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*));
```

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`

i FILE, fopen, fclose, r/w, t/b

`FILE` — структура, описывающая абстракцию для ввода-вывода (файл на диске, клавиатура, экран). Определяет поток, содержит информацию для управления потоком (указатель на буфер, его показатели состояния). Внутри:

- Дескриптор — идентификатор (целое число) файла внутри ОС
- Промежуточный буфер — сначала накапливаем буферъ потом за один вызов записываем на диск (так быстрее, чем писать отдельно)
- Текущее положение в файле
- Индикатор ошибки при последней операции
- Индикатор конца файла

Для работы с этими полями используется `stdio`.

Интерпретация байтов в памяти

- Текстовый формат
  - Интерпретируется как последовательность символов: 100 = 3 символа '1', '0', '0'
  - Есть спецсимволы: перевод строки, табуляция.
  - Проблемы: разные кодировки (например перевод строки)
  - Простая интерпретация, но большой размер
- Бинарный формат
  - Для обработки нужно описание, сложный формат (bmp, wav, elf)
  - Заголовок записывается: первыми четырьмя — ширина, вторыми четырьмя — высота, остальные данные тремя байтами RGB с выравниванием.
  - Сложная интерпретация, но компактнее.

fopen

```c
FILE *fopen(const char* fname, const char* mode);

FILE *f = fopen("in.txt", mode);
if ( f == NULL ) {
    // файл не открылся
}
fclose(f);
```

Если файл не открылся, возвращает нулевой указатель.

```c
FILE* fin = fopen("in.txt", "r");
FILE* fout = fopen("out.txt", "w");
```

fclose Закрывает файл и разъединяет filestream, связанный с потоком. Все буферы записываются и сбрасываются. Если закрыто успешно, возвращает нуль, иначе EOF.

```c
int fclose(FILE *filestream);
```

ii stdin, stdout, stderr

- Поток 0 (stdin) — для чтения команд пользователя и входных данных
- Поток 1 (stdout) — для вывода данных на экран
- Поток 2 (stderr) — для вывода на экран, но без буфера.

stdio.h

```
FILE *f0 = fopen("in.txt", "..."); // файл на диске
FILE *f1 = stdin; // чтение с клавиатуры
FILE *f2 = stdout; // вывод на экран
```

iii printf, scanf, fprintf, fscanf, sprintf, sscanf, fgets

printf Запись отформатированной строки в stdout. Возвращает количество записанных символов. В случае ошибки возвращает отрицательное число.

```
int printf(const char* format, ... );
```

scanf Считывание данных из потока stdin. Возвращает количество успешно считанных символов. В случае ошибки возвращает EOF.

```
int scanf(const char* format, ... );
```

fprintf Форматированный вывод в файл.

```
int fprintf(FILE *stream, const char* format, ... );
```

fscanf Форматированное чтение из файла.

```
int fscanf(FILE *fp, const char* format, ... );
```

sprintf Вывод в массив, указанный buf

```
int sprintf(char* buf, const char* format, args );
```

sscanf Ввод из массива buf

```
int sscanf(char* buf, const char* format, args );
```

iv Обработка ошибок, feof, ferror

```
while (!feof(fin)) {
    fread(...);
    if (ferror(fin)) {
        // сделать что-то
    }
}
```

feof Проверка достижения конца файла, связанного с потоком через параметр fstream. Если файл достиг конца, возвращается не нуль.

```
int feof(FILE* fstream);
```

ferror Отслеживание появления ошибки, связанной с потоком через fstream. Если произошла ошибка, возвращается не нуль. Так как fread сначала перейдет чтением “за конец файла”, а потом установит индикатор. Поэтому правильно выполнять эту операцию после feof, чтобы проинформировать о возникшей ошибке.

```
1 int ferror(FILE* fstream);
```

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

i FILE, fopen, fclose, r/w, t/b, буферизация

См. i.

ii fread, fwrite, fseek, ftell, fflush

```
FILE* fin = fopen("in.txt", "rb");
FILE* fout = fopen("out.txt", "w");
int array[100];
fread(array, sizeof(int), 100, fin);
fwrite(array, sizeof(int), 50, fout);
fclose(fout);
fclose(fin);
```

fread Считывает массив размером count, каждый из которых имеет размер size байт, из потока. Сохраняет в блоке памяти, на который указывает ptrvoid. Индикатор положения увеличивается на количество записанных байтов. Возвращает количество успешно считанных элементов.

```
size_t fread(void* ptrvoid, size_t size, size_t count, FILE* fstream);
```

fwrite Записывает массив размером count элементов, каждый из которых имеет размер size байт, в блок памяти, на который указывает ptrvoid.

```
size_t fwrite(const coid* ptrvoid, size_t size, size_t count, FILE* fstream);
```

fseek Перемещает указатель позиции в потоке, добавляя смещение offset к положению origin.

SEEK_SET начало файла
SEEK_CUR текущее положение файла
SEEK_END конец файла

```
size_t fseek(FILE* fstream, long int offset, int origin);
```

ftell Возвращает текущую позицию потока. Для бинарных — количество байт от начала файла.

```
long int ftell(FILE* fstream);
```

fflush Любые не записанные данные в выходном буфере записываются в файл. Если аргумент равен нулевому указателю, все открытые потоки записываются. Если операция успешна, возвращает нуль.

```
int fflush(FILE* fstream);

int main() {
    FILE* fout = fopen("in", "w");
    int a = 3, b = 5;
```

```
6       fprintf(fout, "\%d \%d", a, b); // может так и остаться в буфере, если malloc сломается
7       fflush(fout); // поэтому можем принудительно записать
8       int *array = malloc(100000000);
9       if (array == NULL)
10          return -1
11      fclose(fout);
12      return 0;
13  }
```

iii FILE, fopen, fclose, r/w, t/b, буферизация

См. iv.

Вопрос 10 Классы и объекты

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

i инкапсуляция: private/public

Три основные идеи ООП: инкапсуляция, наследование и полиморфизм. Инкапсуляция позволяет работать с кодом, классом, как с черным ящиком. При работе без этого программисту нужно не забыть создать, не выйти за границы, не перепутать размер, не забыть удалить. Если за это отвечает некоторый интерфейс, можно не думать о его реализации. Если нужно поменять класс, можно унаследовать новый класс от старого и добавить/удалить что-то.

В классе есть три модификатора доступа (private/public/protected). Публичные функции доступны всем, изменять публичные переменные тоже может кто-угодно. Приватные поля могут использовать только представители класса.

Проверка на соответствие модификатора и использования происходит во время компиляции.

ii конструктор (overloading), деструктор

перегрузка Можно создать несколько функций с одним именем, но разными параметрами, при этом линкер их сможет различить: во время компиляции им присваиваются новые имена (name mangling). Так можно, например, создать одному классу несколько конструкторов.

конструктор Конструкторы нужны, чтобы инцилизировать поля. Так же может определятся конструктор по умолчанию (default constructor).

```
class Array {
    Array(int s) { size = s; data = new int[size]; }
    Array() { size = 100; data = new int[size]; }
}

int main() {
    Array a; // вызов default конструктора
    Array b(100);
}
```

По умолчанию инициализируются пустые конструктор и деструктор.

iii деструктор

Вызывается автоматически, когда объект выходит из области видимости.

iv инициализация полей

Все поля инициализорованы чем-то до конструктора. Инициализация может происходить через двоеточие после конструктора. Такой метод позволяет избежать коллизии имен и проинициализировать ссылки.

```
Figure::Figure (int id, int x, int y) : id(id), x(x), y(y);
```

Вопрос 11 Работа с кучей на C++

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

i new/delete

Это аналоги malloc и free. Но new и delete — операторы, их транслирует компилятор, а не линкуется из библиотеки. Нельзя смешивать две пары, так как они могут иметь внутренние различия.

```
int* pi = new int[5]; // создаем массив
my_class* a = new my_class;
delete [] a;
delete my_class;
```

ii Создание объектов в куче

new и delete вызывают конструкторы и деструкторы классов соответственно. При создании массива через new не стоит удалять без [], может вызвать undefined behavior. Если вызываем delete [], он проходит по массиву и вызывает деструктор у каждого элемента.

iii Конструктор копий

Используется, когда нужен оператор присваивания. Есть один объект нашего классы, мы хотим проинициализировать новый с такими же значениями полей.

```
class Array {
    Array (const Array& a) {
        size = a.size;
        data = new int[size];
        // если просто скопировать объекты через знак равно,
        //будет побайтовое копирование и будут ссылаться на одну память
        for (size_t i = 0; i < size; ++i)
            data[i] = a.data[i];
    }
}

Array a = b;
f(a); // здесь тоже копирование
```

В конструктор копий передается именно ссылка, так как иначе потребуется скопировать объект, то есть возникнет рекурсивный бесконечный вызов.

iv Оператор присваивания

Хотим создать два объекта, а потом присвоить первому второй. Для этого можно определить оператор “=”.

```
Array& Array::operator=(const Array& a) {
    if (&a == this) // для a = a;
        return *this;
    delete [] data;
    size = a.size;
```

```
    data - new int[size];
    for (size_t i = 0; i < size; ++i)
        data[i] = a.data[i];
    return *this; // для a = b = c;
}
```

Еще можно сделать это же с помощью swap:

```
Array& Array::operator=(Array a) { //copy-swap
    std::swap(_data, a._data);
    return *this;
}
```

Вопрос 12 Наследование и полиморфизм

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

→ Наследование — создание класса на основе другого, при этом производный класс будет иметь те же методы, но возможно добавлять новые поля, методы, переопределять или дописывать старые.

→ Полиморфизм — возможность сделать объект базового типа объектом производных типов, в также возможность переписать/дописать методы базового в производном.

i protected

`protected` позволяет видеть содержимое не только представителям класса, но и всем классам-наследникам. Это позволяет использовать поля и методы родителя в классах наследниках, которые при этом не должны быть публичными.

ii virtual (overriding)

Виртуальные функции нужны для того, чтобы эти функции могли быть там перекрыты (переписаны) у детей. Это требуется для того, чтобы программа понимала какую функцию нужно вызывать при работе. Конструктор не может быть виртуальным, но деструктор может. Если виртуальная функция объявлена с присвоением нуля, она обязательно должна будет перекрыта в дочернем классе, а в этом реализации нет.

list.h

```
class List {
protected:
    int val;
    List *next;
public:
    List(int val);
    ~List();
    virtual void push_back(int val);
    size_t length() const;
};
```

doubleList.h

```
#include "list.h"
class DoubleList: public List {
private:
    DoubleList* prev;
public:
    DoubleList(int val);
    ~DoubleList();
    void push_back(int val);
    void pop_back(int val); // new method
};
```

list.cpp

```cpp
#include "list.h"
List::List(int val) {
    this->val = val;
    this->next = NULL;
}
~List::List() { delete this->next; }

void List::push_back(int val) {
    List* cur = this;
    while (cur->next != NULL)
        cur = cur->next;
    cur->next = new List(val);
}

size_t List::length() const {
    size_t count = 0;
    const List* cur = this;
    while (cur->next != NULL) {
        cur = cur->next;
        count++;
    }
    return count;
}
```

doubleList.cpp

```cpp
#include "doubleList.h"
DoubleList::DoubleList(int val): List(cal) {
    this->prev = NULL;
}
~DoubleList::DoubleList() {}

void DoubleList::push_back(int val) {
    DoubleList* cur = this;
    while (cur->next != NULL)
        cur = (DoubleList* )cur->next;
    cur->next = new DoubleList(val);
    ((DoubleList* )cur->next)->prev = cur;
}
void DoubleList::pop_back() {
    DoubleList* cur = this;
    while (cur->next != NULL)
        cur = cur->next;
    this->prev = NULL;
    delete this;
}
```

iii таблица виртуальных функций

Все виртуальные методы класса автоматически записываются в таблицу, которая идет перед классом. При вызове виртуальной функции программа смотрит в таблицу, находит адрес нужной виртуальной функции, и вызывает ее.

Каждый объект с виртуальными функциями хранит скрытое поле `vptr` на начало этой таблицы.

iv статическое/динамическое связывание

Связывание — сопоставление имени функции и ее адреса в памяти.

- Статическое связывание — имя можно заменить на адрес в этапе построения.

```cpp
Sale s(...);
int r = s.getSalary();
```

- Динамическое связывание — имя можно заменить на адрес только на этапе выполнения.

```cpp
Employee *e;
int t, salary;
scanf("\%d \%d", &t, &salary);
if (d = 0)
    e = new Developer(salary);
else
    e = new SaleManager(salary);
printf("\%d", w->getSalary());
```

По умолчанию в C++ используется статическое связывание, но если у метода есть ключевое слово `virtual`, то для него используется динамическое.

Вопрос 13 Умные указатели

- scoped_ptr
- unique_ptr
- shared_ptr

i scored_ptr

Хотим сделать обертку над указателем, чтобы не думать о памяти (об ее освобождении), при выходе из области видимости указатель сам должен очистить.

scoped_ptr.h

```
class scoped_ptr {
private:
    Person* myPointer;
public:
    scoped_ptr(Person* ptr);
    ~scoped_ptr();

    Person* ptr();
    Person& operator *() const;
    Person* operator ->() const;
    bool isNull() const;
private:
    scoped_ptr(const scoped_ptr& p);
    const scoped_ptr&
         operator=(const scoped_ptr& p);
}
```

scoped_ptr.cpp

```
scoped_ptr::scoped_ptr(Person* ptr) {
    myPointer = ptr;
}
scoped_ptr::~scoped_ptr() {
    if (myPointer != 0)
        delete myPointer;
}
Person& scoped_ptr::operator *() const {
    return *myPointer;
}
Person* scoped_ptr::operator ->() const {
    return myPointer;
}
bool scoped_ptr::isNull() const {
    return myPointer == 0;
}
```

Использование:

```
scoped_ptr p = new Person("Bob");
p->getName();
*p.getName(); // не скомпилируется,
// сначала . потом *
(*p).getName();
```

Но есть проблемы:

```
scoped_ptr p1 = new Person("Bob");
scoped_ptr p2 = new Person("Alice");
scoped_ptr p3 = p1; // дважды деструкор в конце
PrintPerson(p1); // аналогично
p1 = p2; // утечка из p2 и проблема с delete
```

ii unique_ptr

Можно перемещать и один раз передать в функцию, после этого будет NULL.

unique_ptr.h

```
class unique_ptr {
private:
    Person* p;
public:
    unique_ptr(Person* ptr);
    ~unique_ptr();

    Person* ptr();
    Person& operator *() const;
    Person* operator ->() const;
    bool isNull() const;
private:
    unique_ptr(unique_ptr& p);
}
    unique_ptr& operator=(unique_ptr& p);
```

unique_ptr.cpp

```
unique_ptr::unique_ptr(Person* ptr) {
    myPointer = ptr;
}
unique_ptr::~unique_ptr() {
    if (myPointer != 0)
        delete myPointer;
}
Person& unique_ptr::operator *() const {
    return *myPointer;
}
Person* unique_ptr::operator ->() const {
    return myPointer;
}
bool unique_ptr::isNull() const {
    return myPointer == 0;
}
unique_ptr::unique_ptr(unique_ptr& p) {
    myPointer = p.myPointer;
    p.myPointer = 0;
}
unique_ptr& unique_ptr::operator =
        (unique_ptr& p) {
    if (this != &p) {
        if (myPointer != )
            delete myPointer;
        myPointer = p.myPointer;
        p.myPointer = 0;
    }
    return *this;
}
```

iii shared_ptr

- Создаем вспомогательный объект Storage, где есть счетчик ссылок и указатель на Object.

```
shared_ptr p1 = new Object;
```

- Создадим второй указатель и присвоим ему значение первого. Нужно увеличить счетчик на 1.

```
shared_ptr p2 = p1;
```

- Теперь область видимости заканчивается, вызывается деструктор p2. Уменьшим счетчик на 1, так как он не 0, ничего более не делаем.
- Вызывается деструктор p1, уменьшаем счетчик еще на 1, получаем нуль, поэтому вызываем деструктор для Object, а потом и для Storage.

Вопрос 14 Перегрузка операторов

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

i бинарные и унарные опрераторы

Унарные операторы требуют только один объект. С помощью перегрузки операторов можно определить короткую запись некоторых операций для своего класса. Операторы "." и "a ? b : c" перегружать нельзя.

```
class BigInt {
    char operator [](size_t i) const; // для print(const BigInt&);
    char& operator [](size_t i);  // для BigInt a(239); a[3] = 5;
    size_t size_;
    char* digits_;
    BigInt(const BigInt& num) {
        size_ = num.size_;
        digits_ = num.digits_;
    }

    void swap(BigInt& b) {
        std::swap(size_, b.size_);
        std::swap(digits_, b.digits_);
    }
    BigInt& operator=(const BigInt& num) {
        if (this != &num) {
            BigInt tmp(num);
            tmp.swap(*this);
        }
        return *this;
    }
    BigInt& operator++() { // prefix
        ...
        return *this;
    }
    BigInt& operator++(int) { // postfix
        BigInt t(*this);
        ++(*this);
        return t;
};
// достаточно реализовать только эти операторы сравнения
bool operator <(BigInt const & a , BigInt const & b ) { ... }
bool operator ==(BigInt const & a , BigInt const & b) { ... }
```

Унарные операторы лучше реализовывать внутри класса. Бинарные и операторы сравнения снаружи.

ii в классе/вне класса

Операторы могут быть переопределены как внутри класса, так и вне (это будет функцией от одной/двух переменных и не будет методом класс). Это полезно, когда нет доступа к классу.

```
// outside class
BigInt operator+(const BigInt a, const BigInt& b) {
    a += b;
    return a;
}
// inside class: `this` == `a`
```

```
BigInt BigInt::operator+(const BigInt& b) {
    (*this) += b;
    return *this;
}
```

iii приведение типов

- int к BigInt через конструктор

```
class BigInt {
    BigInt(int a) { .. };
};
BigInt a = 3;
BigInt a = (BigInt)3;
```

- Это не всегда удобно и понятно. `Matrix m = 3;` Можно запретить использование конструктора для приведения типов.

```
class Matrix {
    explicit Matrix(size_t a) { .. }
};
```

- BigInt к int

```
class BigInt {
    operator int() const {
        return ...;
    }
};
BigInt a(23919); int b = a;
```

Вопрос 15 Ключевые слова extern, static, inline

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

i extern

Если есть два файла, в первом создается переменная `int a = 0;`, во втором в функции мы пытаемся `a++;`. Выдаст ошибку, так как не знает, что это за переменная. Можно создать заголовочный файл, записать туда эту переменную. Либо в во втором файле написать `extern int a;`. Компилятор не будет создавать переменную, а только будет знать, что такая есть.

ii static у переменных и функций

1. Глобальные static переменные
   Будут видны только в своем файле, так как обрабатывается на внутренней стадии линковки: для каждого модуля создается таблица с адресами переменных.
2. Локальные static переменные
   Будет сохранять свое значение между вызовами функции (хранится в области глобальных переменных).

```
1 void func() {
2     static int num_func_calls = 0;
3     printf("\%d" num_func_calls++)
4 }
```

3. static функции
   По умолчанию у всех функций есть ключевое слово extern. Если хотим запретить функции вызов в другом файле, нужно добавить static.
   Может использоваться при написании библиотеки, если мы не хотим пользователю дать пользоваться какой-то функцией.
   Или чтобы не перекрывались имена у разных программистов, пишущих одну программу.

iii static у полей и методов

1. static у полей
   Это глобальная переменная для всех объектов класса, при этом область видимости будет только внутри него.

person.h

```
1 class Person {
2 private:
3     static int last_id;
4     int id;
5     char name[256];
6     int age;
7 public:
8     Person(const char* name, int age);
9 };
```

person.cpp

```
1  int Person::last_id = 0;
2  Person::Person(const char* name, int age) :
3          name(name), age(age) {
4      id = last_id++;
5  }
6
7  int main() {
8      Person p1("Bob", 12);
9      Person p2("Alice", 11);
10 }
```

2. static у методов
   У него нет this, следовательно, не может получить доступ к нестатическим полям, может быть вызван через имя класса.

```
class Person {
    ...
    static int get_next_id() {
        return last_id;
    }
}
int main() {
    int i = Person::get_index_id();
}
```

```
class Point {
    int x, y;
    static int distance(const Point& p1, const Point& p2);
    Point(int x, int y) : x(x), y(y) {}
};
int main() {
    Point p1(1, 3); Point p2(2, 3);
    Point::distance(p1, p2);
```

iv inline у функций

Сообщает оптимизатору, чтобы по возможности он вставил код этой функции в место, где она вызывается. Тем самым экономиться время работы и память, которые были бы потрачены на вызов функции.

```
int max(int a, int b) {
    return a > b : a ? b;
}
int main() {
    int c = 10;
    int b = 20;
    int d = max(c, b);
}
```

После оптимизации:

```
int main() {
    int c = 10;
    int b = 20;
    int d = a > b ? a : b;
}
```

Но это не гарантировано: компилятор сам решает, стоит ли это делать и сможет ли он. Кроме этого, обязательно знать и тело функции тоже. Если оно будет в другом месте, компилятор не поймет, что делать.

Все функции внутри header являются inline. Если хотим этого избежать — пишем определение отдельно.