

# Билеты по C++ Зимняя сессия

---

## Содержание

1. 1. Программа, состоящая из нескольких файлов	1
2. 2. Указатели, массивы, ссылки. Три вида памяти. Работа с кучей на C	3
3. 3. Структуры. Связный список на C	5
4. 4. Функции. Указатели на функции	7
5. 5. Обзор стандартной библиотеки C	8
6. 6. Ввод-вывод на C	10
7. 7. Классы и объекты	13
8. 8. Работа с кучей на C++	15
9. 9. Наследование и полиморфизм	17
10. 10. Умные указатели	19
11. 11. Перегрузка операторов	20
12. 12. Ключевые слова extern, static, inline	21
13. 13. Разное	22

# 1. Программа, состоящая из нескольких файлов

В больших проектах удобно разбивать исходный код на много файлов (модулей). Во-первых, это позволяет упорядочить код и легко искать нужное место в нем. Во-вторых, части кода можно неоднократно переиспользовать. В-третьих, при изменении одной части кода не нужно перекомпилировать весь проект. Теперь обо всём по-порядку.

- **Заголовочные файлы**

Заголовочные файлы нужны в первую очередь, чтобы избежать ошибок в программе, чтобы они обнаружили на стадии компиляции. В них содержатся объявления функций, чтобы компилятор знал, какой интерфейс у функций. Использование заголовочных файлов предотвращает коллизии имён, а также, если программист изменит функцию в своём файле - ему потребуется изменить её и в заголовочном, который подключен к другим файлам, в которых может использоваться эта функция, соответственно эти файлы должны быть перекомпилированы или переписаны, если, например, изменили возвращаемое значение функции. Заголовочные файлы подключаются командой препроцессора `#include`:

```
#include <headername> // для заголовков из стандартной библиотеки
#include "headername" // для собственных заголовочных файлов
```

Она вставляет всё, что написано в заголовочном файле в файл с кодом.

Заголовочные файлы могут также подключать друг друга с помощью `#include`. При этом если файл `a.h` подключает `b.h`, и `b.h` подключает `a.h`, то возникнет циклическая зависимость, и препроцессор сломается. Есть два способа этого избежать – надо писать в каждом заголовочном файле следующие команды (файл `header.h`):

```
#ifndef _HEARED_H_ // начало условия
#define _HEADER_H_ // после этого в следующий раз условие выше не выполнится
/* код заголовочного файла */
#endif // конец условия
```

Все команды выше есть в стандарте языка си, а значит код скомпилируется любым компилятором. Второй способ не входит в стандарт! Будьте внимательны:

```
#pragma once
/* код заголовочного файла */
```

- **Компиляция и линковка**

Сборка кода в программу происходит в несколько этапов. Рассмотрим на примере языка Си.

Сначала работает препроцессор, который редактирует код, выполняя все команды препроцессора (начинающиеся с `#`), например `#include` или `#define`.

Далее идёт трансляция в ассемблерный код, а из него в машинный код (объектные файлы).

Далее на стадии линковки связываются все имена переменных и функций с физическими адресами машины. На выходе получаем executable файл.

К проекту можно также линковать сторонние библиотеки. Такая линковка бывает двух

видов – статическая и динамическая. При статической линковке файлы библиотек (в linux они имеют расширение \*.a) ”вставляются” непосредственно в исполняемый файл. Таким образом его размер сильно увеличивается. При динамической линковке файлы библиотеки (в linux \*.so) необходимо скачивать отдельно.

- **Утилита make**

make служит для упрощённой сборки проектов. Особенно полезен, когда в проекте много файлов. Он построен на простом принципе: есть цели и их зависимости. Если зависимости изменились, то нужно изменить цель - следовательно вызываются команды, написанные ниже. Важной особенностью является то, что команды нужно писать с отступом с помощью tab. Выглядит это как-то так:

```
< цель >: < зависимости >  
    < команды >
```

Порядок записей в makefile тоже важен. По умолчанию выполняется первая цель, поэтому обычно её делают на подобии такого:

```
all: bin main
```

Чтобы вызвались соответствующие зависимости, которые являются целями ниже. Чтобы использовать make, достаточно написать в консоли make, тогда он вызовется со стандартными настройками, можно также указывать явно цели, использовать ключи на принудительное исполнение команд цели или изменения кол-ва потоков и т.д.

## 2. Указатели, массивы, ссылки. Три вида памяти. Работа с кучей на C

- **Применение указателей и ссылок**

Указатель на переменную - это номер ячейки памяти, её адрес, если быть точнее, в котором хранятся данные переменной. Если переменная состоит из нескольких байтов - берётся адрес первого байта. Синтаксически указатель применяется так:

```
int* ptr; // указатель на память размера sizeof(int)
```

Указатели также имеют особое значение - 0 (NULL), но оно именно особое, в нём ничего не хранится, это не нулевой адрес в памяти, а просто пустое значение, означающее, что указатель не указывает в осмысленное место. При работе с указателями и памятью нужно быть очень аккуратными, чтобы не указать в ту память, которую ещё не инициализировали, или, что ещё хуже, в ту память, которая уже занята под что-то другое, куда лезть не стоило. Можно делать указатель на указатель:

```
int *ptr1;
int **ptr2 = &ptr; // указатель на память, в которой лежит ptr1
```

или брать значение, лежащие в памяти, на которую указывает указатель:

```
int x = 3;
int *ptr = &x; // указатель на x
*ptr = 5; // значение x стало 5
/* &x - взять адрес переменной x
   *ptr - взять значение, которое лежит по указанному в ptr адресу
*/
```

Ссылки в C++ - это синтаксический сахар, позволяющий избавиться от звёздочек для работы со значениями. При компиляции это просто заменяется на указатели и разыменование.

```
int x = 3;
int &ref = x; // ref - это ссылка на x
ref = 5 // x тоже стало равно 5
int *ptr = &ref // равносильно int *ptr = &x
```

- **Арифметика указателей**

С указателями можно работать, как с обычными интовыми переменными, операторы +, -, ++ и т.п., при этом адрес сдвинется на нужное кол-во ячеек, например:

```
int* ptr = &b;
ptr += 5; // адрес сдвинется на 5 * sizeof(int) байтов
```

Массивы - это тоже указатели. `int arr[100]`; 'arr' можно использовать, как указатель. Обращение к переменным на самом деле происходит так:

```
int arr[100];
arr[42] = 3 // равносильно *(arr + 42) = 3
```

Благодаря коммутативности сложения в арифметике указателей  $(arr + i)$  равно  $(i + arr)$ . Поэтому  $arr[i]$  равносильно  $i[arr]$ .

- **Глобальная/статическая память, стек, куча**

Глобальная память - это память, выделяющаяся до запуска программы, кусок памяти определённого размера, в котором хранятся константы и всё, что не будет менять размеры в течении использования.

Стек - это динамическая память, которая используется для хранения вызванных функций, локальных переменных функций, и т.п.

Куча - это динамическая память, в которой могут, в отличие от стека, храниться большие объекты, массивы на несколько миллионов переменных, например. При выделении переменных на стеке - они "мусором", при выделении переменных на куче они инициализируются нулями, если выделение было не функцией malloc.

- **malloc/calloc/realloc**

Это сишные функции из библиотеки `stdlib`, они нужны для выделения памяти.

```
malloc(size_t x) //выделяет память (x байтов)
calloc(size_t num, size_t size) //выделяет и заполняет нулями num*size байтов
realloc(void* ptr, size_t size) перевыделяет память, на которую указывает ptr
free(void *ptr); //память, выделенную динамически всегда надо освободить
```

Память выделяется на куче, что довольно важно.

- **void\***

Это упоротый костыль, возможность делать указатели на память без явного указания типа. арифметика с этим работать не будет, ибо размер `void` не определен. Можно приводить указатели к разным типам, таким образом предавать в функцию какие угодно типы.

## 3. Структуры. Связный список на C

- **Неинтрузивная реализация**

Список - это какой-то блок данных и указатель на следующий блок, если список двусвязный, то ещё и на предыдущий.

```
struct Node {
    int x; // данные
    struct Node *next, *prev; // указатели вперед и назад
}

struct list {
    struct Node *head;
}

void new_node(struct list *l) {
    struct Node *n = malloc(sizeof(struct Node));
    n->next = l->head;
    n->prev = l->head->prev;
    l->head->prev->next = n;
    l->head = n;
}

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

- **Интрузивная реализация**

Интрузивный список хранит интрузивные ноды, внутри которых лежат собственно ноды с данными. Интрузивный список связан, а вот блоки со значениями не имеют указателей друг на друга. Интрузивный список позволяет создать лишь одну реализацию списка, а потом можно делать различные блоки со значениями, различными типами и функциями, которые будут работать поверх одного интрузивного списка (что-то вроде полиморфизма, только на си). Приведем реализации структуры и функции добавления:

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

struct intrusive_node
{
    struct intrusive_node *next;
    struct intrusive_node *prev;
};

struct intrusive_list
{
    struct intrusive_node *head;
};
```

```
struct position_node
{
    int x, y;
    struct intrusive_node node;
};

/* Преобразование ноды с данными в интрузивную ноду и добавление в список */
void add_node(struct intrusive_list *list, struct intrusive_node *new_node)
{
    new_node->prev = list->head;
    new_node->next = list->head->next;
    new_node->next->prev = new_node;
    list->head->next = new_node;
}

/* Создание ноды с данными (x, y) */
void add_position(struct intrusive_list *list, int x, int y)
{
    struct position_node *new_node = malloc(sizeof(struct position_node));
    new_node->x = x;
    new_node->y = y;
    add_node(list, &new_node->node);
}
```

Макрос `container_of` получает указатель на интрузивную ноду, тип который её содержит, название поля с нашей интрузивной нодой и возвращает указатель на ноду с данными.

Оставшаяся реализация в качестве упражнения ;]

- **typedef**

Всё просто. `typedef` существующий `_тип` синоним `_для_него`;

```
typedef long long ll;
typedef struct pos{
    //pass
} pos_t;
```

`typedef` не является командой препроцессора, он лишь вводит синоним для существующего типа.

## 4. Функции. Указатели на функции

- **указатели на функции**

У функций, как и у переменных есть свой адрес в памяти. Поэтому можно взять указатель на функцию, чтобы, например, передать её в другую функцию. Как это устроено, можно посмотреть в реализации сортировки.

- **Как происходит вызов функции**

Вызов функции происходит в несколько этапов.

Сначала на стеке выделяется место для хранения возвращаемого значения функции, её локальных переменных, параметров, которые функция получает и адрес, куда нужно вернуться после выполнения этой функции.

Затем параметры копируются на свои места и, собственно, вызывается функция.

После выполнения функции сохраняется возвращаемое значение, вызываются деструкторы локальных объектов.

Наконец программа возвращается в то место, где была вызвана функция, (**TODO**: что происходит с возвращаемым значением?), И место на стеке освобождается.

- **Реализация сортировки**

Ниде приведена пузырьковая сортировка, которая работает для массивов любых типов. Параметры: указатель на начало массива, количество элементов, размер элемента в байтах, указатель на функцию-компаратор. Компаратор возвращает отрицательное число, если первый объект меньше, положительное, если первый объект больше и ноль в случае равенства.

```
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);
}

void sort(void *base, size_t num, size_t size, int (*compar)(const void*,
    const void*)) {
    size_t i, j;
    for (i = 0; i < num; i++)
        for (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);
}
```

Пример использования:

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

int arr[10] = {9, 8, 7, 6, 5, 4, 3, 2, 1, 0};
sort(arr, 9, sizeof(int), compareInt);
```



## 5. Обзор стандартной библиотеки C

- **string.h**

Всем привет, с вами kostya55516, сегодня мы будем обзирать string.h.

-Копирование в destination num байтов из source:

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

-Сравнение кусков памяти. Возвращает 0, если они одинаковые, > 0, если в ptr1 встретился байт с большим unsigned int значением, чем в ptr2 и < 0 в противном случае:

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

-То же самое, но для строк. Если первая строка короче второй – результат будет < 0, если вторая короче – соответственно > 0:

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

-Конкатенация строк. Приклеивает source к destination справа. Возвращает destination:

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

-Поиск подстроки в строке. Возвращает указатель на место в первой строке, где начинается первое вхождение второй. Если первая строка вторую возвращает NULL:

```
char* strstr (const char* str1, const char* str2);  
/* Пример */  
strstr("it's a cpp, baby", "cpp"); //вернет указатель на 'с' в первой строке  
strstr("it's a cpp, baby", "java"); //вернет NULL
```

-Поиск символа в строке:

```
char* strchr (const char* str, int character);
```

- **stdlib.h**

-Первый гость нашей программы, выделяет на куче память такого размера и возвращает указатель на начало выделенного блока. Если выделить не получилось - возвращает нулевой указатель:

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

-Освобождение памяти, выделенной malloc-ом и его друзьями:

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

-Перевыделение памяти с новым размером size. Возвращает указатель на память, данные сохраняются. Может оставить данные на своём месте, а может выделить на новом и скопировать:

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

-Выделение  $\text{num} * \text{size}$  памяти, инициализация нулями. В остальном, как malloc:

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

-Строка в число. Пропускает пробелы, находит знак перед числом, если он есть и переводит цифры из строки в интовое значение, учитывая знак. Если какая-то ошибка, преобразовать не получается - будет неопределенное поведение:

```
int atoi (const char* str);
```

-Строка в long long int. Если endptr не ноль, то endptr указывает на указатель на символ после числа. Делает то же, что и strtol, но для long long. Если всё ок - вернёт число, если невалидна строка - вернёт ноль, если происходит переполнение значения - вернется LLONG\_MAX ( $2^{63} - 1$ ) или LLONG\_MIN ( $-2^{63}$ ):

```
long long int strtoll (const char* str, char** endptr, int base);
```

-Генерация псевдослучайных чисел:

```
void srand (unsigned int seed); //инициализирует генератор рандома числом  
srand(time(NULL)); // обычно используют текущее время (time.h)  
int rand (void); //генерирует псевдослучайное число, опираясь на srand()
```

-Быстрая сортировка. base – массив, num – кол-во элементов, size – размер элемента, compar – указатель на функцию сравнения элементов (компаратор):

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

## 6. Ввод-вывод на C

- **FILE, fopen, fclose, r/w, t/b**

FILE – Этот тип данных определяет поток и содержит информацию, необходимую для управления потоком, в том числе указатель на буфер потока, и его показатели состояния. Файловые объекты обычно создаются с помощью вызова функций fopen или tmpfile, которые возвращают ссылку на объект связанный с файлом.

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

Функция fopen открывает файл, имя которого указано в параметре fname и связывает его с потоком, который может быть идентифицирован для выполнения различных операций с файлом. Файл можно открывать в разных режимах.

«rt»/«rb» Режим открытия файла для чтения. Файл должен существовать.

«wt»/«wb» Режим создания пустого файла для записи. Если файл с таким именем уже существует его содержимое стирается, и файл рассматривается как новый пустой файл.

«at»/«ab» Дописать в файл. Операция добавления данных в конец файла. Файл создается, если он не существует.

Символ 't' означает открытие файла в текстовом режиме. 'a' – в бинарном. Если файл был успешно открыт, функция возвращает указатель на объект файла, который используется для идентификации потока и выполнения операций с файлом. В противном случае, возвращается нулевой указатель.

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

Функция fclose закрывает и разъединяет файл filestream, связанный с потоком. Все внутренние буферы, связанные с потоком сбрасываются: содержание любого незаписанного буфера записывается и содержание любого непрочитанного буфера сбрасывается. Если файл успешно закрыт, возвращается нулевое значение. В случае ошибки, возвращается EOF.

- **Буферизация**

Поскольку чтение/запись на диск, ввод/вывод на экран – операции долгие, потоки накапливают куски в буфере. Поток записи сбрасывает буфер в файл после того как тот переполнился, а потоки чтения считывают кусок файла в буфер, и работают с ним, пока тот не опустеет, после чего считывают ещё кусок.

- **stdin, stdout**

-Поток номер 0 (stdin) зарезервирован для чтения команд пользователя или входных данных.

-Поток номер 1 (stdout) зарезервирован для вывода данных на экран.

- **printf, scanf, fprintf, fscanf, sprintf, scanf**

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

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

-Считывание данных из потока stdin Возвращает количество успешно считанных элементов. В случае ошибки возвращает EOF.

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

-Форматированный вывод в файл. Записывает в указанный поток последовательность символов в формате, указанном аргументом format:

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

-Форматированное чтение из файла:

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

-Вывод в массив, указанный аргументом buf:

```
int sprintf(char *buf, const char *format, arg-list);
```

-Ввод из массива buf:

```
int sscanf(char *buf, const char *format, arg-list);
```

### • fread, fwrite, fseek, ftell, fflush

-Считывание массива размером count элементов, каждый из которых имеет размер size байт, из потока, и сохранение его в блоке памяти, на который указывает ptrvoid. Индикатор положения потока увеличивается на общее число записанных байтов. Возвращается количество, успешно считанных, элементов:

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

-Запись массива размером count элементов, каждый из которых имеет размер size байт, в блок памяти, на который указывает ptrvoid:

```
size_t fwrite(const void* ptrvoid, size_t size, size_t count, FILE* filestream);
```

-Перемещение указателя позиции в потоке. Устанавливает внутренний указатель положения в файле, в новую позицию, которая определяются путем добавления смещения к исходному положению.

SEEK\_SET Начало файла

SEEK\_CUR Текущее положение файла

SEEK\_END Конец файла

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

-Значение указателя текущего положения потока. Для бинарных потоков, возвращается значение соответствующее количеству байт от начала файла:

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

-Если данный поток был открыт для записи (или, если он был открыт для обновления и последняя операция ввода/вывода была операцией вывода) любые незаписанные данные в выходном буфере записываются в файл. Если аргумент является нулевым указателем, то открытый файл очищается. Поток остается открытым после этого вызова. Нулевое значение указывает на успех:

```
int fflush(FILE * filestream);
```

- **Обработка ошибок, feof, ferror**

-Проверка, достигнут ли конец файла, связанного с потоком, через параметр `filestream`. Возвращается значение, отличное от нуля, если конец файла был действительно достигнут:

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

-Отслеживание появления ошибки, связанной с потоком, который передаётся через параметр `filestream`. Если ошибка была обнаружена, возвращается значение, отличное от нуля. Эту функцию целесообразно вызывать после выполнения предыдущей операции с потоком. Таким образом, если предыдущая операция выполнится с ошибкой, функция `ferror` проинформирует об этом:

```
int ferror(FILE* filestream);
```

## 7. Классы и объекты

- **Зачем нужно ООП**

Люди пришли к ООП по нескольким причинам, но по сути, так просто исторически сложилось. Языки программирования делали всё более высокоуровневыми, чтобы не задумываться, как там всё внутри устроено, чтобы ускорить написание кода и его повторное использование. Три основные идеи ООП - инкапсуляция, наследование и полиморфизм. Если коротко, то инкапсуляция позволяет работать с кодом, с классом, как с чёрным ящиком. У тебя есть какой-то объект и ты просто пользуешься его функциями, которые пришиты к нему (интерфейс), не имея представления, как они работают. Программисты всегда двигались в сторону уменьшения сущностей, с которыми нужно работать, чтобы не забивать мозг лишними вещами и работать быстрее. Также можно использовать код повторно. Захотелось изменить класс, добавить туда пару функций, какие-то исправить, а заново код писать не хочется, ведь всё остальное можно оставить - ну тогда просто можно унаследоваться от класса и менять только необходимое. Плюс появилась перегрузка, можно было написать класс, который работал бы по-разному с разными типами, а также можно было бы унаследоваться и перекрыть старые функции, изменив интерфейс класса, к примеру, поменять один блок на другой – это полиморфизм.

- **Инкапсуляция: `private/public`**

В классе есть два(вообще три, но о третьей позже) модификатора доступа к полю - приватный и публичный. Как понятно из названия, публичными полями могут пользоваться все кому не лень, изменять публичные переменные, вызывать публичные функции. Но чтобы программисты, при использовании твоего кода не выстрелили себе в ногу, что они очень любят делать в свободное время, есть приватные поля. К ним доступ имеют только представители самого класса. Но машинный код не знает ничего про модификаторы, их там и нет. Компилятор просто посмотрит на твой код и кинет ошибку, если ты залез туда, куда не надо.

- **Конструктор (`overloading`), деструктор**

В C++ по сравнению с C появилась такая вещь, как перегрузка (`overloading`) функций. Теперь линкер умеет различать функции с одинаковым именем, но с различающимся числом или типом аргументов или типом возвращаемого значения. На стадии разрешения имен функциям присваиваются новые имена, в которые включается указание типов аргументов и типа возвращаемого значения. В частности, перегрузка позволяет сделать одному классу несколько конструкторов. Некоторые классы требуют хитрой инициализации - конструктора. Например `std::vector`. Плюс плюсов - перегрузка. Можно создать несколько конструкторов от разных параметров, на основе которых код будет делать разные вещи. Например если параметры не передались, он может по умолчанию выбрать свои. Если передан один параметр - создаст вектор такого размера, если два - создаст и проанализирует. C++ создаёт свой конструктор по умолчанию (без параметров) для твоего класса. Если классу не требуется выделять память, открывать файлы или захватывать ресурсы, и т.п., то просто не писать конструктор. Если ты написал зоть какой-то конструктор, то конструктор по умолчанию не создается. Деструктор же

нужен, чтобы убить всё то, что ты натворил в конструкторе и работе с классом. Он вызывается автоматически, когда объект класса выходит из области видимости, т.е. если его объект был создан внутри функции, то при её завершении вызовется деструктор. Обычно, когда не нужен конструктор – не нужен и деструктор, компилятор сам справится, а когда пишется конструктор - тогда и деструктор нужен.

- **Инициализация**

До вызова конструктора все поля инициализируются чем-то . Чтобы их проинициализировать, часто используется двоеточие после конструктора и там инициализация полей. Такой метод идеологически устоявшийся, также позволяет избежать коллизии имён и проинициализировать ссылки. Например, для класса фигуры конструктор со списком инициализации может выглядеть вот так:

```
Figure::Figure(int id, int x, int y) : id(id), x(x), y(y) {}
```

как тут видно, имена полей класса и имена аргументов функции одинаковые, но проблем не возникает.

## 8. Работа с кучей на C++

- **new/delete**

Это аналог malloc и free из си. У них могут быть внутренние различия, в зависимости от компилятора, поэтому лучше не смешивать одно с другим.

```
int* a = new int[5]; //квадратные скобки, чтобы забить массив.
my_class* a = new my_class; //без скобок, чтобы создать элемент.
delete [] a;
delete my_class;
```

ну тут тоже видны аналогии.

- **Создание объектов в куче**

delete и new вызывают деструкторы и конструкторы классов соответственно. Также не стоит мешать создание массивов new, а удаление delete без скобок, это может вызвать undefined behavior. Если создан массив классов, то delete[] пройдёт по массиву и вызовет деструктор.

- **Конструктор копий**

Конструктор копий, в основном, используется тогда, когда нужен конструктор и деструктор, тогда же нужен и оператор присваивания. Иначе же, стандартные реализации справляются. По сути оператор копирования - это тот же конструктор, но с объектом того же типа, чтобы можно было сразу инициализировать копию объекта, например:

```
my_class::my_class(my_class& obj) {
    // выделяем память под ресурсы если( надо)
    // копируем поля из obj
}
```

Важно! В конструктор копий передается именно ссылка, потому что при передачи параметра в функцию напрямую объект необходимо скопировать. Таким образом конструктор копирования будет рекурсивно вызывать сам себя и всё сломается :(

Конструктор копий используется, когда мы пишем my\_class a = b; или, когда вызываем функцию и передаём класс, как параметр. Также конструктор копий позволяет сделать my\_class a = 3; Но порой это нам не нужно и мы можем это запретить:

```
explicit my_class(size_t size);
my_class b(3); // OK
my_class b = 3; // Compilation Error
```



- **Оператор присваивания**

Теперь мы хотим создать два объекта, а потом первому присвоить второй. Как мы это хотим? Через привычное равно.  $a = b$ . Для этого нужно переопределить оператор равно.

```
my_class& operator=(const my_class obj) {  
    // swap идиома;  
    return *this;  
}
```

У этого оператора должно быть возвращаемое значение, иначе такие вещи, как тройное присваивание не будут работать. Возвращаем мы ссылку на объект, поэтому нужно разыменовывать указатель `this`. Swap идиома позволяет просто и быстро перекопировать значения. Мы передаём в оператор не ссылку, поэтому копия уже создаётся, далее мы свопаем значения, а копия удаляется сама собой. Удобно.

## 9. Наследование и полиморфизм

Наследование – это создание класса на основе другого, при этом производный класс будет иметь те же методы, что и базовый, но еще есть возможность добавлять новые поля и методы, а также переписывать или дописывать старые.

Полиморфизм — это возможность сделать объект базового типа объектом любого из производных типов, а также возможность переписать или дописать методы базового класса в производном.

- **protected**

В то время, как `private` дает доступ только изнутри данного класса, `protected` позволяет видеть свое содержимое еще и всем классам-наследникам данного класса. `protected`-инкапсуляция нужна как раз для того, чтобы можно было для написания собственных методов производного использовать поля и методы базового класса, которые тем не менее не должны использоваться напрямую вне этих классов.

- **virtual (overriding)**

Виртуальные функции передаются наследникам, подразумевая, что функции могут быть переписаны у них. Эти функции сделаны, чтобы программа понимала во время работы, какую функцию нужно вызывать, основываясь на типе объекта. Для переопределения функции в дочернем классе необходимо, чтобы интерфейс (все аргументы и возвращаемое значение) совпадал с оригиналом, иначе это будет просто перегрузкой. Также необходимо обратить внимание на то, что конструктор не может быть виртуальным, но деструктор может. Например, если у нас есть класс `list`, у него наследник `double_list`, то можно сделать в `list` функцию, которая принимает `list`, при этом `double_list` тоже может быть записан в качестве параметров этой функции при вызове, и проблем никаких не будет, ибо двусвязный список - это тоже список. Но компилятор не может узнать на стадии компиляции, какую ему функцию нужно вызвать, которая работает со списком или с двусвязным списком. Поэтому и существуют виртуальные функции. Также виртуальные функции могут быть объявлены с присвоением нуля. Это будет говорить компилятору о том, что они обязательно должны быть перекрыты в дочерних классах. Такая запись означает также, что у этой функции нет реализации в этом классе.

```
class list {
public:
    virtual void add(int c) {
        //pass
    }
    virtual void do_smth() = 0;
}
class double_list: public list{ //класс double_list – наследник класса list
public:
    void add(int c) {
        //pass
    }
    do_smth() {
        //обязательно надо реализовать
    }
}
```

- **таблица виртуальных функций**

Все виртуальные методы класса автоматически записываются в таблицу, которая идет перед классом. Когда происходит вызов виртуальной функции, программа обращается к этой таблице, смотрит адрес нужной версии виртуальной функции, и вызывает функцию, лежащую по этому адресу.

- **статическое/динамическое связывание**

Связывание – это процесс сопоставления функции ее адреса. Статическое связывание выполняется на стадии компиляции. Динамическое связывание выполняется во время работы программы с помощью таблицы виртуальных функций. Если посмотреть на следующий пример, то станет понятно, почему нужно динамическое связывание:

```
list* l;
srand(time(NULL));
int n = rand();
if (n > 10)
    l = new list(5);
else
    l = new double_list(6); //double_list – наследник класса list, но у них
    разные функции add_value
l->add_value(7); //если бы было статическое связывание, то add_value была бы
    всегда одной и той же
```

Действительно, если нам нужно, чтобы иногда выполнялся метод одного класса, а иногда другого, то статическое связывание нам не поможет.

## 10. Умные указатели

- `scoped_ptr`

Мы не хотим вечно думать о памяти, о том, чтобы её чистить, поэтому давайте сделаем умный указатель, который сам будет заботиться о высвобождении памяти, когда выходит из области видимости.

```
class scoped_ptr
{
private:
    Person *p;
    scoped_ptr(const scoped_ptr p) { } //Чтобы запретить копировать объект
    void operator=() { }
public:
    scoped_ptr(Person *p) { this->p = p; }
    ~scoped_ptr() { delete this->p; }
    Person* ptr() { return p; }
    Person* operator->() { return p; }
    Person& operator*() { return *p; }
}
```

Юзкейсы:

```
scoped_ptr p = new Person(Vasya);
p->hasBirthday(); //будет воспринимать как p1->->
*p1.hasBirthday(); //не скомпилился: сначала выполнится '.', затем '*'
(*p1).hasBirthday(); //скомпилился
```

Но у такой реализации есть проблемы:

```
scoped_ptr p1 = new Person(Vasya);
scoped_ptr p2 = new Person(Dima);
p1->hasBirthday(); //будет воспринимать как p1->->
*p1.hasBirthday();
scoped_ptr p1 = p; //в конце дважды вызовется деструктор, а в нем delete
p1 = p2; //утечка памяти из p2 и та же проблема с delete
```

Можно засунуть оператор присваивания и копирования в приватное поле, но это не избавит от всех проблем и обмануть всё равно получится.

- `auto_ptr`

**TODO:**

- `shared_ptr`

**TODO:**

# 11. Перегрузка операторов

- бинарные и унарные  
**TODO:**
- в классе/вне класса  
**TODO:**
- friend  
**TODO:**
- приведение типов  
**TODO:**

## 12. Ключевые слова extern, static, inline

- extern у переменных  
**TODO:**
- static у переменных и функций  
**TODO:**
- static у полей и методов  
**TODO:**
- inline у функций  
**TODO:**

## 13. Разное

- Автотесты  
**TODO:**
- Декомпозиция программы (Model, View)  
**TODO:**
- Ключевые слова `const`, `enum`  
**TODO:**