C++ Модули 1-2

Тарабонда Герман C++

Содержание

1 Программа, состоящая из нескольких файлов

Заголовочные файлы

Иногда стоит разбивать проект на различные файлы. Есть определенное количество плюсов в этом. Во-первых, если были сделаны какие-то исправления, то после нужно скомпилировать только этот файл заново, а не все сразу. Во-вторых, разбиение на файлы структурирует код. В заголовочных файлах пишется ОБЪЯВЛЕНИЕ функций, а ОПРЕДЕЛЕНИЕ функций не помещается без явной необходимости в заголовочные файлы.

Пример объявление функции в заголовочном файле:

```
#ifndef _HEADER_H_ // Здесь все по стандарту C
#define _HEADER_H_
// пишем наши объявления функций
#endif
```

Мы пишем все эти условия, чтобы при подключении `first.h` в `second.h`, а `second.h` в `first.h` не произошло циклической зависимости.

Этот способом можно написать, но он не православный и не входит в стандарт:

```
#pragma once
// котим
```

А в нашем сишном файле мы подключаем заголовочные файлы вот так:

```
#include <header>  // подключает хедеры из стандратной библиотеки
#include "header" // подключает собственный хедер
```

Компиляция и линковка

Рассмотрим компиляцию программы `proc.c`. Сначала он порходит препроцессинг, то есть происходит:

- Замена комментариев пустыми строками
- Текстовое включение файлов – `#include`
- Макроподстановки – `#define`
- Обработка директив условной компиляции – `#if`, `#ifdef`, `#elif`, `#else`, `#endif`

Потом происходит компиляция, в результате которого компилируется объектный код. Сначала происходит трансляция в ассемблерный код, а затем из него в машинный.

Далее происходит этап линковки, когда связываются все объектные файлы. При этом может произойти ошибка связки, если функция объявлена, но не определена.

Объявления и определения функций

Объявление (declaration) – вводит имя, возможно, не определяя деталей. Например, ниже перечислены объявления:

```
void sum(int a, int b);
```

Определение (definition) – это объявление, дополнительно определяющее детали, необходимые компилятору. Из перечисленных выше объявлений, определениями являются только два:

```
void sum(int a, int b){
   return a + b;
};
```

Утилита make

Часто у нас получается очень большой проект и нам не хочется компилировать и ликовать все каждый раз занова, поэтому у нас есть волшебная утилита make. Идея проста: есть цель и есть зависимости этой цели. Если зависимость меняется, то цель тоже должна поменятся. Сначала в утилете make выполняется первая команда. Обычно там пишется что-то вроде:

```
all: main.o
```

А в общем это выглядит как-то так:

```
< цель >: < зависимость >
   < команды > // Здесь должен быть tab
```

Чтобы вызвалось по умолчанию makefile нужно в консоле написать `make`.

2 Указатели, массивы, ссылки

NULL, nullptr

Согласно cppreference.com `NULL` определяет константу для нулевого указателя, которая может быть типом целового числа и равна 0. Оно означает, что указатель не указывает в осмысленное место.

В C++11 появился `nullptr`, который носит те же функции, что и `NULL`, но имеет собственный тип `std::nullptr_t`.

Применение указателей и ссылок

Лучше сначала сказать что такое указатель, ссылка и с чем его едят (кушац хочу((((((.

Ссылка – механизм языка программирования (C++), позволяющий привязать имя к значению. В частности, ссылка позволяет дать дополнительное имя переменной и переда-

вать в функции сами переменные, а не значения переменных. Оформляется добавлением знака &.

Указатель(pointer) – число, адрес (то есть смещение от начала) соответствующего элемента в памяти.

Ссылка и указатель – это не одно и то же. Ссылка не можнт быть неинициализированной, у ссылки нет нулевого значения и ссылки не могут быть переинициализированны. Примеры:

```
int a = 42;
int *p = &a; // & - берем адрес a
int b = *p; // взять значение по адресу p ( разыменовать )
printf("%p", p); // выводим адрес

int *p; // OK
int &l; // не OK

int *p = 0; // OK
int &l = 0; // не OK

int a = 10, b = 20;
int *p = &a;
p = &a;
int &l = a;
l = b; // не православно
```

Полезно применять указатели в функциях (нпример вот эта функция swap):

```
void swap(int *a, int *b){
   int tmp = *a;
   *a = *b;
   *b = tmp;
}
```

Кстати, `sizeof(int*) == sizeof(double*)`.

Арифметика указателей

С ними можно работать как с int-ами, то есть делаьб ++ и т. д., при этом указатель сдвинется на нужное количество ячеек.

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

Также есть замечательная штука как массивы. Это тоже указатели, и обращение к ним - это своего рода синтаксический сахар.

```
int arr[100];
arr[5] = 55; // == *(arr + 5)
```

```
// arr[i] == i[arr] == *(arr + i)
```

Константность указателей

Указатель на константное значение — это неконстантный указатель, который указывает на неизменное значение. Для объявления указателя на константное значение, используется ключевое слово const перед типом данных:

```
const int value = 7;
const int *ptr = &value; // здесь всё ок: ptr - это неконстантный указатель, который
    указывает на "const int"
*ptr = 8; // нельзя, мы не можем изменить константное значение
```

Также есть **константный указатель** — это указатель, значение которого не может быть изменено после инициализации. Для объявления константного указателя используется ключевое слово const между звёздочкой и именем указателя:

```
int value1 = 7;
int value2 = 8;

int * const ptr = &value1; // ок: константный указатель инициализирован адресом value1
ptr = &value2; // не ок: после инициализации константный указатель не может быть
    изменён
```

3 Три вида памяти. Работа с кучей на C

Глобальная/статическая память, стек, куча

Глобальная память (static variables) – это память, выделяющаяся до запуска программы, кусок памяти определённого размера, в котором хранятся константы и всё, что не будет менять размеры в течении использования. Код для выделения и освобождения генерирует компилятор. Память выделяется при загрузке в память, освобождается при завершении программы. Глобальные инициализируются в каком-то порядке, статические – при входе в функцию. Глобальные переменные инициализируется нулем.

Стек – это динамическая память, которая используется для хранения вызванных функций, локальных переменных функций, параметров функций. Код для выделения и освобождения генерирует компилятор. Выделяется память при входе в функцию, а освобождается при выходе.

Куча – это динамическая память, в которой могут хранится большие объекты. Выделение и очищение памяти пишет сам программист. При выделении пееременных на стеке они заполнены "мусором а на кучи оно инициализируется нулями (если оно не было сделано malloc'ом).

extern-переменные

Классно расказывается тутъ. Если кратко: extern позволяет объявить переменные в одном файле и при подключении этого файла к остальным мы сможем спокойно пользоваться переменной объявленное таким способом. Если такого не сделать, то может произойти что-то такое (главное звук).

sizeof для переменных и типов

`sizeof` - это унарный оператор, возвращающий длину в байтах переменной или типа, помещенных в скобки.

```
float f;
printf("%f ", sizeof f);  // выведится 4
printf("%d", sizeof(int)); // выведится 4
```

malloc/calloc/realloc/free

Сишные фунции из библиотеки из stdlib, которые нужны для выделения памяти:

```
malloc(size_t x) // выделяет память (x байтов) не инициализирует и оставляет мусор
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 освобождать
```

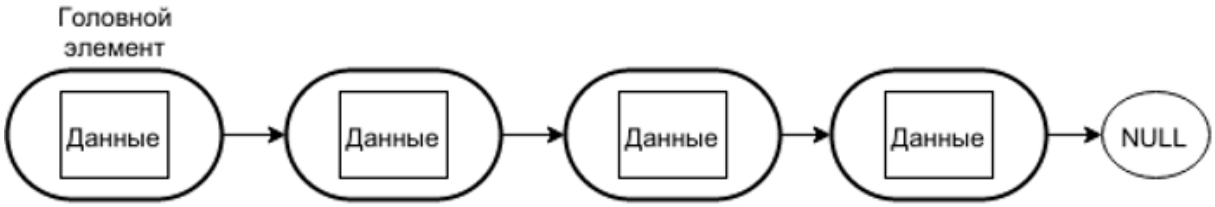
void*

В C существует особый тип указателей – указатели типа void или пустые указатели. Эти указатели используются в том случае, когда тип переменной не известен. Так как void не имеет типа, то к нему не применима операция разадресации (взятие содержимого) и адресная арифметика, так как неизвестно представление данных. Тем не менее, если мы работаем с указателем типа void, то нам доступны операции сравнения. В итоге можно приводить указатели к разным типам, таким образом предавать в функцию какие угодно типы.

4 Структуры. Неинтрузивный связный список на C

Неинтрузивная реализация

Список - это блоки данных, которые ссылаются на следующий такой же блок. В случае двусвязного ссылаются 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);
}
```

Расположение полей в памяти, выравнивание

Вообще структуры в C – это совокупность перемнных, объединенных одним именем.

В структурах происходит выравнивание данных, чтобы процессоры работали быстрее. К примеру, у нас есть такой код:

```
struct cipher {
   char letter; // 1 байт
   int num; // 4  байта
}
```

В данном случае размер структуры будет 8 байт, то есть произойдет дополнение нулями:

```
[ char ] [ пусто ] [ пусто ] [ пусто ] [ int занимает 4 байта ]
```

Чтобы избавиься от выравнивания можно использовать:

```
#pragma pack(push, 1) // выравнивание по 1 байту
// структура
#pragma pack(pop)
```

typedef

Чтобы печатать меньше можно написать `typedef`

```
typedef long long ll;
typedef struct cipher{
// котим
} cipher_t; // так писать православно
```

5 Структуры. Интрузивный связный список на C

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); // нода с данными превращается в интрузивную
}
```

Расположение полей в памяти, выравнивание

Вообще структуры в C – это совокупность перемнных, объединенных одним именем.

В структурах происходит выравнивание данных, чтобы процессоры работали быстрее. К примеру, у нас есть такой код:

```
struct cipher {
   char letter; // 1 байт
   int num; // 4  байта
}
```

В данном случае размер структуры будет 8 байт, то есть произойдет дополнение нулями:

```
[ char ] [ пусто ] [ пусто ] [ пусто ] [ int занимает 4 байта ]
```

Чтобы избавиться от выравнивания можно использовать:

```
#pragma pack(push, 1) // выравнивание по 1 байту
// структура
#pragma pack(pop)
```

typedef

Чтобы печатать меньше можно написать `typedef`

```
typedef long long ll;
typedef struct cipher{
// котим
} cipher_t; // так писать православно
```

6 Функции. Указатели на функции

Как происходит вызов функции

Сначала на стеке выделяется память (кадр, frame). Там храним локальные перемнные, параметров, возвращаемое значение, адрес, куда нужно потом вернуться. Затем параметры копируются в свои места, и вызывается функция. После выполнении функции сохраняется возвращаемое значение. И в конце возвращается в то место, откуда была вызвана.

Реализация сортировки

```
void swap(void *left , void *right , site_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);
}
```

Пользуемся так:

```
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);
```

7 Обзор стандартной библиотеки C

string.h

`memcpy` копирует первые count элементов из source в destination

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

`memcmp` сравнивает куски в памяти prt1 и ptr2, если одинаковы, то возвращает 0, если ptr1 больше, то > 0, иначе <0

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

`strcpy` копирует из str2 в str1.

```
char* strcpy(char* str1, const char* str2);
```

`strcmp` сравнивает ptr1 и ptr2, если одинаковы, то возвращает 0, если ptr1 больше, то > 0, иначе <0

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

`strcat` что-то склеивает и возвращает в `destination` `destination + source`

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

`strstr` ищет подстроку в строке и возвращает указатель на первое вхождение str2. Если не нашли, то возвращаем `NULL`

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

```
/* Пример */
strstr''('its a cpp, ''baby, '"'cpp); //вернет указатель на ''c в первой строке
strstr''('its a cpp, ''baby, '"'java); //вернет NULL
```

`strchr` поиск символа в строке, все тоже самое, что в `strstr`

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

`strtok` делит строку разделителями `delimetrs`. Еще про strtok важно знать ещё, что он внутри хранит как static переменную позицию там какую-то, то есть нельзя работать с двумя строками параллельно с помочщью strtok.

```
char* strtok ( char * str, const char * delimiters );
/* Пример */
#include <iostream>
#include <cstring>

int main ()
{
  char str[] = "Особенности национальной рыбалки - художественный, комедийный фильм.";

  std::cout << "Разделение строки "" << str << "" на лексемы:n";
  char * pch = strtok (str," ,.-"); // во втором параметре указаны разделитель
    пробел(, запятая, точка, тире)

  while (pch != NULL)                          // пока есть лексемы
  {
      std::cout << pch  << "n";
      pch = strtok (NULL, " ,.-");
  }
  return 0;
}
```

stdlib.h

`atoi` конвертирует строку, на которую указывает параметр str, в величину типа int. Строка должна содержать корректную запись целого числа. В противном случае возвращается 0.

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

`strtoll` преобразует строку, доступную по указателю str в целочисленное значение. Функция устанавливает str_end на указатель, который указывает на следующий за символом, который использовался в записи распознанного числа. Если str_end равен NULL, то он игнорируется. Если str пустой или не содержит ожидаемой записи числа, то никакого преобразования не будет сделано, и (если str_end не NULL) значение str будет храниться по указателю str_end.

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

srand/rand генерирует псевдослучайные числа. 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.

```
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()
```

qsort быстрая сортиров очка. base - сам массив, num - количество элеметов, size - размер элементов

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

Строки в стиле Си (null-terminated strings)

Строка C-style — это простой массив символов, который использует нуль-терминатор. Нуль-терминатор — это специальный символ (код ASCII — 0), используемый для обозначения конца строки. Строка C-style ещё называется «нуль-терминированной строкой».

```
char mystring[] = "string";
```

В конце добавляется '\0'. Тоже самое что 0, но не '0'

Контракты функций: кто выделяет/освобождает буфер, какого должен быть размера, какие требования к аргументам, где хранится внутреннее состояние функций

Для того, чтобы пользователь функции знал, что зачем нужно, нужно писать контракты функции, чтобы все корректно выполнялось. Если не выполнять данные контракты, то будет UB. Скорее всего в данном пункте подразумевалось разъяснить контракты вышеописанных функций.

Все выделяет пользователь, освобождает тоже он для strcpy юзер сам должен выделить буфер, в который можно будет записать результирующую строку, такого размера, чтобы туда поместилось все что нужно.

Внутреннее состояние функций,видимо, в статических переменных нужной функции. Например, как в strtok. При этом не должна меняться строка, на которой вызываешь strtok, иначе UB, хз верно ли.

strcpy копирует строку. Ему нужен только указатель на начало и дальше он дойдет до \0 символа. memcpy принимает границы области и копирует их.

8 Ввод-вывод на C. Текстовые файлы

FILE, fopen, fclose, rt/wt

`FILE` – это 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.

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, sscanf, fgets

`printf` загружает данные из данного места, преобразует их в строку символов эквиваленты и записывает результаты в stdout.

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

`scanf` читает данные из различных источников, интерпретирует его в соответствии с format и сохраняет результаты в stdin.

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

`fprintf` загружает данные из данного места, преобразует их в строку символов эквиваленты и записывает результаты в файл.

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

`fscanf` считывает данные из файла stream поток.

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

`sprintf` загружает данные из данного места, преобразует их в строку символов эквиваленты и записывает результаты в буффер.

```
int sprintf( char *buffer, const char *format, ... );
```

`sscanf` reads the data from null-terminated character string buffer.

```
int sscanf( const char *buffer, const char *format, ... );
```

`fgets` считывает символы из потока и сохраняет их в виде строки в параметр string до тех пор пока не наступит конец строки или пока не будет достигнут конец файла.

Символ новой строки прекращает работу функции fgets, но он считается допустимым символом, и поэтому он копируется в строку string.

Нулевой символ автоматически добавляется в строку str после прочитанных символов, чтобы сигнализировать о конце строки.

```
char *fgets(char *str, int num, FILE *stream);
```

Обработка ошибок, feof, ferror

`feof` проверяет 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);
```

`ferror` отслеживает 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);
```

9 Ввод-вывод на C. Бинарные файлы

FILE, fopen, fclose, rb/wb, буферизация

`FILE` – это 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.

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.

fread, fwrite, fseek, ftell, fflush

`fread` читает указанное количество объектов в массиве buffer из данного stream входного потока. Объекты, не интерпретируются в любом случае.

```
size_t fread( void *buffer, size_t size, size_t count, FILE *stream );
```

`fwrite` пишет count объектов в заданном массиве buffer к stream выходной поток. Объекты, не интерпретируются в любом случае.

```
int fwrite( const void *buffer, size_t size, size_t count, FILE *stream );
```

`fseek` устанавливает индикатор позиции файла для stream файл потока к величине, на которую указывает offset. Эта функция может быть использована для установки индика-

тора за фактический конец файла, однако, отрицательные значения положения не принимаются. возвращает 0 на успех, ненулевое значение в противном случае.

SEEK_SET Начало файла

SEEK_CUR Текущее положение файла

SEEK_END Конец файла

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

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

```
long ftell( FILE *stream );
```

`fflush` causes the output file stream to be synchronized with the actual contents of the file. If the given stream is of the input type, then the behavior of the function is undefined.

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

Обработка ошибок, feof, ferror

`feof` проверяет 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);
```

`ferror` отслеживает 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);
```

10 Классы и объекты

Инкапсуляция: private/public

Инкапсуляция – возможность скрытия реализации каких либо частей модуля или объекта от внешнего мира (от клиента). По умолчанию, в классе данные и методы приватные; они могут быть прочитаны и изменены только классом к которому принадлежат. В C++ доступно несколько спецификаторов, и они изменяют доступ к данным следующим образом:

- Публичные `public` данные доступны всем
- Защищенные `private` доступны только классу и дочерним классам

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: по умолчанию, member initialization list, default member initializer (C++11)

Конструктор, который не имеет параметров (или имеет параметры, все из которых имеют значения по умолчанию), называется конструктором по умолчанию. Пример:

```
class Biscuits{
public:
   Biscuits(){ // дада это конструктор по умолчанию
      flour = 1;
      eggs = 1;
   }
private:
   int flour;
   int eggs;
}
```

Можно задать значения по умолчанию:

```
class Biscuits{
public:
   Biscuits(){}
private:
   int flour = 1;
   int eggs = 1;
```

```
}
```

Можно еще сделать конструктор с помощью member initialization list, то есть вот так:

```
Biscuits::Biscuits(int flour, int eggs): flour(flour), eggs(eggs) {}
```

А еще можно сделать такую классную вещь, как default member initializer:

```
class Biscuits{
public:
   Biscuits(int flour=1, int eggs=1){}
private:
   int flour;
   int eggs;
}
```

Время жизни полей: порядок вызова конструкторов и деструкторов полей и конструктора/деструктора класса

Вообщя время жизни полей от конструктора до дестрктора, но важен порядок вызова конструкторов и деструкторов. В конструкторе мы должны сначала инициализировать родителей, затем наши поля, затем самого себя. С деструкторами поступаем в противоположном порядке.

C++11: =default, constructor chaining

Про `=default` написано было. В C++11 можно вызовать свой же конструктор, только не так:

```
Biscuits(int flour, int eggs) {}
Bisuits(int eggs){
   Biscuits(5, eggs); // не православно
}
```

а так:

```
Biscuits(int flour, int eggs) {}
Bisuits(int eggs): Biscuits(5, eggs) {}
```

Правило трёх, правило нуля

Если написали один из методов: деструктор, конструктор копий или `operator=`, то нужно дописать и остальные из этого списка. А правило нуля гласит, что не нужно писать эту фигню, а нужно пользоваться умными указателями.

11 Работа с кучей на C++

new/delete

Аналоги malloc/free в C++. new не совместим с free и все в таком духе.

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

Создание объектов в куче

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

Конструктор копий

Конструктор копирования — это особый тип конструктора, который используется для создания нового объекта через копирование существующего объекта.

```
class_name (const class_name o&) {
// тело конструктора
}
```

Инициализация возникает в трех случаях: когда один объект инициализирует другой, когда копия объекта передается в функцию и когда создается временный объект (обычно если он служит возвращаемым значением). Например, любая из следующих инструкций вызывает инициализацию:

```
myclass x = y; // инициализация
func x(); // передача параметрау
 = func (); // получение временного объекта
```

Если передаем не ссылочку, то произойдет бесконечная рекурсия ;-). Пример конструктора:

```
class Array {
...
Array ( const Array & a ) {
   size = a.size ;
   data = new int [ size ];
   for ( size_t i = 0; i < size ; i ++) data [ i ] = a.data [i ];
}
};
```

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); // свопаем 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 свопаем значения, а копия удаляется сама собой. Удобно.

Если мы не используем семантику copy-swap, то обязательно нужно проверить:

```
if (this == &obj)
   return *this;
```

C++11: =delete

Пишем `=delete`, чтобы не делать ему конструктор по умолчанию:

```
Biscuits() = delete;
```

Используем это, а не кладем в `private`, чтобы заблокировать `friend` и все в этом духе.

Правило трёх, правило нуля

Если написали один из методов: деструктор, конструктор копий или `operator=`, то нужно дописать и остальные из этого списка. А правило нуля гласит, что не нужно писать эту фигню, а нужно пользоваться умными указателями.

12 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.

(Не)явные преобразования указателей

Памагити

protected

В то время, как private дает доступ только изнутри данного класса, protected позволяет видеть свое содержимое еще и всем классам-наследникам данного класса. protectedинкапсуляция 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)

Мы можем перегружать методы. Например вот такой кусочек кода:

```
class Person {
   string name() const { return name_; }
   ...
};
class Professor : Person {
   string name() const {
      return "Prof. " + Person::name();
   }
   ...
};
```

```
Professor pr("Stroustrup");
cout << pr.name() << endl; // Prof. Stroustrup
Person * p = &pr;
cout << p->name() << endl; // Stroustrup
```

Если мы добавим слово `virtual`, то функция, которую мы вызовем будет зависеть не от типа указателя, а от объекта, на который ссылается данный указатель.

```
class Person {
   virtual string name() const { return name_; }
   ...
};
class Professor : Person {
   string name() const {
      return "Prof. " + Person::name();
   }
   ...
};
```

```
Professor pr("Stroustrup");
cout << pr.name() << endl; // Prof. Stroustrup
Person * p = &pr;
cout << p->name() << endl; // Prof. Stroustrup
```

Еще стоило бы сказать, что не бывает виртуального конструктора, но бывает виртувльные деструтор. То есть если мы напишем `virtual` перед деструктором, то он вызовет тот деструктор, на который ссылается данный указатель. Пример:

```
class Person {
   virtual ~Person() {}
   ...
};
class Student : Person {
   string uni_;
   ...
};
```

```
Person *p = new Student("Alex", 21,"Durka");
...
delete p; //  Если бы не virtual, то была бы утечка памяти
```

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

По умолчанию в C++ стоит статическое связывание. Если есть хотя бы одна виртуальная функция, то для него используется динамическое связывание. На идейном уровне, в одном столбце стоит название, в другом адрес. Если линковка была статическая - при исполнении будет сразу комманда "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 нам нужно, чтобы выполнялись методы одного и методы другого класса, то статическое связывание нам не помогло бы.

13 Умные указатели

scoped_ptr

Мы юзеры, которые не хотят думать о памяти, поэтому создадим указатели, которые сами освобождают память, когда надо.

Пример использования:

```
void f () { // мы не так хотим
```

```
   Object * p = new Object (...);
   p - > save_to_file ( " out . txt " );
   delete p ;
}

void f () { // а так
   scoped_ptr p ( new Object (...));
   p - > save_to_file ( " out . txt " );
}
```

Реализация:

```
// scoped_ptr.h
class scoped_ptr {
private :
   Object * myPointer ;
public :
   scoped_ptr ( Object * ptr );
   ~ scoped_ptr ();
public :
   Object * ptr ();
   Object & operator *() const ;
   Object * operator - >() const ;
   bool isNull () const ;

private :
   scoped_ptr ( const scoped_ptr & p );
   const scoped_ptr & operator =( const scoped_ptr & p );
}
```

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

```
}
```

Этот указатель нельзя копировать и передавать в функции.

unique_ptr

Этот указатель можно перемещать и передавать в функцию.

```
unique_ptr :: unique_ptr ( unique_ptr & p ) {
   myPointer = p . myPointer ;
   p . myPointer = 0;
}
unique_ptr & unique_ptr :: operator =( unique_ptr & p ) {
   if ( this != & p ) {
      if ( myPointer != 0) {
         delete myPointer ;
      }
      myPointer = p . myPointer ;
      p . myPointer = 0;
   }
   return * this ;
}
```

shared_ptr

std::shared_ptr – умный указатель, с разделяемым владением объектом через его указатель. Несколько указателей shared_ptr могут владеть одним и тем же объектом; объект будет уничтожен, когда последний shared_ptr, указывающий на него, будет уничтожен или сброшен. Объект уничтожается с использованием delete-expression или с использованием пользовательской функции удаления объекта, переданной в конструктор shared_ptr.

14 Перегрузка операторов

Бинарные и унарные (префиксный и постфиксный)

```
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 строчку матрицы,
            return _data[i]; // в которую можно присвоить
    }
    const int* operator[](std::size_t i) const { // возвращает строчку матрицы,
        return _data[i]; // из которой можно считать
    }
    ...
};
```

Возвращаемое значение

Важно правильно поставить возвращаемое значение: если оператор объявлен вне класса (типа + или тд), то возвращаем сам класс, иначе ссылку на этот класс.

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]++;
}
```

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
```

15 Сложные вызовы функций

Передача параметров в функции и возврат из функций: по значению (по копии) и по ссылке, массивы, структуры, указатели

TODO

Неявные преобразования аргументов и переменных

Если у нас есть такая функция:

```
int Durka(int number);
```

А пихать туда будем `double`, то у нас `double` преобразуется к `int`.

Отсутствие неявного преобразования объекта при вызове методов: в a.foo() к a не применяются неявные преобразования

К `this` неявные преобразования сделать нельзя. Так например, этот код нескомпилиться, если у нас есть `BigInt::operator+(const BigInt&)` и конструктор неявного преобразования `BigInt(int)`:

```
2 + BigInt(10); // не скомпилируется
```

Это нескомпилится, так как 2 не преобразуется к `BigInt` и у нее не будет вышеописанного метода. А наоброт все будет ок, так как 2 преобразуется к `BigInt`.

explicit-конструкторы

Можно запретить создание автоматичекого конвертирующего конструктора:

```
class FruitBasket {
public:      explicit FruitBasket(int);
```

```
}

// ...

FruitBasket fb = 5;              // Не прокатит!
FruitBasket fb = FruitBasket(5); // Прокатит! Сработает копирующий конструктор,
    который тоже создается автоматически
```

Время жизни временных объектов в выражениях

TODO

16 Ключевые слова extern, static, inline

extern у переменных

Классно расказывается тутъ. Если кратко: extern позволяет объявить переменные в одном файле и при подключении этого файла к остальным мы сможем спокойно пользоваться переменной объявленное таким способом. Если такого не сделать, то может произойти что-то такое (главное звук).

static у переменных и функций

Если поставить `static` только у глобальной переменной, то эту переменную можно будет использовать только в этом файле и другие переменные с тем же именем из других файлов не повлияют на нее.

Если поставить `static` у локальной переменной, то она будет скомпилированна только 1 раз. Пример:

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

Если мы хотим запретить вызывать функцию из других файлов, то пишем `static`.

static у полей и методов

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.

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.

inline у функций

Если сказать функции, что она `inline`, то оптимизатор попытается напрямую вставить этот код. Тогда программа будет работать быстрее. Функции, опредленные внутри класса - `inline`. Также можно определить функцию в a.h, а потом вызвать её в двух разных файлах и получить multiple defenition. Если дописать inline то будет ок.

17 Перегрузка функций, константность

Ключевое слово const (C/C++) у переменных, аргументов, полей, методов (cv-qualification), указателей

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. 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 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. `const` у метода позволяет защитить нас от изменения объекта, от которого метод вызван.

Ключевое слово mutable

Основное применение `mutable` – кеширование данных. У нас может быть какая-то долгая операция и мы не хотим ее выполнять каждый раз, когда она не меняется, поэтому мы ее можем закешировать.

```
class Matrix {
   ...
   double get ( size_t i , size_t j ) const ;
   void set ( size_t i , size_t j , double value );
   double determinant () const ;
};
```

Перегрузка функций

Перегрузка функций — это особенность в C++, которая позволяет определять несколько функций с одним и тем же именем, но с разными параметрами. То есть мы сможем сделать так:

```
int ask(int a, int b);
double ask(double a, double b);
```

Но не можем сделать так:

```
int ask(int a, int b);
double ask(int a, int b);
```

Это ошибка компиляции.

Параметры функций по умолчанию

Мы можем хайпануть и сделать так:

```
void printValues(int a, int b=15);
```

18 Наследование: детали

Расположение подобъектов в памяти

Сначала идет, то что у родителей, потом что у наследников.

Сортировка и структуры данных C vs ООП

Говорим объекту “сделай что-то”, более крутой подход, чем в C

```
class Comparable {
   //or virtual bool operator<(const Comparable *v) = 0 const;
   virtual int compare ( const Comparable * v) = 0 const ;
```

```
};
void nsort ( Comparable ** m , size_t size ){
   ...
   m [ i ] =
}
class Point : public Comparable {
private :
   int x , y ;
public :
   virtual int compare ( const Comparable * v) const {...};
};
Point ** points ;
//allocation: points = new ... and for() { points[i] = new }
nsort ( points , N );
```

private/protected наследование

У нас есть класс `A`:

```
class A {
public :
   int x;
protected :
   int y;
private :
   int z;
};
```

И такие наследники, в которых происходит что-то подобное:

```
class B : public A {
   // x is public
   // y is protected
   // z is not accessible from B
};
class C : protected A {
   // x is protected
   // y is protected
   // z is not accessible from C
};
class D : private A { // 'private' is default for classes
   // x is private
   // y is private
   // z is not accessible from D
};
```

Иногда у поля класса нужно вызвать `protected` метод или перекрыть виртуальную

функцию. В этом случае можно сделать так:

```
class Engine {
protected :
   void maintenanceCheck () { ... };
   ...
};
class Car : private Engine {
   void reset () {
      maintenanceCheck ();
   }
};
```

C+11: final у метода и класса, override

`final` запрещает в классах-наследниках переопределение определенных методов. Данный спецификатор также позволяет запрещать наследование от некоторого класса.

А `override` явно показывает "перекрывается"функция или нет.

```
class Base {
public :
   virtual void f (int );
   virtual int g () const ;
   void h (int);
};

class Derived : public Base {
public :
   void f (int) override ; // ok
   int g () override ; // compilation error
   void h (int) override ; // compilation error
};
```