

Билеты к экзамену C++  
МКН, Современное программирование  
семестр I

Тамарин Вячеслав

March 27, 2020

# Contents

## Вопрос 1 Программа, состоящая из нескольких файлов

- компиляция и линковка
- заголовочные файлы
- утилита make

### i компиляция и линковка

```
1 int main() {  
2     return 0;  
3 }
```

Причины разбиения на файлы:

1. Абстракция
2. Несколько программистов
3. Быстродействие

Пусть в программе будет несколько файлов:

```
main.c  
1 int main() {  
2     hello();  
3     return 0;  
4 }
```

```
hello.c  
1 void hello() {  
2     printf("Hello!");  
3 }
```

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 между 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  
1 void hello(int n);
```

hello.c

```
1 #include "hello.h"
2 void hello(int n) {
3     printf("%d", n);
4 }
```

main.c

```
1 #include "hello.h"
2 int main() {
3     hello();
4     return 0;
5 }
```

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

Если мы хотим ссылаться в цикле

a.c

```
1 #ifndef _a_H_
2 #define _a_H_
3 #include "b.h"
4 #endif
```

b.c

```
1 #ifndef _b_H_
2 #define _b_H_
3 #include "a.h"
4 #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

```
1 main: main.o str.o util.o
2     gcc main.o str.o util.o -o main
3 main.o: main.cpp util.h str.h
4     gcc -c main.cpp
5 str.o: str.cpp str.h
6     gcc -c str.cpp
7 util.o: util.cpp util.h
8     gcc -c util.cpp
9
10 clean:
11     rm -rf *.o
```

## Вопрос 2 Указатели, массивы, ссылки

- применение указателей и ссылок
- арифметика указателей

### i Массивы

```
1 int array[10]; // size 10*sizeof(int)
2
3 a[0] = 10;
4 int array[5] = {0, 1, 2, 3, 4};
5 int array[] = {0, 1, 2, 3, 4};
6 int array[5] = {0};
```

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

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

Двухмерные массивы

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

### ii Указатели

Это адрес ячейки:

```
1 int *p; // pointer
2 int a, b;
3 p = &a; // address of a
4 b = *p; // value by address p
5
6 printf("%p", p); // output of address
```

Теперь a равно b;

Адрес массива — указатель на его первый элемент.

```
1 char array[10];
2 char *p;
3 p = array[0];
4 p = array;
```

### iii Арифметика указателей

```
1 int i[10];
2 char c[10];
3 int *pi = &i[0];
4 char *pc = &c[0];
5 pi+=1;
6 pc+=1;
```

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

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

Можно вычитать, но нельзя складывать.

#### iv Различия между указателями разных типов

На C компилируется, а на плюсах нет.

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

#### v Применение

Полезно передавать в функцию указатель на большой объект.

```
1 void strlen(char* ptr) {
2     char *p = ptr;
3     while (*p != '\0')
4         ++p;
5     return p - ptr;
6 }
```

Второй вариант будет быстрее:

```
1 while (p[i] != 1)
2     i++;
3
4 while (*p != 1)
5     p++;
```

```
1 void swap(int *pa, int *pb) {
2     int tmp = *pa;
3     *pa = *pb;
4     *pb = tmp;
5 }
6 int main() {
7     int a = 3; int b = 4;
8     swap(&a, &b);
9     return 0;
10 }
```

#### vi Ссылки

Новая форма записи последнего примера

```
1 void swap(int& pa, int& pb) {
2     int tmp = pa;
3     pa = pb;
4     pb = tmp;
5 }
6 int main() {
7     int a = 3; int b = 4;
8     swap(a, b);
9     return 0;
10 }
```

## 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
1 #ifndef _hello_H_
2 #define _hello_H_
3 extern int a;
4 #endif _hello_H_
```

```
main.c
1 #include "hello.h"
2 int a = 0;
3 int main() {
4     return 0;
5 }
```

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

### Static

```
1 void f() {
2     static int call_count = 0;
3     call_count++;
4 }
5
6 int main() {
7     f(); f(); f();
8 }
```

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:

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

*typedef* позволяет задавать новые типы:

```
1 typedef unsigned int size_t
```

```
1 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

```
1 int *p = (int*)malloc(10000*sizeof(int));  
2 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

```
1 int **m = (int**)malloc(N * sizeof(int*));  
2 for (int i = 0; i < N; ++i) {  
3     m[i] = (int *)malloc(N * sizeof(int));  
4 }  
5  
6 m[42][42] = 42;  
7  
8 for (int i = 0; i < N; ++i) {  
9     free(m[i]);  
10 }  
11 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.

```
1 struct product_s {
2     char label[256];
3     unsigned char weight;
4     unsigned int price;
5 };
6
7 scanf("%s %f %d", p.label, &(p.weight), &(p.price));
8
9 struct product_s array[100];
10 array[0].weight = 42;
11
12 struct product_s* ptr = malloc(sizeof(struct product_s));
13 ptr->weight = 42;
```

Полное копирование: оператор = требует линейное время.

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

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

```
1 struct array_s {
2     int* p;
3     int n;
4 }
5 struct array_s a, b;
6 a.p = malloc(sizeof(int) * 100);
7 a.n = 100;
8 b = a;
9 a.p[0] = 42;
10 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.

```
1 #include <stdlib.h>
2
3 struct Node {
4     int data;
5     struct Node *next, *prev;
6 };
7
8 struct List {
9     struct Node *head;
10    void new_node(int data);
```

```

11     void delete_node(struct Node* n);
12 };
13
14 void List::new_node(int data) {
15     struct Node* n = (struct Node*)malloc(sizeof(struct Node));
16     n->next = this->head;
17     n->data = data;
18     n->prev = this->head->prev;
19     this->head->prev->next = n;
20     this->head = n;
21 }
22
23 void List::delete_node(struct Node* n) {
24     n->prev->next = n->next;
25     n->next->prev = n->prev;
26     free(n);
27 }

```

### iii typedef

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

```

1 typedef long long ll;
2 typedef struct point {
3     //pass
4 } point_s

```

## Вопрос 5 Структуры. Интрузивный связный список на С

- 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.

```
1  #include <stdlib.h>
2
3  struct IntrusiveNode {
4      struct IntrusiveNode *next;
5      struct IntrusiveNode *prev;
6  };
7
8  struct IntrusiveList {
9      struct IntrusiveNode *head;
10 };
11
12 struct Node {
13     int data;
14     struct IntrusiveNode *node;
15 };
16
17 void add_intr_node(struct IntrusiveList *list, struct IntrusiveNode *new_node) {
18     new_node->prev = list->head;
19     new_node->next = list->head->next;
20     new_node->next->prev = new_node;
21     list->head->next = new_node;
22 }
23
24 void add_node(struct IntrusiveList *list, int data) {
25     struct Node *new_node = malloc(sizeof(struct Node));
26     struct IntrusiveNode *new_intr_node = malloc(sizeof(struct IntrusiveNode));
27     new_node->data = data;
28     new_node->node = new_intr_node;
29     add_intr_node(list, new_intr_node);
30 }
31
32 void delete_node(struct IntrusiveList *list, struct Node *dnode) {
33     struct IntrusiveNode *intr_node = dnode->node;
34     intr_node->prev->next = intr_node->next;
35     intr_node->next->prev = intr_node->prev;
36     free(dnode); free(intr_node);
37 }
```

### ii typedef

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

```
1 typedef long long ll;
2 typedef struct point {
3     //pass
4 } 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 хранения 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

```
1  #include <stdlib.h>
2  #include <string.h>
3
4  void swap(void* left, void* right, size_t size) {
5      void* tmp = malloc(size);
6      memcpy(tmp, left, size);
7      memcpy(left, right, size);
8      memcpy(right, tmp, size);
9      free(tmp);
10 }
11
12 void sort(void* base, size_t num, size_t size, int (*compar)(const void*, const void*)) {
13     for (size_t i = 0; i < num; ++i)
14         for (size_t j = i + 1; j < num; ++j)
15             if (compar((char *)base + i * size, (char *)base + j * size) > 0)
16                 swap((char *)base + i * size, (char *)base + j * size, size);
17 }
18
19 int comparInt(const void* left, const void* right) {
20     int a = *(int *)left;
21     int b = *(int *)right;
22     return a < b ? -1 : a > b ? 1 : 0;
23 }
24
25 int main()
26 {
27     int array[10] = {5, 2, 8, 9, 4, 1, 7, 6, 3, 0};
28     sort(array, 10, sizeof(int), comparInt);
29     for (size_t i = 0; i < 10; ++i)
30         printf("%d ", array[i]);
31     return 0;
32 }
```

## 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 memcmp (const void* ptr1, const void* ptr2, size_t num);
```

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

```
1 char* strcpy (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`.

```
1 int atoi(const char* str);
2 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(263 - 1)` или `LONG_MIN(-263)`. `endPtr` — первый символ, на котором сломалось.

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

**srand/rand** Генерация случайных чисел.

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

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

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

## Вопрос 8 Ввод-вывод на С. 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 буферь 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

```
1 FILE *fopen(const char* fname, const char* mode);
2
3 FILE *f = fopen("in.txt", mode);
4 if ( f == NULL ) {
5     // файл не открылся
6 }
7 fclose(f);
```

Если файл не открылся, возвращает нулевой указатель.

```
1 FILE* fin = fopen("in.txt", "r");
2 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.

```
1 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

```
1 FILE *f0 = fopen("in.txt", "..."); // файл на диске
2 FILE *f1 = stdin; // чтение с клавиатуры
3 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.

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

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

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

**fprintf** Форматированный вывод в файл.

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

**fscanf** Форматированное чтение из файла.

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

**sprintf** Вывод в массив, указанный buf

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

**sscanf** Ввод из массива buf

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

## iv Обработка ошибок, feof, ferror

```
1 while (!feof(fin)) {
2     fread(...);
3     if (ferror(fin)) {
4         // сделать что-то
5     }
6 }
```

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

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



**ferror** Отслеживание появления ошибки, связанной с потоком через `fstream`. Если произошла ошибка, возвращается не ноль. 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 С. 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. ??.

### ii fread, fwrite, fseek, ftell, fflush

```
1 FILE* fin = fopen("in.txt", "rb");
2 FILE* fout = fopen("out.txt", "w");
3 int array[100];
4 fread(array, sizeof(int), 100, fin);
5 fwrite(array, sizeof(int), 50, fout);
6 fclose(fout);
7 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.

```
1 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.

```
1 size_t fwrite(const void* 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

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

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

```
1 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 ноль.

```
1 int fflush(FILE* fstream);
2
3 int main() {
4     FILE* fout = fopen("in", "w");
5     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(1000000000);
9   if (array == NULL)
10      return -1
11   fclose(fout);
12   return 0;
13 }

```

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

См. ??.

## 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 инициализировать поля. Так же может определяться конструктор по умолчанию (default constructor).

```
1 class Array {
2     Array(int s) { size = s; data = new int[size]; }
3     Array() { size = 100; data = new int[size]; }
4 }
5
6 int main() {
7     Array a; // вызов default конструктора
8     Array b(100);
9 }
```

По умолчанию инициализируются пустые конструктор и деструктор.

### iii деструктор

Вызывается автоматически, когда объект выходит из области видимости.

### iv инициализация полей

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

```
1 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.

```
1 int* pi = new int[5]; // создаем массив
2 my_class* a = new my_class;
3 delete [] a;
4 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 класса, мы хотим проинициализировать новый с такими же значениями полей.

```
1 class Array {
2     Array (const Array& a) {
3         size = a.size;
4         data = new int[size];
5         // если просто скопировать объекты через знак равно,
6         //будет побайтовое копирование и будут ссылаться на одну память
7         for (size_t i = 0; i < size; ++i)
8             data[i] = a.data[i];
9     }
10 }
11
12 Array a = b;
13 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 “=”.

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

```
6     data = new int[size];
7     for (size_t i = 0; i < size; ++i)
8         data[i] = a.data[i];
9     return *this; // для a = b = c;
10 }
```

Еще можно сделать это же с помощью swap:

```
1 Array& Array::operator=(Array a) { //copy-swap
2     std::swap(_data, a._data);
3     return *this;
4 }
```

## 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
1 class List {
2     protected:
3         int val;
4         List *next;
5     public:
6         List(int val);
7         ~List();
8         virtual void push_back(int val);
9         size_t length() const;
10 };
```

```
doubleList.h
1 #include "list.h"
2 class DoubleList: public List {
3     private:
4         DoubleList* prev;
5     public:
6         DoubleList(int val);
7         ~DoubleList();
8         void push_back(int val);
9         void pop_back(int val); // new method
10 };
```

```

1  list.cpp
2  #include "list.h"
3  List::List(int val) {
4      this->val = val;
5      this->next = NULL;
6  }
7  ~List::List() { delete this->next; }
8
9  void List::push_back(int val) {
10     List* cur = this;
11     while (cur->next != NULL)
12         cur = cur->next;
13     cur->next = new List(val);
14 }
15
16 size_t List::length() const {
17     size_t count = 0;
18     const List* cur = this;
19     while (cur->next != NULL) {
20         cur = cur->next;
21         count++;
22     }
23     return count;
24 }

```

```

1  doubleList.cpp
2  #include "doubleList.h"
3  DoubleList::DoubleList(int val): List(val) {
4      this->prev = NULL;
5  }
6  ~DoubleList::DoubleList() {}
7
8  void DoubleList::push_back(int val) {
9      DoubleList* cur = this;
10     while (cur->next != NULL)
11         cur = (DoubleList*)cur->next;
12     cur->next = new DoubleList(val);
13     ((DoubleList*)cur->next)->prev = cur;
14 }
15
16 void DoubleList::pop_back() {
17     DoubleList* cur = this;
18     while (cur->next != NULL)
19         cur = cur->next;
20     this->prev = NULL;
21     delete this;
22 }

```

### 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 `vp_ptr` 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.

```

1  Sale s(...);
2  int r = s.getSalary();

```

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

```

1  Employee *e;
2  int t, salary;
3  scanf("%d %d", &t, &salary);
4  if (d = 0)
5      e = new Developer(salary);
6  else
7      e = new SaleManager(salary);
8  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 `scoped_ptr`

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

```
1  _____ scoped_ptr.h _____
2  class scoped_ptr {
3  private:
4      Person* myPointer;
5  public:
6      scoped_ptr(Person* ptr);
7      ~scoped_ptr();
8
9      Person* ptr();
10     Person& operator *() const;
11     Person* operator ->() const;
12     bool isNull() const;
13 private:
14     scoped_ptr(const scoped_ptr& p);
15     const scoped_ptr&
16         operator=(const scoped_ptr& p);
17 }
```

Использование:

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

```
1  _____ scoped_ptr.cpp _____
2  scoped_ptr::scoped_ptr(Person* ptr) {
3      myPointer = ptr;
4  }
5  scoped_ptr::~~scoped_ptr() {
6      if (myPointer != 0)
7          delete myPointer;
8  }
9  Person& scoped_ptr::operator *() const {
10     return *myPointer;
11 }
12 Person* scoped_ptr::operator ->() const {
13     return myPointer;
14 }
15 bool scoped_ptr::isNull() const {
16     return myPointer == 0;
17 }
```

Но есть проблемы:

```
1  scoped_ptr p1 = new Person("Bob");
2  scoped_ptr p2 = new Person("Alice");
3  scoped_ptr p3 = p1; // дважды деструктор в конце
4  PrintPerson(p1); // аналогично
5  p1 = p2; // утечка из p2 и проблема с delete
```

### ii `unique_ptr`

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

```

1 unique_ptr.h
2 class unique_ptr {
3     private:
4         Person* p;
5     public:
6         unique_ptr(Person* ptr);
7         ~unique_ptr();
8
9         Person* ptr();
10        Person& operator *() const;
11        Person* operator ->() const;
12        bool isNull() const;
13    private:
14        unique_ptr(unique_ptr& p);
15    }
16    unique_ptr& operator=(unique_ptr& p);

```

```

1 unique_ptr.cpp
2 unique_ptr::unique_ptr(Person* ptr) {
3     myPointer = ptr;
4 }
5 unique_ptr::~unique_ptr() {
6     if (myPointer != 0)
7         delete myPointer;
8 }
9 Person& unique_ptr::operator *() const {
10    return *myPointer;
11 }
12 Person* unique_ptr::operator ->() const {
13    return myPointer;
14 }
15 bool unique_ptr::isNull() const {
16    return myPointer == 0;
17 }
18 unique_ptr::unique_ptr(unique_ptr& p) {
19    myPointer = p.myPointer;
20    p.myPointer = 0;
21 }
22 unique_ptr& unique_ptr::operator =
23    (unique_ptr& p) {
24    if (this != &p) {
25        if (myPointer != 0)
26            delete myPointer;
27        myPointer = p.myPointer;
28        p.myPointer = 0;
29    }
30    return *this;
31 }

```

### iii shared\_ptr

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

```

1 shared_ptr p1 = new Object;

```

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

```

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 класса
- приведение типов

### i бинарные и унарные операторы

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.

```
1 class BigInt {
2     char operator [] (size_t i) const; // для print(const BigInt&);
3     char& operator [] (size_t i); // для BigInt a(239); a[3] = 5;
4     size_t size_;
5     char* digits_;
6     BigInt(const BigInt& num) {
7         size_ = num.size_;
8         digits_ = num.digits_;
9     }
10
11     void swap(BigInt& b) {
12         std::swap(size_, b.size_);
13         std::swap(digits_, b.digits_);
14     }
15     BigInt& operator=(const BigInt& num) {
16         if (this != &num) {
17             BigInt tmp(num);
18             tmp.swap(*this);
19         }
20         return *this;
21     }
22     BigInt& operator++() { // prefix
23         ...
24         return *this;
25     }
26     BigInt& operator++(int) { // postfix
27         BigInt t(*this);
28         ++(*this);
29         return t;
30 };
31 // достаточно реализовать только эти операторы сравнения
32 bool operator <(BigInt const & a , BigInt const & b ) { ... }
33 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.

```
1 // outside class
2 BigInt operator+(const BigInt a, const BigInt& b) {
3     a += b;
4     return a;
5 }
6 // inside class: `this` == `a`
```

```

7 BigInt BigInt::operator+(const BigInt& b) {
8     (*this) += b;
9     return *this;
10 }

```

### iii приведение типов

- int к BigInt через конструктор

```

1 class BigInt {
2     BigInt(int a) { .. };
3 };
4 BigInt a = 3;
5 BigInt a = (BigInt)3;

```

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

```

1 class Matrix {
2     explicit Matrix(size_t a) { .. }
3 };

```

- BigInt к int

```

1 class BigInt {
2     operator int() const {
3         return ...;
4     }
5 };
6 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.

```
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 };
```

```
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.

```

1 class Person {
2     ...
3     static int get_next_id() {
4         return last_id;
5     }
6 }
7 int main() {
8     int i = Person::get_index_id();
9 }

```

```

1 class Point {
2     int x, y;
3     static int distance(const Point& p1, const Point& p2);
4     Point(int x, int y) : x(x), y(y) {}
5 };
6 int main() {
7     Point p1(1, 3); Point p2(2, 3);
8     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 экономится время работы и память, которые были бы потрачены на вызов функции.

```

1 int max(int a, int b) {
2     return a > b ? a : b;
3 }
4 int main() {
5     int c = 10;
6     int b = 20;
7     int d = max(c, b);
8 }

```

После оптимизации:

```

1 int main() {
2     int c = 10;
3     int b = 20;
4     int d = a > b ? a : b;
5 }

```

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.

## Вопрос 16 Разное

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

### i ключевое слово `const`

#### в C

Назначение:

1. избавить себя от описок
2. сделать код более понятным для читающего

```
1 const double pi = 3.14159;
2 char const * ptr1; // pointer to const
3 char * const ptr1; // const pointer
4
5 size_t strlen(const char *s);
6
7 const Array operator=(const Array a) {...}; //значение только rvalue
8
9 void f(const char* str) {
10     char* s1 = (char*)str;
11     s1[0] = 'x';
12 }
```

```
1 char s1[] = "Hello, world!";
2 const char* s2 = "Hello, world";
3 f(s1);
4 f(s2); // RT error
```

`s2` — строковая константа:

- храниться в отдельном блоке памяти рядом с глобальными;
- память, выделенная под такую константу, может быть помечена как `read-only`

#### в C++

```
1 class Array {
2 private:
3     const size_t size;
4     int* data;
5 public:
6     Array(size_t s);
7     Array(const Array& a); // а не будет изменен
8     Array& operation=(const Array& a);
9     ~Array();
10
11     int get(int i) const;
12     void get(int i, int v);
13     size_t get_size() const; // защищает от изменения сам объект
14 }
```

```
1 void print(const Array& a) {
2     for (size_t i = 0; i < a.get_size(); ++i)
3         printf("%d", a.get(i));
4 }
```

## ii перегрузка функций

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.

## iii параметры функций по умолчанию

```
player.h
1 class Player {
2 private:
3     double life;
4     int damage;
5 public:
6     Player(double life = 100.0, int damage = 10);
7     // Player(double life, int damage = 10); ok
8     // Player(double life = 100.0, int damage); error
9 }
```

```
main.cpp
1 #include "player.h"
2 int main() {
3     Player p1(2, 3);
4     Player p2(2);
5     Player p2();
6 }
```

Параметры по умолчанию должны быть указаны в header, так как их необходимо знать при компиляции *main.cpp*, а другие файлы в этот момент не используются.



## Вопрос 17 Наследование: детали

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

### i сортировка и структуры данных C vs ООП

#### сортировка в стиле C

```
1 struct point_s {
2     int x, y;
3 };
4
5 void qsort (void* base, size_t num, size_t size,
6             int (*compar)(const void*, const void*)) {}
7
8 int cmp_point1 (const void* v1, const void* v2) {
9     // здесь нет никакой проверки типов со стороны компилятора
10    const struct point_s *p1 = (const struct point_s*) v1;
11    const struct point_s *p2 = (const struct point_s*) v2;
12    ...
13 }
14
15 struct point_s points[10];
16 qsort(points, 10, sizeof(points[0]), cmp_point1);
```

#### сортировка в стиле ООП

```
1 class Comparable {
2     virtual bool operator<(const Comparable* v)=0 const;
3 };
4 void nsort(Comparable** m, size_t size) {
5     ...
6     m[i] = ;
7 }
8
9 class Point: public Comparable {
10 private:
11     int x, y;
12 public:
13     virtual bool operator<(const Comparable* v) const { ... };
14 };
15
16 int N = 10;
17 Point** points = new Point[N];
18 for (size_t i = 0; i < N; ++i)
19     points[i] = new Point(i, i);
20 nsort(points, N);
```

#### связный список в стиле C

##### неинтрузивный

```
1 struct node_s {
2     void* user_data;
3     struct node_s *next;
4 };
```

```

5 struct list_s {
6     struct node_s *head;
7 };
8
9 void push_back(struct list_s *l, struct node_s *n);

```

```

1 struct node_s *n = malloc(sizeof(struct node_s));
2 n->user_data = malloc(sizeof(struct point_s));
3 // деа malloc
4 push_back(&l, n);

```

## интрузивный

```

1 struct point_s {
2     int x, y;
3     struct node_s node;
4 };
5 list_t l;
6 struct point_s *pn = malloc(sizeof(*pn));
7 // malloc один, но требуется трюк для получения x, y
8 push_back(&l, &pn->node);

```

## связный список в стиле C++

```

1 class Object {
2 public:
3     virtual bool operator<(const Object* o) {
4         return this < o;
5     }
6     virtual bool operator==(const Object* o) {
7         return this == o;
8     }
9     virtual int hash(const Object* o) {
10         return (int)this;
11     }
12 };
13
14 class Node {
15 public:
16     void setData(Object* o);
17 private:
18     Object* o;
19     Node* next;
20 };
21
22 class List {
23     Node* head;
24 public:
25     void push_back(Node *n);
26 };
27
28 class Point: public Object {
29     int x, y;
30     virtual bool operator<(const Object* o) {
31         Point *p = (Point*) o;
32         return this < o;
33     }
34 };

```

```

1 List l;
2 Node *n = new Node;
3 n->setData(new Point(3, 4));
4 // тоже два new
5 l.push_back(n);

```

## private/protected/public наследование

```

1 class P {
2 public:
3     int x;
4 protected:
5     int y;
6 private:
7     int z;
8 };

```

```

1 class A: public P {
2     // x is public
3     // y is protected
4     // z is not accessible from P
5 }
6
7 class B: protected P {
8     // x is protected
9     // y is protected
10    // z is not accessible from P
11 }
12
13 class C: private P {
14     // x is private
15     // y is private
16     // z is not accessible from P
17 }

```

## Два типа отношений

**has a** у машины есть двигатель и тормоза

```

1 class Car {
2     Engine e;
3     Brakes b;
4 };

```

**is a** грузовик это машина, автобус это машина

```

1 class Truck: public Car {};
2 class Bus: public Car {};

```

Еще можно показать, что машина не двигатель:

```

1 class Engine {
2     protected:
3     void maintenanceCheck() { .. };
4 };
5
6 class Car: private Engine {
7     void reset() {

```

```

8      maintenanceCheck();
9  }
10 };

```

## ii C++11: final, override

### override

```

1 class Base {
2     public:
3     virtual void f(int);
4     virtual int g() const;
5     void h(int);
6 };

```

```

1 class Derived: public Base {
2     public:
3     void f(int); // не знаем virtual?
4     int g(); // перезагрузка вместо перекрытия
5     void h(int); // не была virtual
6 }

```

Это можно исправить, используя override;

```

1 class Derived: public Base {
2     public:
3     void f(int) override; // ok
4     int g() override; // CE
5     void h(int) override; // CE
6 };

```

### final

```

1 struct Base {
2     virtual void f();
3 };
4 struct Derived: public Base {
5     void f() final; // дети не могут больше унаследовать f()
6 }
7 struct DerivedD: public Derived {
8     void f(); // CE
9 }

```

## Вопрос 18 Элементы проектирования

- декомпозиция программы (Model, View)
- автотесты

## Вопрос 19 Множественное наследование

- разрешение конфликтов имен
- виртуальное наследование
- наследование интерфейсов

Множественное наследование — возможность наследовать сразу несколько классов. Виртуальное наследование — один экземпляр родителя.

```
1  class Person {
2      private:
3          string name_;
4      public:
5          string name() const { return name_; }
6  };
7
8  class Student: virtual public Person {
9      private:
10         string university_;
11     public:
12         string university() const { return university_; }
13 };
14
15 class Employee: virtual public Person {
16     private:
17         string company_;
18     public:
19         string company() const { return company_; }
20 };
21
22 class BadStudent: public Student, public Employee {};
```

Здесь не нужно виртуальное наследование: даты создания могут быть разными.

```
1  class Song: public Music, public Lyrics {
2      void show() { cout << Music::date << " " << Lyrics::date << endl; }
3  };
```

### Конструктор при виртуальном наследовании

```
1  class Person {
2      private:
3          string name_;
4      public:
5          explicit Person(string const& name): name_(name) {}
6  };
7
8  class Student: virtual public Person {
9      public:
10         explicit Student(string const& name): Person(name) {}
11 };
12
13 class Employee: virtual public Person {
14     public:
15         explicit Employee(string const& name): Person(name) {}
16 };
17
18 class BadStudent: public Student, public Employee {
19     explicit BadStudent(string const& name): Person(name), Student(name), Employee(name) {}
20 };
```

Но лучше использовать интерфейсы, где все методы абстрактные, нет полей. Тогда при наследовании не будет проблем с дублированием переменных и методов.

```
1 class Person {
2 public:
3     string name () const { return name_ ; }
4 private:
5     string name_ ;
6 };
7 class IStudent {
8 public:
9     virtual string name () const = 0;
10    virtual string university () const = 0;
11    virtual ~ IStudent () {}
12 };
13 class IFullTimeEmployee {
14 public:
15     virtual string name () const = 0;
16     virtual string company () const = 0;
17     virtual ~ IFullTimeEmployee () {}
18 };
19 class BadStudent : public Person , public IStudent , public IFullTimeEmployee {
20 public:
21     string name () const { return Person :: name (); }
22     string university () const { return university_ ; }
23     string company () const { return company_ ; }
24 private:
25     string university_ , company_ ;
26 };
```