

Указатель на неопределенный
тип

void*

Тип указатель `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.

Особенности использования `void*` (1)

- В языке C 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;
```

Особенности использования `void*` (2)

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

Указатели и одномерные динамические массивы

Динамическое выделение памяти

Иногда в процессе выполнения программы удобно «создавать» переменные.

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

- `malloc` (выделяет блок памяти и не инициализирует его);
- `calloc` (выделяет блок памяти и заполняет его нулями);
- `realloc` (перевыделяет предварительно выделенный блок памяти).

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.

Особенности malloc, calloc, realloc (2)

- 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 (2)

```
int *a = NULL;
size_t n = 5;

// Выделение памяти
a = malloc(n * sizeof(int));
// Проверка успешности выделения
if (a == NULL)
{
    return ...
}

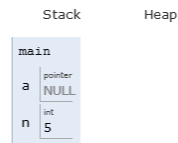
// Использование памяти
for (size_t i = 0; i < n; i++)
    a[i] = i;

// Освобождение памяти
free(a);
```

malloc (3)

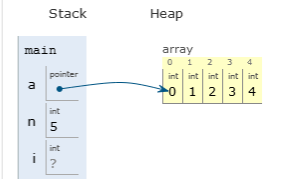
1. Перед выделением памяти

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(void)
5 {
6     int *a = NULL;
7     int n = 5;
8
9     // Выделение памяти
10    a = malloc(n * sizeof(int));
11
12    printf("a %p\n", a);
13
```



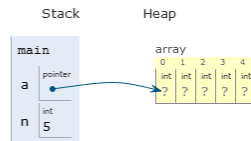
3. Использование выделенной памяти

```
14 // Проверка успешности выделения
15 if (a == NULL)
16 {
17     fprintf(stderr, "Memory allocation error\n");
18     return -1;
19 }
20
21 // Использование памяти
22
23 for (int i = 0; i < n; i++)
24     a[i] = i;
25
26 for (int i = 0; i < n; i++)
27     printf("%d ", a[i]);
```



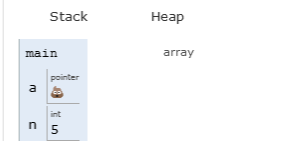
2. Сразу после выделения памяти

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(void)
5 {
6     int *a = NULL;
7     int n = 5;
8
9     // Выделение памяти
10    a = malloc(n * sizeof(int));
11
12    printf("a %p\n", a);
13
14    // Проверка успешности выделения
```



4. Сразу после освобождения

```
15 if (a == NULL)
16 {
17     fprintf(stderr, "Memory allocation error\n");
18     return -1;
19 }
20
21 // Использование памяти
22
23 for (int i = 0; i < n; i++)
24     a[i] = i;
25
26 for (int i = 0; i < n; i++)
27     printf("%d ", a[i]);
28
29 // Освобождение памяти
30 free(a);
31
32 return 0;
```



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 и 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.

calloc (2)

```
int *a;
size_t n = 5;

// Выделение памяти
a = calloc(n, sizeof(int));
// Проверка успешности выделения
if (a == NULL)
{
    return ...
}

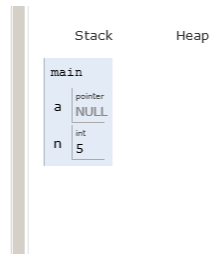
// Использование памяти
for (size_t i = 0; i < n; i++)
    printf("%d ", a[i]);

// Освобождение памяти
free(a);
```

calloc (3)

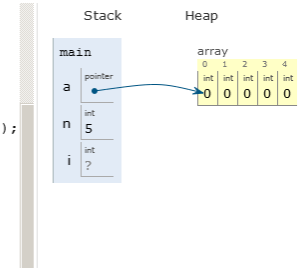
1. Перед выделением памяти

```
2 #include <stdlib.h>
3
4 int main(void)
5 {
6     int *a = NULL;
7     int n = 5;
8
9     // Выделение памяти
10    a = calloc(n, sizeof(int));
11
12    printf("a %p\n", a);
13
```



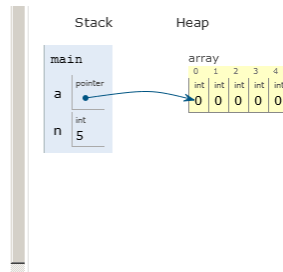
3. Использование выделенной памяти

```
12    printf("a %p\n", a);
13
14    // Проверка успешности выделения
15    if (a == NULL)
16    {
17        fprintf(stderr, "Memory allocation error\n");
18        return -1;
19    }
20
21    // Использование памяти
22    for (int i = 0; i < n; i++)
23        printf("%d ", a[i]);
```



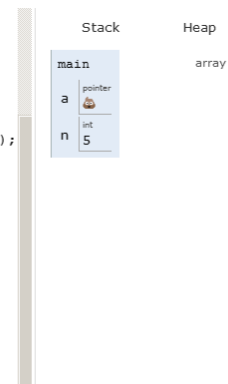
2. Сразу после выделения памяти

```
3
4 int main(void)
5 {
6     int *a = NULL;
7     int n = 5;
8
9     // Выделение памяти
10    a = calloc(n, sizeof(int));
11
12    printf("a %p\n", a);
13
14    // Проверка успешности выделения
```



4. Сразу после освобождения

```
12    printf("a %p\n", a);
13
14    // Проверка успешности выделения
15    if (a == NULL)
16    {
17        fprintf(stderr, "Memory allocation error\n");
18        return -1;
19    }
20
21    // Использование памяти
22    for (int i = 0; i < n; i++)
23        printf("%d ", a[i]);
24
25    // Освобождение памяти
26    free(a);
27
28    return 0;
```



free

```
#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.

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
```

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.

Возвращение динамического массива из функции (прототип)

- Как возвращаемое значение

```
int* create_array(FILE *f, size_t *n);
```

- Как параметр функции

```
int create_array(FILE *f, int **arr, size_t *n);
```

Возвращение динамического массива из функции (вызов)

- Как возвращаемое значение

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

- Как параметр функции

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

Типичные ошибки (1)

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

Типичные ошибки (2)

- 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 ☹

Отладчик использования памяти (англ. memory debugger)

Отладчик использования памяти – специальное программное обеспечение для обнаружения ошибок программы при работе с памятью, например, таких как утечки памяти и переполнение буфера. [wiki]

- **valgrind**
- Dr. Memory

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

Указатель на функцию

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);
```

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);
```

Особенности использования указателей на функции (1)

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;
```

Особенности использования указателей на функции (2)

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

```
int (*p3)(int, int) = *add;  
int (*p4)(int, int) = *****add;
```

Указатели на функции можно сравнивать

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

Особенности использования указателей на функции (3)

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);
```

Указатель на функцию и void* (1)

C99 6.3.2.3 #1

A pointer to void may be converted to or from a pointer to any incomplete or object type. A pointer to any incomplete or object type may be converted to a pointer to void and back again; the result shall compare equal to the original pointer.

Функция - *не объект* в терминологии стандарта.

Указатель на функцию и void* (2)

C99 6.3.2.3 #8

A pointer to a function of one type may be converted to a pointer to a function of another type and back again; the result shall compare equal to the original pointer. If a converted pointer is used to call a function whose type is not compatible with the pointed-to type, the behavior is undefined.

Указатель на функцию и void* (3)

Согласно C99 6.3.2.3 #1 и C99 6.3.2.3 #8, указатель на функцию не может быть преобразован к указателю на void и наоборот.

Но POSIX требует, чтобы такое преобразование было возможно при работе с динамическими библиотеками.

- C99 J.5.7 Function pointer casts (расширение стандарта)
- POSIX dlsym RATIONALE
- Generic Function Pointer C2X (будущее (?))

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.

VLA, alloca

Variable Length Array

```
#include <stdio.h>

int main(void)
{
    int n;

    printf("n: ");
    scanf("%d", &n);

    int a[n];

    printf("n = %d, sizeof(a) = %d\n",
           n, (int) sizeof(a));
```

```
    for (int i = 0; i < n; i++)
        a[i] = i;

    for (int i = 0; i < n; i++)
        printf("%d ", a[i]);

    return 0;
}
```

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.

alloca

```
#include <alloca.h>
```

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

Функция *alloca* выделяет область памяти, размером `size` байт, на стеке. Функция возвращает указатель на начало выделенной области. Эта область автоматически освобождается, когда функция, которая вызвала *alloca*, возвращает управления вызывающей стороне.

Если выделение вызывает переполнение стека, поведение программы не определено.

alloca

<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;</pre>	<pre> 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);
 ...
 }
}
```

make

# 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
```

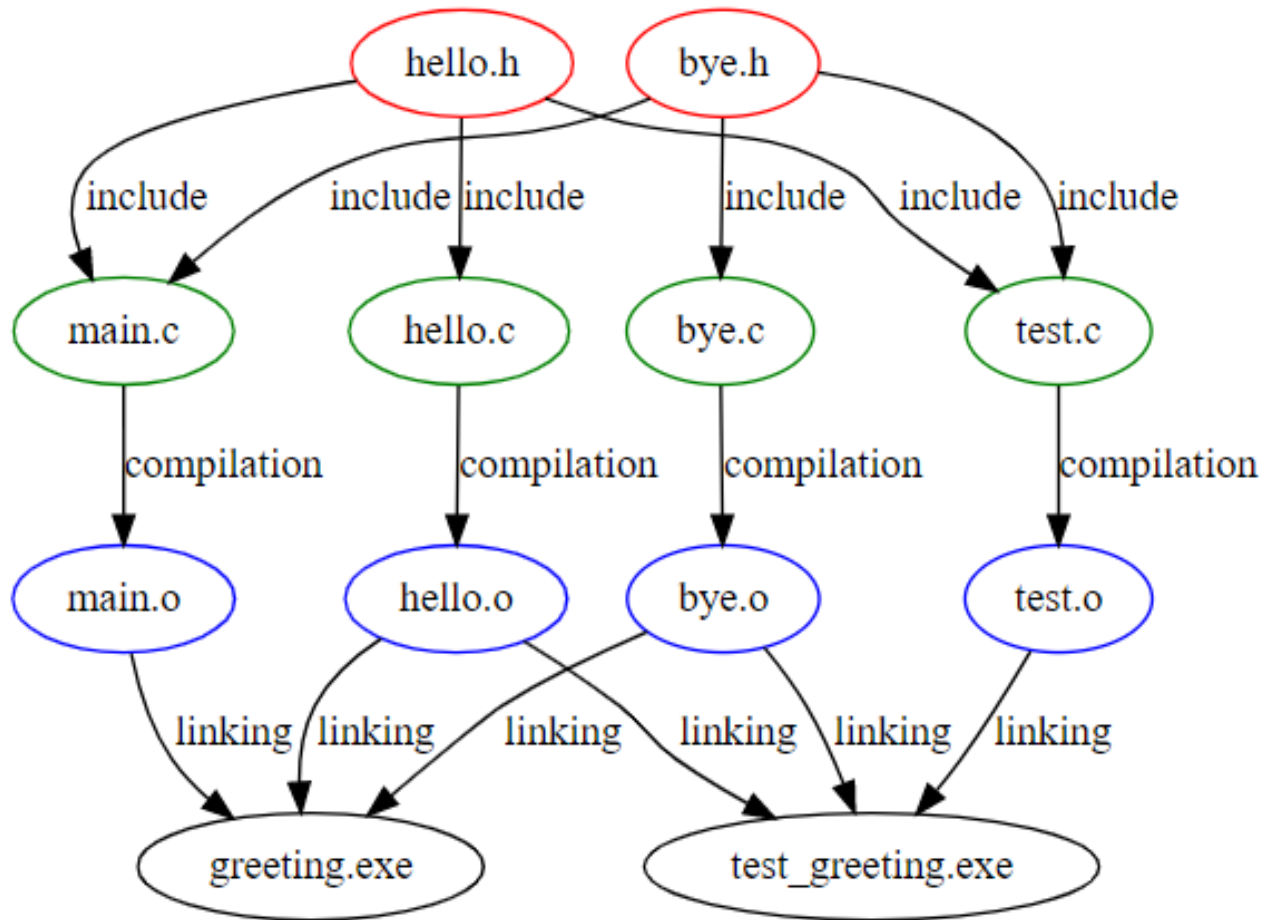
## Компоновка

```
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
```

# Граф зависимостей



# Утилита make

**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.

# Сценарий сборки проекта

цель: зависимость\_1 ... зависимость\_n

[tab]команда\_1

[tab]команда\_2

...

[tab]команда\_m

что создать/сделать: из чего создать

как создать/что сделать



# Простой сценарий сборки

```
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
```

# Алгоритм работы make (1)

## Первый запуск make

- make читает сценарий сборки и начинает выполнять первое правило

```
greeting.exe : hello.o bye.o main.o
gcc -o greeting.exe hello.o bye.o main.o
```
- Для выполнения этого правила необходимо сначала обработать зависимости

```
hello.o bye.o main.o
```
- make ищет правило для создания файла hello.o

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

# Алгоритм работы make (2)

## Первый запуск make

- Файл `hello.o` отсутствует, файлы `hello.c` и `hello.h` существуют. Следовательно, правило для создания `hello.o` может быть выполнено  

```
gcc -std=c99 -Wall -Werror -pedantic -c hello.c
```
- Аналогично обрабатываются зависимости `bye.o` и `main.o`.
- Все зависимости получены, теперь правило для построения `greeting.exe` может быть выполнено  

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

# Алгоритм работы make (3)

Второй запуск make (hello.c был изменен)

- make читает сценарий сборки и начинает выполнять первое правило

```
greeting.exe : hello.o bye.o main.o
gcc -o greeting.exe hello.o bye.o main.o
```
- Для выполнения этого правила необходимо сначала обработать зависимости

```
hello.o bye.o main.o
```
- make ищет правило для создания файла hello.o

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

# Алгоритм работы make (4)

Второй запуск make (hello.c был изменен)

- Файлы hello.o, hello.c и hello.h существуют, но время изменения hello.o меньше времени изменения hello.c. Придется пересоздать файл hello.o

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

# 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 -Wall -Werror -pedantic
```

```
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
```

# Использование переменных и комментариев (3)

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

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

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

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

# Фиктивные (.PHONY) цели

В make-файле могут встречаться цели, которые не являются именами файлов. Такие цели называются *фиктивными* и используются для выполнения каких-то действий (очистки, установки и т.п.).

Чтобы make даже не пытался интерпретировать таких как цели как имена файлов их помечают атрибутом .PHONY.

```
.PHONY: clean
```

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

# 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 $<
```

# Автоматические переменные (3)

```
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 $@
```

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

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

# Автоматические переменные (4)

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

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

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

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



# 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
```

# 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)
```

# 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) *.o *.exe
```

# 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

# Присваивание переменных, зависящих от цели (2)

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

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

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

```
.PHONY : clean debug release
```

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

# Генерация зависимостей (1)

# Компилятор

CC := gcc

# Опции компиляции

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

# Общие объектные файлы

OBJS := hello.o bye.o

# Все с-файлы (или так SRCS := \$(wildcard \*.c))

SRCS := hello.c bye.c test.c main.c

greeting.exe : \$(OBJS) main.o

\$(CC) \$^ -o \$@

# 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
с расширением ".c" на имена с расширением ".d"
include $(SRCS:.c=.d)
```

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



# Функции в make

## Вызов функции

```
$(function_name [arguments])
```

## Функция patsubs

```
$(patsubst pattern, replacement, text)
```

```
cfiles := main.c hello.c bye.c
```

```
objs := $(patsubst %.c, %.o, $(cfiles))
```

или краткая форма

```
objs := $(cfiles:%.c=%.o)
```

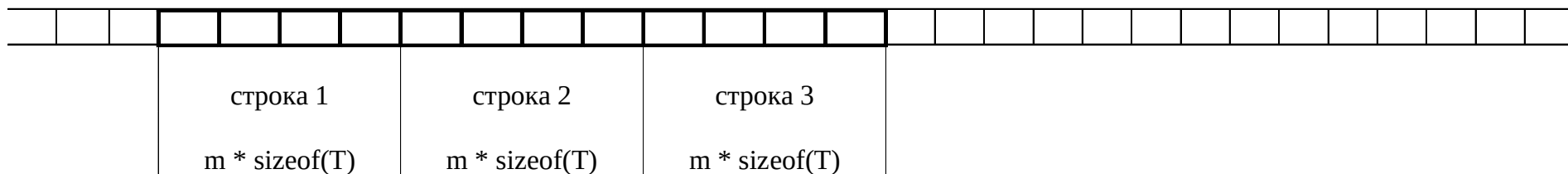
# Особенности выполнения команд

- Ненулевой код возврата может прервать выполнение сценария.
- Каждая команда выполняется в своем shell.

# Динамические матрицы

# Матрица как одномерный массив

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



|     |     |     |     |     |     |     |     |     |     |     |     |     |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| k   | 0   | 1   | 2   | 3   | 4   | 5   | 6   | 7   | 8   | 9   | 10  | 11  |
| i;j | 0;0 | 0;1 | 0;2 | 0;3 | 1;0 | 1;1 | 1;2 | 1;3 | 2;0 | 2;1 | 2;2 | 2;3 |

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

# Матрица как одномерный массив

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

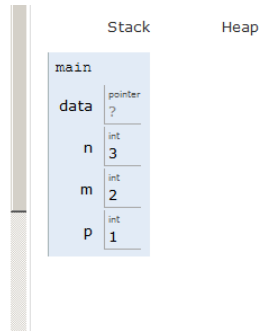
// Выделение памяти под "матрицу"
data = malloc(n * m * sizeof(double));
if (data)
{
 // Работа с "матрицей"
 for (size_t i = 0; i < n; i++)
 for (size_t j = 0; j < m; j++)
 // Обращение к элементу i, j
 data[i * m + j] = 0.0;

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

# Матрица как одномерный массив

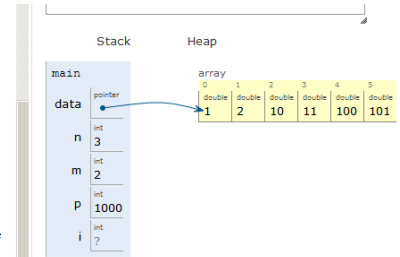
## 1. Перед выделением памяти

```
4 int main(void)
5 {
6 double *data;
7 int n = 3, m = 2, p = 1;
8
9 // Выделение памяти под "матрицу"
10 data = malloc(n * m * sizeof(double));
11 if (data)
12 {
13 for (int i = 0; i < n; i++)
14 {
15 for (int j = 0; j < m; j++)
16 // Обращение к элементу i, j
```



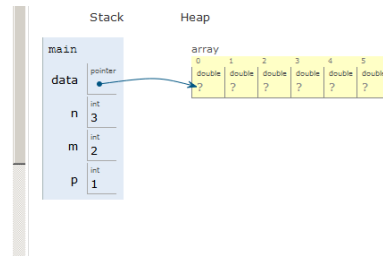
## 3. Использование выделенной памяти

```
14 {
15 for (int j = 0; j < m; j++)
16 // Обращение к элементу i, j
17 data[i * m + j] = p + j;
18
19 p *= 10;
20 }
21
22 for (int i = 0; i < n; i++)
23 {
24 for (int j = 0; j < m; j++)
25 printf("%5.1f", data[i * m + j]);
26 }
```



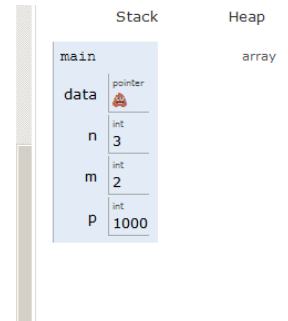
## 2. Сразу после выделения памяти

```
4 int main(void)
5 {
6 double *data;
7 int n = 3, m = 2, p = 1;
8
9 // Выделение памяти под "матрицу"
10 data = malloc(n * m * sizeof(double));
11 if (data)
12 {
13 for (int i = 0; i < n; i++)
14 {
15 for (int j = 0; j < m; j++)
16 // Обращение к элементу i, j
```



## 4. Сразу после освобождения

```
19 p *= 10;
20 }
21
22 for (int i = 0; i < n; i++)
23 {
24 for (int j = 0; j < m; j++)
25 printf("%5.1f", data[i * m + j]);
26
27 printf("\n");
28 }
29
30 // Освобождение памяти
31 free(data);
```



# Матрица как одномерный массив

Преимущества:

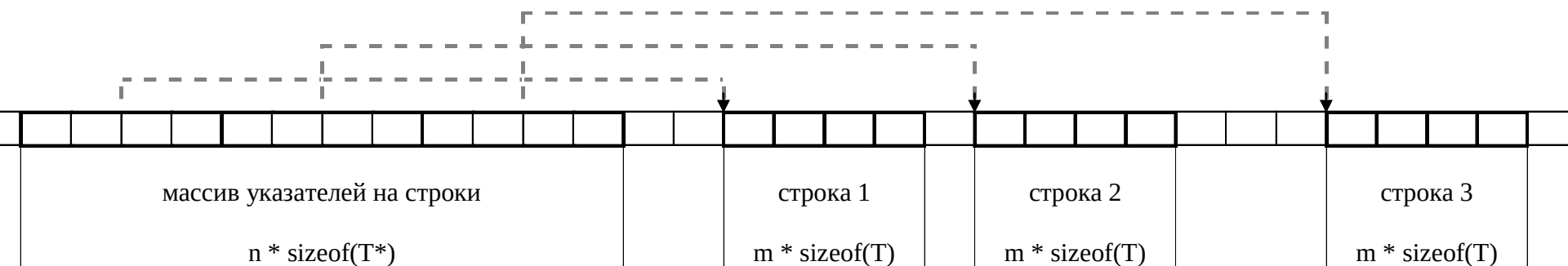
- Простота выделения и освобождения памяти.
- Возможность использовать как одномерный массив.

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





# Матрица как массив указателей

*Алгоритм выделения памяти*

*Вход:* количество строк ( $n$ ) и количество столбцов ( $m$ )

*Выход:* указатель на массив строк матрицы ( $p$ )

- Выделить память под массив указателей ( $p$ )
- Обработать ошибку выделения памяти
- В цикле по количеству строк матрицы ( $0 \leq i < n$ )
  - Выделить память под  $i$ -ую строку матрицы ( $q$ )
  - Обработать ошибку выделения памяти
  - $p[i]=q$

# Матрица как массив указателей

*Алгоритм освобождения памяти*

*Вход:* указатель на массив строк матрицы (p) и количество строк (n)

- В цикле по количеству строк матрицы ( $0 \leq i < n$ )
  - Освободить память из-под  $i$ -ой строки матрицы
- Освободить память из-под массива указателей (p)

# Матрица как массив указателей

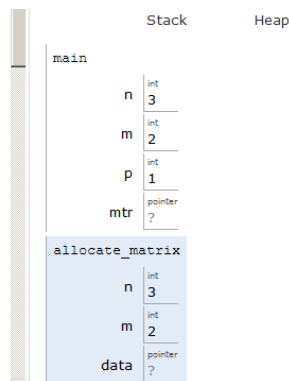
```
void free_matrix(double **data, size_t n);

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;
}
```

# Матрица как массив указателей

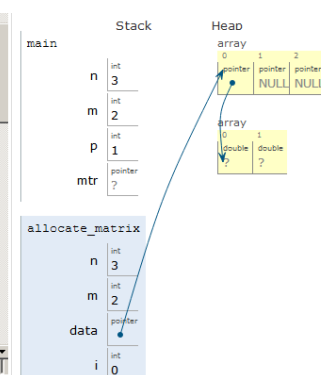
## 1. Перед выделением памяти

```
6 #include <stdlib.h>
7
8 void free_matrix(double **data, int n);
9
10 double** allocate_matrix(int n, int m)
11 {
12 double **data;
13 data = calloc(n, sizeof(double*));
14 if (!data)
15 return NULL;
16 for (int i = 0; i < n; i++)
17 {
```



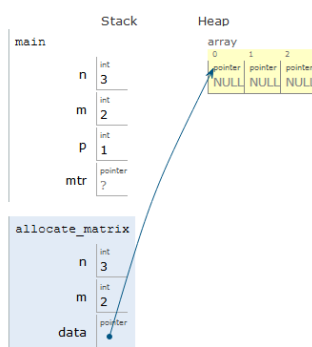
## 3. Выделена память под первую строку

```
16 data = calloc(n, sizeof(double*));
17 if (!data)
18 return NULL;
19
20 for (int i = 0; i < n; i++)
21 {
22 data[i] = (double*) malloc(m * sizeof(double));
23 if (!data[i])
24 {
25 free_matrix(data, n);
26 return NULL;
27 }
28 }
29
30 return data;
```



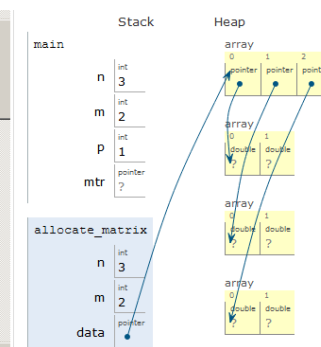
## 2. Выделена память под массив указателей

```
6 #include <stdlib.h>
7
8 void free_matrix(double **data, int n);
9
10 double** allocate_matrix(int n, int m)
11 {
12 double **data;
13 data = calloc(n, sizeof(double*));
14 if (!data)
15 return NULL;
16 for (int i = 0; i < n; i++)
17 {
```



## 4. Окончание выделения памяти

```
16 data = calloc(n, sizeof(double*));
17 if (!data)
18 return NULL;
19
20 for (int i = 0; i < n; i++)
21 {
22 data[i] = (double*) malloc(m * sizeof(double));
23 if (!data[i])
24 {
25 free_matrix(data, n);
26 return NULL;
27 }
28 }
29
30 return data;
```



# Матрица как массив указателей

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

 free(data);
}
```

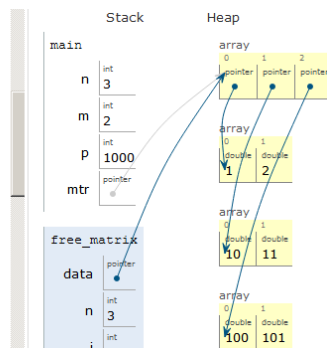
# Матрица как массив указателей

## 1. Перед освобождением памяти

```

29 }
30
31 return data;
32 }
33
34
35 void free_matrix(double **data, int n)
36 {
37 for (int i = 0; i < n; i++)
38 if (data[i])
39 free(data[i]);
40
41 free(data);
42 }
43
44

```

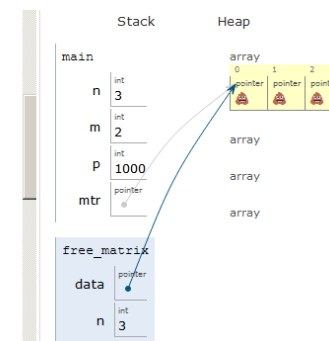


## 3. Освобождена память из-под строк

```

29 }
30
31 return data;
32 }
33
34
35 void free_matrix(double **data, int n)
36 {
37 for (int i = 0; i < n; i++)
38 if (data[i])
39 free(data[i]);
40
41 free(data);
42 }
43
44

```

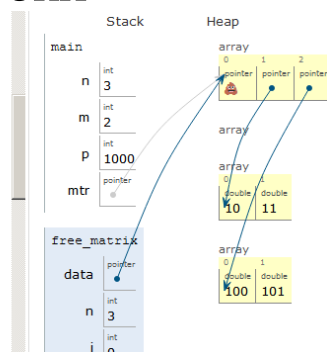


## 2. Освобождена память из-под первой строки

```

29 }
30
31 return data;
32 }
33
34
35 void free_matrix(double **data, int n)
36 {
37 for (int i = 0; i < n; i++)
38 if (data[i])
39 free(data[i]);
40
41 free(data);
42 }
43
44

```

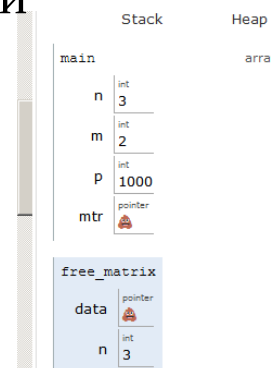


## 4. Освобождена память из-под массива указателей

```

29 }
30
31 return data;
32 }
33
34
35 void free_matrix(double **data, int n)
36 {
37 for (int i = 0; i < n; i++)
38 if (data[i])
39 free(data[i]);
40
41 free(data);
42 }
43
44

```

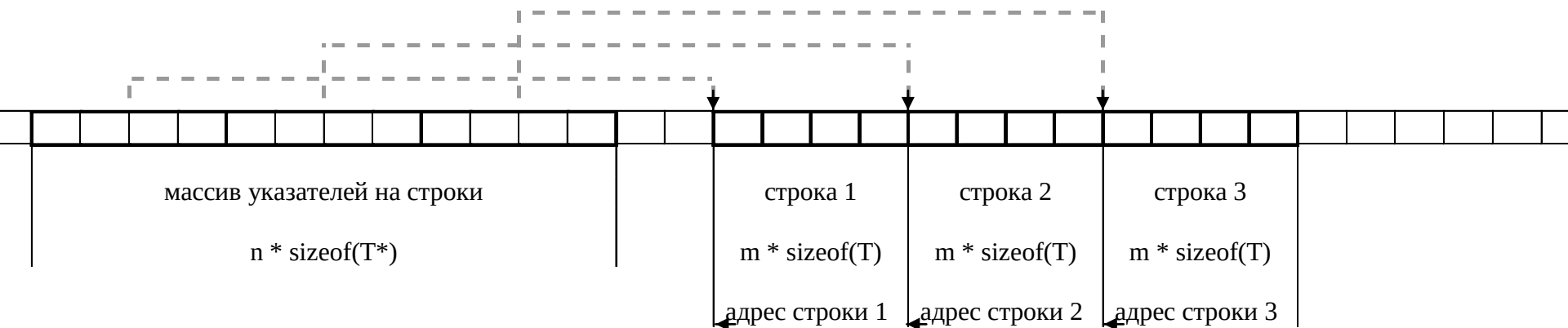


# Матрица как массив указателей

- 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





# Объединение подходов (1)

*Алгоритм выделения памяти*

*Вход:* количество строк ( $n$ ) и количество столбцов ( $m$ )

*Выход:* указатель на массив строк матрицы ( $p$ )

- Выделить память под массив указателей на строки ( $p$ )
- Обработать ошибку выделения памяти
- Выделить память под данные (т.е. под строки,  $q$ )
- Обработать ошибку выделения памяти
- В цикле по количеству строк матрицы ( $0 \leq i < n$ )
  - $p[i]$ =адрес  $i$ -ой строки в массиве  $q$

# Объединение подходов (1)

*Алгоритм освобождения памяти*

*Вход: указатель на массив строк матрицы (p)*

- Освободить память из-под данных (адрес данных = адрес строки 0)
- Освободить память из-под массива указателей (p)

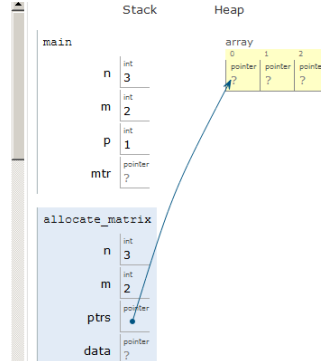
# Объединение подходов (1)

```
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;
}
```

# Объединение подходов (1)

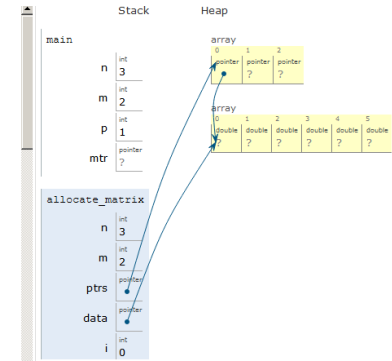
## 1. Выделение памяти по массив указателей

```
1 double** allocate_matrix(int n, int m)
2 {
3 double **ptrs = malloc(n * sizeof(double*));
4
5 if (ptrs)
6 {
7 double *data = malloc(n * m * sizeof(double));
8
9 for(int i = 0; i < n; i++)
10 ptrs[i] = data + i * m;
11 }
12 return ptrs;
13 }
14
15 void free_matrix(double **ptrs)
16 {
17 free(ptrs[0]);
18 }
```



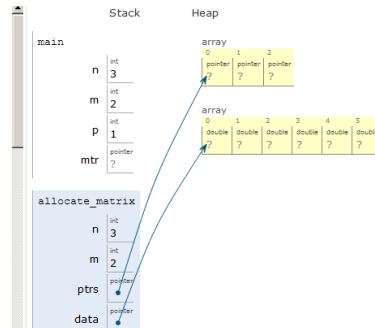
## 3. Вычисление адреса первой строки

```
1 double** allocate_matrix(int n, int m)
2 {
3 double **ptrs = malloc(n * sizeof(double*));
4
5 if (ptrs)
6 {
7 double *data = malloc(n * m * sizeof(double));
8
9 for(int i = 0; i < n; i++)
10 ptrs[i] = data + i * m;
11 }
12 return ptrs;
13 }
14
15 void free_matrix(double **ptrs)
16 {
17 free(ptrs[0]);
18 }
19
20 free(ptrs);
```



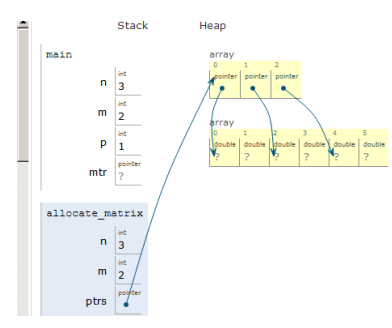
## 2. Выделение памяти под данные

```
1 double** allocate_matrix(int n, int m)
2 {
3 double **ptrs = malloc(n * sizeof(double*));
4
5 if (ptrs)
6 {
7 double *data = malloc(n * m * sizeof(double));
8
9 for(int i = 0; i < n; i++)
10 ptrs[i] = data + i * m;
11 }
12 return ptrs;
13 }
14
15 void free_matrix(double **ptrs)
16 {
17 free(ptrs[0]);
18 }
```



## 4. Адреса всех строк вычислены

```
1 double** allocate_matrix(int n, int m)
2 {
3 double **ptrs = malloc(n * sizeof(double*));
4
5 if (ptrs)
6 {
7 double *data = malloc(n * m * sizeof(double));
8
9 for(int i = 0; i < n; i++)
10 ptrs[i] = data + i * m;
11 }
12 return ptrs;
13 }
14
15 void free_matrix(double **ptrs)
16 {
17 free(ptrs[0]);
18 }
```



# Объединение подходов (1)

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

 free(ptrs);
}
```

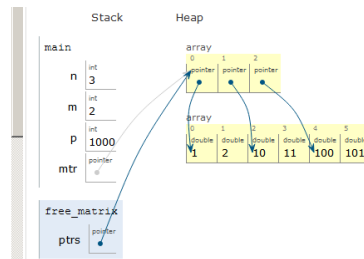
ВНИМАНИЕ

Здесь скрывается потенциальная ошибка.

# Объединение подходов (1)

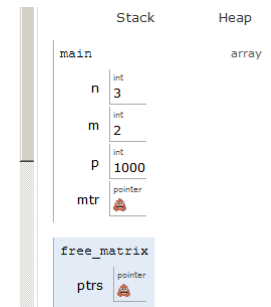
## 1. Перед освобождением памяти

```
10 ptrs[i] = data + i * m;
11 }
12
13 return ptrs;
14 }
15
16 void free_matrix(double **ptrs)
17 {
18 free(ptrs[0]);
19
20 free(ptrs);
21 }
22
23
```



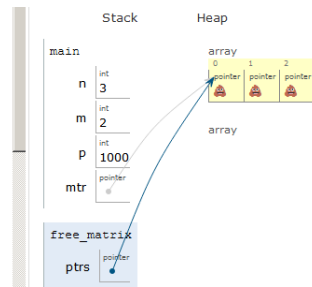
## 3. Память под указатели освобождена

```
10 ptrs[i] = data + i * m;
11 }
12
13 return ptrs;
14 }
15
16 void free_matrix(double **ptrs)
17 {
18 free(ptrs[0]);
19
20 free(ptrs);
21 }
22
23
```



## 2. Память под данные освобождена

```
10 ptrs[i] = data + i * m;
11 }
12
13 return ptrs;
14 }
15
16 void free_matrix(double **ptrs)
17 {
18 free(ptrs[0]);
19
20 free(ptrs);
21 }
22
23
```



# Объединение подходов (1)

## Преимущества:

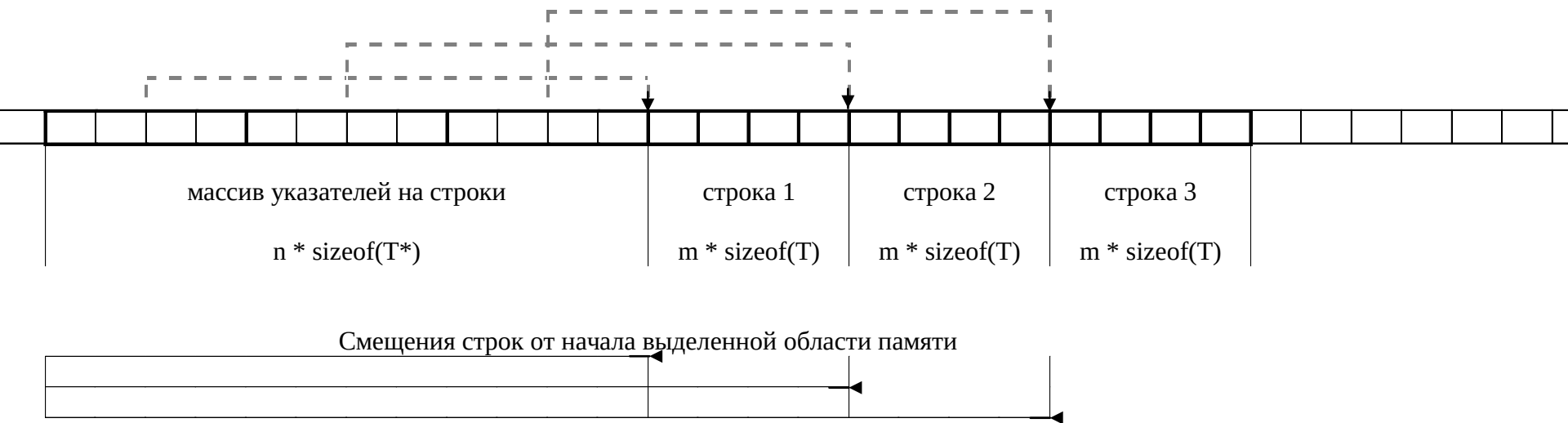
- Относительная простота выделения и освобождения памяти.
- Возможность использовать как одномерный массив.
- Перестановка строк через обмен указателей. (Возможна ошибка, см. слайд 19.)

## 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





# Объединение подходов (2)

*Алгоритм выделения памяти*

*Вход:* количество строк ( $n$ ) и количество столбцов ( $m$ )

*Выход:* указатель на массив строк матрицы ( $p$ )

- Выделить память под массив указателей на строки и элементы матрицы ( $p$ )
- Обработать ошибку выделения памяти
- В цикле по количеству строк матрицы ( $0 \leq i < n$ )
  - Вычислить адрес  $i$ -ой строки матрицы ( $q$ )
  - $p[i]=q$

# Объединение подходов (2)

```
double** allocate_matrix(size_t n, size_t m)
{
 double **data = malloc(n * sizeof(double*) +
 n * m * sizeof(double));

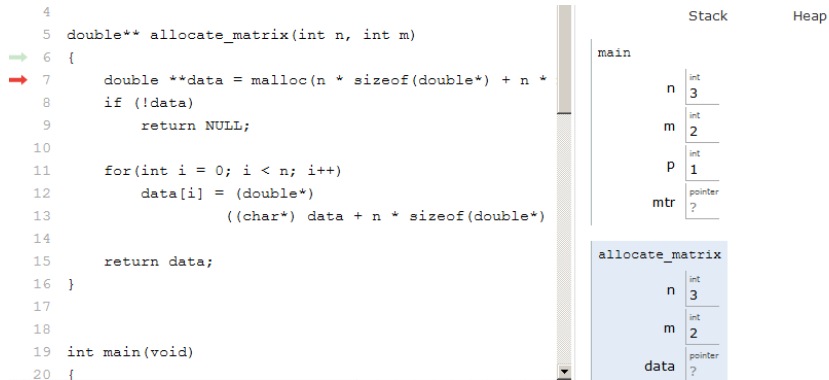
 if (!data)
 return NULL;

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

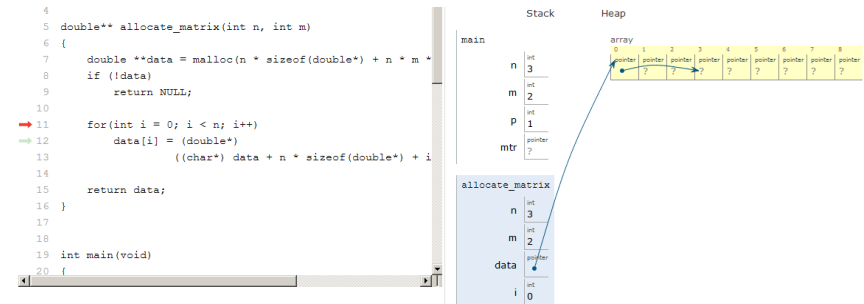
 return data;
}
```

# Объединение подходов (2)

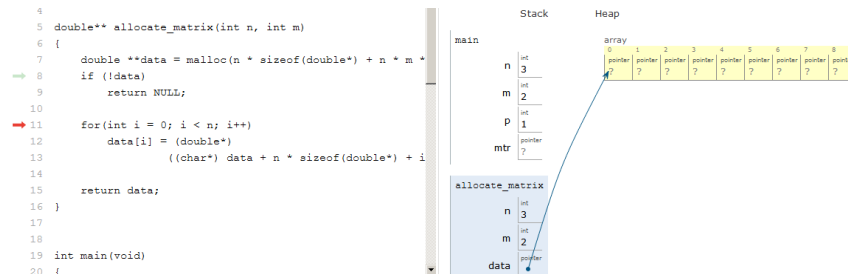
## 1. Перед выделением памяти



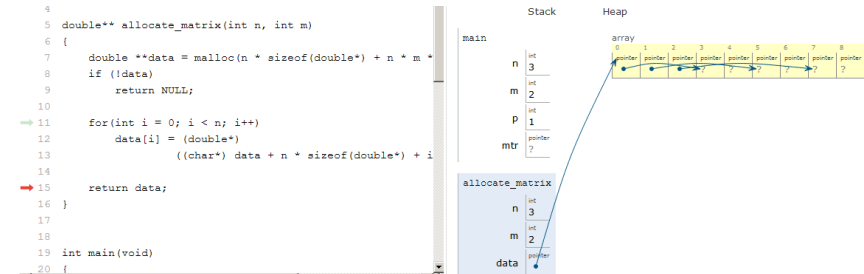
## 3. Вычисление адреса первой строки



## 2. Выделение памяти



## 4. Адреса всех строк вычислены



# Объединение подходов (2)

// Найдите ошибки, если они есть

```
double** allocate_matrix(int n, int m)
{
 double **matrix = malloc(n * sizeof(double*) +
 m * sizeof(double));

 if (matrix == NULL)
 free(matrix);

 matrix[0] = matrix + n;
 for (int i = 1; i < n; i++)
 matrix[i] = matrix[0] + m * i;

 return matrix;
}
```

# Объединение подходов (2)

// Ошибки выделены красным

```
double** allocate_matrix(int n, int m)
{
 double **matrix = malloc(n * sizeof(double*) +
 m * sizeof(double));

 if (matrix == NULL)
 free(matrix);

 matrix[0] = matrix + n;
 for (int i = 1; i < n; i++)
 matrix[i] = matrix[0] + m * i;

 return matrix;
}
```

# 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.

# Передача матрицы в функцию

- Внимательно изучить текст программы-примера.
- Ответить на вопросы, заданные в комментариях.
- Проверить правильность ответа с помощью компилятора.
  - test\_05.c
  - test\_06.c
  - test\_07.c

# Идея для реализации тестов

```
#define N 3
#define M 2

void foo_2(int **a, size_t n, size_t m)
{
}

int main(void)
{
 int a[N][M];
 size_t n = N, m = M;
 int* b[N] = {a[0], a[1], a[2]};

 foo_2(b, n, m);

 return 0;
}
```



# Строки/структуры и динамическое выделение памяти

(часть 1)

# 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 (+ strdup)

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

 return 0;
}
```

# Строки и динамическая память

```
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);
}
```

# Строки и динамическая память

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

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

lineptr - либо NULL (и тогда в n - 0), либо указатель на буфер, выделенный с помощью malloc (и тогда в n - размер буфера). Если буфера не хватает, он будет перевыделен.

```
// с glibc 2.10
```

```
#define _POSIX_C_SOURCE 200809L
```

```
// до glibc 2.10
```

```
#define _GNU_SOURCE
```

# Строки и динамическая память

```
int n, m;

n = snprintf(NULL, 0, "My name is %s. I live in %s.", NAME, CITY);
if (n > 0)
{
 char *line = malloc((n + 1) * sizeof(char));
 if (line)
 {
 m = snprintf(line, n + 1, "My name is %s. I live in %s.", NAME, CITY);

 printf("n = %d, m = %d\n", n, m);
 printf("%s\n", line);

 free(line);
 }
}
```

# Строки и динамическая память

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

// ...
{
 char *line = NULL;
 int n;

 n = asprintf(&line, "My name is %s. I live in %s.", NAME, CITY);
 if (n > 0)
 {
 printf("n = %d\n", n);
 printf("%s\n", line);

 free(line);
 }
}
```

# 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 ссылаться поля структуры.



# Структуры с полями-указателями

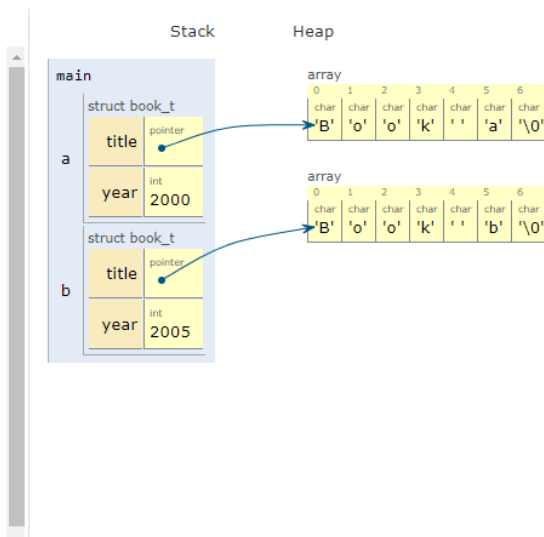
Иногда стратегия «поверхностного копирования» может приводить к ошибкам.

До присваивания

После присваивания

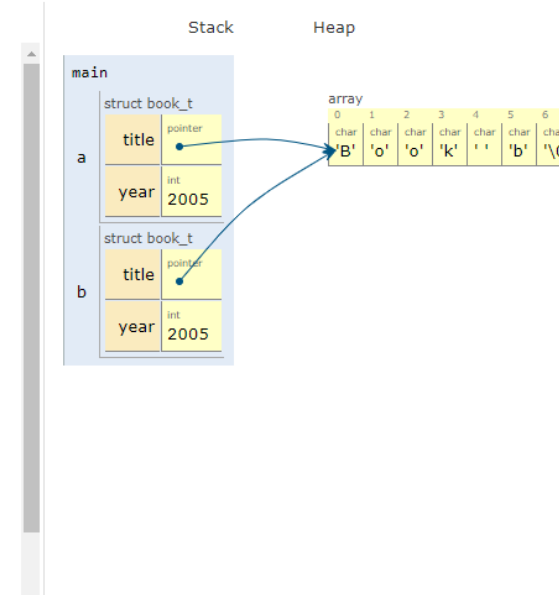
C (gcc 4.8, C11)  
(known limitations)

```
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 } a = { 0 }, b = { 0 };
11
12 a.title = strdup("Book a");
13 a.year = 2000;
14
15 b.title = strdup("Book b");
16 b.year = 2005;
17 → a = b;
18
19 → a = b;
20
```



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 } a = { 0 }, b = { 0 };
11
12 a.title = strdup("Book a");
13 a.year = 2000;
14
15 b.title = strdup("Book b");
16 b.year = 2005;
17
18 a = b;
19 → a = b;
20
21 → free(a.title);
22 → free(b.title);
```



# Структуры с полями-указателями

Стратегия так называемого «глубокого копирования» (англ., *deep copying*) 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;
}
```

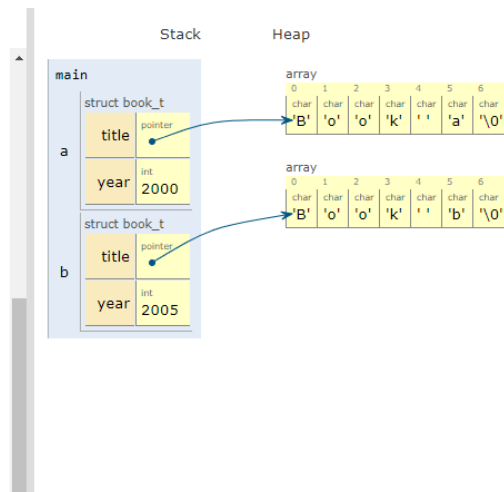
# Структуры с полями-указателями

## Стратегия «глубокого копирования».

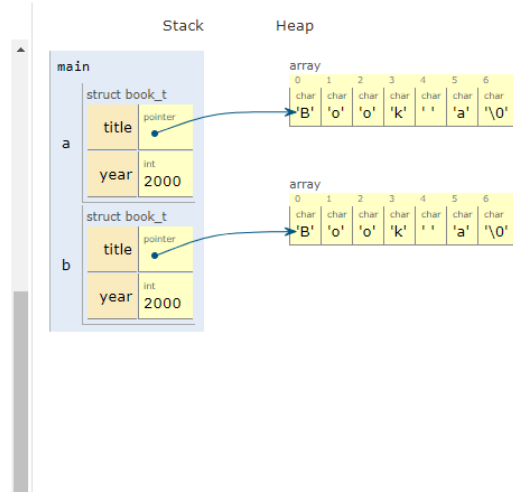
До копирования

После копирования

```
44 C (gcc 4.8, C11)
45 (known limitations)
23 dst->year = src->year;
24
25 return 0;
26 }
27
28 int main(void)
29 {
30 struct book_t a = { 0 }, b = { 0 };
31
32 a.title = strdup("Book a");
33 a.year = 2000;
34
35 b.title = strdup("Book b");
36 b.year = 2005;
37
38 book_copy(&b, &a);
39
40 free(a.title);
41 free(b.title);
42 }
```



```
44 C (gcc 4.8, C11)
45 (known limitations)
23 dst->year = src->year;
24
25 return 0;
26 }
27
28 int main(void)
29 {
30 struct book_t a = { 0 }, b = { 0 };
31
32 a.title = strdup("Book a");
33 a.year = 2000;
34
35 b.title = strdup("Book b");
36 b.year = 2005;
37
38 book_copy(&b, &a);
39
40 free(a.title);
41 free(b.title);
42 }
```



# 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 освободить память?
 free(pbook);
}
```

# Строки/структуры и динамическое выделение памяти

(часть 2)

# 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];
};
```

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

```
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;
}
```

# Flexible array member vs pointer field

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

# Динамически расширяемые массивы

# Функция realloc

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

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

# 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
```

# 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;
 }
}
```

# Динамически расширяемые массивы: особенности реализации

- Удвоение размера массива при каждом вызове `realloc` сохраняет средние «ожидаемые» затраты на копирование элемента.
- Поскольку адрес массива может измениться, программа должна обращаться к элементам массива по индексам.
- Благодаря маленькому начальному размеру массива, программа сразу же «проверяет» код, реализующий выделение памяти.

# 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`, `memscr` или `memmove`?
  - `for`
  - `memscr` НЕЛЬЗЯ (как и `strscr`), `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.

# Сложные объявления



# Чтение сложных объявлений

Не возникает проблем с чтением следующих объявлений:

```
int foo[5];
char *foo;
double foo(void).
```

Но как только объявление становится сложнее, трудно точно сказать что это. Например,

```
char *(*(**foo[][8]))[];
```

# 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);`

# Чтение сложных объявлений

(семантические ограничения)

- Невозможно создать массив функций.  
`int a[10](int);`
- Функция не может возвращать функцию.  
`int g(int)(int);`
- Функция не может вернуть массив.  
`int f(int)[];`
- В массива только левая лексема `[]` может быть пустой.
- Тип `void` ограниченный.  
`void x;           // ошибка`  
`void x[5];       // ошибка`

# Чтение сложных объявлений

(использование typedef для упрощения)

```
int *(*x[10])(void);
```

```
typedef int* func_t(void);
```

```
typedef func_t* func_ptr;
```

```
typedef func_ptr* funt_ptr_arr[10];
```

```
funt_ptr_arr x;
```

# Списки

# Массив

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

Преимущества и недостатки массива объясняются стратегией выделения памяти: память под все элементы выделяется в одном блоке.

“+” Минимальные накладные расходы.

“+” Константное время доступа к элементу.

“–” Хранение меняющегося набора значений.



# СВЯЗНЫЙ СПИСОК

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.

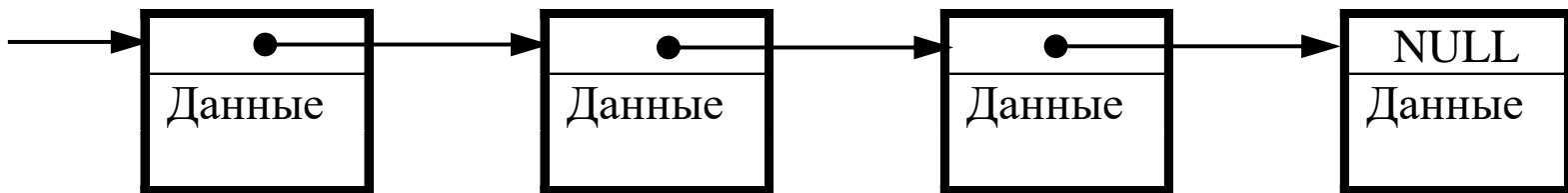
*Связный список* — это набор элементов, причем каждый из них является частью узла, который также содержит ссылку на [узел. Седжвик (с)] следующий и/или предыдущий узел списка.

# СВЯЗНЫЙ СПИСОК

*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).



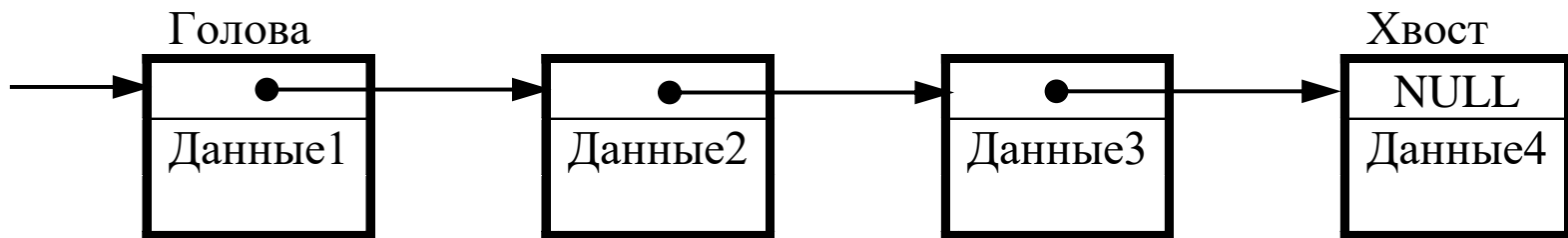
# СВЯЗНЫЙ СПИСОК

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

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

# Линейный односвязный список

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



# Линейный односвязный список

Узел, на который нет указателя, является первым элементом списка. Обычно этот узел называется *головой списка*.

Последний элемент списка никуда не ссылается (ссылается на NULL). Обычно этот узел называется *хвостом списка*.

# Линейный односвязный список

## Свойства односвязного списка

- Передвигаться можно только в сторону конца списка.
- Узнать адрес предыдущего элемента, опираясь только на содержимое текущего узла, нельзя.

# Линейный односвязный список

## Базовые операции

- Добавить элемент в начало или конец списка.
- Найти указанный элемент.
- Удалить элемент.
- Добавить новый элемент до или после указанного.

# Элемент списка

```
struct person_t
{
 const char *name;
 int born_year;

 struct person_t *next;
};
```

```
typedef struct person_t person_t;
```

```
struct person_t
{
 const char *name;
 int born_year;

 person_t *next;
};
```



# Создание/удаление узла списка

```
struct person_t* person_create(const char *name, int born_year)
{
 struct person_t *pers = malloc(sizeof(struct person_t));

 if (pers)
 {
 pers->name = name;
 pers->born_year = born_year;
 pers->next = NULL;
 }

 return pers;
}

void person_free(struct person_t *pers)
{
 free(pers) ;
}
```

# Добавление элемента в список

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

NB: функции, изменяющие список, должны возвращать указатель на новый первый элемент.

# Добавление элемента в список

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

## ИСПОЛЬЗОВАНИЕ

```
head = add_front(head, pers);
```

# Добавление элемента в список

```
struct person_t* list_add_end(struct person_t *head,
 struct person_t *pers)
{
 struct person_t *cur = head;

 if (!head)
 return pers;

 for (; cur->next; cur = cur->next)
 ;

 cur->next = pers;

 return head;
}
```

# Добавление элемента в список

Добавление элемента в конец нашего простого списка – операция порядка  $O(N)$ . Чтобы добиться времени  $O(1)$ , можно завести отдельный указатель на конец списка.

```
struct list_t
{
 struct person_t *head;
 struct person_t *tail;
};
```

# Поиск элемента в списке

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

 return NULL;
}
```

Поиск занимает время порядка  $O(N)$  и эту оценку не улучшить.

# Обработка всех элементов списка

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

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

# Обработка всех элементов списка

```
// печать информации из элемента списка
void person_print(struct person *pers, void *arg)
{
 char *fmt = arg;
 printf(fmt, pers->name, pers->born_year);
}

// list_apply(l1, person_print, "l1: %s %d\n");

// подсчет количества элементов списка
void person_count(struct person *pers, void *arg)
{
 int *counter = arg;
 (*counter)++;
}

// list_apply(l2, person_count, &n); // где int n = 0;
```



# Освобождение списка

Так делать НЕЛЬЗЯ! Почему?

```
void list_free_all(struct person_t *head)
{
 for (; head; head = head->next)
 person_free(head) ;
}
```

# Освобождение списка

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

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

Наша функция `free_all` не освобождает память из поля `name` (см. `person_create`).

# Удаление элемента по имени

```
struct person* del_by_name(struct person *head,
 const char *name)
{
 struct person *cur, *prev = NULL;

 for (cur = head; cur; cur = cur->next)
 {
 if (strcmp(cur->name, name) == 0)
 {
 if (prev)
 prev->next = cur->next;
 else
 head = cur->next;
 person_free(cur);
 return head;
 }
 prev = cur;
 }

 return NULL;
}
```

У этой реализации есть недостаток, который не замечали лет пять. Какой?

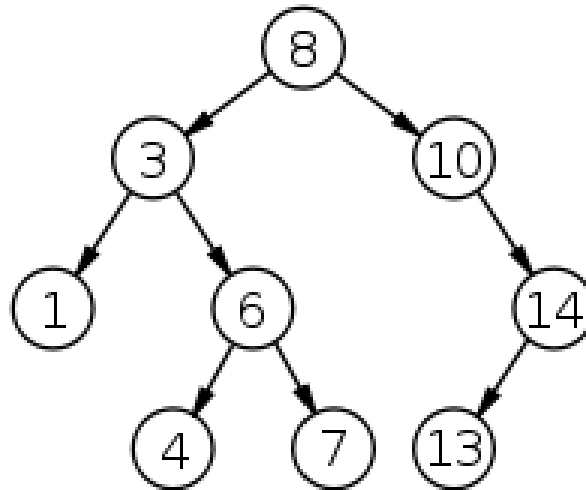
# Списки: дальнейшее развитие

- Представление элемента списка
  - Универсальный элемент (`void*`).
- Двусвязные списки
  - Требуется больше ресурсов.
  - Поиск последнего и удаление текущего – операции порядка  $O(1)$ .

# Двоичные деревья поиска

# Двоичное дерево поиска

- *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.



# Двоичное дерево поиска

## Базовые операции

- Добавление узла.
- Поиск узла.
- Обход дерева.
- Удаление узла.

# Узел дерева

```
struct tree_node
{
 const char *name;

 // родитель
 struct tree_node *parent;
 // меньшие
 struct tree_node *left;
 // большие
 struct tree_node *right;
};
```



# Узел дерева

```
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 узла в дерево

```
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;
}
```

# Поиск в дереве (1)

```
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);
}
```

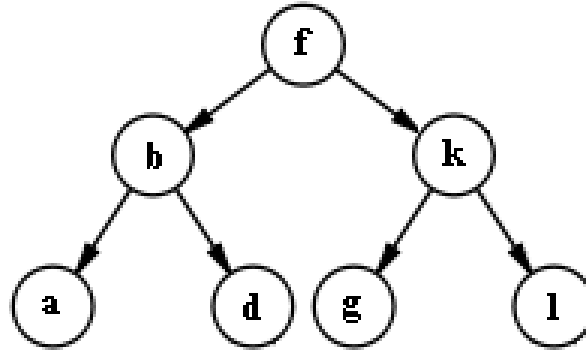
# Поиск в дереве (2)

```
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;
}
```

# Обход дерева



- Прямой (pre-order)
  - **f b a d k g l**
- Фланговый или поперечный (in-order)
  - **a b d f g k l**
- Обратный (post-order)
  - **a d b g l k f**

# Обход дерева

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

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

# DOT

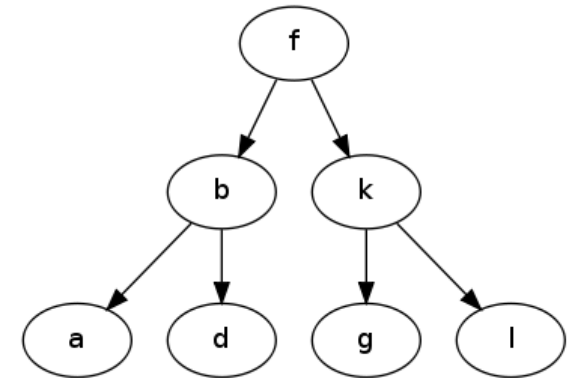
- DOT - язык описания графов.
- Граф, описанный на языке DOT, обычно представляет собой текстовый файл с расширением .gv в понятном для человека и обрабатывающей программы формате.
- В графическом виде графы, описанные на языке DOT, представляются с помощью специальных программ, например Graphviz.

В основном Wiki (с)

# DOT

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

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

[Edit this page](#)[Attach file](#)[Rename page](#)

Powered by Trac 0.12.2  
By Edgewall Software.



# DOT

```
void to_dot(struct tree_node *tree, void *param)
{
 FILE *f = param;

 if (tree->left)
 fprintf(f, "%s -> %s;\n", tree->name, tree->left->name);

 if (tree->right)
 fprintf(f, "%s -> %s;\n", tree->name, tree->right->name);
}

void export_to_dot(FILE *f, const char *tree_name,
 struct tree_node *tree)
{
 fprintf(f, "digraph %s {\n", tree_name);

 apply_pre(tree, to_dot, f);

 fprintf(f, "}\n");
}
```

# Абстрактные типы данных

# Модуль (1)

- Программу удобно рассматривать как набор независимых модулей.
- Модуль состоит из двух частей: интерфейса и реализации.
- *Интерфейс* описывает, что модуль делает. Он определяет идентификаторы, типы и подпрограммы, которые будут доступны коду, использующему этот модуль.
- *Реализация* описывает, как модуль выполняет то, что предлагает интерфейс.

# Модуль (2)

- У модуля есть один интерфейс, но реализаций, удовлетворяющих этому интерфейсу, может быть несколько.
- Часть кода, которая использует модуль, называют *клиентом*.
- Клиент должен зависеть только от интерфейса, но не от деталей его реализации.

# Преимущества использования модулей

- Абстракция (как средство борьбы со сложностью)  
Когда интерфейсы модулей согласованы, ответственность за реализацию каждого модуля делегируется определенному разработчику.
- Повторное использование  
Модуль может быть использован в другой программе.
- Сопровождение  
Можно заменить реализацию любого модуля, например, для улучшения производительности или переноса программы на другую платформу.

# Модули в языке Си (1)

- 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`.

# Модули в языке Си (2)

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

# Типы модулей (1)

- Набор данных

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

- Библиотека

Набор связанных функций.

- Абстрактный объект

Набор функций, который обрабатывает скрытые данные.



# Типы модулей (2)

- 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.

# Обход дерева без рекурсии

Обход дерева без рекурсии (используя стек)

```
void preorder (root)
{
 stack s
 push (&s, root)
 while (!empty (s))
 {
 node = pop (s)
 обработать node
 push (&s, node->right)
 push (&s, node->left)
 }
}
```

# Пример 1: абстрактный объект стек

1/stack.h

1/stack.c

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

# Пример 2: «абстрактный» тип данных «стек»

2/stack.h

2/stack.c

К сожалению `stack_t` не является абстрактным типом данных, потому что `2/stack.h` показывает все детали реализации.

# Неполный тип с языке Си (1)

- 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.

# Неполный тип с языке Си (2)

- Допустимо определять указатель на неполный тип  
`typedef struct t *T;`
- Можно
  - определять переменные типа T;
  - передавать эти переменные как аргументы в функцию.
- Нельзя
  - применять операцию обращения к полю (->);
  - разыменовывать переменные типа T.

# Пример 3: абстрактный тип данных «стек»

3/stack.h

3/stack.c

# Пример 4: абстрактный тип данных «стек» + еще реализация

4/stack.h

4/stack\_1.c (аналогична 3/stack.c)

4/stack\_2.c



# Трудности, улучшения и т.п. (1)

- Именование

В примерах использовались имена функций, которые подходят для многих АД (create, destroy, is\_empty). Если в программе будет использоваться несколько разных АД, это может привести к конфликту. Поэтому имеет смысл добавлять название АД в название функций (stack\_create, stack\_destroy, stack\_is\_empty).

# Трудности, улучшения и т.п. (2)

- Обработка ошибок
  - Интерфейс это своего рода контракт.
  - Интерфейс обычно описывает *проверяемые* ошибки времени выполнения и *непроверяемые* ошибки времени выполнения и исключения.
  - Реализация не гарантирует обнаружение непроверяемых ошибок времени выполнения. Хороший интерфейс избегает таких ошибок, но должен описать их.
  - Реализация гарантирует обнаружение проверяемых ошибок времени выполнения и информирование клиентского кода.

# Трудности, улучшения и т.п. (3)

- «Общий» АТД
  - Хотелось бы чтобы стек мог «принимать» данные любого типа без модификации файла `stack.h`.
  - Программа не может создать два стека с данными разного типа.

Решение — использовать `void*` как тип элемента, НО:

- элементами могут быть динамически выделяемые объекты, но не данные базовых типов `int`, `double`;
- стек может содержать указатели на что угодно, очень сложно гарантировать правильность.

# Функции с переменным числом параметров

# 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

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

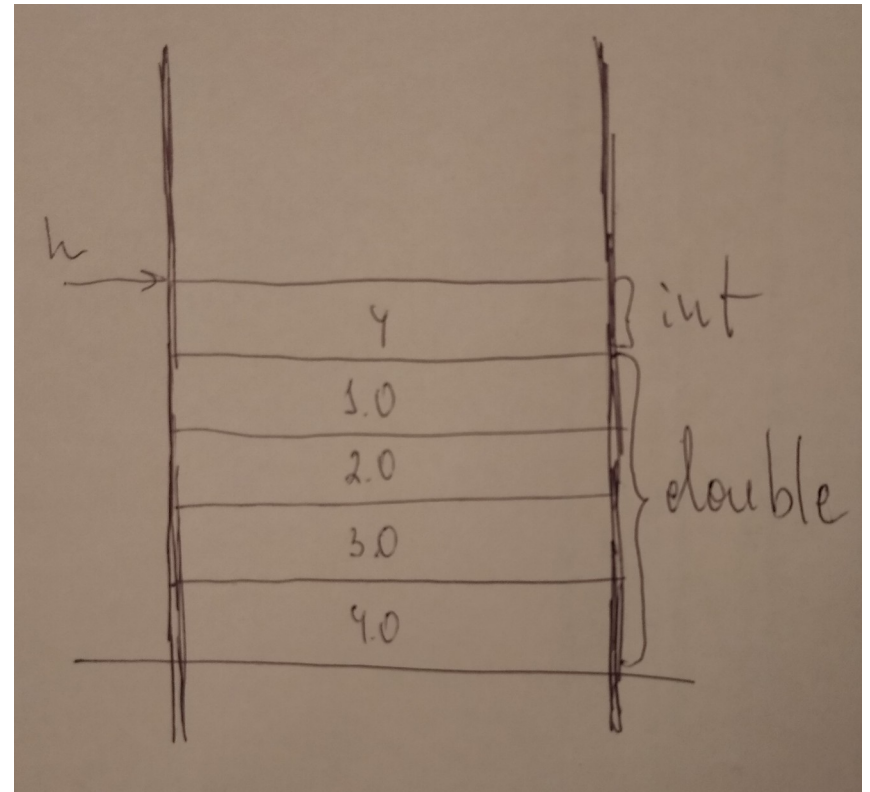
```
#include <stdio.h>

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

int 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(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;
}
```



# Функции с переменным числом параметров

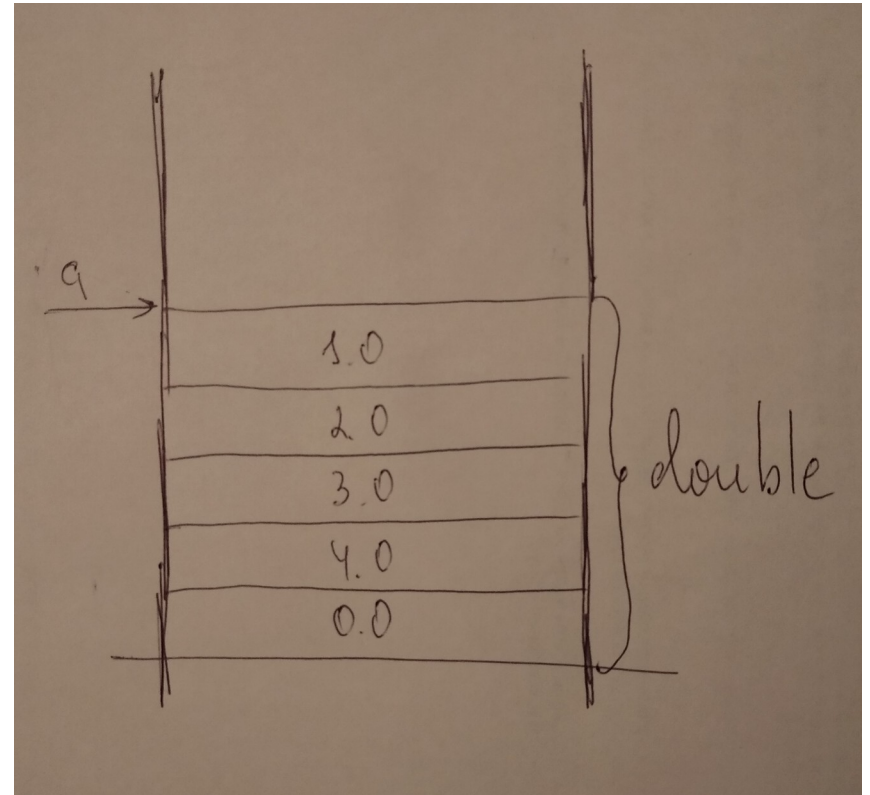
```
#include <stdio.h>

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

```
#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;
}
```

# Функции с переменным числом параметров

```
#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;
}
```

# Стандартный способ работы с параметрами функций с переменным числом параметров

stdarg.h

- va\_list
- va\_start(va\_list argptr, last\_param)
- type va\_arg(va\_list argptr, type)
- 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;
}
```

# 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;
 }
```

inline-функции

# 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
```

## 6.4.7 Function specifiers

6 ... An inline definition *does not provide an external definition* for the function, and *does not forbid an external definition* in another translation unit. An inline definition provides an alternative to an external definition, which *a translator may use* 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.

- inline-реализация не предоставляет и не запрещает реализацию со внешней линковкой.
- Транслятор волен сам выбирать.

# Способы исправления проблемы «unresolved reference»

- Использовать ключевое слово `static`

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

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

 return i;
}
```

Такая функция доступна только в текущей единице трансляции.

# Способы исправления проблемы «unresolved reference»

- Использовать ключевое слово `extern`

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

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

 return i;
}
```

Такая функция доступна из других единиц трансляции.

# Способы исправления проблемы «unresolved reference»

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

Самый **плохой** способ решения проблемы, потому что реализации могут не совпасть.

# Способы исправления проблемы «unresolved reference»

- Убрать ключевое слово `inline` из определения функции.

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

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

 return i;
}
```

Компилятор «умный» :), сам разберется.

# Директивы препроцессора

# Директивы препроцессора

- Макроопределения
  - #define, #undef
- Директива включения файлов
  - #include
- Директивы условной компиляции
  - #if, #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.

Продолжение на следующем слайде.

# 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.

# Макросы с параметрами

`#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

```
i = MAX(j + k, m - n);
// i = ((j + k) > (m - n) ? (j + k) : (m - n));

if (IS_EVEN(i))
// if (((i) % 2 == 0))
 i++;
```

# Макросы с переменным числом параметров (C99)

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

# Общие свойства макросов

- Список-замены макроса может содержать другие макросы.
- Препроцессор заменяет только целые лексемы, не их части.
- Определение макроса остается «известным» до конца файла, в котором этот макрос объявляется.
- Макрос не может быть объявлен дважды, если эти объявления не тождественны.
- Макрос может быть «разопределен» с помощью директивы `#undef`.

# Макросы с параметрами vs функции

## *Преимущества*

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

## *Недостатки*

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

```
n = MAX(i++, j);
```



# Скобки в макросах

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

```
#define TWO_PI 2 * 3.14
```

```
f = 360.0 / TWO_PI;
// f = 360.0 / 2 * 3.14;
```

```
#define SCALE(x) (x * 10)
```

```
j = SCALE(i + 1);
// j = (i + 1 * 10);
```

# Создание длинных макросов

```
// 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);
```

# Создание длинных макросов

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

# 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)

# 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
```

# Остальные директивы

`#error` сообщение

```
#if defined(OS_WIN)
...
#elif defined(OS_LIN)
...
#elif defined(OS_MAC)
...
#else
#error Unsupported OS!
#endif
```

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

# «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);
```



# «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.

# Библиотеки (часть 1)

# Библиотеки

Библиотека включает в себя

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

Библиотеки делятся на

- статические;
- динамические.

# Статические библиотеки

Связываются с программой в момент компоновки.  
Код библиотеки помещается в исполняемый файл.

«+»

- Исполняемый файл включает в себя все необходимое.
- Не возникает проблем с использованием не той версии библиотеки.

«-»

- «Размер».
- При обновлении библиотеки программу нужно пересобирать.

# Динамические библиотеки

Подпрограммы из библиотеки загружаются в приложение во время выполнения. Код библиотеки не помещается в исполняемый файл.

«+»

- Несколько программ могут «разделять» одну библиотеку.
- Меньший размер приложения (по сравнению с приложением со статической библиотекой).
- Модернизация библиотеки не требует перекомпиляции программы.

# Динамические библиотеки

окончание

«+»

- Могут использовать программы на разных языках.

«-»

- Требуется наличие библиотеки на компьютере.
- Версионность библиотек.

Способы использования динамических библиотек

- динамическая компоновка;
- динамическая загрузка.

# Linux: использование статической библиотеки

## Сборка библиотеки

- КОМПИЛЯЦИЯ

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

- уПАКОВКА

```
ar cr libarr.a arr_lib.o
```

- индексирование

```
ranlib libarr.a
```

## Сборка приложения

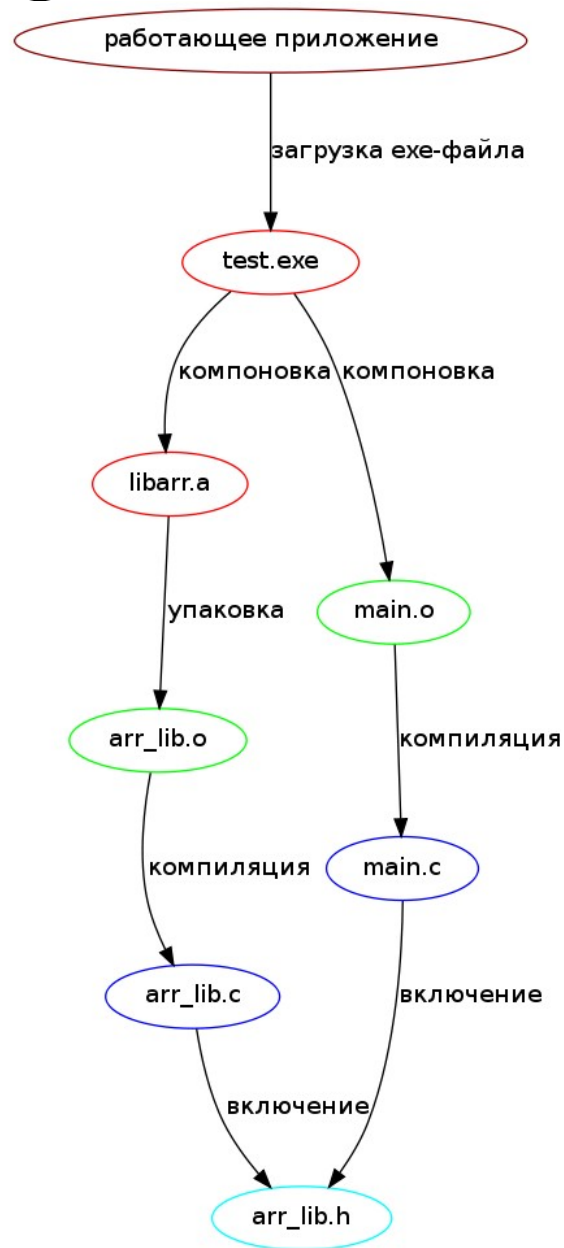
```
gcc -std=c99 -Wall -Werror main.c libarr.a -o app.exe
```

ИЛИ

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



# Граф зависимостей



# Linux: использование динамической библиотеки (динамическая компоновка)

## Сборка библиотеки

— КОМПИЛЯЦИЯ

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

— КОМПОНОВКА

```
gcc -o libarr.so -shared arr_lib.o
```

## Сборка приложения

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

```
gcc -o app.exe main.o -L. -larr
```

# Граф зависимостей



# Linux: использование динамической библиотеки (динамическая загрузка)

## Сборка библиотеки

— КОМПИЛЯЦИЯ

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

— КОМПОНОВКА

```
gcc -o libarr.so -shared arr_lib.o
```

## Сборка приложения

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

```
gcc -o app.exe main.o -ldl
```

# Linux API для работы с динамическими библиотеками

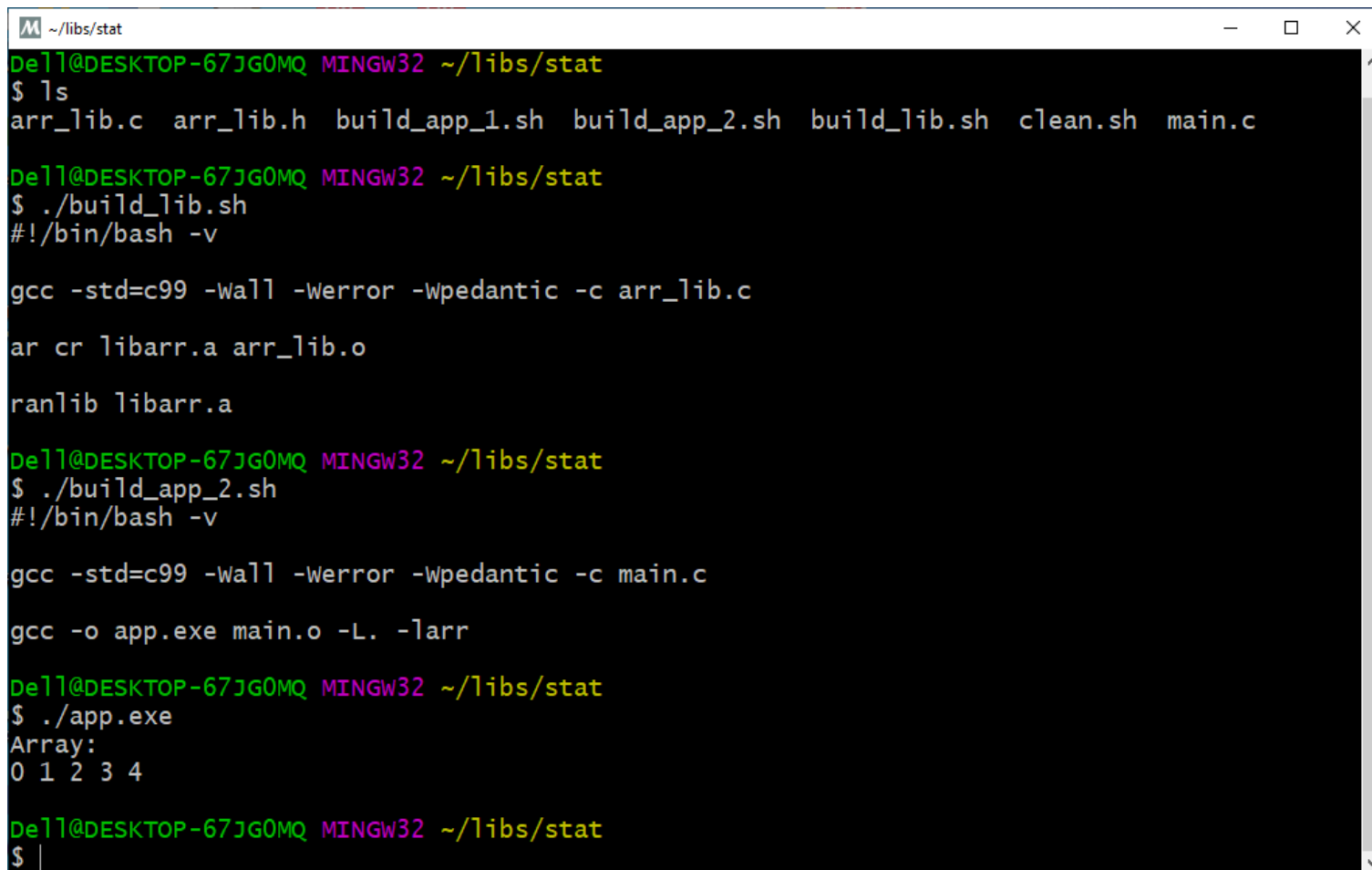
## `dlfcn.h`

- `void* dlopen(const char *file, int mode);`
- `void* dlsym(void *restrict handle,  
                    const char *restrict name);`
- `int dlclose(void *handle);`

# Граф зависимостей



# Windows: статическая библиотека



```
De11@DESKTOP-67JG0MQ MINGW32 ~/libs/stat
$ ls
arr_lib.c arr_lib.h build_app_1.sh build_app_2.sh build_lib.sh clean.sh main.c

De11@DESKTOP-67JG0MQ MINGW32 ~/libs/stat
$./build_lib.sh
#!/bin/bash -v

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

ar cr libarr.a arr_lib.o

ranlib libarr.a

De11@DESKTOP-67JG0MQ MINGW32 ~/libs/stat
$./build_app_2.sh
#!/bin/bash -v

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

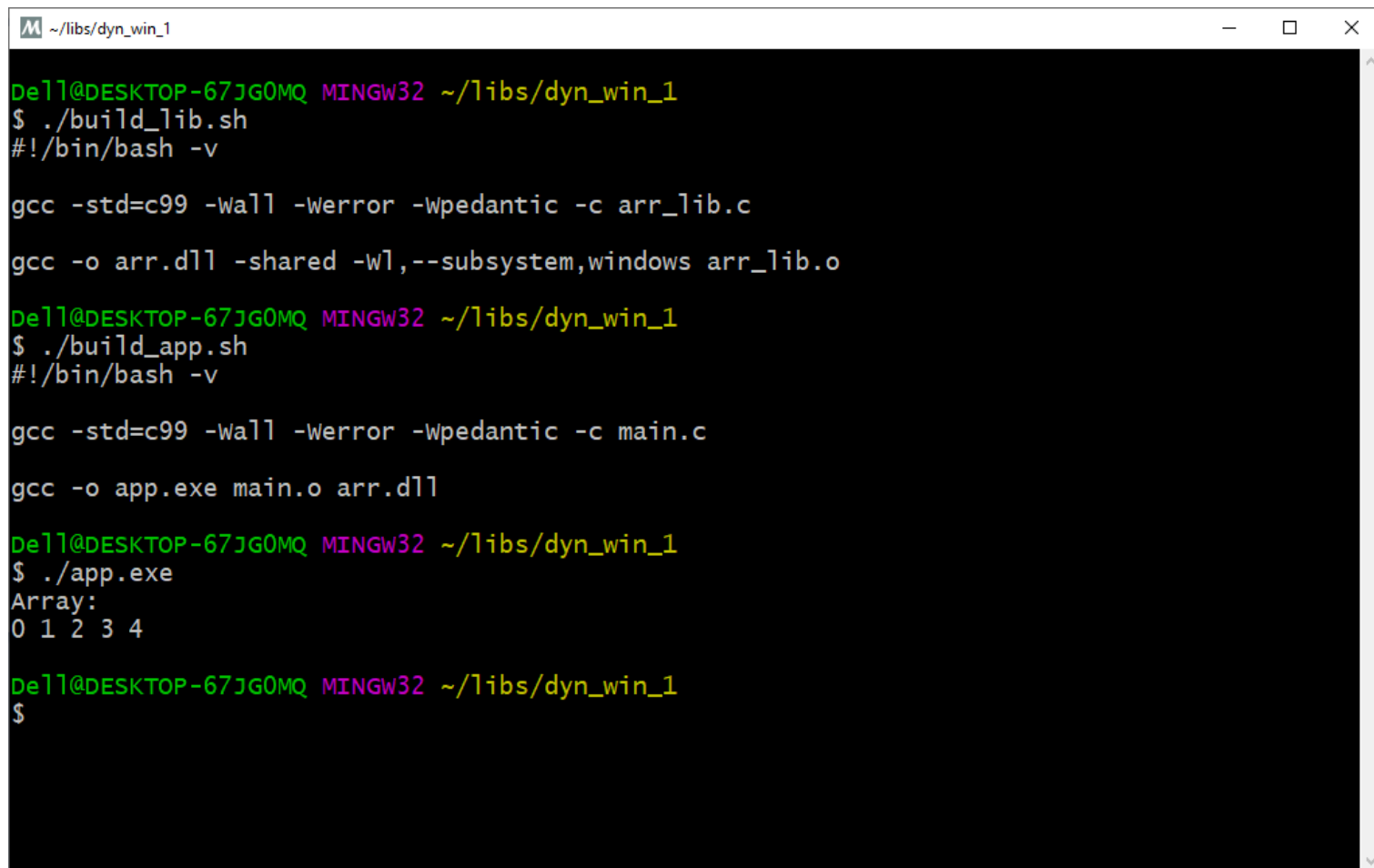
gcc -o app.exe main.o -L. -larr

De11@DESKTOP-67JG0MQ MINGW32 ~/libs/stat
$./app.exe
Array:
0 1 2 3 4

De11@DESKTOP-67JG0MQ MINGW32 ~/libs/stat
$ |
```

Никаких отличий от Linux.

# Windows: динамическая библиотека (компоновка)



```
De11@DESKTOP-67JG0MQ MINGW32 ~/libs/dyn_win_1
$./build_lib.sh
#!/bin/bash -v

gcc -std=c99 -Wall -Werror -Wpedantic -c arr_lib.c
gcc -o arr.dll -shared -Wl,--subsystem,windows arr_lib.o

De11@DESKTOP-67JG0MQ MINGW32 ~/libs/dyn_win_1
$./build_app.sh
#!/bin/bash -v

gcc -std=c99 -Wall -Werror -Wpedantic -c main.c
gcc -o app.exe main.o arr.dll

De11@DESKTOP-67JG0MQ MINGW32 ~/libs/dyn_win_1
$./app.exe
Array:
0 1 2 3 4

De11@DESKTOP-67JG0MQ MINGW32 ~/libs/dyn_win_1
$
```



# Windows API для работы с динамическими библиотеками

**windows.h**

- **HMODULE LoadLibrary(LPCSTR);**
- **FARPROC GetProcAddress(HMODULE, LPCSTR);**
- **FreeLibrary(HMODULE);**

# Windows: динамическая библиотека (загрузка)

```
De11@DESKTOP-67JG0MQ MINGW32 ~/libs/dyn_win_3
$./build_lib.sh
#!/bin/bash -v

gcc -std=c99 -Wall -Werror -Wpedantic -DARR_EXPORT -c arr_lib.c
gcc -o arr.dll -shared -Wl,--subsystem,windows arr_lib.o

De11@DESKTOP-67JG0MQ MINGW32 ~/libs/dyn_win_3
$./build_app.sh
#!/bin/bash -v

Приходится убирать Wpedantic
gcc -std=c99 -Wall -Werror -c main.c
gcc -o app.exe main.o

De11@DESKTOP-67JG0MQ MINGW32 ~/libs/dyn_win_3
$./app.exe
Library is loaded at address 738e0000.
arr_form function is located at address 738e150c.
arr_print function is located at address 738e153f.
Array:
0 1 2 3 4

De11@DESKTOP-67JG0MQ MINGW32 ~/libs/dyn_win_3
$ |
```

# Библиотеки (часть 2)

# Библиотека на Си, приложение на Python.

Предположим, что у нас есть динамическая библиотека со следующим набором функций:

- Простая (с точки зрения Python) функция `add`:

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

- Функция `divide` возвращает несколько значений, одно из которых возвращается с помощью указателя:

```
int divide(int a, int b, int *remainder);
```

# Библиотека на Си, приложение на Python.

ОКОНЧАНИЕ

- Функции `avg`, `fill_array`, `filter` обрабатывают массив:

```
double avg(double *arr, int n);
void fill_array(double *arr, int n);
int filter(double *src, int src_len,
double *dst, int *dst_len);
```

Нам необходимо вызвать эти функции из программы на Python без написания какого-либо дополнительного кода на Си или использования каких-либо утилит.

# ctypes (шаг 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.

# ctypes (шаг 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 и restype.

# ctypes: не Python поведение (1)

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)
```



# ctypes: не Python поведение (2)

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
```

# ctypes: массивы

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

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

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

# ctypes: итоги

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

# Модуль расширения (1)

Полное и исчерпывающее описание алгоритма написания модуля расширения может быть найдено в документации Python:

“Extending and Embedding the Python Interpreter” (<https://docs.python.org/3/extending/index.html>).

Сейчас будут рассмотрены только наиболее важные моменты.

# Модуль расширения (2)

Обычно функции модуля расширения имеют следующий вид

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

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

# Модуль расширения (4)

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

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

# Модуль расширения (5)

- 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);
```

# Модуль расширения (6)

- Ближе к концу модуля расширения располагаются таблица методов модуля PyMethodDef и структура PyModuleDef, которая описывает модуль в целом.
- В таблице PyMethodDef перечисляются
  - Си функции;
  - имена, используемые в Python;
  - флаги, используемые при вызове функции,
  - строки документации.
- Структура PyModuleDef используется для загрузки модуля.
- В самом конце модуля располагается функция инициализации модуля, которая практически всегда одинакова, за исключением своего имени.



# Модуль расширения: КОМПИЛЯЦИЯ

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.  
Внимательно прочитать:

- <https://github.com/valtron/llvm-stuff/wiki/Building-Python-3.4--extension-modules-with-MinGW>
- <https://stackoverflow.com/questions/34135280/valueerror-unknown-ms-compiler-version-1900>

Куча. Алгоритмы работы malloc/free

# 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.

# Особенности использования динамической памяти

Для хранения данных используется «куча».

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

“+”

Все «минусы» локальных переменных.

“-”

Ручное управление временем жизни.

# Свойства области, выделенной malloc

- 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.



# Реализация malloc/free

- 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)?

# Реализация malloc/free

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.





# Реализация malloc/free

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;
};
```

# Реализация malloc/free

```
#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;
}
```

# Реализация malloc/free

## 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.

# Реализация malloc/free

```
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;
```

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

 return result;
}
```

# Реализация malloc/free

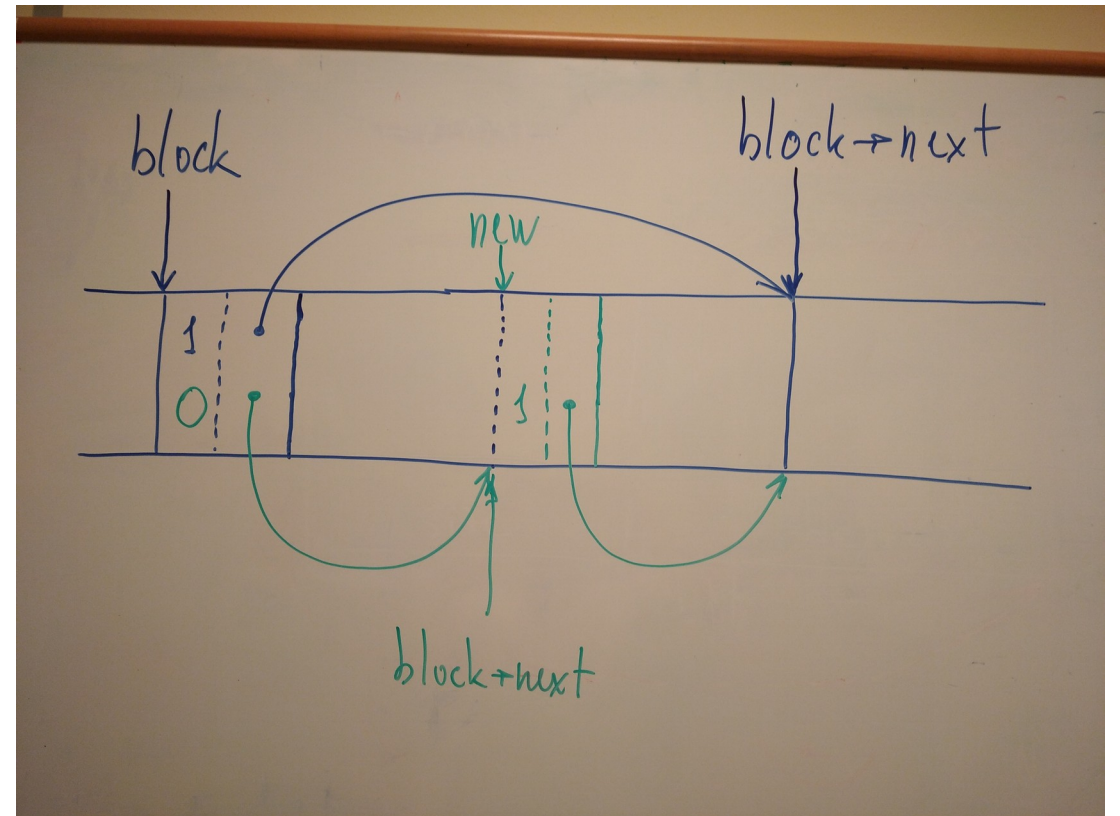
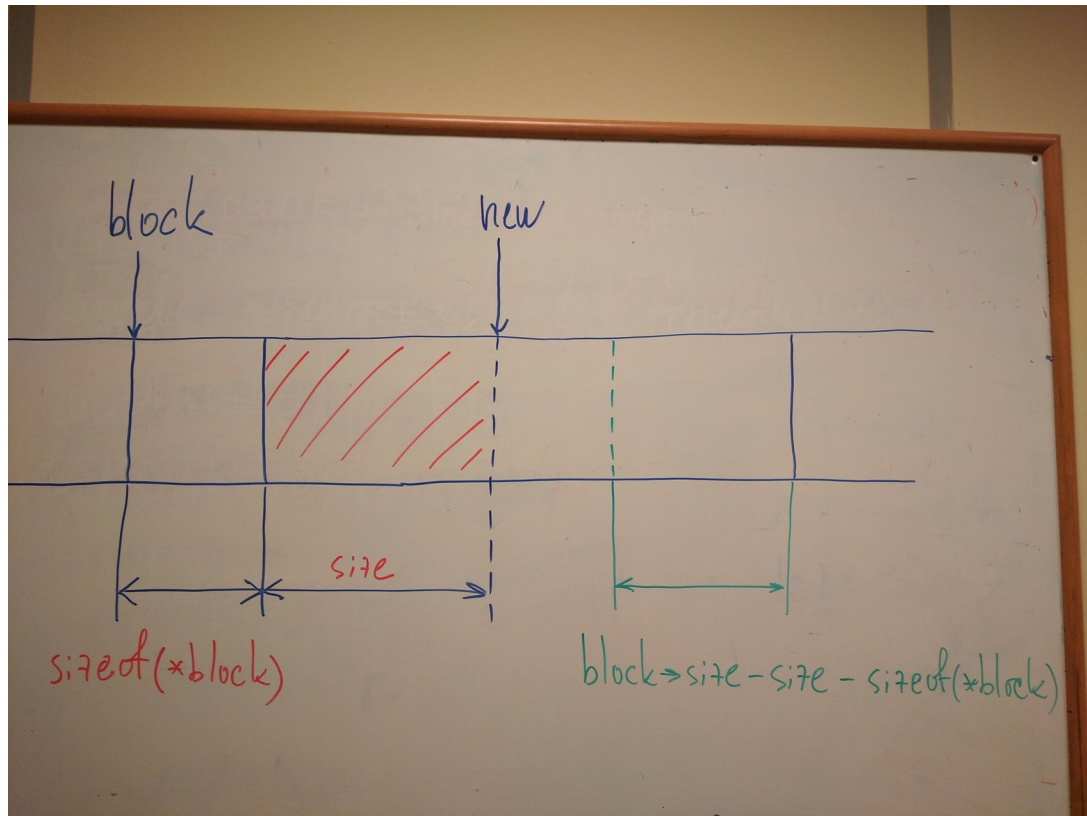
```
static void split_block(struct block_t *block, size_t size)
{
 size_t rest = block->size - size;

 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;
}
```

# Реализация malloc/free



# Реализация malloc/free

## 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.

# Реализация malloc/free

```
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");
}
```

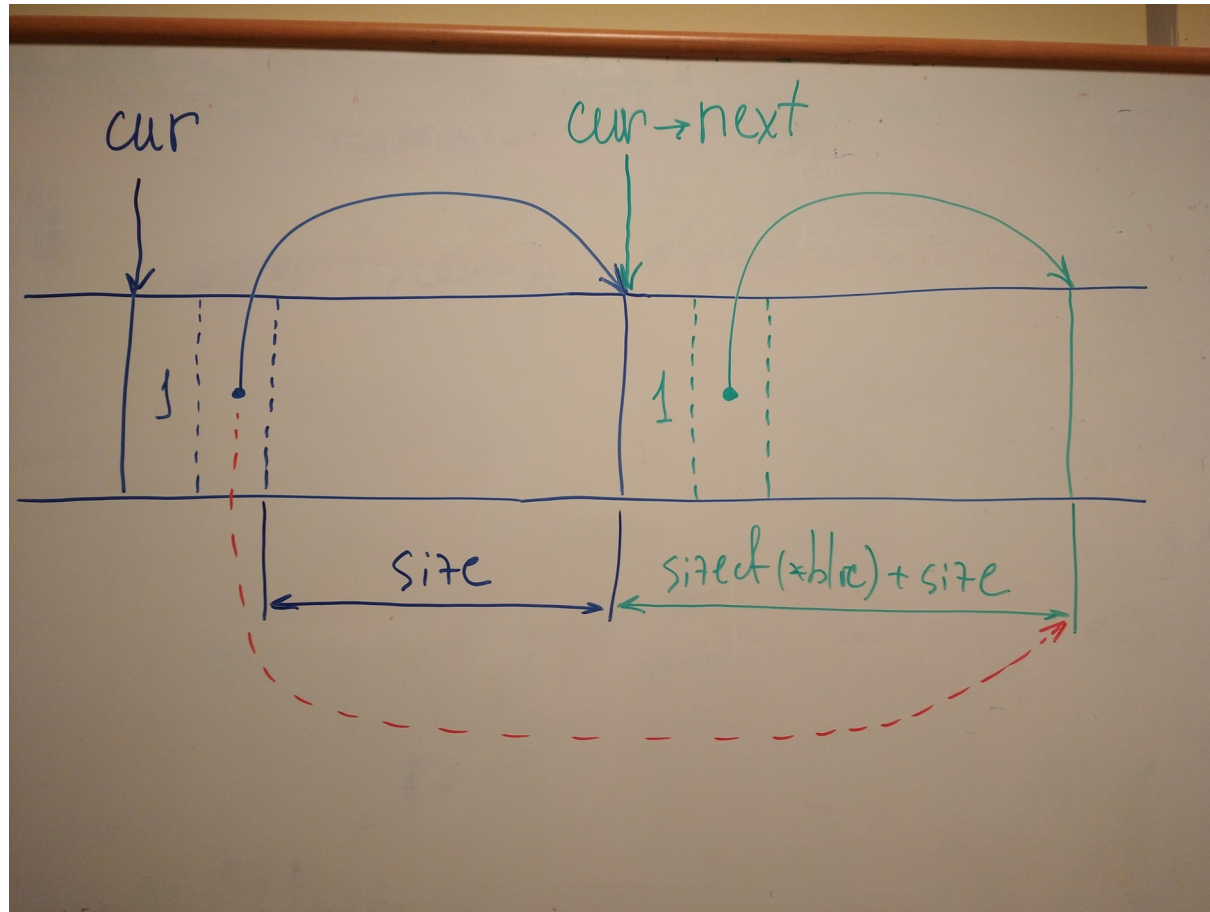


# Реализация malloc/free

```
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;
 }
}
```

# Реализация malloc/free



# Реализация malloc/free: недодделки и т.п.

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

# Реализация malloc/free: недоделки и т.п.

## *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.

# Реализация malloc/free: недодделки и т.п.

```
// 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));
```

# Реализация malloc/free: недодделки и т.п.

## Фрагментация



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

# Использованные материалы

- Б. Керниган, Д. Ритчи «Язык программирования С»
- М. Burelle «A Malloc Tutorial»
- Т. Madurapperuma «How to write your own Malloc and Free using C?»
- Лабораторная работа по курсу CS170 - Operating Systems

Списки в стиле университета  
Беркли  
(вариант реализации из ядра  
Linux)



# Списки Беркли: реализация

Текущая реализация:

<https://github.com/torvalds/linux/blob/master/include/linux/list.h>

Используемая реализация есть в примерах. Взята из какой-то версии ядра и немного «адаптирована».

Отличия между используемой реализацией и текущей предлагается найти самостоятельно.

# 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;
 ...
};
```

# 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`.

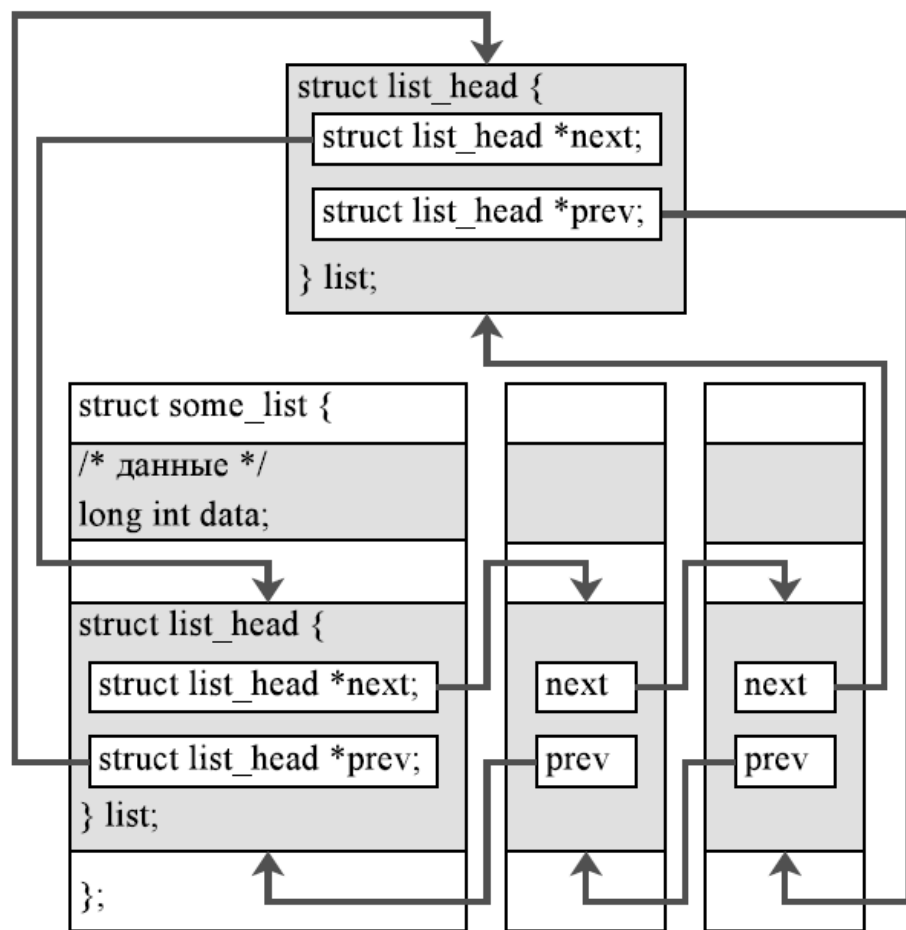
# Списки Беркли: создание «ГОЛОВЫ»

```
#define LIST_HEAD_INIT(name) { &(amp;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 в стиле Беркли



# 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);
}
```

# 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;
}
```



# 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

```
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));
```

# offsetof: идея

```
int offset = (int) (&((struct s*) 0)->i);
```

- `((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.
- `((struct s*) 0)->i`
  - Получаем поле `i` структуры `s`. Компилятор думает, что это поле расположено по адресу `0 + смещение i`.
- `&((struct s*) 0)->i`
  - Вычисляем адрес поля `i`, т.е. смещение `i` в структуре `s`.
- `(unsigned int) (&((struct s*) 0)->i)`
  - Преобразовываем адрес члена `i` к целому числу.

# Списки в стиле Беркли: анализ

«+» И «-»

- + Одно выделение памяти на узел списка.**
- Независимо от того в списке узел или нет присутствуют два дополнительных указателя.**