# Chapter 1

# Вопросы

## Вопрос 1 Программа, состоящая из нескольких файлов

- компиляция и линковка
- заголовочные файлы
- утилита make

## Вопрос 2 Указатели, массивы, ссылки

- применение указателей и ссылок
- арифметика указателей

## Вопрос 3 Три вида памяти. Работа с кучей на C

- глобальная/статическая память, стек, куча
- `malloc/calloc/realloc/free`
- `void*`

## Вопрос 4 Структуры. Неинтрузивный связный список на C

- неинтрузивная реализация
- typedef

## Вопрос 5 Структуры. Интрузивный связный список на C

- интрузивная реализация
- typedef

## Вопрос 6 Функции. Указатели на функции

- как происходит вызов функции
- реализация сортировки
- `void sort(void* base, size_t num, size_t size, int (*compar)(const void*,const void*));`

## Вопрос 7 Обзор стандартной библиотеки C

- `string.h (memcpy, memcmp, strcpy, strcmp, strcat, strstr, strchr, strtok)`
- `stdlib.h (atoi, strtoll, srand/rand, qsort)`

## Вопрос 8 Ввод-вывод на C. Текстовые файлы

- FILE, fopen, fclose, r/w, t/b
- stdin, stdout, stderr
- printf, scanf, fprintf, fscanf, sprintf, sscanf, fgets
- обработка ошибок, feof, ferror

## Вопрос 9 Ввод/вывод на C. Бинарные файлы

- FILE, fopen, fclose, r/w, t/b, буферизация
- fread, fwrite, fseek, ftell, fflush
- обработка ошибок, feof, ferror

## Вопрос 10 Классы и объекты

- инкапсуляция: private/public
- конструктор (overloading), деструктор
- инициализация полей (в том числе C++11)
- C++11: =default, constructor chaining

## Вопрос 11 Работа с кучей на C++

- new/delete
- создание объектов в куче
- конструктор копий
- оператор присваивания
- C++11: =delete

## Вопрос 12 Наследование и полиморфизм

- protected
- virtual (overriding)
- таблица виртуальных функций
- статическое/динамическое связывание

## Вопрос 13 Умные указатели

- `scoped_ptr`
- `unique_ptr`
- `shared_ptr`

## Вопрос 14 Перегрузка операторов

- бинарные и унарные
- в классе/вне классе
- приведение типов

## Вопрос 15 Ключевые слова extern, static, inline

- extern у переменных
- static у переменных и функций
- static у полей и методов
- inline у функций

## Вопрос 16 Разное

- ключевое слово const (C/C++)
- перегрузка функций
- параметры функций по умолчанию

## Вопрос 17 Наследование: детали

- сортировка и структуры данных C vs ООП
- private/protected наследование
- C+11: final, override

## Вопрос 18 Элементы проектирования

- декомпозиция программы (Model, View)
- автотесты

## Вопрос 19 Множественное наследование

- разрешение конфликтов имен
- виртуальное наследование
- наследование интерфейсов

# Chapter 2

## Вопрос 1 Программа, состоящая из нескольких файлов

- компиляция и линковка
- заголовочные файлы
- утилита make

### i компиляция и линковка

```c
int main() {
    return 0;
}
```

Причины разбиения на файлы:

1. Абстракция
2. Несколько программистов
3. Быстродействие

Пусть в программе будет несколько файлов:

main.c

```c
int main() {
    hello();
    return 0;
}
```

hello.c

```c
void hello() {
    printf("Hello!");
}
```

Для компиляции:

```
$ gcc main.c hello.c -o main
```

Что происходит во время выполнения команды? Каждый файл компилируется по отдельности. В памяти каждый блок соответствует функции.

1. компиляция: из $*.c$ получаются объектные файлы $*.o$
2. линковка: выполняется линковщиком, задача состоит в том, чтобы собрать в один исполняемый файл, объединить блоки в памяти межу собой и выполнить разрешение адресов. Линковщик устанавливает относительные адреса, линковка более быстрый процесс.
   Если мы запустим компиляцию от объектных файлов, будет выполняться только линковка.

Пусть теперь сигнатура *hello* поменялась: теперь там есть параметры. Если скомпилируем по отдельности проблем не будет, но ошибка останется не замеченной. Для этого используются заголовочные файлы.

## ii заголовочные файлы

У функции есть

1. Определение (definition)
2. Объявление (declaration) : сигнатура - получаемое и возвращаемое

hello.h
```c
void hello(int n);
```

hello.c
```c
#include "hello.h"
void hello(int n) {
    printf("\%d", n);
}
```

main.c
```c
#include "hello.h"
int main() {
    hello();
    return 0;
}
```

Теперь одно определение подключается в оба файла. Можно не указывать имя переменной в определении.

Если мы хотим ссылаться в цикле

a.c
```c
#ifndef _a_H_
#define _a_H_
#include "b.h"
#endif
```

b.c
```c
#ifndef _b_H_
#define _b_H_
#include "a.h"
#endif
```

## iii утилита make

GCC:

- только препроцессор → все #... `$ gcc -E`
- только компиляция `$ gcc -c`
- только перевод в ассемблер `$ gcc -s`

Для автоматизации используется *make*.

b.c
```
main: main.o str.o util.o
    gcc  main.o str.o util.o -o main
main.o: main.cpp util.h str.h
    gcc -c main.cpp
str.o: str.cpp str.h
    gcc -c str.cpp
util.o: util.cpp util.h
    gcc -c util.cpp

clean:
     rm -rf *.o
```

# Вопрос 2 Указатели, массивы, ссылки

- применение указателей и ссылок
- арифметика указателей

## i Массивы

```
int array[10]; // size 10*sizeof(int)

a[0] = 10;
int array[5] = {0, 1, 2, 3, 4};
int array[] = {0, 1, 2, 3, 4};
int array[5] = {0};
```

При создании вторым способом компилятор дописывает в конец ”0”.

```
char array[] = {'H', 'e', 'l', 'l', 'o'}
char array[] = "Hello"
```

Двухмерные массивы

```
int m[10][10];
int m[2][2] = { {1, 2}, {3, 4}};
```

## ii Указатели

Это адрес ячейки:

```
int *p; // pointer
int a, b;
p = &a; // address of a
b = *p; // value by addres p

printf("%p", p); // output of addres
```

Теперь a равно b;

Адрес массива — указатель на его первый элемент.

```
char array[10];
char *p;
p = array[0];
p = array;
```

## iii Арифметика указателей

```
int i[10];
char c[10];
int *pi = &i[0];
char *pc = &c[0];
pi+=1;
pc+=1;
```

Когда прибавляем 1 к адресу, он увеличивается на размер типа: в первом случае на 4 байта, во втором на 1 байт.

```
i[3] === *(i+3) === 3[i]
```

Можно вычитать, но нельзя складывать.

## iv Различия между указателями разных типов

На C компилируется, а на плюсах нет.

```
char c[10];
int *pi = &c[0];
```

## v Применение

Полезно передавать в функцию указатель на большой объект.

```
void strlen(char* ptr) {
    char *p = ptr;
    while (*p != '\0')
        ++p;
    return p - ptr;
}
```

Второй вариант будет быстрее:

```
while (p[i] != 1)
    i++;

while (*p != 1)
    p++;
```

```
void swap(int *pa, int *pb) {
    int tmp = *pa;
    *pa = *pb;
    *pb = tmp;
}
int main() {
    int a = 3; int b = 4;
    swap(&a, &b);
    return 0;
}
```

## vi Ссылки

Новая форма запси последнего примера

```
void swap(int& pa, int& pb) {
    int tmp = pa;
    pa = pb;
    pb = tmp;
}
int main() {
    int a = 3; int b = 4;
    swap(a, b);
    return 0;
}
```

# Вопрос 3 Три вида памяти. Работа с кучей на C

- глобальная/статическая память, стек, куча
- `malloc/calloc/realloc/free`
- `void*`

## i Глобальная/статическая

Видна везде, определяется вне функций, память выделяется, когда загружается программа.

hello.h

```h
#ifndef _hello_H_
#define _hello_H_
extern int a;
#endif _hello_H_
```

main.c

```c
#include "hello.h"
int a = 0;
int main() {
    return 0;
}
```

hello.c

```c
int a=3;
void hello() {
}
```

Static

```c
void f() {
static int call_count = 0;
call_count++;
}

int main() {
    f(); f(); f();
}
```

Если объявить *static* вне функции, она не будет видна даже с *extern*.

## ii Стек (stack)

Поддерживает операции *push* и *pop*. Когда функция заканчивается, память, выделенная для нее, освобождается. Вычисляется в момент компиляции, как и глобальная.

При запуске рекурсии большой глубины может переполниться *stack* и произойти аварийное завершение.

## iii Динамическая (heap)

По запросу программиста во время работы:

```c
#include <stdlib.h>
int *p = (int*)malloc(10000*sizeof(int));
p[0] = 1;
free p;
```

*typedef* позволяет задавать новые типы:

```
typedef unsigned int size_t
```

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

*void** — указатель на все, универсальный указатель, который можно привести к любому типу, но запрещена адресная арифметика.

### Утечка памяти. Memory leak

```
int *p = (int*)malloc(10000*sizeof(int));
p = (int*)malloc(10*sizeof(int));
```

### Особенности динамической памяти

1. Скорость выделения меньше, чем у любой другой
2. Имеет смысл выделять только объекты большого размера. Так как кроме самого объекта нужно создать ссылку на место в памяти.

## iv Выделение массива массивов

```
int **m = (int**)malloc(N * sizeof(int*));
for (int i = 0; i < N: ++i) {
    m[i] = (int *)malloc(N * sizeof(int));
}

m[42][42] = 42;

for (int i = 0; i < N; ++i) {
    free(m[i]);
}
free(m);
```

## v Еще

- calloc — выделяет память и инициализирует нулями
- realloc — изменяет размер существующего массива:
    1. если нужное число байт не занято рядом, просто увеличиваем
    2. иначе находим другое место и переносим туда весь массив
    3. если нет вообще памяти, возвращаем 0.

# Вопрос 4 Структуры. Неинтрузивный связный список на C

- неинтрузивная реализация
- typedef

## i Структуры

Сущность — набор переменных: точка, товар.

```
struct product_s {
    char label[256];
    unsigned char weight;
    unsigned int price;
};

scanf("\%s \%f \%d", p.label, &(p.weight), &(p.price));

struct product_s array[100];
array[0].weight = 42;

struct product_s* ptr = malloc(sizeof(struct product_s));
ptr->weight = 42;
```

Полное копирование: оператор = требует линейное время.

```
struct product_s a, b;
b = a;
```

Если мы копируем указатель, копируется только значение: новый указатель ссылается на тот же объект.

```
struct array_s {
    int* p;
    int n;
}
struct array_s a, b;
a.p = malloc(sizeof(int) * 100);
a.n = 100;
b = a;
a.p[0] = 42;
b.p[0] == a.p[0] ? printf(1) : printf(0); // equal values
```

Структура может быть размещена на стеке, в глобальной памяти, на куче.

## ii Связный список

Список — блок данных и указатель на следующий. Если список связный, то еще на предыдущий.

```
#include <stdlib.h>

struct Node {
    int data;
    struct Node *next, *prev;
};

struct List {
    struct Node *head;
    void new_node(int data);
```

```
    void delete_node(struct Node* n);
};

void List::new_node(int data) {
    struct Node* n = (struct Node*)malloc(sizeof(struct Node));
    n->next = this->head;
    n->data = data;
    n->prev = this->head->prev;
    this->head->prev->next = n;
    this->head = n;
}

void List::delete_node(struct Node* n) {
    n->prev->next = n->next;
    n->next->prev = n->prev;
    free(n);
}
```

### iii typedef

Не команда для препроцессора, это просто синоним для существующего типа.

```
typedef long long ll;
typedef struct point {
    //pass
} point_s
```

# Вопрос 5 Структуры. Интрузивный связный список на C

- интрузивная реализация
- typedef

## i Интрузивная реализация

Такой список хранит интрузивные вершины, внутри которых лежат вершины с данными. Внутренние блоки не имеют связи между собой. За счет этого мы можем создать один список и использовать его для разных типов.

```c
#include <stdlib.h>

struct IntrusiveNode {
    struct IntrusiveNode *next;
    struct IntrusiveNode *prev;
};

struct IntrusiveList {
    struct IntrusiveNode *head;
};

struct Node {
    int data;
    struct IntrusiveNode *node;
};

void add_intr_node(struct IntrusiveList *list, struct IntrusiveNode *new_node) {
    new_node->prev = list->head;
    new_node->next = list->head->next;
    new_node->next->prev = new_node;
    list->head->next = new_node;
}

void add_node(struct IntrusiveList *list, int data) {
    struct Node *new_node = malloc(sizeof(struct Node));
    struct IntrusiveNode *new_intr_node = malloc(sizeof(struct IntrusiveNode));
    new_node->data = data;
    new_node->node = new_intr_node;
    add_intr_node(list, new_intr_node);
}

void delete_node(struct IntrusiveList *list, struct Node *dnode) {
    struct IntrusiveNode *intr_node = dnode->node;
    intr_node->prev->next = intr_node->next;
    intr_node->next->prev = intr_node->prev;
    free(dnode); free(intr_node);
}
```

## ii typedef

Не команда для препроцессора, это просто синоним для существующего типа.

```c
typedef long long ll;
typedef struct point {
    //pass
} point_s
```

# Вопрос 6 Функции. Указатели на функции

- как происходит вызов функции
- реализация сортировки
- `void sort(void* base, size_t num, size_t size, int (*compar)(const void*,const void*));`

## i указатель на функцию

Указатель на функцию — тоже адрес в памяти, его можно передавать, например, чтобы вызвать в другой функции вызывать.

## ii как происходит вызов функции

- Сначала выделяется место на стеке для хранение возвращаемого значения, локальных переменных, параметров, которые она получает и адрес возврата.
- Далее параметры копируются на выделенные места и выполняется функция
- После выполнения сохраняется возвращаемое значение, вызываются деструкторы локальных объектов
- Программа возвращается по адресу возврата, туда, где была вызвана функция, ее место на стеке освобождается.

## iii Сортировка

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

void swap(void* left, void* right, size_t size) {
    void* tmp = malloc(size);
    memcpy(tmp, left, size);
    memcpy(left, right, size);
    memcpy(right, tmp, size);
    free(tmp);
}

void sort(void* base, size_t num, size_t size, int (*compar)(const void*, const void*)) {
    for (size_t i = 0; i < num; ++i)
        for (size_t j = i + 1; j < num; ++j)
            if (compar((char *)base + i * size, (char *)base + j * size) > 0)
                swap((char *)base + i * size, (char *)base + j * size, size);
}

int comparInt(const void* left, const void* right) {
    int a = *(int *)left;
    int b = *(int *)right;
    return a < b ? -1 : a > b ? 1 : 0;
}

int main()
{
    int array[10] = {5, 2, 8, 9, 4, 1, 7, 6, 3, 0};
    sort(array, 10, sizeof(int), comparInt);
    for (size_t i = 0; i < 10; ++i)
        printf("%d ", array[i]);
    return 0;
}
```

# Вопрос 7 Обзор стандартной библиотеки C

- string.h (memcpy, memcmp, strcpy, strcmp, strcat, strstr, strchr, strtok)
- stdlib.h (atoi, strtol, srand/rand, qsort)

## i string.h

**memcpy** Копирование в destination из source num байтов.

```
1 void* memcpy (void* destination, const void* source, size_t num);
```

**memcmp** Сравнение кусков памяти. Возвращаемое значение: 0, если одинаковы, положительное число, если в ptr1 встретился байт с большим unsigned int значением, отрицательное, иначе.

```
1 int memcpy (const void* ptr1, const void* ptr2, size_t num);
```

**strcpy** Копирование C-строки в destination из source.

```
1 char* memcpy (char* dstination, const char* source);
```

**strcmp** Сравнение строк: если первая длиннее, результат положительный.

```
1 int strcmp (const char* str1, const char* str2);
```

**strcat** Конкатенация строк: приклеивает source к destination справа.

```
1 char* strcat (char* destination, const char* source);
```

**strstr** Поиск подстроки в строке: возвращает указатель на место в первой строке или NULL;

```
1 char* strstr (const char* str1, const char* str2);
```

**strchr** Поиск символа в строке: возвращает указатель на место в первой строке или NULL;

```
1 char* strchr (const char* str1, int symbol);
```

**strtok** Разбить строку на токены по разделителям: внутри лежит статическая переменная, которая хранит место в строке (откуда начать при следующем вызове), строку разрушает, расставляя нули, чтобы printf "знал", где останавливаться.

```
1 char str[] = "This ,a simple string.";
2 char *pch = strtok(str, " ,");
3 while (pch != NULL) {
4     printf("\%s\n", pch);
5     pch = strtok(NULL, " ,.-");
6 }
```

## ii stdlib.h

**atoi** Переводит строку в число int. Пробелы пропускаются, знак перед числом учитывается. При ошибке undefined behavior.

```
int atoi(const char* str);
int res = atoi("-1");
```

**strtol** Переводит строку в число long long int. Если успешно, то возвращает число, если строка неправильная — вернет 0, если переполнение, то вернет LLONG_MAX($2^{63} - 1$) или LONG_MIN ($-2^{63}$). endPtr — первый символ, на котором сломалось.

```
long long int strtol(char* buffer, char** endPtr, int base);
char *end; char *ptr = "25a";
int N = strtol(ptr, &end, 10);
if (ptr == end) {}
```

**srand/rand** Генерация рандомных чисел.

```
void srand(unsigned int seed);
srand(time(NULL)); // time.h
int r1 = rand();
srand(time(NULL));
```

**qsort** Быстрая сортировка. base — массив, num — количество элементов, size — размер элемента, compar — указатель на функцию сравнения.

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