор или

директива `pragma` - это инструкция, которая предоставляет информацию и настройки для компилятора или среды выполнения программы. Директива `pragma` используется

интерпретатор встречает директиву `error`, он немедленно прекращает сборку или выполнение программы и выводит сообщение об ошибке. Это может быть полезно, чтобы предотвратить компиляцию или выполнение кода, который имеет неправильное использование, отсутствующие зависимости или другие проблемы.

Пример использования директивы `error` на языке C++:

```
```cpp
#include <iostream>

#define VERSION 1

#ifndef VERSION
#if VERSION < 2
#error "This version is no longer supported.
Please update your code."
#endif
#endif

int main() {
 std::cout << "Hello, World!" << std::endl;
 return 0;
}
```

для управления компиляцией, оптимизацией или поведением программы в различных контекстах.

Пример использования директивы `pragma` на языке C++:

```
```cpp
#include <iostream>

#pragma GCC optimize("O3")
#pragma once

int main() {
    std::cout << "Hello, World!" << std::endl;
    return 0;
}
```

```

В этом примере использованы две директивы `pragma`. Первая `pragma GCC optimize("O3")` указывает компилятору GCC включить оптимизацию уровня 3 для данного файла. Такие оптимизации могут повысить производительность программы, но могут увеличить время компиляции. Вторая директива `pragma once` гарантирует, что файл будет включен только один раз в

|                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   | процессе компиляции, даже если он был включен из нескольких исходных файлов.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| Директива <code>if</code> в языке программирования позволяет проверить условие и выполнить определенный блок кода, если условие выполняется.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      | Директива <code>ifdef</code> (или <code>ifndef</code> ) также проверяет условие, но в зависимости от того, создана ли макроопределение с таким именем. Если оно создано (или не создано), то выполняется определенный блок кода.                                                                                                                                                                                                                                                                                                                                                      |
| <p><b>Преимущества директивы <code>if</code>:</b></p> <ol style="list-style-type: none"> <li>1. Более мощная и гибкая, поскольку позволяет использовать различные типы условий, такие как сравнения и логические выражения.</li> <li>2. Позволяет выполнять блок кода только в случае, когда условие истинно.</li> </ol> <p><b>Недостатки директивы <code>if</code>:</b></p> <ol style="list-style-type: none"> <li>1. Требует более сложного синтаксиса, поскольку требуется явно написать условие.</li> <li>2. Может привести к тому, что код становится более сложным и трудночитаемым из-за включения большого количества условий.</li> </ol> | <p><b>Преимущества директивы <code>ifdef</code>:</b></p> <ol style="list-style-type: none"> <li>1. Простой и понятный синтаксис, поскольку директива проверяет только наличие макроопределения.</li> <li>2. Легко использовать при работе с макросами и параметрами компиляции.</li> </ol> <p><b>Недостатки директивы <code>ifdef</code>:</b></p> <ol style="list-style-type: none"> <li>1. Ограниченнность в возможности проверки, поскольку проверяется только наличие макроопределения.</li> <li>2. Не позволяет использовать сложные логические условия или сравнения.</li> </ol> |

### Директива #if

```
1 #if value
2 // код, который выполнится, в случае, если value - истина
3 #elseif value1
4 // этот код выполнится, в случае, если value1 - истина
5 #else
6 // код, который выполнится в противном случае
7 #endif
```

Директива `#if` проверяет, является ли значение `value` истиной и, если это так, то выполняется код, который стоит до закрывающей директивы `#endif`. В противном случае, код внутри `#if` не будет компилироваться, он будет удален компилятором, но это не влияет на исходный код в исходнике.

Обратите внимание, что в `#if` могут быть вложенные директивы `#elseif` и `#else`. Ниже показан пример кода для комментирования блоков кода, используя следующую конструкцию:

```
1 #if 0
2 // код, который необходимо закомментировать
3 #endif
```

Если у вас в программе есть блоки кода, которые содержат многострочные комментарии и вам требуется обернуть полностью этот блок кода в комментарий — ничего не получится, если вы воспользуетесь `/*многострочный комментарий*/`. Другое дело — конструкция директив `#if #endif`.

### «Операция» #

«Операция» # конвертирует аргумент макроса в строковый литерал.

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

#define TEST(condition, ...) ((condition) ? \
 printf("Passed test %s\n", #condition) : \
 printf(_VA_ARGS_))
```

Где-то в программе

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

TEST(voltage <= max_voltage,
 "Voltage %d exceed %d", voltage, max_voltage);
```

### Директива #ifdef

```
1 #ifdef nameToken
2 // код, который выполнится, если nameToken определен
3 #else
4 // код, который выполнится, если nameToken не определен
5 #endif
```

Директива `#ifdef` проверяет, был ли ранее определен макрос или символьическая константа как `#define`. Если — да, компилятор включает в программу код, который находится между директивами `#ifdef` и `#else`, если `nameToken` ранее определен не был, то выполняется код между `#else` и `#endif`, или, если нет директивы `#else`, компилятор сразу переходит к `#endif`. Например, макрос `_cpp` определен в C++, но не в Си. Вы можете использовать этот факт для смешивания С и C++ кода, используя директиву `#ifdef`:

```
1 #ifdef _cpp
2 // C++ код
3 #else
4 // Си код
5 #endif
```

### «Операция» ##

«Операция» ## объединяет две лексемы в одну.

```
#define MK_ID(n) i##n
```

Где-то в программе

```
int MK_ID(1), MK_ID(2);
// int i1, i2;
```

Более содержательный пример

```
#define GENERAL_MAX(type)
type type##_max(type x, type y)
{
 return x > y ? x : y;
}
```

# Шаги обработки макроса с параметрами (6.10.3.4)

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

## 35. Встраиваемые функции.



- Что такое ключевое слово inline? Почему появилось в языке?
- Сравнить встраиваемые функции с макросами / с обычными функциями
- Способы борьбы с ошибкой undefined reference при использовании inline

### inline-функции (C99)

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.

### inline-функции (C99)

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
```

|                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |                                                                                                                                                                                                                                                                                                                                                                                                                    |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <h2>6.4.7 Function specifiers</h2> <p>6 ... An inline definition <i>does not provide an external definition</i> for the function, and <i>does not forbid an external definition</i> in another translation unit. An inline definition provides an alternative to an external definition, which a translator <i>may use</i> to implement any call to the function in the same translation unit. It is unspecified whether a call to the function uses the inline definition or the external definition.</p> <ul style="list-style-type: none"> <li>inline-реализация не предоставляет и не запрещает реализацию со внешней линковкой.</li> <li>Транслятор волен сам выбирать.</li> </ul> | <p>6 ... Встроенное определение не предоставляет внешнего определения для функции и не запрещает внешнее определение в другом блоке трансляции. Встроенное определение предоставляет альтернативу внешнему определению, которое транслятор может использовать для реализации вызова функции в том же блоке трансляции. Не указано, использует ли вызов функции встроенное определение или внешнее определение.</p> |
| <p><b>Способы исправления проблемы «unresolved reference»</b></p> <ul style="list-style-type: none"> <li>Использовать ключевое слово static</li> </ul> <pre>static inline int add(int a, int b) {return a + b; }  int main(void) {     int i = add(4, 5);      return i; }</pre> <p>Такая функция доступна только в текущей единице трансляции.</p>                                                                                                                                                                                                                                                                                                                                     | <p><b>Способы исправления проблемы «unresolved reference»</b></p> <ul style="list-style-type: none"> <li>Использовать ключевое слово extern</li> </ul> <pre>extern inline int add(int a, int b) {return a + b; }  int main(void) {     int i = add(4, 5);      return i; }</pre> <p>Такая функция доступна из других единиц трансляции.</p>                                                                        |
| <p><b>Способы исправления проблемы «unresolved reference»</b></p> <ul style="list-style-type: none"> <li>Добавить еще одно <b>такое же не-inline</b> определение функции <b>где-нибудь</b> в программе.</li> </ul> <p>Самый <b>плохой</b> способ решения проблемы, потому что реализаций могут не совпасть.</p>                                                                                                                                                                                                                                                                                                                                                                         | <p><b>Способы исправления проблемы «unresolved reference»</b></p> <ul style="list-style-type: none"> <li>Убрать ключевое слово inline из определения функции.</li> </ul> <pre>int add(int a, int b) {return a + b; }  int main(void) {     int i = add(4, 5);      return i; }</pre> <p>Компилятор «умный» :), сам разберется.</p>                                                                                 |
| <p>Встраиваемые функции:</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            | <p>Макросы:</p>                                                                                                                                                                                                                                                                                                                                                                                                    |

|                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"> <li>- Встраивание функции означает, что код функции вставляется непосредственно в вызывающий код во время компиляции. Это позволяет избежать накладных расходов на вызов функции, так как нет необходимости сохранять и восстанавливать контекст выполнения функции при ее вызове.</li> <li>- Встраиваемые функции могут приводить к более быстрому выполнению программы, поскольку их код выполняется прямо в вызывающем коде. Это особенно полезно для небольших функций, вызываемых множеством раз.</li> <li>- Однако, использование встраиваемых функций может привести к увеличению размера программы, поскольку код функции будет размещен в каждом месте вызова функции.</li> </ul> | <ul style="list-style-type: none"> <li>- Макросы представляют собой подстановки текста, которые выполняются препроцессором до этапа компиляции. Макросы могут быть использованы для определения констант, выполнения простых операций или создания гибких шаблонов кода.</li> <li>- Макросы обычно имеют компактный размер, поскольку они заменяются простым текстом во время препроцессинга. Однако, некорректное использование макросов может приводить к ошибкам и неожиданному поведению.</li> <li>- Макросы могут быть полезными в случаях, когда требуется выполнить простые операции без вызова функций или когда требуется сгенерировать код в зависимости от аргументов.</li> </ul> |
| <p><b>Обычные функции:</b></p> <ul style="list-style-type: none"> <li>- Обычные функции являются наиболее универсальным способом организации кода в С. Они предлагают абстракцию и повторное использование кода.</li> </ul>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |

- Вызов обычных функций требует сохранения контекста выполнения и перехода к вызываемой функции, что может привести к накладным расходам по сравнению с встраиванием функции.
- Обычные функции полезны для крупных и сложных операций, требующих повторного использования и модульности кода.

До стандарта C99, если функция имела класс памяти `static` в заголовочном файле, то она заменялась в файле на свое тело. Это привело к проблеме большого увеличения размера исполняемого файла. Поэтому было добавлено ключевое слово `inline`, которое уже и советует компилятору заменить вызов функции на ее тело.

### **36-41. Библиотеки. Статические библиотеки / Динамические библиотеки, динамическая компоновка / Динамические библиотеки, динамическая загрузка / Динамические библиотеки, подходы к реализации функций, которым требуется создавать буфер динамически / Динамическая библиотека на Си, приложение на Python (ctypes) / Динамическая библиотека на Си, приложение на Python (модуль расширения)**

- Что такое библиотека? Какие функции обычно выносятся в библиотеку? В какой форме распространяются библиотеки? Какие виды библиотек знаете, какими преимуществами и недостатками эти виды обладают
- **Статические библиотеки:** как собрать статическую библиотеку, как собрать приложение, которое эту статическую библиотеку использует,
- **Динамические библиотеки:** как собрать динамическую библиотеку, какие особенности сборки динамической библиотеки есть в зависимости от ОС (в Windows иначе), как оформляется код приложения при использовании дин. библиотеки, сравнить динамическую компоновку и динамическую загрузку
  - **Динамическая загрузка:** в каждой ОС есть API, которое позволяет загрузить динамическую библиотеку в память, это API предусматривает функцию, которая позволяет найти в библиотеке интересующую Вас функцию, это же API

предусматривает функцию, с помощью которой библиотеку позже можно из памяти выгрузить

- **Подходы к реализации функций, которым требуется создавать буфер динамически:** 1) реализация функций внутри библиотеки выделения памяти и освобождения памяти; 2) переложение вопросов выделения и освобождения памяти на вызывающую сторону (предусмотреть механизм сообщения вызывающей стороне размер того буфера, который требуется программе)
- **Приложение на Python:** рассказать о проблемах, которые возникают, когда библиотека на одном языке, а приложение на другом (необходимо библиотеку загрузить, спроектировать типы одного языка на другой), как эти проблемы решить (ctypes / модуль расширений)
  - **Общий подход использования ctypes на примере сложения и деления:** как загрузить библиотеку, как написать функцию обертку, когда ее можно не писать, когда нужно писать
  - **Общий подход использования ctypes на примере обработки массивов:** в деталях изложить как реализовать обработку массивов с помощью этого модуля (функции которые принимают массив только для чтения / функции которые принимают массив только для записи / функции которые принимают массив как для чтения так и на запись)
  - **Организация модуля расширений**
  - **Общий подход использования модуля расширений на примере обработки массивов**
- Ключи –l и –L и особенности именования библиотек в Linux

**Библиотека – это набор специальным образом оформленных объектных файлов**

**Библиотека** – сборник подпрограмм или объектов, используемых для разработки программного обеспечения (ПО). В некоторых языках **программирования** (например, в

Python) то же, что и модуль, в некоторых – несколько модулей.

*Библиотека включает в себя*

- заголовочный файл;
- откомпилированный файл самой библиотеки:
  - библиотеки меняются редко – нет причин перекомпилировать каждый раз;
  - двоичный код предотвращает доступ к исходному коду.

**Библиотека – это набор специальным образом оформленных объектных файлов – в виде объектных файлов.**

В языке программирования С код библиотек представляет собой функции, размещенные в файлах, которые скомпилированы в объектные файлы, а те, в свою очередь, объединены в библиотеки. В одной библиотеке объединяются функции, решающие определенный тип задач. Например, существует библиотека математических функций.

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

Библиотека содержит функции и описания объектов (структур), связанные логически (libarray, libmatrix и т.д)

Библиотека распространяется в виде – файл с двоичным кодом библиотеки (объектный) + заголовочный файл библиотеки

Библиотеки делятся на

- **СТАТИЧЕСКИЕ** - Связываются с программой в момент компоновки. Код библиотеки помещается в исполняемый файл. Статическая библиотека по сути – архив из объектных файлов. Их придумали, чтобы было удобно таскать кучу объектных файлов
- **ДИНАМИЧЕСКИЕ** - Подпрограммы из библиотеки загружаются в приложение во время выполнения. Код библиотеки не помещается в исполняемый файл. Примечание: Компоновщик, таблица импорта, она содержит сведения о том, что программа использует такую функцию, и она находится в такой библиотеке – это в исполняемый файл попадает, но сам код – нет.

**Статические** - Связываются с программой в момент компоновки. Код библиотеки помещается в исполняемый файл. Статическая библиотека – архив из объектных файлов. Их придумали, чтобы было удобно таскать кучу объектных файлов.

«+»

- Исполняемый файл включает в себя все необходимое. – программа может работать на любом компьютере, независимо от того, есть ли на нем библиотека или нет.
- Не возникает проблем с использованием не той версии библиотеки. – если и возникают, то у вас, как у программиста – ты не можешь собрать приложение (на этапе компиляции или компоновки), но как знающий человек поймешь, что не так с версией и исправишь – пользователь с проблемой не столкнется

«-»

- «Размер». – если несколько приложений используют одну и ту же библиотеку, то код библиотеки в нескольких экземплярах присутствует на компьютере. На жестком диске +- норм, но если запустить их все вместе, и они попадут в оперативную память – уже проблема.
- При обновлении библиотеки программу нужно пересобрать.
- Только один язык программирования (потому что все сильно завязано на компиляторе) (в динамике не так, поэтому можно использовать на других языках)

Способы использования статические – только в момент компоновки

**Динамические** - Подпрограммы из библиотеки загружаются в приложение во время выполнения. Код библиотеки не помещается в исполняемый файл. Примечание: Компоновщик, таблица импорта, она содержит сведения о том, что программа использует такую функцию, и она находится в такой библиотеке – это в исполняемый файл попадает, но сам код – нет.

“+”

- Несколько программ могут «разделять» одну библиотеку.
- Меньший размер приложения (по сравнению с приложением со статической библиотекой). (выигрыш происходит только тогда, когда несколько приложений используют одну и ту же библиотеку)
- Средство реализации плагинов. Плагин – механизм расширения функциональности приложения – когда в приложении есть интерес, который позже может быть реализован в библиотеки. Можно какой-то функционал раздавать бесплатно, а другой – в виде плагина и за деньги
- Модернизация библиотеки не требует перекомпиляции программы.
- Могут использовать программы на разных языках.

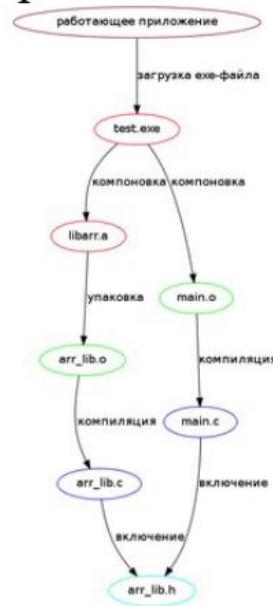
«-»

- Требуется наличие библиотеки на компьютере.
- Версионность библиотек. (если поменялся интерфейс). Необходимо обеспечить работу приложений, которые знают только о старой версии, и те, которые знают о старой и новой. С этим столкнулись Windows, dependency hell – ад из зависимостей

Способы использования динамических библиотек

- **динамическая компоновка;** – когда мы делегируем компоновщику часть функций по загрузке библиотеки и поиску функций в этой библиотеке.
- **динамическая загрузка.** – всю работу выполняем сами, используя интерфейс, который предоставляет ОС

# Граф зависимостей



## 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
```

Компиляция - просто получаем объектный файл (файлы)

Упаковка – получить архив из объектных файлов. Выполняется с помощью утилиты ar, работа управляется ключами. С – create – создать файл библиотеки, если он еще не существует, r – replace – если уже есть файл с библиотекой и объектный файл. Функции из него будут помещаться, если библиотека содержит их более старые версии. Затем имя библиотеки и составляющие объектные файлы

Индексирование (работает и без него). Если библиотека состоит из большого числа объектных файлов и функций, то, чтобы ускорить работу **компоновщика**, рекомендуется создать индекс. Это делается с помощью утилиты ranlib, которая модифицирует сам файл библиотеки libarr.a

сбор самой библиотеки

```
%.a : $(OBJS)
 ar rc $@ $(OBJS)
 ranlib $@
```

сбор приложения

```
31
32 app.exe: libapp.a $(OUT)app.o
33 $(CC) $(OUT)app.o -L. -lapp -o $@
34
```

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

-lapp - название библиотеки (сокращение, l == lib)

если написать -llibapp.a - то будет искать файл с названием liblibapp.a

## ВИНДА

Динамические библиотеки представлены форматом .dll – dynamic linked library.

Ключ shared говорит о том, что мы собираем не исполняемый файл, а динамическую библиотеку.

Затем перечисляем объектные файлы библиотеки

Ключ **-Wl**, уходит компоновщику и сообщает о подсистеме **-subsystem**, которой присваиваем значение windows.

В конце – имя библиотеки

## 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,window -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,window -o arr.dll
```

### Сборка приложения

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



10

## ЛИНУКС

```
сбор библиотеки
расширение *.so
-fpic === Флаги -fPIC и -fpic разрешают генерацию "позиционно-независимого кода" (position independent code), требующегося для разделяемых библиотек.
librarr.so: $(OUT)sort.o $(OUT)filter.o $(OUT)work_with_file.o $(SRC)sort.c $(SRC)filter.c $(SRC)work_with_file.c
$(CC) $(CFLAGS) -fpIC -c $(SRC)sort.c $(SRC)filter.c $(SRC)work_with_file.c
$(CC) -o librarr.so -shared $(OUT)sort.o $(OUT)filter.o $(OUT)work_with_file.o

сбор приложения
LD_LIBRARY_PATH
```

Для какого-то конкретного запуска приложения, можно подменить библиотеки. В Linux переменная окружения LD\_LIBRARY\_PATH - это список отделённых двоеточиями имён каталогов, где библиотеки следует искать перед тем, как их будут искать по стандартным путям. Полезно для отладки новых библиотек или для использования нестандартной библиотеки для чего-то этакого.

```
dynamic_out: librarr.so $(OUT)main.o $(OUT)utils.o
$(CC) -o app2.out $(OUT)main.o $(OUT)utils.o -L .
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:.

dl_out: $(OUT)main_dl.o $(OUT)utils.o
$(CC) -o app3.out $(OUT)main_dl.o $(OUT)utils.o -ldl
```

-ldl это обозначение библиотеки для компоновщика. Он сообщает компоновщику найти и связать файл с именем libdl.so(или иногда libdl.a). Это имеет тот же эффект, что и размещение полного пути к рассматриваемой библиотеке в той же позиции командной строки.

### дин заргудка

- + меньше памяти
- надо больше писать кода приложения

### дин компоновка

- + ничего не исправлять (линукс)
- исправлять код самой библиотеки (винда)

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

### УКАЗЫВАЕМ БИБЛИОТЕКУ ПРИ КОМПОНОВКЕ

#### Сборка приложения

- Получаем объектный файл приложения  
`gcc -std=c99 -Wall -Werror -c main.c`
- И все компонуем
  - `gcc main.o -L. -larr -o test.exe`

При сборке библиотеки на компиляторе Microsoft (Visual Studio) получаем arr.dll и arr.lib. При компоновке указывать lib файл

ilomovskoy:xyz.dll xyz.lib

ilomovskoy:xyz.lib для компоновки

вообще gcc лучше))))))))

## ЛИНУКС

```
Igor@Igor-PC:~/work/L_20_src_1/dyn_lin_1$ gcc -std=c99 -Wall -Werror -fPIC -c arr_lib.c
Igor@Igor-PC:~/work/L_20_src_1/dyn_lin_1$ gcc -o libarr.so -shared arr_lib.o
Igor@Igor-PC:~/work/L_20_src_1/dyn_lin_1$ ls
arr_lib.h arr_lib.o libarr.so main.c
Igor@Igor-PC:~/work/L_20_src_1/dyn_lin_1$ gcc -std=c99 -Wall -Werror -c main.c
Igor@Igor-PC:~/work/L_20_src_1/dyn_lin_1$ gcc -o app.out main.o -L .
Igor@Igor-PC:~/work/L_20_src_1/dyn_lin_1$./app.out
./app.out: error while loading shared libraries: libarr.so: cannot open shared object file: No such file or directory
Igor@Igor-PC:~/work/L_20_src_1/dyn_lin_1$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:.
Igor@Igor-PC:~/work/L_20_src_1/dyn_lin_1$./app.out
Array:
0 1 2 3 4
Igor@Igor-PC:~/work/L_20_src_1/dyn_lin_1$
```

Вроде библиотека есть, но пробуем как обычно собрать приложение с динамической компоновкой – ошибка-загрузить библиотеку не удалось.

LD\_LIBRARY\_PATH отвечает за пути, по которым ищутся динамические библиотеки. Тут не удалось, поэтому нужно добавить наш путь к этим путям при помощи команды `export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:.` – добавили текущий каталог. Эту команду нужно добавлять в каждой новой сессии.

Или можно задать путь

**Динамическая загрузка** – это делегирование части функции по загрузке библиотеки и поиску нужных функций самостоятельно, без участия компоновщика с использованием интерфейса, который предоставляет операционная система.

Когда всю работу по загрузке библиотеки и поиске нужных функций мы берем на себя. С помощью API, которую предоставляет ОС

## ВИНДА

### windows.h

- **HMODULE LoadLibrary(LPCSTR)** Функция `LoadLibrary` отображает заданный исполняемый модуль в адресное пространство вызывающего процесса.  
Если строка определяет путь, но файл не существует в указанном каталоге, функция завершается ошибкой. Когда определяется путь, убедитесь, что использованы наклонные черты влево(обратные слэши (\)), а не прямые слэши (/). Если символьная строка не определяет путь, функция использует стандартную стратегию поиска файла.
- **FARPROC GetProcAddress(HMODULE, LPCSTR)** Если функция завершается успешно, возвращаемое значение – адрес экспортируемой функции или переменной.  
Если функция завершается ошибкой, возвращаемое значение – ПУСТО (NULL). Чтобы получить дополнительную информацию об ошибке, вызовите [GetLastError](#).
- **FreeLibrary(HMODULE)** Функция `FreeLibrary` уменьшает итоговое число ссылок на загруженные динамически подключаемые библиотеки (DLL). Когда итоговое число ссылок достигает нуля, модуль отменяет отображение в адресном пространстве вызывающего процесса, а дескриптор становится больше не допустим.

поэтому сразу может не получиться выгрузить

## ЛИНУКС

### #include <dlfcn.h>

- **dlopen** загружает динамическую библиотеку, имя которой указано в строке `filename`, и возвращает прямой указатель на начало динамической библиотеки. Если `filename` не является полным именем файла (т.е. не начинается с "/"), то файл ищется в следующих местах:
  - в разделенном двоеточием списке каталогов, в переменной окружения пользователя `LD_LIBRARY_PATH`.  
В списке библиотек, кэшированных в файле `/etc/ld.so.cache`.  
В `/usr/lib` и далее в `/lib`.  
Если `filename` указывает на NULL, то возвращается указатель на основную программу.
- **dlclose** уменьшает на единицу счетчик ссылок на указатель динамической библиотеки `handle`. Если нет других загруженных библиотек, использующих ее символы и если счетчик ссылок принимает нулевое значение, то динамическая библиотека выгружается.
- **dlsym** использует указатель на динамическую библиотеку, возвращаемую `dlopen`, и оканчивающееся нулем символьное имя, а затем возвращает адрес, указывающий, откуда загружается этот символ. Если символ не найден, то возвращаемым значением `dlsym` является NULL; тем не менее, правильным способом проверки `dlsym` на наличие ошибок является сохранение в переменной результата выполнения `dlerror`, а затем проверка, равно ли это значение NULL.

Для указания GCC на библиотеки, которые нужно использовать при сборке, понадобятся ещё два ключа:

- **l** — для имени библиотеки,
- **L** — для указания пути к ней.

Зачем:

- 1) Пусть необходим ресурсоемкий, но эффективно реализованный алгоритм – си отлично подходит. На нем реализуем алгоритм в виде библиотеки и используем в питоне.
- 2) Предоставить доступ к платформо/аппаратно зависимым вещам. Пишем библиотеку для работы с устройством на си, а потом на питоне – остальное
- 3) Уже много чего на Си и просто так переписывать все на питон – ну такое. Проще подключать

В первую очередь нужно обращать внимание на динамическое выделение памяти. В таком случае лучше всего перекладывать выделение памяти на вызывающую сторону.

Также может произойти проблемы с несоответствием типов и действий над ними. Тот же Python не знает об указателях в языке С и не может вернуть значение через параметр функции. Здесь и возникает трудность, которая требует написания обертки для функций, чтобы перевести функцию одного языка в понятный вид для другого языка

**Модуль `ctypes`** – это мощный инструмент в Python, который позволяет вам использовать существующие библиотеки в других языках путем написания простых **декораторов** в самом Python.

1 - Чтобы загрузить библиотеку необходимо создать объект класс CDLL:

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

Классов для работы с библиотеками в модуле `ctypes` несколько:

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

Класс выбирается в зависимости от соглашения о вызовах, которое использует библиотека. (в нашем случае cdecl - соглашение о вызове в СИ)

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

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

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

В языке Си используются идиомы, которых нет в языке Python. Например, функция divided возвращает одно из значений через свой аргумент. Поэтому решение «в лоб» обречено на неудачу.

```
int devide(int, int, int*)
_divide = lib.divide
_divide.argtypes = (ctypes.c_int, ctypes.c_int, \
 ctypes.POINTER(ctypes.c_int))
_divide.restype = ctypes.c_int

x = 0
_divide(7, 5, x)
```

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

```
def divide(x, y):

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

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

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

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

## ИТОГО

- 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.
- Альтернативные подходы – использование Swig или Cython.

## Модуль расширений

Это более правильный метод использования динамической библиотеки, так как здесь мы можем правильней написать обертки

1) Для каждой функции нашей библиотеки написать соответствующую обертку.

### 1.1) Какой прототип?

Модуль расширения пишется на Си и состоит из функций, у всех – одинаковый прототип. Обычно функции модуля расширения имеют следующий вид

```
static PyObject* py_func(PyObject* self, PyObject* args)
```

```
{
 ...
}
```

- PyObject – это тип данных Си, представляющий любой объект Python.
- Функция модуля расширения получает кортеж таких объектов (args)- аргументов функции- и возвращает новый Python объект в качестве результата.
- Аргумент self не используется в простых функциях.

1.2) Как из массива аргументов **достать аргументы**, которые обычно используют на Си?

- Функция PyArg\_ParseTuple используется для конвертирования переменных из представления Python в представление Си.
- На вход (похоже на scanf) эта функция принимает строку форматирования, которая описывает тип требуемого значения, и адреса переменных, в которые будут помещены значения.
- В ходе конвертации функция PyArg\_ParseTuple выполняет различные проверки. Если что-то пошло не так, функция возвращает NULL.

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

... . . . . .

### 1.3) Как создать объект – **возвращаемое значение**

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

```
int a, b, c;

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

c = add(a, b);

return Py_BuildValue("i", c);
```

При работе с массивами пишем вспомогательную копирования функцию (из Си переводим в Питон)

```
#include <stdlib.h>
#include "Python.h"
#include "example.h"

// int int add(int a, int b);
static PyObject* py_add(PyObject *self, PyObject *args)
{
 int a, b, c;

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

 c = add(a, b);

 return Py_BuildValue("i", c);
}

// int divide(int, int, int *);
static PyObject* py_divide(PyObject *self, PyObject *args)
{
 int a, b, quotient, remainder;

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

 quotient = divide(a,b, &remainder);

 return Py_BuildValue("(ii)", quotient, remainder);
}
```

```

static PyObject* copy_to_py_list(const double *arr, int n)
{
 PyObject *plist, *pitem;

 plist = PyList_New(n);
 if (!plist)
 return NULL;

 for (int i = 0; i < n; i++)
 {
 pitem = PyFloat_FromDouble(arr[i]);
 if (!pitem)
 {
 Py_DECREF(plist);
 return NULL;
 }

 PyList_SET_ITEM(plist, i, pitem);
 }

 return plist;
}

// void fill_array(double *arr, int n);
static PyObject* py_fill_array(PyObject *self, PyObject *args)
{
 PyObject *plist;
 int n;
 double *pbuf;

 if (!PyArg_ParseTuple(args, "i", &n))
 return NULL;

 pbuf = malloc(n * sizeof(double));
 if (!pbuf)
 return NULL;

 fill_array(pbuf, n);

 plist = copy_to_py_list(pbuf, n);

 free(pbuf);

 return plist;
}

```

```
static PyObject *py_avg(PyObject *self, PyObject *args)
{
 PyObject *seq, *item, *item_float;
 double *a;
 int n;
 double res;

 // Получим аргументы как последовательность объектов Python
 if (!PyArg_ParseTuple(args, "O", &seq))
 return NULL;

 seq = PySequence_Fast(seq, "Argument must be iterable");

 if (!seq)
 return NULL;

 //
 // Преобразуем последовательность в массив double (если получится)
 //

 n = PySequence_Fast_GET_SIZE(seq);

 a = malloc(n * sizeof(double));
 if (!a)
 {
 Py_DECREF(seq);

 return NULL;
 }
```

```
for (int i = 0; i < n; i++)
{
 item = PySequence_Fast_GET_ITEM(seq, i);
 if (!item)
 {
 Py_DECREF(seq);
 free(a);

 return NULL;
 }

 item_float = PyNumber_Float(item);
 if (!item_float)
 {
 Py_DECREF(seq);
 free(a);

 return NULL;
 }

 a[i] = PyFloat_AS_DOUBLE(item_float);

 Py_DECREF(item_float);
}

Py_DECREF(seq);

res = avg(a, n);

free(a);

return Py_BuildValue("d", res);
}
```

```
static PyObject *py_avg_2(PyObject *self, PyObject *args)
{
 PyObject *obj, *iterator, *item;
 double *a;
 int n, i, bad_data;
 double res;

 // Получим аргументы как последовательность объектов Python
 if (!PyArg_ParseTuple(args, "O", &obj))
 {
 PyErr_SetString(PyExc_TypeError, "Cant parse arguments");

 return NULL;
 }

 iterator = PyObject_GetIter(obj);
 if (!iterator)
 {
 PyErr_SetString(PyExc_TypeError, "Cant get iterator");

 return NULL;
 }
```

```
// Посчитаем кол-во
n = 0;
bad_data = 0;
while ((item = PyIter_Next(iterator)) && !bad_data)
{
 if (!PyFloat_Check(item) && !PyLong_Check(item))
 {
 bad_data = 1;
 }
 else
```

```

 {
 n++;
 }

 Py_DECREF(item);
}

Py_DECREF(iterator);

if (bad_data)
{
 Py_DECREF(obj);

 PyErr_SetString(PyExc_TypeError, "Bad arguments (only numbers are required)");

 return NULL;
}

a = malloc(n * sizeof(double));
if (!a)
{
 Py_DECREF(obj);

 PyErr_SetString(PyExc_TypeError, "Memory allocation error");

 return NULL;
}

iterator = PyObject_GetIter(obj);

i = 0;
while ((item = PyIter_Next(iterator)))
{
 a[i] = PyFloat_AsDouble(item);
 i++;
 Py_DECREF(item);
}

Py_DECREF(iterator);

Py_DECREF(obj);

res = avg(a, n);

free(a);

return Py_BuildValue("d", res);
}

```

2) Ближе к концу модуля располагаются таблица методов модуля PyMethodDef и структура PyModuleDef, которая описывает модуль в целом.

- В таблице PyMethodDef перечисляются
  - Си функции;
  - имена, используемые в Python;
  - флаги, используемые при вызове функции,
  - строки документации.
- В конце – нулевая структура (нигде количество методов не писали, видимо, для понимания их числа)

```
// Таблица методов модуля
static PyMethodDef example_methods[] =
{
 {"add", py_add, METH_VARARGS, "Integer addition"},
 {"divide", py_divide, METH_VARARGS, "Integer division"},
 {"fill_array", py_fill_array, METH_VARARGS, "Filling array of doubles"},
 {"avg", py_avg, METH_VARARGS, "Calculation of average value of array of doubles"},
 {"avg_2", py_avg_2, METH_VARARGS, "Calculation of average value of array of doubles"},
 { NULL, NULL, 0, NULL}
};
```

3) Структура PyModuleDef используется для загрузки модуля.

```
// Структура, описывающая модуль
static struct PyModuleDef example_dll_module =
{
 PyModuleDef_HEAD_INIT,
 "example_dll", // имя модуля
 "An example_dll module", // строка для документации
 -1,
 example_methods // таблица методов
};
```

4) В самом конце модуля располагается функция инициализации модуля, которая практически всегда одинакова, за исключением своего имени.

```
// Функция инициализации модуля
PyMODINIT_FUNC PyInit_example_dll(void)
```

```
{
 return PyModule_Create(&example_dll_module);
}
```

5) Пишем скрипт setup.py. Он содержит информацию о модуле расширения – где заголовки, какие define определить/разопределить, где файлы с нужными библиотеками

```
from distutils.core import setup, Extension

setup(name='example_dll',
 ext_modules=[
 Extension('example_dll',
 ['wrap.c'],
 include_dirs = ['.'],
 define_macros = [('FOO', '1')],
 undef_macros = ['BAR'],
 library_dirs = ['.'],
 libraries = ['example'])
]
)
```

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

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

На Windows компиляция может сразу не заработать.

```
C:\msys64\home\Ira\05_12_20\l_20_src_2\app>>>python setup.py build_ext --inplace
running build_ext
building 'example_dll' extension
error: Unable to find vcvarsall.bat
```

1 - получаем объектный файл модуля расширения, используя заголовочные файлы питона (поэтому пишем путь полный)  
gcc -std=c99 -Wall -I C:/Users/Ira/AppData/Local/Programs/Python/Python38/include -c wrap.c  
2 - указываем специальный файл libarray\_dll.pyd, а так же указываем путь до всех библиотек питона и указываем -lpython3 -lpython38  
gcc -shared -o libarray\_dll.pyd wrap.o libarray.dll -L C:/Users/Ira/AppData/Local/Programs/Python/Python38/libs -lpython3 -lpython38  
9  
0 app : libarray.dll  
1 gcc -std=c99 -Wall -I C:/Users/Ira/AppData/Local/Programs/Python/Python38/include -c wrap.c  
2 gcc -shared -o libarray\_dll.pyd wrap.o libarray.dll -L C:/Users/Ira/AppData/Local/Programs/Python/Python38/libs  
3 rm \*.o  
4

## 42. Неопределенное поведение

- Что такое побочный эффект? Что с точки зрения стандарта относится к побочному эффекту?

- Пример, анализ примера справа налево слева направо, получение двух разных результатов (за собственный пример плюсик ☺)
- Что такое точка следования? Какие точки следования стандарт выделяет? С использованием точек следования анализируете Ваше выражение ещё раз и рассказываете, как с этим явлением можно в программе бороться
- Виды неопределенного поведения в Си (примеры)
- Undefined behaviour: средства борьбы

Побочный эффект — **не основные, дополнительные (как желательные, так и нежелательные) последствия работы функции.**

Побочный эффект выражается в неявном изменении значения переменной в процессе вычисления выражения. Все операции присваивания могут вызывать побочный эффект. Вызов функции, в которой изменяется значение какой-либо внешней переменной, либо путем явного присваивания, либо через указатель, также имеет побочный эффект.

- Модификация данных.
- Обращение к переменным, объявленным как volatile.
  - Ключевое слово volatile – это спецификатор, применяемый при объявлении переменной. Он сообщает компилятору, что значение переменной может изменяться в любой момент – без какого-либо действия со стороны кода, который компилятор обнаруживает поблизости.
- Вызов системной функции, которая производит побочные эффекты (например, файловый ввод или вывод).
- Вызов функций, выполняющих любое из вышеперечисленных действий.

## Порядок вычисления выражений

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

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

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

## Точки следования

- 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.

## Точки следования

Определены следующие точки следования:

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

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

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

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

# Точки следования

продолжение

- В конце полного выражения.

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

# Точки следования

продолжение

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

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

## Порядок вычисления выражений

Почему результата выражения « $x[i] = i++ + 1;$ » не определен?

- Порядок вычисления выражений и подвыражений между точками следования не определен.
- В выражении « $x[i] = i++ + 1;$ » есть единственная точка следования, которая находится в конце полного выражения.
- В выражении « $x[i] = i++ + 1;$ » есть два обращения к переменной  $i$ . **Множественный доступ и является источником проблемы.**

**Избегайте сложных выражений!**

## Порядок вычисления выражений

Почему спецификация языка оставляет открытым вопрос, в каком порядке компиляторы должны вычислять выражения между точками следования?

Причина неопределенности порядка вычислений – простор для оптимизации.

```

int main()
{
 int p1, p2, p3, p4, p6, p7, p8;
 int x[3] = {0}, i = 1;

 p1 = f1() + f2();
 printf("\n");

 p2 = f1() - f2();
 printf("\n");

 p3 = f1() * f2();
 printf("\n");

 p4 = f1() / f2();
 printf("\n");

 f3(f1(), f2());
 printf("\n");

 f3(c(a()), d(b()));
 printf("\n");

 x[i] = i++ + 1;
 printf("%d %d %d\n\n", x[0], x[1], x[2]);
 0 0 2

 p6 = f1() && f2();
 printf("\n");

 p7 = (f1(), f2());
 printf("\n");

 p8 = f1() ? f2(): 3;
 printf("\n");

 return 0;
}

```

```

f1
f2
f1
f2
f1
f2
f2
f1
b
d
a
c
f1
f2
f1
f2
f1
f2

```

█

Какие точки следования выделяет стандарт с99?

- Между вычислением левого и правого operandов в операциях `&&`, `||` и `,"`  
`*p++ != 0 && *q++ != 0` (на знаке операции)
- Между вычислением первого и второго или третьего operandов в тернарной операции.  
`a = (*p++) ? (*p++) : 0;` (на знаке вопроса)

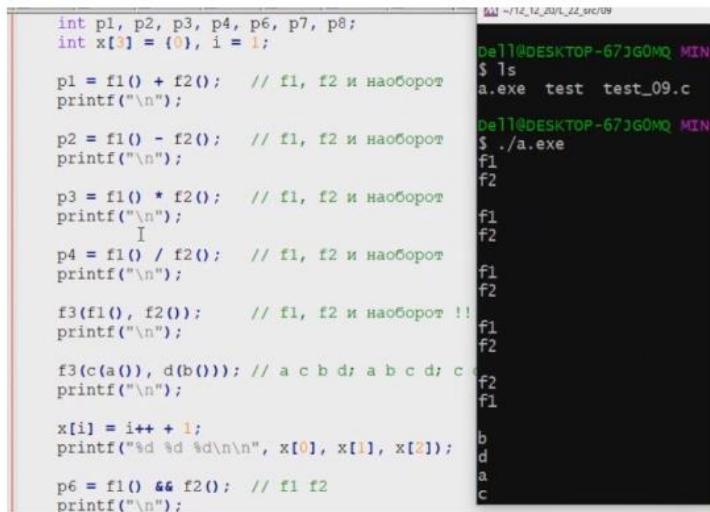
- Перед входом в вызываемую функцию.

Порядок, в котором вычисляются аргументы не определен, но эта точка следования гарантирует, что все ее побочные эффекты проявятся на момент входа в функцию.

- В объявлении с инициализацией на момент завершения вычисления инициализирующего значения.

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

Примеры:



```
int p1, p2, p3, p4, p5, p6, p7, p8;
int x[3] = {0}, i = 1;

p1 = f1() + f2(); // f1, f2 и наоборот
printf("\n");

p2 = f1() - f2(); // f1, f2 и наоборот
printf("\n");

p3 = f1() * f2(); // f1, f2 и наоборот
printf("\n");
 I
p4 = f1() / f2(); // f1, f2 и наоборот
printf("\n");

f3(f1(), f2()); // f1, f2 и наоборот !!
printf("\n");

f3(c(a()), d(b())); // a c b d; a b c d; c
printf("\n");

x[i] = i++ + 1;
printf("%d %d %d\n", x[0], x[1], x[2]);
 I
p6 = f1() && f2(); // f1 f2
printf("\n");
```

получили bdac и f2 f1 из-за того, что

аргументы функции в стек, а выходят в обратном порядке

У студийного компилятора – все так же, кроме

В простейших случаях компилятор замечает

```
test_09.c: In function 'main':
test_09.c:71:13: error: operation on 'i' may be undefined [-Werror=sequence-point]
 71 | x[i] = i++ + 1;
 ^~~~~~
```

- *Unspecified behavior.* не специфицированное поведение

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

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

- *Implementation-defined behavior.* поведение от реализации

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

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

- *Undefined behavior*

Такое поведение возникает как следствие неправильно написанной программы или некорректных данных. Стандарт ничего не гарантирует, может случиться все что угодно. (может отработать правильно или упасть или вообще ничего.)

## 9. Почему "неопределенное" поведение присутствует в языке Си?

Чтобы

- освободить разработчиков компиляторов от необходимости обнаруживать ошибки, которые трудно диагностировать. (в особенности – undefined behavior, остальные 2 – с остальными видами)
- избежать предпочтения одной стратегии реализации другой.

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

## 11. Как бороться с неопределенным поведением?

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

-ftrapv

This option generates traps for signed overflow on addition, subtraction, multiplication operations.

Эта опция генерирует ловушки для переполнения со знаком при операциях сложения, вычитания, умножения.

## 12. Приведите примеры неопределенного поведения.

- Использование неинициализированных переменных.
- Переполнение знаковых целых типов.
- Выход за границы массива.
- Использование «диких» (неинициализированных) указателей
- Выражение размера в объявлении массива не является постоянным выражением и во время выполнения программы вычисляется как отрицательное значение.
- Разыменование NULL-указателя
- Деление на 0

переполнение знаковых типов с gcc будет работать, так как его просто не будет...!!!!

Примеры неопределенного поведения:

- [доступ к элементам массива вне разрешенных границ](#). Например, выделено 5 элементов, а пытаемся прочитать элемент с индексом  $\geq 5$ .
- многократное изменение переменной или неупорядоченные изменения и независимое чтение при отсутствии порядка вычислений. Например:  $i = i++ + i;$

*implementation-defined behavior* - определённое в реализации (компиляторе) поведение. Поведение, которое чётко должно быть прописано в документации на **любой** компилятор.

Примеры:

- Размер **int** может быть от 2 до 4 байт в зависимости от реализации или настроек компилятора.
- **char** представляет собой один байт, однако в разных системах байт имеет разное количество бит (8, 16, 32, 64)
- [наличие дополнительных вариантов функции main](#), кроме как `int main()` и `int main(int, char**)`.

*unspecified behavior* - неустановленное поведение. Поведение для корректной программы и корректных данных, которое зависит от реализации (компилятора). Такое поведение не обязано быть описаным в документации на компилятор.

- Все аргументы функции должны быть вычислены до вызова функции, но они могут быть вычислены в любом порядке.
- Результат округления, когда значение выходит за пределы диапазона
- Порядок двух элементов, сравниваемых как равные в массиве, отсортированном функцией **qsort** (неустойчивая сортировка)

## 43-44. АТД, понятие модуль, разновидности модулей, абстрактный объект – стек целых чисел / АТД – стек целых чисел

- Определение понятия модуль
- Преимущества при модульной организации программы
- Из каких частей модуль состоит, какие требования к этим частям предъявляются, какие средства язык си предоставляет для реализации модулей
- Что такое тип данных, что такое АТД, чем они отличаются
- Реализация абстрактного объекта – стек целых чисел

**Модуль** в языке **си** - это такой источник кода, функционал которого вы можете включить в свой проект в виде отдельного файла и/или прописав имя его заголовка в `#include` - то есть **модуль в си** - это вообще говоря - файл исходного кода, который компилируется отдельно , от остальных составляющих программы.

Модуль – функционально законченный фрагмент программы, оформленный в виде отдельного файла с исходным кодом или поименованной непрерывной её части, предназначенный для использования в других программах.

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 (могут отличаться используемым типом данных/алгоритмом)

Часть кода, которая использует модуль, называют *клиентом*.

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

## Преимущества модульной организации

- Абстракция (как средство борьбы со сложностью)

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

Каждый модуль – черный ящик. Мы знаем, что он делает, но не знаем деталей. Можем вносить изменения в основную программу, не думая о модуле.

- 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.

1) Исправление ошибок

2) Реагирование на требования клиентов: важность производительности или ресурсов

Типы модулей:

- Набор данных

Набор связанных переменных и/или констант. В Си модули этого типа часто представляются только заголовочным файлом. (float.h, limits.h.). Переменные – не лучшая идея, но константы – удобно.

- Библиотека

Набор связанных функций. (stdio.h, string.h, собственные библиотеки)

- Абстрактный объект

Набор функций, который обрабатывает скрытые данные. Журналирование – последняя реализация. Создание журнала, вывод информации, закрытие.

- 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.

- 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.
- Допустимо определять указатель на неполный тип

- Нельзя

- применять операцию обращения к полю (->);
- разыменовывать переменные типа T.

<https://docs.microsoft.com/ru-ru/cpp/c-language/incomplete-types?view=msvc-160&viewFallbackFrom=vs-2019>

*Неполный тип* — это тип, который описывает идентификатор, но не содержит информацию, необходимую для определения размера идентификатора. Неполным типом может быть:

**Тип данных (тип)** — множество значений и операций над этими значениями

Язык Си поддерживает *статическую* типизацию. Тип данных определяет

- внутренне представление данных в памяти;
- множество значений, которые могут принимать величины этого типа;
- операции и функции, которые можно применять к величинам этого типа.

Простые (скалярные) типы

- целый (int);
- вещественный (float и др.);
- символьный (char);
- перечисляемый тип;
- логический тип (c99);
- void;
- указатели.

Составные (строктурированные) типы

- массивы;
- структуры;
- объединения.

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.

## Примеры АТД [ [править](#) | [править код](#) ]

- [Список](#)
- [Стек](#)
- [Очередь](#)
- [Ассоциативный массив](#)
- [Очередь с приоритетом](#)

АТД, в первую очередь, представляет собой тип данных, что означает следующее:

- наличие определенных доступных операций над элементами этого типа
- данные, относительно которых эти операции выполняются (диапазон значений). Что же означает слово “абстрактный”?
- этот тип данных будет скрывать те данные, с помощью которых реализовано данное поведение.

АО - набор функций, которые влияют на скрытые данные. АТД - тип данных со скрытой внутренней структурой + функции-операции над ним.

- 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.

ПРИМЕР 1 абстрактный объект стек

stack\_0.h, stack\_0.c, main\_0.c

static инкапсулирует данные внутри файла

```

#include "stack_0.h"
#define STACK_SIZE 10
static int content[STACK_SIZE];
static int top;
void make_empty(void)
{
 top = 0;
}
bool is_empty(void)
{
 return top == 0;
}
bool is_full(void)
{
 return top == STACK_SIZE;
}
int push(int i)
{
 if (is_full())
 return -1;
 content[top++] = i;
 return 0;
}
int pop(int *i)
{
 if (is_empty())
 return -1;
 *i = content[--top];
 return 0;
}

#ifndef _STACK_0_H_
#define _STACK_0_H_
#include <stdbool.h>
void make_empty(void);
bool is_empty(void);
bool is_full(void);
int push(int i);
int pop(int *i);
#endif

```

```

#include <stdio.h>
#include <assert.h>
#include "stack_0.h"

int main(void)
{
 int i;
 int rc;

 make_empty();

 i = 0;
 while (!is_full())
 {
 rc = push(i);
 assert(rc == 0);

 i++;
 }

 while (!is_empty())
 {
 rc = pop(&i);
 assert(rc == 0);

 printf("%d\n", i);
 }

 return 0;
}

```

НЕДОСТАТОК - Серьезный недостаток – не существует способа, создать несколько экземпляров стека.

Решение: создать структуру. Тогда параметром функций должен быть стек, к которому эта функция применяется

#### ПРИМЕР 2 «абстрактный» тип данных «стек»

stack\_1.h, stack\_1.c, main\_1.c

```
#include "stack_1.h"

void make_empty(stack_t *s)
{
 s->top = 0;
}

#ifndef _STACK_1_H_
#define _STACK_1_H_
#include <stdbool.h>
#define STACK_SIZE 10

typedef struct
{
 int content[STACK_SIZE];
 int top;
} stack_t;

void make_empty(stack_t *s);
bool is_empty(const stack_t *s);
bool is_full(const stack_t *s);
int push(stack_t *s, int i);
int pop(stack_t *s, int *i);
#endif

#include <stdio.h>
#include <assert.h>
#include "stack_1.h"

int main(void)
{
 stack_t s;
 int i;
 int rc;

 make_empty(&s);

 i = 0;
 while (!is_full(&s))
 {
 rc = push(&s, i);
 assert(rc == 0);
 i++;
 }

 while (!is_empty(&s))
 {
 rc = pop(&s, &i);
 assert(rc == 0);
 printf("%d\n", i);
 }

 return 0;
}
```

НЕДОСТАТОК - К сожалению, stack\_t не является абстрактным типом данных, потому что stack\_1.h показывает все детали реализации. И клиент может работать со стеком напрямую и сломать его.

Как скрыть? – использовать неполный тип данных

#### ПРИМЕР 3 абстрактный тип данных «стек»

Добавляются функции create и destroy, так как пользователю мы должны вернуть указатель на эту структуру и освобождать память.

stack\_2.h, stack\_2.c, main\_2.c

```

#include <stdlib.h>
#include "stack_2.h"
#define STACK_SIZE 10

struct stack_type
{
 int content[STACK_SIZE];
 int top;
};

stack_t create(void)
{
 stack_t s = malloc(sizeof(struct stack_type));
 if (s)
 make_empty(s);
 return s;
}

void destroy(stack_t s)(free(s));
void make_empty(stack_t s){s->top = 0;}
bool is_empty(const stack_t s){return s->top == 0;}
bool is_full(const stack_t s)
{
 return s->top == STACK_SIZE;
}

stack_t create(void);
void destroy(stack_t s);
void make_empty(stack_t s);
bool is_empty(const stack_t s);
bool is_full(const stack_t s);
int push(stack_t s, int i);
int pop(stack_t s, int *i);

#endif
}

```

```

#include <stdio.h>
#include <assert.h>
#include "stack_2.h"

int main(void)
{
 stack_t s;
 int i;
 int rc;

 s = create();
 i = 0;
 while (!is_full(s))
 {
 rc = push(s, i);
 assert(rc == 0);

 i++;
 }

 while (!is_empty(s))
 {
 rc = pop(s, &i);
 assert(rc == 0);
 printf("%d\n", i);
 }

 destroy(s);
 return 0;
}

```

Стек (stack\_2.h) реализован только для целых чисел. Это слишком сильное ограничение. «решение» - вынести тип отдельно, чтобы потом быстро исправить

## 45-46. Списки ядра Linux. Идея и основные моменты использования / Идея и основные моменты реализации.

- И. В. выдаст распечатанный заголовочный файл list.h для вопроса 45.
- **Идея и основные моменты использования:** рассказать, что из себя представляет список ядра Linux с точки зрения структур данных, рассказать, как достигается универсальность реализации списка (чем отличается от void \*data), как создать список ядра Linux, как добавить элемент в список (начало и конец), как обойти список, как удалить элемент, как освободить память, сравнить списки ядра Linux с обычными списками
- **Идея и основные моменты реализации:** рассказать, что из себя представляет список ядра Linux с точки зрения структур данных, рассказать, как достигается универсальность реализации списка (чем отличается от void \*data), какие обходы списка ядра Linux вы знаете

(какая особенность?), containerof, offsetof (идея реализации, идейный способ использовать нельзя)

## Списки Беркли: идея

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

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

В отличие от обычных списков, где данные содержатся в элементах списка, структура list\_head должна быть частью самих данных

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

3

## Списки Беркли: создание «ГОЛОВЫ»

```
#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;
}
```

## Списки Беркли: 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);
}
```

7

## Списки Беркли: описание

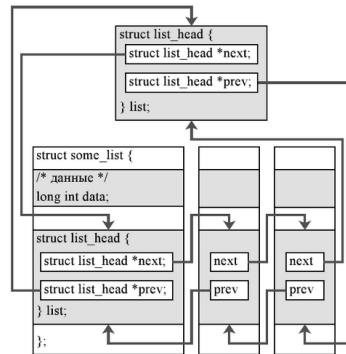
```
#include "list.h"

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

Следует отметить следующее:

- Структуру struct list\_head можно поместить в любом месте в определении структуры.
- struct list\_head может иметь любое имя.
- В структуре может быть несколько полей типа struct list\_head.

## Списки в стиле Беркли



## Списки Беркли: 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)
```

## Списки Беркли: удаление

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

static inline void __list_del_entry(struct list_head *entry)
{
 __list_del(entry->prev, entry->next);
}

static inline void list_del(struct list_head *entry)
{
 __list_del(entry->prev, entry->next);
 entry->next = NULL;
 entry->prev = NULL;
}
```

## offsetof: идея

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

...

printf("offset of i is %d\n", offsetof(struct s, i));
```

В нашем случае TYPE – struct s, MEMBER – i, size\_t – unsigned int. После работы препроцессора получим

```
printf("offset of i is %d\n",
 (unsigned int) (&((struct s*) 0)->i));
```

11

## Списки Беркли: 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)
```

## offsetof: идея

```
int offset = (int) (&((struct s*) 0)->i);
```

- $((\text{struct } s^*) 0)$ 
  - Приводим число ноль к указателю на структуру s. Эта строчка говорит компилятору, что по адресу 0 располагается структура, и мы получаем указатель на нее.
- $((\text{struct } s^*) 0)->i$ 
  - Получаем поле i структуры s. Компилятор думает, что это поле расположено по адресу 0 + смещение i.
- $\&((\text{struct } s^*) 0)->i$ 
  - Вычисляем адрес поля i, т.е. смещение i в структуре s.
- $(\text{unsigned int}) (\&((\text{struct } s^*) 0)->i)$ 
  - Преобразовываем адрес члена i к целому числу.

12

## Списки в стиле Беркли: анализ

«+» и «-»

+ Одно выделение памяти на узел списка.

- Независимо от того в списке узел или нет присутствуют два дополнительных указателя.

[https://translated.turbopages.org/proxy\\_u/en-ru.ru.bd83a1f9-6599858a-bf6c725c-74722d776562/www.makelinux.net/ldd3/chp-11-sect-5.shtml](https://translated.turbopages.org/proxy_u/en-ru.ru.bd83a1f9-6599858a-bf6c725c-74722d776562/www.makelinux.net/ldd3/chp-11-sect-5.shtml)