1. Указатель на void. Функции обработки областей памяти

Указатель на void

Указатель типа ***void*** (обобщенный указатель) 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

```
int compare_int_nums(const void *l, const void *r)
{
        const int *pl = l;
        const int *pr = r;

        return *pl - *pr;
}
```

В Си допускается присваивание указателя на void указателю любого типа (без явного преобразования) и наоборот.

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

void *p = pd;
pd = p;
```

- указатель типа ***void*** нельзя разыменовывать

```
int x = 10;
void *px = &x;

printf("x: %d\n", *px); // нельзя
```

- к нему не применима адресная арифметика

```
int x = 10;
int *px = &x;
void *p = px;

p++; // нельзя
```

Стандартные функции обработки областей памяти. memcpy, memmove, memset, memcmp.

- **memcpy** - копирует заданное количество байт *n* из одного участка памяти *src* в другой *dst*. Поведение не определено в случае пересечения блоков памяти внутри *src* и *dst*.

```
#include <string.h>

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

- **memmove** - аналогично *memcpy*, но корректно обрабатывает перекрывание областей памяти (в реализации сначала создается временная копия данных)

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

- **memcmp** - лексикографически сравнивает первые *n* байт двух областей памяти. Элементы интерпретируются как *unsigned char*.
- `> 0` - если первая строка больше второй
- `== 0` - если строки равны
- `< 0` - если вторая строка больше первой

```
int memcmp(const void *l, const void *r, size_t n);
```

- **memset** - Заполняет область памяти *dst* размером *n* байт указанным значением ch (преобразуется в *unsigned char*).

```
void *memset(void *dst, int ch, size_t n);
```

Примеры использования:

- Копируем строку (включая `\0`)

```
char src[] = "Hello!";
char dest[10];
memcpy(dest, src, strlen(src) + 1);
```

- Перемещаем "abcd" вправо на 2 позиции

```
char str[] = "abcdef";
memmove(str + 2, str, 4); // в результате str = "ababcd"
```

- сравнение двух массивов типа int

```
int arr1[] = {1, 2, 3, 4};
int arr2[] = {1, 2, 3, 5};
int result = memcmp(arr1, arr2, sizeof(arr1));
```

- инициализация буфера символами `*`

```
char buffer[10];
memset(buffer, '*', sizeof(buffer) - 1);
```

Использование указателей на void совместно с указателями на функции.

- указатель на **void** может быть преобразован в указатель на любой неполный или объектный тип и наоборот. Но функция - **не объект** в терминологии стандарта, поэтому 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. Например, это используется, в `dlsym`, чтобы получить адрес функции.
- указатель на функцию одного типа может быть преобразован в указатель на функцию другого типа и обратно. Если преобразованный указатель используется для вызова функции, тип которой несовместим с указанным типом, поведение не определено.

```
void my_function(int x)
{
        printf("Value: %d\n", x);
}

...

/* Первый пример: */
// Преобразование типов
void (*func_ptr) () = (void (*)()) my_function;

// Вызов функции с несовместимым типом (нет аргумента)
func_ptr(); // Поведение не определено

...

/* Второй пример: */
// Преобразование типов
void (*func_ptr) (int) = my_function;

// Совместимые типы
func_ptr(42); // Value: 42
```

2. Функции динамического выделения памяти

- Функции не создают переменные, они лишь выделяют память.
- Возвращают указатели на **void**, так как функция не знает с каким типом она будет работать.

malloc

- Выделяет блок памяти указанного размера в байтах ***size***
- Выделенный блок памяти не инициализируется ничем (содержит "мусор")

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

calloc

- Выделяет блок памяти для массива из ***nmemb*** элементов, каждый из которых имеет размер ***size*** байт
- Выделенный блок памяти инициализируется так, чтобы каждый бит имел значение *0*.

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

- В случае если запрашиваемую область памяти выделить не удалось, функции *calloc u malloc* вернут значение ***NULL***.
- После использования блока памяти он должен быть освобожден. Это можно сделать с помощью функции ***free***.

free

- Освобождает (делает возможным для повторного использования) ранее выделенный блок памяти, на который указывает ***ptr***.
- Если значение ***ptr == NULL***, ничего не происходит
- Если указатель ***ptr*** указывает на блок памяти, который ранее не был получен с помощью одной из функций ***malloc, calloc, realloc***, то поведение функции ***free*** не определено.

```
void free(void *ptr);
```

realloc

- Если **ptr == NULL** и **size != 0**, функция аналогична по поведению ***malloc***
- Если **ptr != NULL** и **size == 0**, функция аналогична по поведению ***free*** (**UB** - Impl. defined)

- Если **ptr != NULL** и **size != 0**, *realloc* перевыделяет ранее выделенный блок памяти, на который указывает ***ptr***, делая его размер равным ***size*** байт. В худшем случае:
  - выделить новую область памяти
  - скопировать данные из старой в новою область
  - освободить старую область

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

Типичная ошибка вызова realloc

- Если запрашиваемый блок памяти выделить не удалось, функция вернет значение **NULL**., но **ptr** не освободится. Поэтому необходимо вводить временную переменную, для предотвращения утечки памяти.

```
void *pbuf = realloc(pbuf, 2 * n); // неправильно

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

Явное приведение типа

```
int n = 5;
int *arr = (int *) malloc(n * sizeof(int));
```

+	-
компиляция с помощью c++ компилятора	начиная с ANSI C приведение не нужно
у функции *malloc* до стандарта ANSI C был другой прототип **char *malloc(size_t size);**	может скрыть ошибку, если забыли подключить **stdlib.h**
дополнительная "проверка" аргументов разработчиком	в случае изменения типа указателя придется менять тип в приведении

Вопрос про выделение 0 байт

Результат вызова функции *malloc, calloc* и *realloc*, когда запрашиваемый размер блока равен 0, зависит от реализации (*implementation-defined*).

- 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 0*.

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

Два способа выделения памяти под динамический массив:

- Как возвращаемое значение

```
int *create_array(FILE *f, size_t *n)
{
        if (f == NULL || n == NULL)
                return NULL;

        *n = 0;
        int tmp;
        while (fscanf(f, "%d", &tmp) == 1)
                (*n)++;

        rewind(f);

        int *arr = malloc((*n) * sizeof(int));
        if (!arr)
                return NULL;

        for (size_t i = 0; i < *n; i++)
                if (fscanf(f, "%d", &arr[i]) != 1)
                {
                        free(arr);
                        return NULL;
                }

        return arr;
}

...

int *arr;
size_t n;
arr = create_array(f, &n);
if (!arr)
{
        // обработка ошибочной ситуации
}
```

- Как параметр функции

```c
int *create_array(FILE *f, int **arr, size_t *n)
{
        if (f == NULL || arr == NULL || n == NULL)
                return ERR_PARAM;

        *n = 0;
        int tmp;
        while (fscanf(f, "%d", &tmp) == 1)
                (*n)++;

        rewind(f);

        *arr = malloc((*n) * sizeof(int));
        if (!(*arr))
                return ERR_MEM;

        for (size_t i = 0; i < *n; i++)
                if (fscanf(f, "%d", &(*arr)[i]) != 1)
                {
                        free(*arr);
                        *arr = NULL;
                        return ERR_IO;
                }

        return ERR_OK;
}

...

int *arr, rc;
size_t n;
rc = create_array(f, &arr, &n);
if (rc != ERR_OK)
{
        // обработка ошибочной ситуации
}
```

Типичные ошибки при работе с динамической памятью

- неверный расчет количества выделяемой памяти
- отсутствие проверки успешности выделения памяти
- утечки памяти
- логические ошибки
    - *wild pointer* - использование не проинициализированного указателя

```c
int *p;
*p = 10;
```

    - dangling pointer - использование указателя сразу после освобождения памяти

```
int *p = malloc(sizeof(int));
free(p);
*p = 10;
```

- изменение указателя, который вернула функция выделения памяти

```
int *p = malloc(10 * sizeof(int));
p++; // Потеря адреса начала выделенной памяти
free(p); // Ошибка
```

- двойное освобождение памяти

```
int *p = malloc(10 * sizeof(int));
free(p);
free(p); // Ошибка
```

- освобождение невыделенной или нединамической памяти

```
int a = 10;
free(&a); // Ошибка: попытка освободить статическую память
```

- выход за границы динамического массива

```
int* array = malloc(5 * sizeof(int));
if (array != NULL)
{
        array[5] = 10; // Ошибка: индекс выходит за пределы массива
        free(array);
}
```

- прочее (например, освобождение памяти неправильным способом)

```
int* array = malloc(10 * sizeof(int));
free(array + 1); // Ошибка: передан некорректный указатель
```

Подходы к обработке ситуации отсутствия динамической памяти

- **возвращение ошибки (return failure)**
  этот подход предполагает, что функции, работающие с памятью, проверяют результат вызова, например, *malloc* и возвращают специальный код ошибки, если выделение памяти не удалось.

```
int n = 5;
int *p = malloc(n * sizeof(int));
if (p == NULL)
        return ERR_MEM;

...
```

- **ошибка сегментации (segfault)**
  выделение памяти явно не обрабатывается, и программа падает при попытке использовать `NULL`-указатель. Могут возникнуть проблемы с безопасностью.

```
int n = 5;
int *p = malloc(n * sizeof(int)); // Не проверяем p
*p = 42; // Если malloc вернул NULL, будет segfault
```

- **аварийное завершение (exit / xmalloc)**
  если память не может быть выделена, программа немедленно завершает свою работу. *xmalloc* - это функция-обертка над malloc, описанная Керниганом и Ритчи

```
void *xmalloc(size_t size)
{
        void *ptr = malloc(size);
        if (ptr == NULL)
        {
                fprintf(stderr, "Ошибка выделения памяти\n");
                exit(EXIT_FAILURE);
        }
        return ptr;
}
```

- **восстановление (recovery)**
  попытка освободить часть ранее выделенной памяти, использовать резервные механизмы или уменьшить потребности программы. Данный подход использовался в исходном коде *git*.

```
void *recoverable_malloc(size_t size)
{
        void *ptr = malloc(size);
        if (ptr == NULL)
        {
                // Попытка освободить память из резервного пула
                release_some_memory();
                ptr = malloc(size);
        }
```

```
        return ptr;
}
```

4. Указатели на функцию

Общее

- Указатели на функцию используются для того, чтобы передавать функции как аргументы и возвращать их из других функций.

Объявление указателя на функцию

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

Получение адреса функции

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

Вызов функции по указателю

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

qsort

Заголовочный файл - *<stdlib.h>*. **qsort** - это стандартная функция, которая позволяет упорядочить массив на основе отношения порядка, которую задает функция *comparator*.

```
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;
}
...
int a[10];
...
qsort(a, sizeof(a) / sizeof(a[0]), sizeof(a[0]), compare_int);
```

Особенности использования указателей на функции

Выражение из имени функции неявно преобразуется в указатель на функцию. Операция `&` для функции возвращает указатель на функцию, но это лишняя операция.

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

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

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

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

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

Указатели на функцию можно сравнивать (адресная арифметика). Только это.

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

Указатель на функцию может быть типом возвращаемого значения функции

```
int (*get_action(char ch)) (int, int);

typedef int (*ptr_action_t) (int, int);

ptr_action_t get_action(char ch);
```

Использование указателей на функции

С помощью указателей на функции в языке Си реализуются:

- **функции обратного вызова (callback)** используются для передачи логики в качестве параметра

```
void print_num(int n)
{
        printf("Num: %d\n", n);
}

void execute_callback(void (*callback) (int)), int n)
{
        callback(n);
}
```

```
...
execute_callback(print_num, n);
```

- **таблицы переходов (jump table)** удобны для организации выбора функций по индексу

```
void add(int a, int b)
{
        printf("%d\n", a + b);
}

void subtract(int a, int b)
{
        printf("%d\n", a - b);
}

void multiply(int a, int b)
{
        printf("%d\n", a * b);
}

...

size_t choice = 1;
void (*operations[3]) (int, int) = {add, subtract, multiply};
operations[choice](10, 5); // вызов функции subtract [1]
```

- **динамическое связывание (binding)** позволяет выбирать реализацию функций в зависимости от входных данных

```
void linux_run()
{
        printf("Running on Linux\n");
}

void windows_run()
{
        printf("Running on Windows\n");
}

void mac_run()
{
        printf("Running on MacOS\n");
}

void execute(const char *os)
{
        void (*run)();
        if (strcmp(os, "Linux") == 0)
                run = linux_run
        else if (strcmp(os, "Windows") == 0)
```

```
                run = windows_run
        else if (strcmp(os, "MacOS") == 0)
                run = mac_run
        else
                return;

        run();
}

...
execute("Windows");
```

Использование указателей на void совместно с указателями на функции.

- указатель на **void** может быть преобразован в указатель на любой неполный или объектный тип и наоборот. Но функция - **не объект** в терминологии стандарта, поэтому 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. Например, это используется, в `dlsym`, чтобы получить адрес функции.
- указатель на функцию одного типа может быть преобразован в указатель на функцию другого типа и обратно. Если преобразованный указатель используется для вызова функции, тип которой несовместим с указанным типом, поведение не определено.

```
void my_function(int x)
{
        printf("Value: %d\n", x);
}

...

/* Первый пример: */
// Преобразование типов
void (*func_ptr) () = (void (*)()) my_function;

// Вызов функции с несовместимым типом (нет аргумента)
func_ptr(); // Поведение не определено

...

/* Второй пример: */
// Преобразование типов
void (*func_ptr) (int) = my_function;

// Совместимые типы
func_ptr(42); // Value: 42
```

5-7. Утилита make

- Утилита make, назначение. Простой сценарий сборки
- Утилита make, назначение, переменные, шаблонные правила
- Утилита make, назначение, условные конструкции, анализ зависимостей

Общее

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

Принцип работы

Необходимо создать так называемый сценарий сборки проекта *Makefile*. Этот файл описывает:
- отношения между файлами программы;
- содержит команды для обновления каждого файла.

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

Разновидности

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

Сценарий сборки

Содержит переменные и правила, из каких составных частей, назначения у правил.

```
зависимость_1 ... зависимость_n
        команда_1
        команда_2
        ...
        команда_m
```

`что создать/сделать: из чего создать` как создать/что сделать`

Особенности выполнения команд

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

Простой сценарий сборки (5)

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

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

bye.o: bye.c bye.h
        gcc -std=c99 -Wall -Werror -Wpedantic -c bye.c

main.o: main.c hello.h bye.h
        gcc -std=c99 -Wall -Werror -Wpedantic -c main.c

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

Как *make* будет его обрабатывать:

1. **если проект не собирался ранее**
   make читает сценарий сборки и начинает выполнять первое правило *greeting.exe*. Для его выполнения необходимо сначала обработать зависимости, поэтому make ищет правила для создания *hello.o*, *bye.o*, *main.o*.

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

   Аналогично обрабатываются остальные зависимости.

   Эти правила могут быть выполнены. Все зависимости получены, теперь правило для построения greeting.exe может быть выполнено.
2. **проект собирался ранее, но после этого был изменен файл hello.o***
   make читает сценарий сборки и начинает выполнять первое правило *greeting.exe*. Для его выполнения необходимо сначала обработать зависимости, поэтому make ищет правила для создания *hello.o*, *bye.o*, *main.o*.

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

   Аналогично обрабатываются зависимости bye.o и main.o, но эти файлы были изменены позже соответствующих си-файлов, т.е. ничего делать не нужно.

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

Ключи запуска утилиты

- 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
```

Переменные, шаблонные правила (6)

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 `$`. Строки, которые начинаются с символа `#`, являются комментариями.

```
# Компилятор
CC := gcc

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

...
main.o: main.c list.h
        $(CC) $(CFLAGS) -I inc -c main.c -o main.o
```

Неявные правила

В примере *main.o* создастся неявно без правила

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

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

Фиктивные цели

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

```
.PHONY: clean
```

Автоматические переменные

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 **$<** означает *первую зависимость*.

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

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

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

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

Шаблонные правила

`%` задает зависимость от подстановки в цели.

```
%.расш_файлов_целей: %.расш_файлов_зависимостей
	команда_1
	команда_2

	...
	команда_m
```

в правилах задает зависимость от всех файлов по маске

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

Условные конструкции, анализ зависимостей (7)

Условные конструкции

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

Переменные, зависящие от цели

```
debug: CFLAGS += -g3
debug: app.exe

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

Автоматическая генерация зависимостей

Ручная установка зависимостей

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

С помощью компилятора

```
# c-файлы
SRCS := ...

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

include $(SRCS:.c=.d)
```

Функции в make

Вызов функции

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

Функция **patsubs**

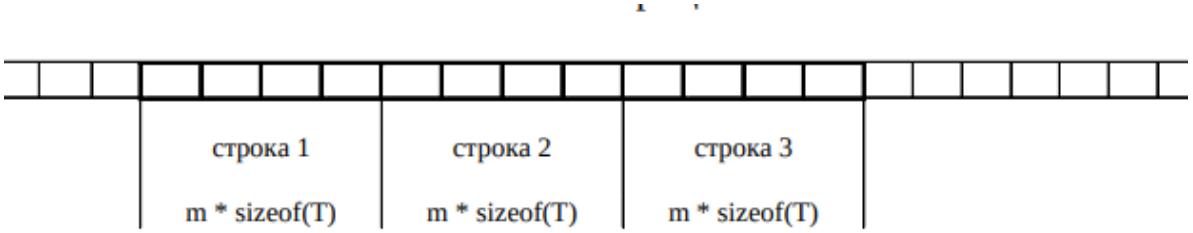
```
$(patsubst pattern, replacement, text)
cfiles := main.c hello.c bye.c

# полная форма
```

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

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

8-13. Динамические матрицы

- Динамические матрицы. Представление в виде одномерного массива и в виде массива указателей на строки. Анализ преимуществ и недостатков.
- Динамические матрицы. Представление в виде одномерного массива и в виде массива указателей на строки матрицы (1 подход). Анализ преимуществ и недостатков.
- Динамические матрицы. Представление в виде одномерного массива и в виде массива указателей на строки матрицы (2 подход). Анализ преимуществ и недостатков.
- Динамические матрицы. Представление в виде массива указателей на строки матрицы и в виде массива указателей на строки матрицы (1 подход). Анализ преимуществ и недостатков.
- Динамические матрицы. Представление в виде массива указателей на строки матрицы и в виде массива указателей на строки матрицы (2 подход). Анализ преимуществ и недостатков.

Матрица как одномерный массив



Выделение памяти

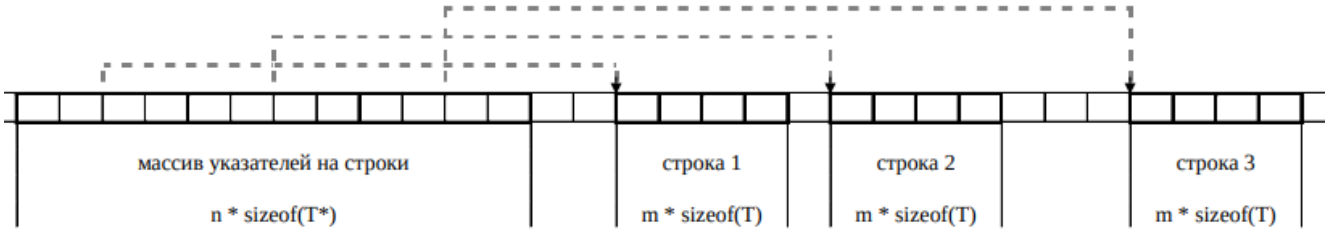
```
double *data = malloc(n * m * sizeof(double));
```

Очищение памяти

```
free(data);
```

+	-
простота выделения и освобождения памяти	отладчик использования памяти не может отследить выход за пределы строки
возможность использовать как одномерный массив	нужно писать `i * m + j`

Матрица как массив указателей на строки



Выделение памяти

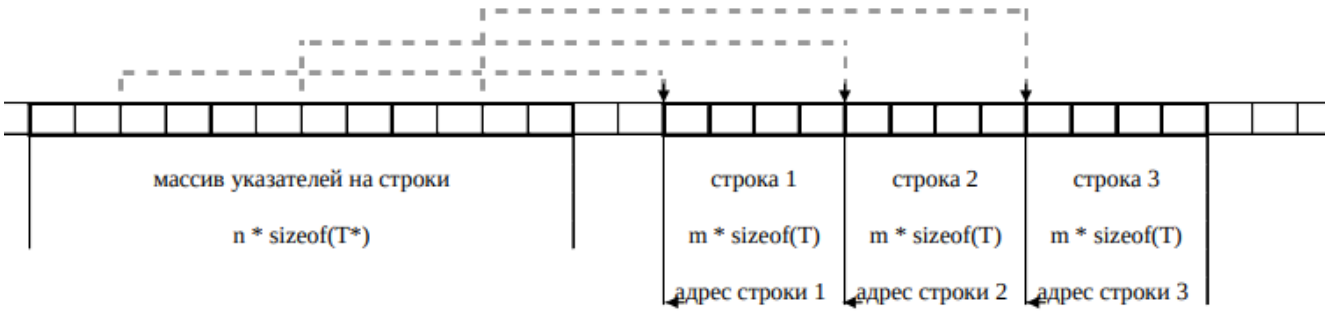
```
double **allocate_matrix(size_t n, size_t m)
{
        double **data = calloc(n, sizeof(double *));
        if (!data)
                return NULL;
        for (size_t i = 0; i < n; i++)
        {
                data[i] = malloc(m * sizeof(double));
                if (!data[i])
               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

Очищение памяти

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

        free(data);
}
```

+	-
возможность обмена строки через обмен указателей	сложность выделения и освобождения памяти
отладчик использования памяти может отследить выход за пределы строки	память под матрицу "не лежит" одной областью

Объединенный подход (1)



Выделение памяти

```
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;
}
```

Очищение памяти

```
void free_matrix(double **data, size_t n)
{
        free(data[0]); // скрывается потенциальная ошибка
        free(data);
}

void free_matrix(double **data, size_t n)
{
        if (data)
        {
                double *prow = data[0];
                for (size_t i = 1; i < n; i++)
                        if (data[i] < prow)
                                prow = data[i];
                free(prow);
                free(data);
```

```
        }
}
```

+	-
относительная простота выделения и освобождения памяти	отладчик использования памяти не может отследить выход за пределы строки
возможность использовать как одномерный массив	относительная сложность начальной инициализации
перестановка строк через обмен указателей	

Объединенный подход (2)

Выделение памяти

```
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;
}
```

Очищение памяти

```
free(data);
```

+	-
перестановка строк через обмен указателей	отладчик использования памяти не может отследить выход за пределы строки
возможность использовать как одномерный массив	сложность начальной инициализации
простота выделения и освобождения памяти	

Сравнительная таблица

-	одномерный массив	массив указателей	объединение (1)	объединение (2)
возможность использовать как одномерный массив	есть	нет	есть	есть
выделение и освобождение памяти	просто	сложно	относительно просто	просто
начальная инициализация	-	-	относительно сложно	сложно
отладчик использования памяти может отследить выход за пределы строк	нет	да	нет	нет
перестановка строк через обмен указателей	-	да	да [?]	да

Передача динамической матрицы в функцию. Примеры

```
void print_mtx(double **mtx);

...
double **mtx = alloc_mtx(N, M);
print_mtx(mtx);
```

Функция, которая может обрабатывать как статические, так и динамические матрицы.

```c
void print_matrix(double *mtx, size_t n, size_t m)
{
        for (size_t i = 0; i < n; i++)
                for (size_t j = 0; j < m; j++)
                        printf("%lf ", mtx[j + i * m]);
}

...
// статическая матрица
print_matrix((double *) a, N, M);

// динамическая матрица
print_matrix((double *) (b + N), N, M);
```

14. Чтение сложных объявлений

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

Читать нужно изнутри наружу (т.е. с имени сущности). Или же: двигаемся вправо пока можем, иначе влево. Отправная точка - идентификатор.
Нужно отдавать предпочтение `[]` и `()`, а не `*`

```
*name[] – массив типа, а не указатель на ...
*name() – функция, принимающая, а не указатель на ...
```

Примеры

```
long **foo[7] – массив из 7 элементов типа указатель на указатель на long

int *(*x[10])(int, int) – массив из 10 элементов типа указатель на функцию,
которая принимает два аргумента типа int и возвращает указатель на int.

char *(*(**foo[][8])())[] – массив типа массив из 8 элементов типа указатель на
указатель на функцию, которая ничего не принимает и возвращает указатель на
массив типа указатель на char
```

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)[];
```

- для массива только левая лексема `[]` может быть пустой

```
int a[][];
```

- тип *void* ограниченный

```
void x;
void x[5];
```

Использование typedef

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

typedef int *func_t(void);
typedef func_t *func_ptr;
typedef func_ptr func_ptr_arr[10];

func_ptr_arr x;
```

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

strdup, strndup

- Функция дублирует строку, на которую указывает аргумент *str*, с выделением динамической памяти под новую строку. При использовании функции *strndup* на длину дублируемой строки вводится ограничение в *n* байт.
- Память под дубликат строки выделяется с помощью функции *malloc*, и по окончанию работы с дубликатом должна быть очищена с помощью функции *free*.
- Заголовочный файл - `string.h`

```
char *strdup(const char *str); // POSIX

char *strndup(const char *str, size_t n); // GNU
```

Использование

```
#define _GNU_SOURCE

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

#define STRING "BMSTU is the best university."

int main(void)
{
        char *str = strdup(STRING);
        if (str)
        {
                printf("%s\n", str);
                free(str);
        }
        else
                printf("MEMORY ERROR!\n");

        return 0;
}
```

Моя реализация

```
char *my_strdup(const char *str)
{
        if (str == NULL)
```

```
                return NULL;

        size_t len = strlen(str);

        char *dst = malloc(len + 1); // sizeof(char) == 1
        if (dst == NULL)
                return NULL;
        memcpy(dst, str, len); // strcpy(dst, str)
        dst[len] = '\0';
        return dst;
}

char *my_strndup(const char *str, size_t n)
{
        if (str == NULL)
                return NULL;

        size_t len = strlen(str);
        if (len > n)
                len = n;

        char *dst = malloc(len + 1);
        if (dst == NULL)
                return NULL;

        memcpy(dst, str, len); // strncpy(dst, str, len)
        dst[len] = '\0';
        return dst;
}
```

getline

- Функция считывает целую строку из *stream*, сохраняет адрес буфера с текстом в **lineptr*.
- *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.
- При успешном выполнении *geltine()* возвращает количество считанных символов, включая символ разделителя, но не включая завершающий байт *'\0'*.
- При ошибке чтения строки функция возвращает *-1*.
- Заголовочный файл - `stdio.h`

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

Использование

```
#define _GNU_SOURCE

#include <stdio.h>
```

```c
#include <stdlib.h>

int main(void)
{
    FILE *stream = fopen("1.txt", "r");
    if (!stream)
        return ERR_IO;

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

    while ((read = getline(&line, &len, stream)) != -1)
    {
        printf("Размер буфера: %d, считано символов: %d\n", (int) len, (int)
read);
        printf("Полученная строка: %s\n", line);
    }

    free(line);
    fclose(stream);
    return 0;
}
```

Моя реализация

```c
ssize_t my_getline(char **lineptr, size_t *n, FILE *stream)
{
    if (!lineptr || !n || !stream)
        return -1;

    size_t size = *n;
    ssize_t read = 0;
    int ch;

    while ((ch = getc(stream)) != EOF)
    {
        if (read >= (ssize_t) size - 1)
        {
            if (!size)
                size = 120; // init size
           else
                size *= 2;
            char *tmp = realloc(*lineptr, size);
            if (!tmp)
                return -1;
            *lineptr = tmp;
        }

        (*lineptr)[read++] = (char) ch;
        if (ch == '\n')
```

```
            break;
    }

    if (read)
    {
        (*lineptr)[read] = '\0';
        *n = size;
        return read;
    }
    return -1;
}
```

sprintf, snprintf, asprintf

- Функция *sprintf* аналогична функции *printf*, за исключением того, что вывод производится в массив, указанный аргументом *buf*. Возвращается количество символов, занесённых в массив. Пользователь должен самостоятельно выделить достаточное количество памяти.
- Функция *snprintf* аналогична *sprintf*, но появляется ограничение в *n* байт на размер строки.
- Функция *asprintf* аналогична *sprintf*, но память для строки выделяется автоматически.
- Заголовочный файл - `stdio.h`

```
int sprintf(char *buf, const char *format, ...); // ISO C90

int snprintf(char *buf, size_t n, const char *format, ...); // C99

int asprintf(char **buf, const char *format, ...); // GNU
```

Использование

```
#define _GNU_SOURCE

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

#define NAME "Valera"
#define CITY "Moscow"

void use_snprintf(void)
{
        int n, m;

    // получить размер буфера
    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)
```

```
            {
                    // заполнить буфер
                    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);
            }
    }
}

void use_asprintf(void)
{
        char *line = NULL;
    int 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);
    }
}
```

Моя реализация

```
int my_asprintf(char **strptr, const char *fmt, ...)
{
    va_list vl;
    va_start(vl, fmt);

    int size = vsnprintf(NULL, 0, fmt, vl);
    if (size < 0)
    {
        va_end(vl);
        return -1;
    }

    *strptr = malloc(size + 1); // sizeof(char) == 1
    if (!*strptr)
    {
        va_end(vl);
        return -1;
    }

    va_end(vl);

    va_start(vl, fmt);

    vsprintf(*strptr, fmt, vl);
```

```
    va_end(vl);
    return size;
}
```

Feature test macros

Feature Test Macros (FTM) используются для включения или выключения определенных функциональных возможностей или расширений в стандартной библиотеке.

```
// >= glibc 2.10
# define _POSIX_C_SOURCE 200809L

// < glibc 2.10
# define _GNU_SOURCE
```

16. Использование структур с полями-указателями

```
struct book_t
{
        char *title;
        int year;
};
```

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.

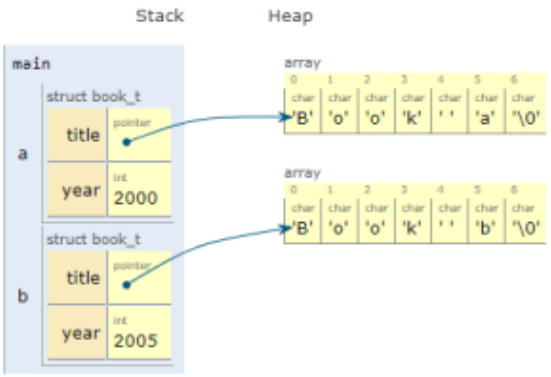
Поверхностное копирование

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

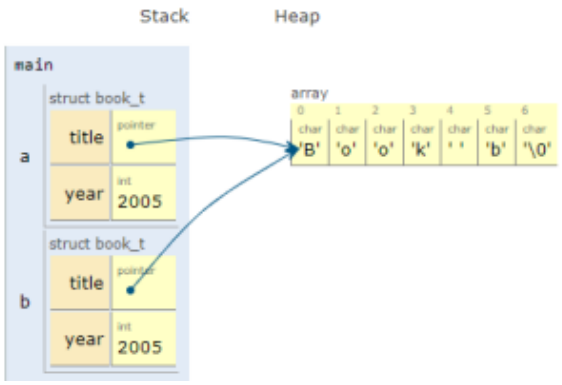
Иногда стратегия «поверхностного копирования» может приводить к ошибкам.

```
struct book_t a = { 0 }, b = { 0 };

a.title = strdup("Book a");
a.year = 2000;

b.title = strdup("Book b");
b.year = 2005;

a = b;

free(a.title);
free(b.title); // двойное освобождение
```

До присваивания:



После присваивания:

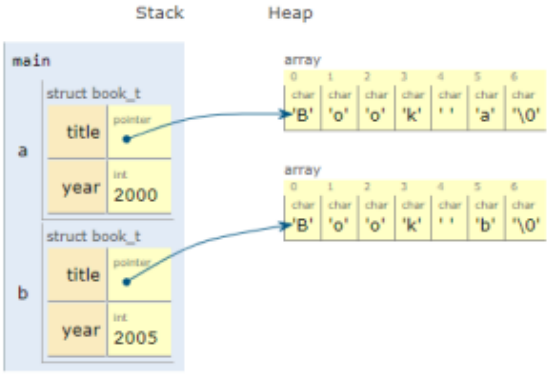


Глубокое копирование

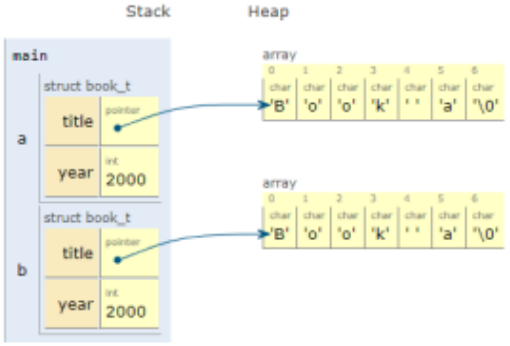
Стратегия так называемого «глубокого копирования» (англ., deep coping) подразумевает создание копий объектов, на которые ссылаются поля структуры.

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

                return 0;
        }

        return 1;
}

...

book_copy(&b, &a); // вместо b = a

free(a.title);
free(b.title);
```

До копирования:



После копирования:



Рекурсивное освобождение памяти

Для выделенной динамически структуры с полем-указателем, при освобождении памяти — сначала надо освободить память из-под внутренних полей, потом из-под самой структуры.

```
free(book.title);
free(book);
```

17. Структуры переменного размера

Структуры переменного размера (примеры)

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.

TLV кодирование *используется* в:

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

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

- подобное поле должно быть последним
- нельзя создавать 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 - UB

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

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

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

        return elem;
}
```

Flexible array member (до C99)

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

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

        // чтобы успокоить valgrind
        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 поле с обычным указателем

- экономия памяти
- локальность данных

- атомарность выделения памяти
- не требует "глубокого копирования" и освобождения

18. Динамически расширяемый массив

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

Динамический массив — это массив, размер которого задаётся в момент выполнения программы, а не на этапе компиляции.

Динамически расширяемый массив - структура данных, представляющая собой увеличивающийся блок памяти, который хранит элементы одного типа.

Отличие от динамического массива.

Динамически расширяемый массив может автоматически изменять размер выделенной памяти при необходимости, а в динамическом массиве это нужно делать вручную.

Особенности реализации

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

Почему память надо выделять крупными блоками

- Для уменьшение 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.

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

#define DA_INIT_SIZE 1
#define DA_STEP      2

#define DA_OK        0
#define DA_ERR_MEM   -1
#define DA_ERR_RANGE -2

void da_init(struct dyn_array_t *parr)
```

```
{
        parr->data = NULL;
        parr->len = 0;
        parr->allocated = 0;
}

void da_free(struct dyn_array_t *parr)
{
        assert(parr);

        free(parr->data);
        da_init(parr); // заполнить нулями структуру
}

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;
}

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;
}
```

Достоинства и недостатки массивов

`+`:

- простота использования
- константное время доступа к любому элементу
- не тратят лишние ресурсы
- хорошо сочетаются с двоичным поиском

`-`:

- хранение меняющегося набора значений

19-21. Линейный односвязный список

- Линейный односвязный список. Узел списка. Сравнение с массивом. Добавление в начало и конец, удаление.
- Линейный односвязный список. Узел списка. Сравнение с массивом. Вставка перед узлом, после узла.
- Линейный односвязный список. Узел списка. Сравнение с массивом. Универсальный обход.

Общее

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

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

Узел обычно состоит из 2х частей:

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

Отличие списков от массивов

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

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

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

- 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.**

Свойства односвязного списка:

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

```
typedef struct node node_t;

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

Добавление в начало и конец, удаление

```
node_t *push_back(node_t *head, node_t *elem)
{
        if (!head)
                return elem;

        node_t *cur = head;
        while (cur->next)
                cur = cur->next;

        cur->next = elem;
        return head;
}

node_t *push_front(node_t *head, node_t *elem)
{
        if (!head)
                return elem;

        elem->next = head;
        return elem;
}
```

```
void list_remove(node_t **head, int data)
{
    while (*head)
    {
        if (data == (*head)->data)
        {
            node_t *tmp = *head;
            *head = (*head)->next;
            free(tmp);
        }
        else
           head = &(*head)->next;
    }
}
```

Освобождение

```
void free_list(node_t *head)
{
        node_t *cur = head;
        while (cur)
        {
                node_t *next = cur->next;
                free(cur);
                cur = next;
        }
}
```

Вставка перед узлом, после узла

```
int insert_back(node_t **head, node_t *elem, int data)
{
    if (!(*head) || !elem)
        return -1;

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

        if (cur == NULL)
        return ERR_NOT_FOUND;

        elem->next = cur->next;
    cur->next = elem;
    return ERR_OK;
}

int insert_front(node_t **head, node_t *elem, int data)
{
```

```
    if (!(*head) || !elem)
        return -1;

    node_t *cur = *head;
    if (cur->data == data)
    {
        elem->next = cur;
        *head = elem;
        return ERR_OK;
    }

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

        if (cur->next == NULL)
        return ERR_NOT_FOUND;
    elem->next = cur->next;
    cur->next = elem;
    return ERR_OK;
}
```

Универсальный обход

```
void list_map(node_t *head, void map(node_t *node, void *param), void *param)
{
        node_t *cur = head;
        while (cur)
        {
                map(cur, param);
                cur = cur->next;
        }
}
```

22-25. Двоичное дерево поиска

- Двоичное дерево поиска. Узел дерева. Сравнение с деревом. Добавление.
- Двоичное дерево поиска. Узел дерева. Сравнение с деревом. Поиск. Рекурсивный и итеративный.
- Двоичное дерево поиска. Узел дерева. Сравнение с деревом. Универсальный обход.
- Двоичное дерево поиска. Узел дерева. Сравнение с деревом. Удаление.

Дерево - связный ациклический граф.

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

У двоичного дерева значение левого потомка меньше значения родителя, а значение правого потомка больше значения родителя для каждого узла дерева. Это свойство позволяет реализовать эффективный поиск.

Узел дерева обычно состоит из двух частей:

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

Узел

```
struct tree_node
{
        // данные
        const char *name;

        // родитель
        // struct tree_node *parent;

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

struct tree_node *create_node(const char *name)
{
        struct tree_node *node = malloc(sizeof(struct tree_node));
        if (node)
        {
                node->name = name;
```

```
                node->left = NULL;
                node->right = NULL;
        }
        return node;
}

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

Добавление узла в дерево. Рекурсивное и итеративное

```
// рекурсивное
struct tree_node *insert(struct tree_node *tree, struct tree_node *node)
{
        int cmp;
        if (tree == NULL)
                return node;

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

        return tree;
}

// итеративное
struct tree_node *insert(struct tree_node *tree, struct tree_node *node)
{
        int cmp;
        if (tree == NULL)
                return node;

        struct tree_node *cur = tree;
        struct tree_node *parent = NULL;

        while (cur)
        {
                parent = cur;
                cmp = strcmp(node->name, cur->name);
                if (cmp == 0)
                        assert(0);
                else if (cmp < 0)
                        cur = cur->left;
                else
                        cur = cur->right;
```

```
        }

        if (cmp < 0)
                parent->left = node;
        else
                parent->right = node;

        return tree;
}
```

Освобождение памяти

```
void free_tree(struct tree_node *tree)
{
        if (!tree)
                return;

        free_tree(tree->left);
        free_tree(tree->right);
        free(tree);
}
```

Поиск. Рекурсивный и итеративный

```
// рекурсивный
struct tree_node *lookup(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 loookup(tree->left, name);
        else
                return lookup(tree->right, name);
}

// итеративный
struct tree_node *lookup_iter(struct tree_node *tree, const char *name)
{
        int cmp;
        while (tree)
        {
                cmp = strcmp(name, tree->name);
                if (cmp == 0)
                        return tree;
                else if (cmp < 0)
                        tree = tree->left;
```

```
                else
                        tree = tree->right;
        }
        return NULL;
}
```

Удаление. Рекурсивное и итеративное

```
struct tree_node *find_min(struct tree_node *tree)
{
        while (tree->left)
                tree = tree->left;
        return tree;
}

// рекурсивное
struct tree_node *delete_node(struct tree_node *tree, const char *name)
{
        if (!tree)
                return NULL;
        int cmp = strcmp(name, tree->name);
        if (cmp < 0)
                tree->left = delete_node(tree->left, name);
        else if (cmp > 0)
                tree->right = delete_node(tree->right, name);
        else
        {
                // узел с одним потомком: заменить указатель родителя на потомка
                if (!tree->left)
                {
                        struct tree_node *tmp = tree->right;
                        free(tree);
                        return tmp;
                }
                else if (!tree->right)
                {
                        struct tree_node *tmp = tree->left;
                        free(tree);
                        return tmp;
                }

                // узел с двумя потомками

                // 1. найти минимальный элемент в правом поддереве (или макс. в
левом)
                struct tree_node *tmp = find_min(tree->right);

                // 2. заменить узел на его потомка
                strcpy(tree->name, tmp->name);

                // 3. затем удалить найденный минимальный элемент
```

```
                tree->right = delete_node(tree->right, tmp->name);
        }

        return tree;
}

// итеративное
struct tree_node *delete_node(struct tree_node *tree, const char *name)
{
        struct tree_node *parent = NULL, *cur = tree;

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

        // узел не найден
        if (!cur)
                return tree;

        // узел - лист
        if (!cur->left && !cur->right)
        {
                if (!parent)
                        return NULL;
                if (parent->left == cur)
                        parent->left = NULL;
                else
                        parent->right = NULL;

                free(cur);
                return tree;
        }

        // узел с одним потомком: заменить указатель родителя на потомка
        if (!current->left || !current->right)
        {
        struct tree_node *child = (current->left) ? current->left : current-
>right;
        if (!parent)
                return child; // Удаляемый узел - корень
        if (parent->left == current)
                parent->left = child;
        else
                parent->right = child;
        free(current);
        return root;
    }
```

```
        // узел с двуми потомками

        // 1. найти минимальный элемент в правом поддереве

        struct tree_node *successor_parent = cur;
        struct tree_node *successor = cur->right;

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

        // 2. заменить узел этим элементом

        strcpy(cur->name, successor->name);

        // 3. затем удалить найденный узел

        if (successor_parent->left == successor)
                successor_parent->left = successor->right;
        else
                successor_parent->right = successor->right;

        free(successor);

        return tree;
}
```

Обход

```
// префиксный (копирование дерева)
void prefix(node_t *tree, void (*f) (struct tree_node *tree, void *arg), void
*arg)
{
        if (tree == NULL)
                return;

        f(tree, arg);
        prefix(tree->left);
        prefix(tree->right);
}

// инфиксный (сортировка)
void prefix(node_t *tree, void (*f) (struct tree_node *tree, void *arg), void
*arg)
{
        if (tree == NULL)
                return;
```

```
        prefix(tree->left);
        f(tree, arg);
        prefix(tree->right);
}

// постфиксный (удаление дерева)
void prefix(node_t *tree, void (*f) (struct tree_node *tree, void *arg), void
*arg)
{
        if (tree == NULL)
                return;

        prefix(tree->left);
        prefix(tree->right);
        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.

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

```
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");
}
```

26. Куча в Си. Алгоритмы работы malloc, free

- Куча в программе на си. Алгоритм работы функции *malloc*. Пример реализации
- Куча в программе на си. Алгоритм работы функции *free*. Пример реализации
- Куча в программе на си. Проблемы выравнивания выделенной области памяти.

Области, в которых программа может размещать данные

1. сегменты данных - константы и глобальные переменные
2. стек - для вызова функций и создания локальных переменных
3. куча - для динамического выделения памяти

Общее

Куча – пул доступной памяти. В куче нет определенного порядка в расположении элементов.

Происхождение термина куча. Вероятно, как противопоставление термину **стека**, т.к. в стеке элементы расположены строго один над другим, а в куче нет определенного порядка в расположении элементов.

Свойства динамическая памяти:

- Для хранения данных используется *куча*
- Создать переменную в «куче» нельзя, но можно выделить память под нее.

Преимущества и недостатки динамической памяти.

+

1. размер данных известен на этапе выполнения программы и не нужен на этапе компиляции
2. размер данных, размещающихся в куче на несколько порядков больше размера данных, размещаемых на стеке
3. время жизни данных в куче никак не связано со временем жизни того блока, в котором выделялась память под эти данные. Т.е. можно выделить память в одной функции, а очистить в другой.

   –
4. Ручное управление временем жизни (сами выделили память, сами освободили память)

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.

Область

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

Инициализация области

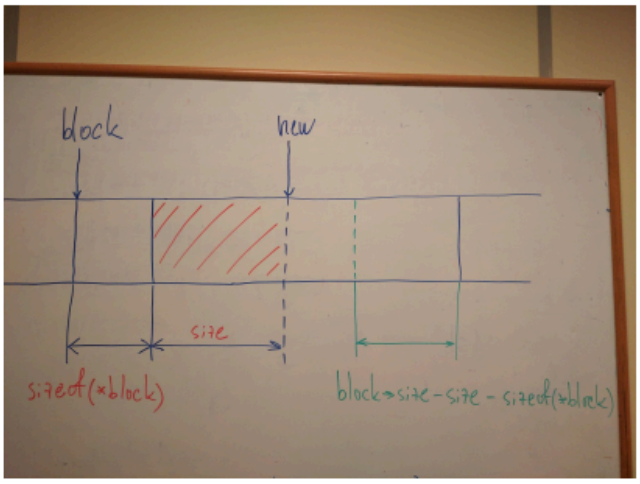
```
#define MY_HEAP_SIZE 1000000

// 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;
}
```

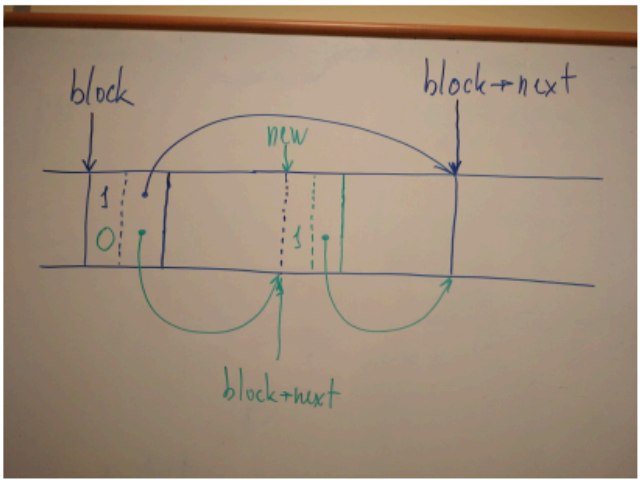
Алгоритм работы malloc

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 свободной.
- Если область не найдена, вернуть нулевой указатель.

Реализация malloc/free





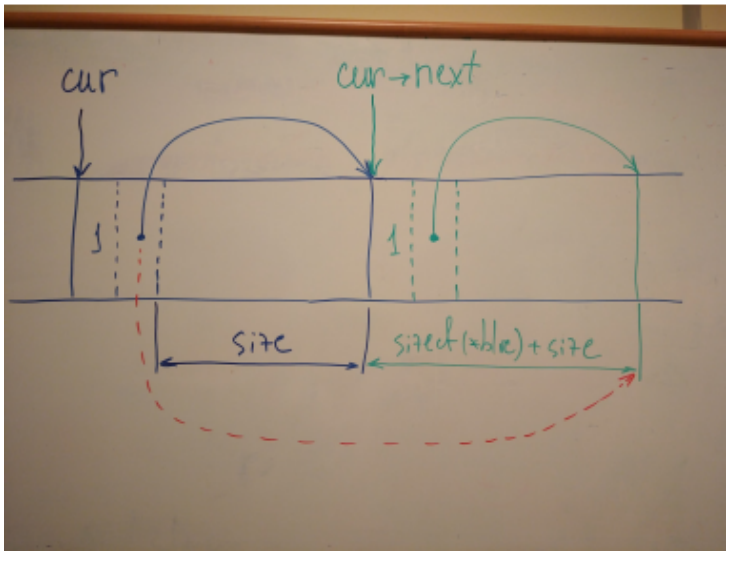

```
static void split_block(struct block_t *block, size_t size)
{
        // остаток области памяти после выделения
        size_t rest = block->size - size;

        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;
}

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

        // все глобальные переменные в Си неявно инициализируются нулями
        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;
}
```

Алгоритм работы free

Освобождение области памяти (free):

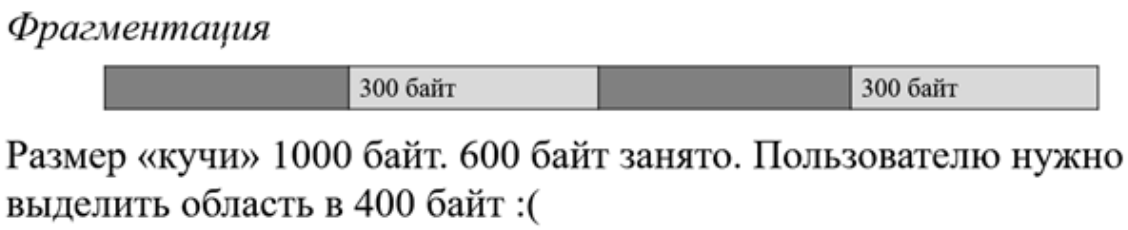
- Просмотреть список занятых/свободных областей памяти в поисках указанной области.
- Пометить найденную область как свободную.
- Если освобожденная область вплотную граничит со свободной областью с какой-либо из двух сторон, то объединить их в единую область большего размера.




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

        while (cur && cur->next)
        {
                if (cur->free && cur->next->free)
```

```
                {
                        cur->size += cur->next->size + sizeof(struct block_t);
                        cur->next = cur->next->next;
                }
                else
                        cur = cur->next;
        }
}

void my_free(void *ptr)
{
        struct block_t *cur = ptr;
        --cur;
        cur->free = 1;

        merge_blocks();
}
```

Дефрагментация



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

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

Проблемы выравнивания выделенной области памяти.

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.

```
// определить самый требовательный тип
typedef long long align_t;

// объединение будет выравнено по правильному адресу – самого требовательного типа
```

```
union block_t
{
        // структура будет выравнена по максимальному элементу
        struct
        {
                size_t size;
                int free;
                union block_t *next;
        } block;

        align_t x;
};

// чтобы все были выравнены одинаковыми, размер выделенной памяти кратной
размеру метаинформации - размер области в блоках - затем в байтах, но правильно
```

Запрашиваемый размер области обычно округляется до размера кратного размеру заголовка.

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

Выравнивание - размещение значений в памяти по адресам, кратным некоторому целому числу, больше единицы.

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

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

27. VLA, alloca

VLA

Появились в C99.

```
scanf("%d", &n);
int a[n];
```

- 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.

```
scanf("%d", &n);
int *a = alloca(n * sizeof(int));

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

`+`:

- Выделение происходит быстро
- Выделенная область освобождается автоматически

`-`:

- функция нестандартная

- серьезные ограничения по размеру области

```
// VLA, когда тело цикла закончится, массив разрушится (т.к. переменная вышла из области видимости)
void foo(int size)
{
        ...
        while (b)
        {
                char tmp[size];
                ...
        }
}

// alloca, массив выделится, но память останется до конца работы функции и легко получить переполнение стека
void foo(int size)
{
        ...
        while (b)
        {
                char *tmp = alloca(size);
                ...
        }
}
```

28. 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, ...);
```

Напишем функцию, вычисляющую среднее арифметическое своих аргументов.
Проблемы:

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

Наивная реализация. Причина некорректной работы.

По *cdecl* параметры в функцию передаются справа налево, т.е. можно получить адрес параметра так:

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

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

Однако работать это не будет, так как аргументы могут передаваться с *выравниванием*.

Стандартный способ работы с параметрами функций с переменным числом параметров - stdarg.h

- va_list
- void va_start(va_list argptr, last_param)

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

```
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("%lf\n", double);
                sum += num;
        }
        va_end(vl);
        return sum / n;
}
```

29-32. Препроцессор

- Препроцессор. Общие понятия. Простые макросы. Директивы условной компиляции.
- Препроцессор. Общие понятия. Макросы с параметрами.
- Препроцессор. Общие понятия. Операция решетка и двойная решетка.

Общее

Препроцессор - программа, подготавливающая код программы к компиляции. При запуске программы, препроцессор просматривает код сверху вниз, файл за файлом, в поиске директив.

Задачи препроцессора:

- Удаление комментариев
- Выполнение директив

Директивы - это специальные команды, которые начинаются с символа `#` и заканчиваются на символе `\n`. Любое количество пробельных символов может разделять лексемы в директиве.

```
#define     N   100

#define DISK_CAPACITY   (SIDES *         \
                                        TRACKS_PER_SIDE *   \
                                        SECTORES_PER_TRACK)
```

Типы директив:

- макроопределения `#define, #undef`
- директива включения файлов `#include`
- директивы условной компиляции `#if, #ifdef, #ifndef, #endif, ...`
- остальные директивы (редко используются) `#pragma, #error, ...`

Простые макросы

`#define идентификатор список-замены`

Используются:

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

```
#define PI 3.14
```

```
#define EOS '\0'
```

2. незначительного изменения синтаксиса языка

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

3. переименования типов

```
#define BOOL int
```

4. предотвращение повторного включения заголовочных файлов (`include guard`)

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`.

Директивы условной компиляции

Директива `if` в языке программирования позволяет проверить условие и выполнить определенный блок кода, если условие выполняется.

Директива `ifdef` (или `ifndef`) также проверяет условие, но в зависимости от того, создано ли макроопределение с таким именем. Если оно создано (или не создано), то выполняется определенный блок кода.

```
// одно и то же

// здесь можно написать сложное условие
#if defined(OS_WIN)

// здесь нельзя
#ifdef OS_WIN
```

Использование условной компиляции:

```
// программа, которая должна работать под несколькими операционными системами
#ifdef OS_WIN
...
#elif OS_MAC
...
```

```
#else
...
#endif
---
// программа, которая должна собираться различными компиляторами
#if __GNUC__
...
#elif __clang__
...
#endif
---
// начальное значение макросов
#ifndef BUF_SIZE
#define BUF_SIZE 256
#endif
---
// временное выключение кода
#if 0
for (int i = 0; i < n; i++)
        a[i] = 0.0;
#endif
```

Макросы с параметрами

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

- Не должно быть пробела между `именем макроса` и `(`.
- Список параметров может быть пустым

Используются:

```
#define MAX(x, y) ((x) > (y) ? (x) : (y))
#define IS_EVEN(x) ((x) % 2 == 0)

...

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

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

Макросы с переменным числом параметров (C99)

используется условная компиляция с помощью `NDEBUG`:

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

```
#define DBG_PRINT(s, ...) ((void) 0)
#endif
```

Макросы с параметрами vs функции

`+`:

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

`-`:

- скомпилированный код становится больше;

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

- 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

Часто используется *do-while* для возможности поставить `;` в конец макроса.

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

Оператор `,`

```
#define ECHO(s) (gets(s), puts(s))
```

Обертка в блок

```
#define ECHO(s) {gets(s); puts(s);}
```

Предопределенные макросы

Эти идентификаторы нельзя переопределять или отменять директивой `undef`.

- `__LINE__` – номер текущей строки (десятичная константа)
- `__FILE__` – имя компилируемого файла
- `__DATE__` – дата компиляции
- `__TIME__` – время компиляции
- и др.
- `__func__` – имя функции как строка (GCC only, C99 и не макрос)

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
Часто используемые команды:

- `pragma once`
- Выравнивание (упаковка структурных переменных `pragma pack`)
- Прочее... (отключение/включение ошибок компилятора, запрет на использование функции и др.)

Операции `#` и `##`

- операция `#` конвертирует аргумент макроса в строковый литерал

```
// скобки вокруг параметра не нужны
#define PRINT_INT(n) printf(#n " = %d\n", (n))

...
PRINT_INT(i / j);
```

- операция `##` объединяет две лексемы в одну

```c
#define CONCAT(a, b) a##b

...
int var1 = 42;
printf("Value of var1: %d\n", CONCAT(var, 1)); // Превращается в var1
```

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.

33. Встраиваемые функции

Ключевое слово inline. Цель добавления. Сравнение с макросами

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

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

В C99 `inline`-реализация не предоставляет и не запрещает реализацию со внешней линковкой. `inline` означает, что определение функции предоставляется только для подстановки и где-то в программе должно быть другое такое же определение этой же функции.

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

int main(void)
{
        int i = add(4, 5);
        return i;
}
// main.c:(.text+0x1e): undefined reference to `add'
// collect2.exe: error: ld returned 1 exit status
```

Способы исправления проблемы «unresolved reference»

- использовать ключевое слово `static`. Такая функция доступна только в текущей единице трансляции.

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

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

```
        return i;
}
```

- использовать ключевое слово `extern`. Такая функция доступна из других единиц трансляции.

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

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

- Добавить еще одно такое же `не-inline` определение функции где-нибудь в программе. Самый плохой способ решения проблемы, потому что реализации могут не совпасть.
- Не использовать `inline`: компиляторы и так могут оптимизировать функции.

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

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

`inline` был введен в C99 с целью **оптимизации производительности** за счет накладных расходов на вызов функции. До этого программисты часто пользовались макросами для решения похожих задач, но макросы имеют недостатки (отсутствие контроля типов, непредсказуемое поведение при использовании сложных выражений, сложность отладки). `inline` - это безопасная альтернатива макросам.

34-37. Библиотеки

- Библиотеки. Статические библиотеки. Порядок компоновки библиотек в Linux.
- Библиотеки. Динамические библиотеки. Динамическая компоновка. Видимость функций в Linux и в Windows.
- Библиотеки. Динамические библиотеки. Динамическая загрузка.
- Библиотеки. Динамические библиотеки. PIC, GOT, PLT. (Про Linux.)
- Библиотеки. Динамические библиотеки. Подходы к реализации функций, которым требуется создать буфер динамически.
- Библиотеки. Динамические библиотеки на Си. Приложение на Python. Использование модуля ctypes на примере функции целочисленного сложения и деления.
- Библиотеки. Динамические библиотеки на Си. Приложение на Python. Использование модуля ctypes на примере функций обработки массивов.
- Библиотеки. Динамические библиотеки на Си. Приложение на Python. Реализация модуля расширения на примере функции целочисленного сложения и деления.
- Библиотеки. Динамические библиотеки на Си. Приложение на Python. Обработка массивов в модуле расширения.

Общее

Библиотека включает в себя

- заголовочный файл;
- откомпилированный файл самой библиотеки (библиотеки меняются редко - нет причин перекомпилировать каждый раз).

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 пересобрать

Динамические библиотеки - подпрограммы из библиотеки загружаются в приложение во время выполнения. Код библиотеки не помещается в исполняемый файл.

`+`:

- Несколько программ могут «разделять» одну библиотеку.
- Меньший размер приложения (по сравнению с приложением со статической библиотекой).
- Модернизация библиотеки не требует перекомпиляции программы.
- Могут использовать программы на разных языках.
  `–`:
- Требуется наличие библиотеки на компьютере
- Версионность библиотек

Способы использования динамических библиотек:

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

Подходы к реализациям функций, которым требуется создавать буфер динамически.

- Реализация функции выделения и освобождения внутри библиотеки;
- Перекладывание ответственности на вопросы, связанные с выделением памяти и освобождения, на вызывающую сторону

Использование статической библиотеки

Сборка библиотеки
– компиляция
`gcc -std=c99 -Wall -Werror -c arr_lib.c`

– упаковка (с заменой объектного файла на новый)
`ar cr libarr.a arr_lib.o`

– индексирование (необязательно)
`ranlib libarr.a`

Сборка приложения

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

# gcc -std=c99 -Wall -Werror main.c -o app.exe -L. -larr
# -L - в какой директории осуществить поиск библиотеки
# -l [arr] - поиска библиотеки lib[arr].a
```

Граф зависимостей

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

# библиотека может быть не найдена операционной системой
# во время запуска
# LD_LIBRARY_PATH=$LD_LIBRARY_PATH:. ./app.exe
# ---
# во время динамической компоновки
# gcc -std=c99 -Wall -Werror main.c -L. -larr -o app.exe -Wl,-rpath=.
```

Использование динамической библиотеки (динамическая загрузка)

Сборка библиотеки

– компиляция

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

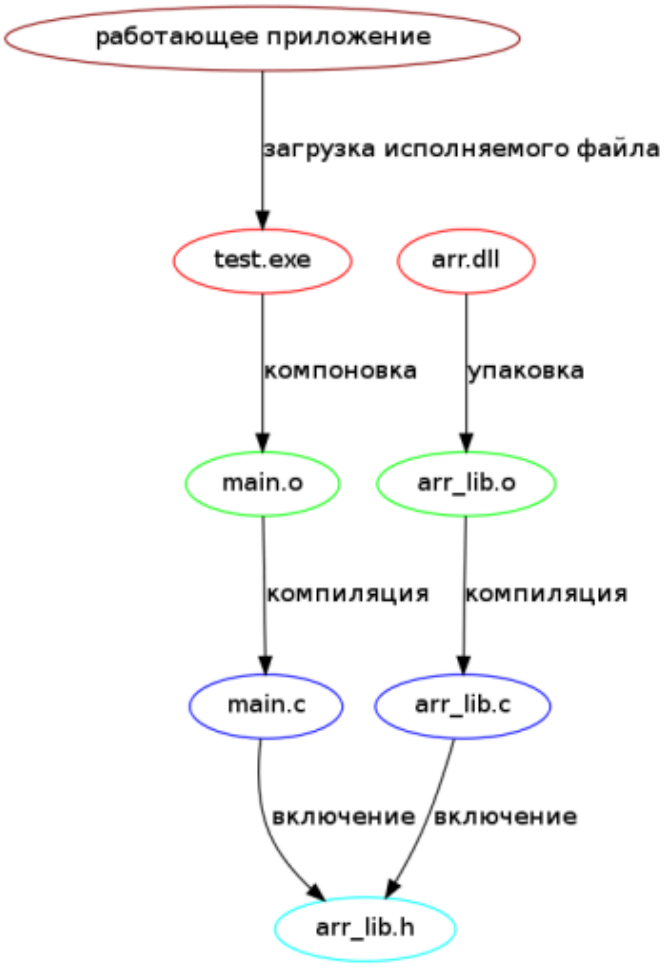
– компоновка

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

Сборка приложения

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

Граф зависимостей



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);
```

Динамические библиотеки в Windows

Любая функция без модификатора `static`, которая помещается в динамическую библиотеку в *Linux*, доступна наружу. В *Windows* - наоборот, все функции по умолчанию из библиотеки наружу не доступны.

```
// при сборке дин. библиотеки под Windows в исходном коде – должны произойти изменения

// -- файл arr_lib.h (библиотека) --
```

```
// с помощью __declspec помечаются функции - доступные наружу
// cdecl - соглашение о вызове
__declspec(dllexport) void __cdecl arr_form(int *arr, int n);

__declspec(dllexport) void __cdecl arr_print(const int *arr, int n);


// -- файл main.c (приложение) --

__declspec(dllimport) void __cdecl arr_form(int *arr, int n);

__declspec(dllimport) void __cdecl arr_print(const int *arr, int n);

int main(void)
...
```

Динамическая библиотека (динамическая компоновка)

```
# при сборке библиотеки без -fPIC тоже все работает (почему?)
gcc -std=c99 -Wall -Werror -Wpedantic -fPIC -c arr_lib.c
gcc -o arr.dll -shared -Wl,--subsystem,windows arr_lib.o

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

Динамическая библиотека (динамическая загрузка)

```
// arr_lib.h (нужно подключать и на стороне приложения в main.c)

#ifndef __ARR__LIB__H__
#define __ARR__LIB__H__

#ifdef ARR_EXPORT
#define ARR_DLL __declspec(dllexport)
#else
#define ARR_DLL __declspec(dllimport)
#endif

#define ARR_DECL __cdecl

ARR_DLL void ARR_DECL arr_form(int *arr, int n);

ARR_DLL void ARR_DECL arr_print(const int *arr, int n);

#endif // __ARR__LIB__H__
```

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

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

Windows API для работы с динамическими библиотеками

`<windows.h>`

```
HMODULE LoadLibrary(LPCSTR);

FARPROC GetProcAddress(HMODULE, LPCSTR);

FreeLibrary(HMODULE);
```

Динамическая библиотека на C, приложение на Python. Модуль ctypes. Функции сложения и деления. Функции обработки массивов.

Нужно подключить библиотеку и для функции указать типы (принимаемые и возвращаемые):

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.

```
# Использование *ctypes* на примере функций сложения, деления, функций для работы с массивами
import ctypes

lib = ctypes.CDLL('./arr.so')

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

# int divide(int, int, int *)
_div = lib.divide
_div.argtypes = (ctypes.c_int, ctypes.c_int, ctypes.POINTER(c_int))
_div.restype = ctypes.c_int

def div(x, y):
        rem = ctypes.c_int()
        quot = _div(x, y, rem)

        return quot, rem.value
```

```
# функция принимает массив только на чтение
# double 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)

# функция принимает массив только на запись
# void fill_array(double *, int)
_fill = lib.fill_array
_fill.argtypes = (ctypes.POINTER(ctypes.c_double), ctypes.c_int)
_fill.restype = None

def fill(n):
        arr = (ctypes.c_double * n)()
        _fill(arr, n)

        return list(arr)

# функция принимает массив на чтение и на запись
# int filter(double *, int, double *, int *)
_filter = lib.filter
_filter.argtypes = (ctypes.POINTER(ctypes.c_double), ctypes.c_int, \
                                    ctypes.POINTER(ctypes.c_double),
ctypes.POINTER(ctypes.c_int))
_filter.restype = ctypes.c_int

def filter(nums):
        src_len = len(nums)
        src = (ctypes.c_double * src_len)(*nums)
        dst_len = ctypes.c_int(0)

        rc = _filter(src, src_len, None, dst_len)
        if not rc:
                return rc, list()

        dst = (ctypes.c_double * dst_len.value)()
        rc = _filter(src, src_len, dst, dst_len)
        return rc, list(dst)


print(add(5, 3))  # 8
print(add(-8, -5))  # -13

print(div(7, 5))  # (1, 2)
print(div(4, 2))  # (2, 0)
```

```
print(avg([1, 2, 3, 4, 5]))  # 3.0
print(avg([1]))  # 1.0

print(fill(5))  # [0.0, 1.0, 2.0, 3.0, 4.0]
print(fill(1))  # [0.0]

print(filter([1, 2, 3, 4, 5]))  # (0, [])
print(filter([-1, -2, -3, -4, -5]))  # (0, [-1.0, -2.0, -3.0, -4.0, -5.0])
print(filter([-1, -2, 3, 4, -5]))  # (0, [-1.0, -2.0, -5.0])
```

Проецирование типов:

`type *` -> `POINTER(type)`
`int` -> `c_int`
`void` -> `None`
`int[N]` -> `(c_int * N)`

Итоги:

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

Динамическая библиотека на C, приложение на Python. Модуль расширения. Функции сложения и деления

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.

```
#include <Python.h>

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

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

// int divide(int, int, int *);
static PyObject *divide_numbers(PyObject *self, PyObject *args)
{
        int a, b, quotient, remainder;
        if (!PyArg_ParseTuple(args, "ii", &a, &b))
                return NULL;
        quotient = divide(a, b, &remainder);
        return Py_BuildValue("(ii)", quotient, remainder);
}

// метаинформация
static PyMethodDef methods[] = {
        // имя функции на Питон, имя самой функции, набор флагов для вызова этой
фукнции, строка документирования
        {"add", add_numbers, METH_VARARGS, "Integer Addiction"},
        {"divide", divide_numbers, METH_VARARGS, "Integer Division"},
        {NULL, NULL, 0, NULL} // сигнал о том, что таблица методов заполнена
}

// описание модуля
static struct PyModuleDef module = {
        PyModuleDef_HEAD_INIT, // по стандарту
        "extension_module", // имя модуля
        NULL,  // строка документирования
        -1, // набор флагов, которые говорят каким именно образом вызывается
интерпретатор
        methods  // таблица функций модуля расширения
};

// функция инициализации модуля расширения
PyMODINIT_FUNC PyInit_module(void)
{
        return PyModule_Create(&module);
}
```

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

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

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

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

Модуль расширения. Обработка массивов в модуле расширения

PIC, GOT, PLT.

`-fPIC` : генерирует код, который может быть загружен в любое место памяти, что необходимо для динамических библиотек. Функции и данные внутри библиотеки скомпилированной с `-fPIC` используют относительные адреса, а не абсолютные. В Linux динамические библиотеки должны быть позиционно-независимыми, потому что в некоторых архитектурах (например, *x86_64*) механизм релокации сложнее и накладывает ограничения на использование абсолютных адресов.

компоновщик - однопроходный. Значит порядок такой:

1. передать `что нуждается`
2. передать `где это реализовано`

```
# скомпоновалось только потому, что main.o -> a.a -> b.a
gcc -o app.exe main.o -L. -la -lb

# если изменить порядок, то будет ошибка компоновки
```

Если библиотеки ссылаются друг на друга, то

```
# передали компоновщику несколько o-файлов,
# чтобы он сам разбирался с порядком
gcc -o app.exe -L. -Wl,-\( -la -lb main.o -Wl,\)
```

LD_PRELOAD

Пример: соберем библиотеку `failmalloc.c` :

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

```
        return NULL;
}
```

```
gcc -std=c99 -Wall -Werror -fPIC -c failmalloc.c
gcc -shared -o libfailmalloc.so failmalloc.o
```

Для того, чтобы приложение, запускающее библиотеку схватило наш `malloc`, а не из `stdlib.h` используем `LD_PRELOAD`

```
LD_PRELOAD=./libfailmalloc.so ls  # ls: memory exhausted
```

attribute ((visibility=("default")))

По умолчанию любая функция без модификатора `static`, которая помещается в динамическую библиотеку в *Linux*, доступна наружу. В *Windows* - наоборот, все функции по умолчанию из библиотеки наружу не доступны.

```
__attribute__ ((visibility=("default")))
void lib(void)
{
        func1();

        func2();
}
```

и во время компиляции библиотеки добавляем ключ `-fvisibility=hidden`, чтобы по умолчанию все функции были наружу недоступны.

38-39. Абстрактные типы данных

- АТД, модуль, разновидности модулей, АО - стек целых чисел
- АТД, модуль, разновидности модулей, АТД - стек целых чисел

Модуль и его разновидности

Программу удобно рассматривать как набор независимых модулей.

- *Модуль* состоит из двух частей: интерфейса (он один) и реализации (может быть несколько).
- *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`.
- Часть кода, которая использует модуль, называют *клиентом*.
- Клиент должен зависеть только от интерфейса, но не от деталей его реализации.

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

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

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`

Типы модулей

- **Набор данных**
  Набор связанных переменных и/или констант. В Си модули этого типа часто представляются только заголовочным файлом. (`float.h`, `limits.h`)
- **Библиотека**
  Набор связанных функций
- **Абстрактный объект**
  Набор функций, который обрабатывает скрытые данные.
- **Абстрактный тип данных**
  Интерфейс, который определяет тип данных и операции над этим типом. Тип данных называется абстрактным, потому что интерфейс скрывает детали его представления и реализации.

Пример АО стек целых чисел

```h
#ifndef __STACK__O__H__
#define __STACK__O__H__

#include <stdbool.h>

void make_empty(void);
bool is_empty(void);
bool is_full(void);
int push(int i);
int pop(int *i);

#endif // __STACK__O__H__
```

```c
#include <stddef.h>
#include "stack_ao.h"

#define STACK_SIZE 10

static int content[STACK_SIZE];
static size_t top;

void make_empty(void)
{
        top = 0;
}

bool is_empty(void)
{
        return top == 0;
}

bool is_full(void)
{
        return top >= STACK_SIZE;
}
```

```c
int push(int i)
{
        if (is_full())
                return 1;
        content[top++] = i;
        return 0;
}

int pop(int *i)
{
        if (is_empty())
                return 1;
        *i = content[--top];
        return 0;
}
```

Неполный тип в Си

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

- Пока тип неполный его использование ограничено.
- Описание неполного типа должно быть закончено где-то в программе
- Допустимо определять указатель на неполный тип `typedef struct t *T;`

Можно:

- определять переменные типа *T*
- передавать эти переменные как аргументы в функцию

Нельзя:

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

Пример АТД стек целых чисел

```c
#ifndef __STACK__H__
#define __STACK__H__

#include <stdbool.h>

typedef struct stack_type *stack_t;

void make_empty(stack_t s);
bool is_empty(const stack_t s);
bool is_full(const stack_t s);
int push(stack_t s, int i);
int pop(stack_t s, int *i);
```

```
stack_t create(void);
void destroy(stack_t s);

#endif // __STACK__H__
```

```
#include <assert.h>
#include <stddef.h>
#include "stack.h"

#define STACK_SIZE 10

struct stack_type
{
        int content[STACK_SIZE];
        size_t top;
};

stack_t create(void)
{
        stack_t s = malloc(sizeof(struct stack_type));
        if (s)
                make_empty(s);
        return s;
}

void destroy(stack_t s)
{
        free(s);
}

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

bool is_empty(const stack_t s)
{
        assert(s);
        return s->top == 0;
}

bool is_full(const stack_t s)
{
        assert(s);
        return s->top >= STACK_SIZE;
}

int push(stack_t s, int i)
{
```

```
        assert(s);
        if (is_full(s))
                return 1;
        s->content[(s->top)++] = i;
        return 0;
}

int pop(stack_t s, int *i)
{
        assert(s);
        if (is_empty(s))
                return 1;
        *i = s->content[--(s->top)];
        return 0;
}
```

40-41. Списки ядра Linux. Идея, основные моменты, реализация

- Списки ядра Linux, идея, основные моменты использования
- Списки ядра Linux, идея, основные моменты реализации

Общее

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

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

В отличие от обычных списков, где данные содержатся в элементах списка, структура *list_head* должна быть частью сами данных.

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

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

Универсальная реализация достигается макросами.

```
// main.c
...

struct data_t
{
        int num;
        struct list_head list;
};

int main(void)
{
        LIST_HEAD(num_list);
```

```c
#if 0
        struct data_t num_list;
        INIT_LIST_HEAD(&(num_list.list));
#endif

        // добавление
        struct data_t *item;
        for (int i = 0; i < 10; i++)
        {
                item = malloc(sizeof(*item));
                if (!item)
                        break;
                item->num = i;
                INIT_LIST_HEAD(&(item->list));

                list_add(&(item->list), &num_list);
        }

        // обход (1)
        struct list_head *iter;
        list_for_each(iter, &num_list)
        {
                item = list_entry(iter, struct data, list);
                printf("LIST: %d\n", item->num);
        }

        // обход (2)
        list_for_each_entry(item, &num_list, list)
                printf("LIST: %d\n", item->num);

        // обход с целью изменения списка (например, удаления каких-то
элементов)
        // освобождение
        struct list_head *safe;
        list_for_each_safe(iter, safe, &num_list)
        {
                item = list_entry(iter, struct data, list);
                // удалить узел из списка
                list_del(iter);

                free(item);
        }

        return 0;
}
```

Реализация

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

#define LIST_HEAD(name) \
```

```
        struct list_head name = LIST_HEAD_INIT(name)

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

// 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)


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

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

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

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

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

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

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

// 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;
}
```

Макрос container_of

Макрос *container_of* расширения GNU в ядре Linux используется для получения указателя на структуру, содержащую определенное поле. Непосредственно используется в *list_entry*.

offsetof - это стандартная функция C, которая возвращает смещение поля от начала структуры.

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

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

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

offsetof: идея

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

`((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

```
struct s
{
        char c;
        int i;
        double d;
};

...
// В нашем случае TYPE - struct s, MEMBER - i, size_t - unsigned int
printf("offset of i is %d\n", offsetof(struct s, i));
```