

Билеты по C++ Зимняя сессия

Авторы: Недиков Константин, Купоросов Василий

Отдельное спасибо Орловой Александре и Ютману Михаилу за помощь в написании

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. Неинтрузивный связный список на C	5
4. 4. Структуры. Интрузивный связный список на C	6
5. 5. Функции. Указатели на функции	8
6. 6. Обзор стандартной библиотеки C	9
7. 7. Ввод-вывод на C. Текстовые файлы	11
8. 8. Ввод-вывод на C. Бинарные файлы	13
9. 9. Классы и объекты	15
10. 10. Работа с кучей на C++	17
11. 11. Наследование и полиморфизм	19
12. 12. Умные указатели	21
13. 13. Перегрузка операторов	23
14. 14. Ключевые слова extern, static, inline	25
15. 15. Разное	27
16. 16. Элементы проектирования	30

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, это пустое значение, означающее, что указатель не указывает в осмысленное место. Не пытайтесь обратиться в NULL!

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 заполнены "мусором", при выделении переменных на куче они инициализируются нулями, если выделение было не функцией malloc.

- **malloc/calloc/realloc/free**

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***

Этот тип даёт возможность делать указатели на память без 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. Неинтрузивный связный список на C

- **Неинтрузивная реализация**

Список - это какой-то блок данных и указатель на следующий блок, если список дву-связный, 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);
}
```

- **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. Структуры. Интрузивный связный список на C

- **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;
};

/* Добавление интрузивной ноды 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);    // нодасданнымипревращаетсявинтрузивную
}

```

Макрос `container_of` получает указатель на интрузивную ноду, тип которой её содержит, название поля с нашей интрузивной нодой и возвращает указатель на ноду с дан-

ными.

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.

5. 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 ниже.

- **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, И место на стеке освобождается.

- **Реализация сортировки**

Ниже 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);
```

6. 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*));
```

7. Ввод-вывод на C. Текстовые файлы

- **FILE, fopen, fclose, rt/wt**

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» Режим открытия файла для чтения. Файл должен существовать.

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

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

Символ 't' означает открытие файла в текстовом режиме. 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, stderr**

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

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

-Поток номер 2 (`stderr`) как и `stdout` выводит данные на экран, но у него нет буфера. Так что текст попадает на экран сразу.

- **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);
```

- **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);
```

8. Ввод-вывод на C. Бинарные файлы

- **FILE, fopen, fclose, rb/wb**

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.

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

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

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

Символ 'b' означает открытие файла в бинарном режиме. 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.

- **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 определяется 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 все открытые потоки флешатся. Поток остается открытым после этого вызова. Нулевое возвращаемое 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 передается 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);
```

9. 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. 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 можно не писать конструктор. Если ты написал хоть 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 полей**

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.

10. 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=(my_class obj) { // такой способ называется copy-swap
    std::swap(_data, obj._data);    // свапаем все поля
    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 свапаем значения, а копия удаляется сама собой. Удобно.

11. 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**

Если в классе есть виртуальная функция, то при компиляции создаётся таблица виртуальных функций, где грубо говоря, на идейном уровне, в одном столбце стоит название, в другом адрес. Если линковка была статическая - при исполнении будет сразу команда "вызвать функцию по такому-то адресу", при динамической же будет команда "найди такую функцию в таблице виртуальных функций и вызови её". У каждого объекта есть указатель на виртуальную функцию, чтобы программа могла понять, у какого класса какую таблицу нужно смотреть. Естественно, мы платим за виртуальные функции скоростью работы программы.

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.

12. 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;
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 p->->
*p.hasBirthday(); //не скомпилился: сначала выполнится '.', затем '**'
(*p).hasBirthday(); //скомпилился
```

Но у такой реализации есть проблемы:

```
scoped_ptr p1 = new Person(Vasya);
scoped_ptr p2 = new Person(Dima);
scoped_ptr p3 = p1; //в конце дважды вызовется деструктор, а в нем delete
PrintUser(p1); //аналогично
p1 = p2; //утечка памяти из p2 и та же проблема с delete
```

Можно сделать оператор присваивания и конструктор копирования приватными, но это не избавит от всех проблем и обмануть всё равно получится.

- **auto_ptr**

Давайте избавимся от проблем утечки памяти и многократного удаления:

```
class auto_ptr {
private:
    Person *p;
public:
    auto_ptr(Person *p) { this->p = p; }
    ~auto_ptr() { if (p) delete p; } //удаляем только один раз
    Person* ptr() { return p; }
    Person* operator->() { return p; }
    Person& operator*() { return *p; }

    auto_ptr(auto_ptr &o) { //вместо конструктора копирования – конструктор
        перемещения
        this->p = o->p;
        o->p = NULL;
    }
};
```

Но тут также остались проблемы. Мы всё ещё не можем использовать его многократно при вызове функций или копировании.

```
PrintUser(p); //норм
PrintUser(p); // там уже Null, можно вызвать только один раз
void func(auto_ptr &p); // можно передавать по ссылке, но это неудобно
```

Мы хотели максимально облегчить себе жизнь, а не следить за жизнью указателя. Нужен счётчик созданных копий.

- **shared_ptr**

//шарит ptr действительно шарит, что происходит.

Пусть каждое копирование вызывает увеличение счётчика, а каждый выход за область видимости (вызов деструктора) – уменьшение. А если счётчик ссылок достиг нуля – удаляется и сам storage – то, что мы храним.

Разные указатели на один объект будут работать с одним и тем же storage, который хранит наш объект (количество таких указателей мы и будем считать).

Хорошая вещь, писать её мы, конечно, не будем :)

Но есть уже написанная полная версия по [ссылке](#).

13. Перегрузка операторов

Бывает удобно вместо длинной записи названия функции или метода использовать короткую запись через какой-то математический знак, например, через '+', '-', '<', и т.п.

- бинарные и унарные

Обратите внимание на особенности операторов '++' и '[']':

```
class Matrix {
...
    Matrix& operator+=(const Matrix& m) {
        for (std::size_t i = 0; i < _rows; i++)
            for (std::size_t j = 0; j < _cols; j++)
                _data[i][j] += m._data[i][j];
        return *this;
    }
    Matrix operator+(const Matrix& m) {
        Matrix tmp(*this);
        return tmp += m;
    }
    Matrix& operator++() { // ++matr;
        for (std::size_t i = 0; i < _rows; i++)
            for (std::size_t j = 0; j < _cols; j++)
                _data[i][j]++;
        return *this;
    }
    Matrix operator++(int n) { // matr++; n всегда равно нулю,
        Matrix tmp(*this); // это просто такой костыль
        for (std::size_t i = 0; i < _rows; i++)
            for (std::size_t j = 0; j < _cols; j++)
                _data[i][j]++;
        return tmp;
    }
    int* operator[](std::size_t i) { // возвращает строку матрицы,
        return _data[i]; // в которую можно присвоить
    }
    const int* operator[](std::size_t i) const { // возвращает строку матрицы,
        return _data[i]; // из которой можно считать
    }
...
};
```

Константный оператор '[']' помогает избегать проблем, когда есть константная матрица, и мы из неё читаем. Во имя переиспользования кода и во благо быстрдействию некоторые операторы часто выражаются через другие.

Пример с практики (реализация матриц).

Интересный факт. Поскольку операторы являются методами класса, их можно вызывать в функциональном виде:

```
Matrix a, b;
if (a.operator[](10)[10] == 42) // равносильно if (a[10][10] == 42)
    a.operator+=(b).operator++(0); // равносильно (a += b)++;
a.operator+=(b).operator++(); // равносильно ++(a += b);
```


- **в классе/вне класса**

Оператор можно переопределять и вне класса. Это будет выглядеть, как определение функции от одной или двух переменных, и тогда это уже не будет являться методом класса. Но все равно оператор можно будет вызывать длинно, как функцию от двух параметров.

Это бывает полезно если у нас нет доступа к коду этого класса. А ещё если нам хочется написать оператор, у которого слева, например, `int`, а справа какой-нибудь класс. Тогда мы сможем, например, прибавить к инту наш класс:

```
int operator+=(int &number, const Matrix& matr) {
    for (std::size_t i = 0; i < _rows; i++)
        for (std::size_t j = 0; j < _cols; j++)
            number += _data[i][j]++;
}
```

Упражнение: что делает эта функция и зачем она может понадобиться? ;)

- **приведение типов**

```
BigInt a = 3; // Приведение
BigInt a = (BigInt)3; // Теперь более явно.
```

Начнём с первого случая. Что это и с чем это есть? Чтобы оно сработало - у нашего класса должен быть конструктор вот такой:

```
BigInt(int) {}
```

Как происходит приведение:

```
BigInt a = (BigInt)3;
      ^         |
      |         v
      a(tmp)<----tmp
```

Современные компиляторы могут это сделать быстрее, без создания третьей переменной. Проблема в том, что это может быть не слишком разумно, ибо `BigInt(3)` просто создаст массив размера 3 (а не число 3), поэтому можно запретить подобное присвоение с помощью `explicit BigInt(size_t)`;

Теперь о приведении типов при наследовании.

Элементы производного класса можно просто так присваивать переменным базового класса (например присвоить `double list` переменной типа `list`). Можно сделать конструктор с параметром, являющимся принимаемым для приведения значения и чтобы привести надо написать перед значением или переменной второго типа в скобках имя первого типа. **Пример.**

14. Ключевые слова `extern`, `static`, `inline`

- **`extern` у переменных**

Вот у нас есть два файла – 1.cpp и 2.cpp. Мы создаём в первом переменную `int c = 0`, а во втором в функции мы делаем `c++`; Вылезет ошибка, компилятор просто не знает о существовании этой переменной. Как это исправить? Ну можно создать заголовочный файл, а можно в 2.cpp написать `extern int c`; Тогда компилятор не будет создавать переменную, но он будет знать, что такая существует.

- **`static` у переменных и функций**

1. **Глобальные `static` переменные**

В 1.cpp создадим `static int c = 0`; во втором файле создадим обычную глобальную переменную `int c = 0`; В первом файле переменная 'с' будет глобальной только внутри своего модуля(файла), притом переменная из второго файла на это не повлияет. Как это происходит?

Все такие вещи обрабатываются на стадии линковки, есть 2 вида линковки – внутренняя(`internal`) и внешняя(`external`). Линковщик для каждого модуля создаёт свою табличку, где переменным сопоставляются адреса.

2. **Локальные `static` переменные**

```
int f(int k) {
    static int c = k;
    c++;
    return c;
}
```

Мы создали переменную один раз, а дальше при вызове функции она не будет создаваться заново, она будет увеличивать старую переменную, при этом область видимости у неё всё ещё локальная. При этом проинициализируется она переменной `k` только один раз, остальные будут всё равно, что мы передали. Переменная создается не в самой функции, она также записывается в таблицу, а инициализация (или вызов конструктора, если там класс) произойдёт один раз при первом вызове функции.

Здесь полезно вспомнить функцию `strtok()`, которая один раз получает строку, а при последующих вызовах постепенно разбивает её на токены. Подробно эта функция описана [здесь](#).

3. **`static` функции**

У всех функций по умолчанию имеется ввиду ключевое слово `extern`, если мы хотим запретить вызов функции в другом файле – необходимо дописать `static`. Например мы пишем библиотеку, мы не хотим, чтобы пользователь пользовался какой-то внутренней функцией, поэтому можно запретить это и написать статик. Или другой пример, когда несколько программистов пишут блоки и чтобы имена не перекрывались, они могут вспомогательные функции обозначать, как `static`.

- **static** у полей и методов

Теперь о том, что добавили в c++:

1. **static** поля

```
class process{
public:
    process(){
        ++c;
    }
    ~process(){
        --c;
    }
    static int c = 0;
}
```

Это будет переменная глобальная для всех объектов этого класса, но область видимости будет только внутри него. Можно и снаружи получить к ней доступ так `process::c`; Такое поле нужно явно инициализировать в `cpp` файле `int process::c = 0`; Но мы не хотим, чтобы её изменяли вне класса. Тогда запишем эту переменную в `private` модификатор доступа. Вроде всё хорошо, но вот теперь мы не можем снаружи узнать, сколько у нас процессов создано. Можно создать функцию `int getCounter()`; однако если у нас не будет ни одного объекта, мы не сможем узнать, что процессов 0. Для этого можно сделать эту функцию тоже статической.

2. **static** методы

```
static int getCounter();
```

`process::getCounter()` можно вызвать и без процессов. Нам не нужны объекты, но мы и не можем обращаться к нестатическим переменным. Однако если в неё передать объект, то она сможет это сделать.

3. **inline** у функций

Ключевое слово `inline` у функции сообщает оптимизатору, чтобы при возможности он вставил код этой функции в место, где она вызывается. Таким образом оптимизируются память и время работы, которые были бы потрачены на вызов функции. `inline` не обещает, что функция будет встроена в код, компилятор сам решит, стоит ли это делать и сможет ли он это сделать. Также инлайн функциям нужно небольшое ограничение - они должны знать тело. Если у нас тело функции будет в одном месте, а при вызове мы будем знать лишь объявление - компилятор не поймёт, что ему нужно делать, но если в каждом файле будут одинаковые функции - линковщик сойдёт сума. Для этого для инлайн функций сделано исключение, линковщик выбирает какую-то одну из тех, что одинаковые, а другие реализации просто игнорирует. Так что это приведёт к `undefined behavior`, если сделать разные реализации инлайн функций с одинаковым интерфейсом.

Также все функции, определённые внутри класса в `header` файле по умолчанию являются `inline` функциями. Чтобы они не были инлайн - нужно написать определение отдельно.

15. Разное

- friend

Ключевое слово friend появляется в теле класса и дает функции или другому классу доступ к private или protected членам класса при появлении этого ключевого слова. friend функции и классы не являются методами и подклассами данного класса.

```
class gauss {
private:
    int x, y;
public:
    friend class printer;
    gauss(int x = 0, int y = 0): x(x), y(y) { }
    friend gauss& operator++(gauss &one) {
        x++;
        y++;
        return *this;
    }
};

class printer {
public:
    printer operator<<(gauss one) {
        printf("(%d, %d)", one.x, one.y);
        return *this;
    }
};

int main() {
    gauss a(1, 2), b(3);
    printer out;
    out << a << " " << ++b;
}
```

- **Ключевые слова const, enum**

1. **const** - это ключевое слово означает, что данные не будут изменяться.

`const int N = 10;` означает, что переменную менять нельзя.

`const` делает неизменяемым то, после чего стоит. Можно в некоторых случаях, как выше, менять порядок написания, но в некоторых случаях он будет важен.

`char const *s` сделает указатель на `char`, данные которого менять нельзя. `const char* s` будет делать то же самое.

`char * const s` будет уже запрещать менять сам адрес, но позволяет менять значения.

В конце концов, `char const * const s` будет запрещать менять всё.

`char ss[10]` - массив, у его указатель (`ss`) нельзя менять, т.е. это аналогично такому описанию: `char * const ss;`

Для чего же это? Во-первых, это позволяет подсказать программисту, какие поля могут изменяться, а какие точно не будут, а во-вторых, это защита от самого себя, чтобы ничего не испортить.

Ссылки тоже константы по указателю.

Если сделать константное поле в классе, то ему нельзя будет присвоить какое-то значение в конструкторе, но для этого идеально подойдёт список инициализации.

const у параметра функции, например, у конструктора копий или у функции сравнения строк

```
Array(const Array a){...}
int strcmp(const char *s1, const char *s2);
```

с одной стороны предостерегает того кто пишет функцию от изменения того, что меняться не должно по логике, а с другой стороны сообщает пользователю функции, что ничего плохого с его данными не случится, если он их засунет в эту функцию.

У возвращаемого значения

```
const Array operator=(const Array a){...}
```

защищает нас от ... чего? Ну всё не так уж и прозрачно, вообще бред, если честно, но может когда-то это вас защитит. `const` у возвращаемого значения говорит, что значение только `rvalue`, т.е. оно находится в правой части выражения. Ну например, мы можем написать так: `a = (b = c)`, но не можем написать так:

`(a = b) = c`, что по сути своей логично, но кажется очень надуманной ситуацией

У метода

```
int getSize() const;
```

позволяет защитить нас от изменения объекта, от которого метод вызван.

2. Теперь поговорим о такой прекрасной вещи, как **enum**, которая в плюсах реализована так себе, и её все ненавидят. Это эnumерации – списки значений, у которых есть имена. Проблемы заключаются сразу в том, что значения эти только целочисленные и это глобальные переменные.

```
enum key_code{shift_code, z_code, x_code = 10, c_code, v_code}.
```

Мы можем делать переменные, которые должны будут принимать по логике только эти значения. Здесь у `'shift_code'` будет 0, у `'z_code'` 1, у `'x'` понятно, дальше 11

и 12. Почему так? Ну каждому последующему даётся значение на 1 больше предыдущего, а также можно задавать значения вручную. Проблемы у этого такие же, как и у глобальных переменных, а также могут получиться одинаковые значения у нескольких переменных, мы ожидаем, что получив 'z_code', пользователь нажал на 'z', а на самом деле он мог нажать на 'f', у которой мы случайно указали такой же код. Плюс чтобы сделать значения не интовые, а строковые – в дело подключаются костыли всемогущие. Нам практик показывал, как с помощью макросов и генераторов получилось сделать более менее адекватное подобие эnumерации со строковыми значениями, но это вышел довольно адский и плохо читаемый код. В с++ появились также enum классы, но о них мы формально не говорили. Значения в эnumерациях, если я не ошибаюсь, это long long или что-то больше, лень гуглить.

16. Элементы проектирования

- **Автотесты**

Чтобы избежать появления ошибок во время изменения программы, особенно, когда над программой работает сразу несколько человек, люди придумали договорённость писать автотесты - набор функций, которые вызывают функции из программы и проверяют, что результат соответствует ожиданию. Притом тесты делают это автоматически, на выходе лишь говоря, что всё прошло успешно или нет, и где оно споткнулось, чтобы программисту не терять время на ввод тестов руками.

Существует пара религиозных фанатизмов по тестам – это unit тесты и test driven код. Первое – это идеология того, что тесты должны быть у каждой программы и должны полностью покрывать все строки кода, каждую функцию программы. Второе ещё более странное, это когда люди пишут сначала тесты, т.е. ожидаемую работу программы, а уже потом пишут сам код и реализации этих функций.

Автотесты также являются хорошей документацией программы, ибо они отчётливо показывают поведение функций и ожидаемый от них результат.

Пример.

- **Декомпозиция программы (Model, View)**

Есть разные шаблоны проектирования программ, например, описанные в одной популярной книжке Design patterns. Бездумное использование шаблонов может привести к бессмысленному усложнению программы. Декомпозиция программы – разбиение её на разные более простые и маленькие блоки, пусть и связанные, каждый из которых выполняет только одну свою задачу. Связь может быть очень спутанной между блоками, но в основном стараются, чтобы зависимости были попроще. Например, если мы пишем крестики-нолики – можно разделить всё на model – наше игровое поле, на котором будет просчитываться логика и храниться результаты, а также на View – графическое отображение игры. При этом Model не обязательно знать, что происходит с View, зато View нужно работать с доской. Model получается более универсальным, мы для одного Model можем написать разные, зависящие от него, блоки, отвечающие за вывод игры на компьютере, телефоне, в консоли или с графическим интерфейсом.