День 1-ый. Ответы на вопросы 22.08.2023

Заголовок
Подзаголовок
Обычный текст
`Код`

- Важно
- Вопрос
- Внезапные идеи для to-do
- Определение
- Что-то отстраненное от темы
- Выполнить

- В *printf* тип *float* автоматически расширяется до *double* только если передается не переменная, а фактическое значением => для передачи значения переменной используем %lf или %lg ии %le для *double*

- Если не указывать амперсант в *scanf("...", a)*, то *scanf* создаст свою переменную *a* и даст ей значение (а после завершение оператора, эта переменная удалится) , а начальная переменная не поменяется
В таком случае *scanf* ничего не введет
Пример с Ромой: для того чтобы подарить Роме кружку, нужно не взять картинку комнаты Ромы и пририсовать туда кружку, а узнать, где эта комната находится (узнать АДРЕС комнаты) и принести туда кружку

- В записи *printf("...", &a)* оператор вернет адрес переменной а в шестнадцатиричной сс

- После праты полезно прочитать кернигана и ритчи (а после кернигана и ритчи - прату)

- В какой момент через scanf выводится рандомное число?

- В ардуино нет входной точки в программу (мэйна)?

- Создать отдельную функцию для квадратки, а в мейне ее уже вызывать

- Выводить автора и дату проги при запуске программы

- Возможность объявлять переменные не только в начале функции - одна из причин использовать g++

Разработка "сверху вниз":
- Пишем снчала главную программму
- Считаем, что "все уже написано"
- После этого пишем сами функции, при необходимости повторяя процесс

- Есть смысл инициализировать переменные не нулем, а NaN (Not a Number). NaN является "ядовитым" числом, которое вызывает ошибки при всех вычислениях. Это позволит быстро заметить проблемы. NaN работает только для вещественных чисел, а для int такое ядовитое число нужно придумывать
Существуют функции:
isnan() -- проверяет является ли число NaN
isfinite() -- проверяет является ли число бесконечностью или NaN

- assert(условие) - перехватчик ошибок

- Doxygen - прога которая автоматически генерирует документацию, используя уже написанные в коде комментарии (в ТХBook есть глава про это)

- Про сравнение действительного числа с нулем:

Тогда ты можешь сравнить модуль своего коэффициента (подключаешь библиотеку math.h-> модуль числа с плавающей точкой это оператор fabs(<число>)) с числом, очень близким к нулю>
Например, я делаю так:
#define DEFFERENCE 0.000001 // тут задаю константу "разница"
И потом сравниваю не вот так: a == 0 (a - число с плавающей точкой)
А вот так: fabs(a) < DIFFERENCE
Вообще это в прате написано, так что идея не новая, вотс

Источник <https://vk.com/im?sel=c120&sel=c121> =>
fabs(a) < DIFFERENCE аналогично a==0

День 2-ой. Массив коэффицентов для квадратки,
введение в Unit-тесты 23.08.2023

Заголовок

Подзаголовок

Обычный текст

```
Код
```

Передача в функцию массив коэффициентов

Реализация функции для решения квадратного уравнения:

```
int SolveSquare(double a, double b, double c, double *x1, *x2) {}
```

Слишком много аргументов. При повышении степени уравнения, аргументов становится еще больше.

Перейдем в идеальный мир. Хочу так:

```
int SolveSquare(double coffs... , double roots...)
{
    roots0 = ...
    coffs1 = ...
}
```

Перейдем в реальный мир. Конструкция, которая нам удобна, называется массивом.

```
int SolveSquare(double coffs[] , double roots[])
{
    roots[0] = ...
    coffs[1] = ...
}

int main()
{
    double coffs[3] = {0}; //все элементы - нули
    double coffs[3] = {}; //то же самое
}
```

Если массив двумерный, можно ли объявлять начальные значения? Ответ:

```
double coffs[3][5] = { {}, //строка из нулей
                       {5} //строка с первым элементом пятеркой, дальше нули
                        } //остальные строки нулевые

int main()
{
    double coffs[3] = {};
    double roots[2] = {};
    InputCoffs(coffs, 3); //функция для ввода коэффицентов
                          // в ней мы 3 раза через цикл вызовем ввод одного числа
    SolveSquare(coffs, 3, roots, 2) // 3 - число коэффицентов, 2 - число корней

    PrintRoots(roots, nRoot)
}
```

Есть смысл создать для ввода отдельную функцию InputCoffs

Также есть смысл сделать отдельную функцию для вывода массива корней

Зачем мы передаем массив корней определенной длины, если еще не знаем, какое кол-во корней выйдет? Есть смысл создать массив roots прямо в функции SolveSquare, а потом передавать его в main.
Однако возвращать копии массивов - долго. Поэтому в си так не делают. Делать указатель на массив, создающийся в SolveSquare, нельзя. Потому что после завершения SolveSquare массив удалится и указатель станет неактуальным.

Поэтому создаем roots в функции main, и уже из функции SolveSquare менять массив с помощью указателя.

Так, массив roots будет жить вплоть до завершения main'a, то есть до самого выхода из программы.

Еще есть смысл написать отдельную функцию для нахождения длины массива и вызывать ее в

Unit-тесты

Реализация функции для решения квадратного уравнения:

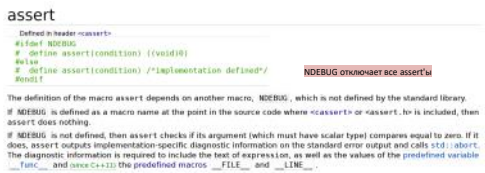
```
int TestOne(double a, double b, double c, double x1ref, double x2ref, int nRootsref)
{
    double x1 = 0, x2 = 0;
    int nRoots = SolveSquare(a,b,c,&x1,&x2);
    if (x1 != x1ref || x2 != x2ref || nRoots != nRootsref) {printf("ERR"); return 0;}
    else {printf("OK"); return 1;}
}

int TestAll()
{
    int nOk = 0;
    nOk += TestOne(1, 0, -4, -2, 2, 2);
    ...
    return nOk;
}
```

передавать в функцию теста массивы (и не забыть добавить assert() если размеры массивов не совпадают)

Что если printf выводит -0? Ответ:
Создать функцию которая при приближении к нулю выводит не само число, а четко 0

- Важно
- Вопрос
- Внезапные идеи для to-do
- Определение
- Что-то отстраненное от темы
- Выполнить

Технология защитного программирования
Defensive programming (**защитное**, безопасное **программирование**) — принцип разработки ПО, при котором разработчики пытаются учесть все возможные ошибки и сбои, максимально изолировать их и при возможности восстановить работоспособность программы в случае неполадок
Существует такая практика: расписывать assert'ы еще до алгоритма



Файл по типу "config,h" - файл, содержащий глобальные настройки, глобальные переменные, определяющие, например, режимы работы программы

В switch можно не ставить break'и для выполнения одного и того же блока действий для разных значений кейса

В буфере ввода сохраняются только символы ASCII. Для того чтоб считать нажатия клавиш типа shift, alt нужно обрщаться напрямую к драйверу клавиатуры

Существует какой-то способ очистки буфера через scanf("%*s", ...) НО этот способ неэффективенН

getchar() возвращает 257+1 символ, а char содержит только 256 значений => результат getchar() стоит записывать в int, а не в char. В случае с char возникнут проблемы с EOF

Есть какая то общепринятая точность для сравнения числа с нулем?
Ответ: нет, в больших проектах нужная точность рассчитывается с помощью матана

exit() и abort() - зло. Слишком жесткие методы окончания программы

В C++ стандартные функции ввода-вывода писались очень быстро накануне выхода стандарта. Были совершены некоторые архитектурные ошибки, которые нельзя было впоследствии изменить, тк все уже вышло в стандарт. В итоге функции ввода-выволв в C++ сильно проигрывют аналогичным функциям из C

Один из методов теорфиза - переходить в идеальный мир. Подобный метод можно использовать в программировании: выдумывать свои конструкции, которые было бы удобно применять, и только потом пытаться их реализовывать в соответствии с синтаксисом языка

Швабрами с досок стирают только в МФТИ. В остальных вузах не так. В МИФИ Дед даже делал туториал с фотографиями как использовать швабры

В питоне смешиваются понятия массивов и списков. Он создан для экспериментальных коротких программ. Для больших проектов питон не подходит (как минимум потому что он медленный).

Что предпочтительнее: использовать or и and или || и &&?
Ответ: лучше II и && по инженерному принципу наименьшего удивления

добавить защиту ввода от сильно больших чисел (больше или меньше +-37 степени)

в TXLib функция random() мяукает

Почему дед сказал аргумент командной строки --test а не test? Так лучше? Что означают --? Ответ:
Принято, что без тире идут только имена файлов. Параметры состоящие из одной буквы пишутся с одним минусом, параметры из большего количества букв - с двумя минусами

Массивы по типу char[] - та самая динамическая память о которой говорил дед?

EOF - не признак конца файла, а признак ошибки. EOF создан для того, чтобы передавать -1 в качестве "символ считать не удалось. ошибки выкидывать тебе не буду, вместо краша программы отправлю EOF"

Заголовок

Подзаголовок

Обычный текст

`Код`

- Важно
- Вопрос
- Внезапные идеи для to-do
- Определение
- Что-то отстраненное от темы
- Выполнить

Unit-тесты (продолжение): массивы, СТРУКТУРЫ

Рассмотрим наш уже написанный код

```
void TestAll()
{
    TestOne(0, 0, 0, 0, 0, SS_INF_ROOTS);
    TestOne(...);
}
```

Неудобно и некрасиво. Лучше вызывать TestOne в цикле, передавая туда разные аргументы, хранящиеся в массиве.
Можно создать один большой массив в котором будут последовательно храниться шестерки чисел.

```
double data[] = {0, 0, 0, 0, 0, SS_INF_ROOTS, ...};

for(int i = 0; i < nTests*6; i+=6)
    TestOne(data[i], data[i+1], ...);
```

Проблемы этого решения:
1. Сюда влезут только данные одного типа
2. Решение неявное, неочевидное и запутанное

Давайте создадим двумерный массив, чтобы решить вторую проблему

```
double data[5][6] = { {0, 0, 0, 0, 0, SS_INF_ROOTS},
                      {...},
                      {...} };

for(int i = 0; i < nTests; i++)
    TestOne(data[i][0], data[i][1], ...); //i - это номер теста или номер строки в data
```

Однако проблема с типами еще осталась
А еще теперь нужно запоминать в каком "столбце" какое значение содержится, так как они просто бессмысленно пронумерованы и не содержат значимых имен

"Нам нужна деревенская улица, а не городской нумерованный проспект"

Переходим к еще одному методу решения - использовать структуру

```
struct TestData
{
    double a, b, c;
    double x1, x2;
    int nRoots;
    const char name[10]; //или const char* name - так более культурно
}
```

Объявляя структуру, никакие переменные еще не созданы. Это всего лишь чертеж объекта, который мы сейчас создадим

```
TestData data1 = {0, 0, 0, 0, 0, SS_INF_ROOTS}; // или = {a=0, b=0, c=0, x1=0, ...}
TestOne(data1.a, data1.b, data.c, ...); //вызываем функцию тестирования с данными из структуры
```

Все равно некрасиво, ведь теперь мы сначала компонуем значения, а потом снова раскладываем на отдельные аргументы.
Сделаем так, чтобы TestOne принимала в качестве аргумента структуру

```
TestOne(data1); //но...
```

Здесь есть нюанс с памятью. В отличие от массивов (где передается только первый элемент) структура полностью копируется при передаче в функцию. Это влияет на время работы. Поэтому очень распространенная практика - передавать указатель на структуру

```
TestOne(&data1);
```

В функции TestOne:

```
void TestOne(TestData* data)
{
    double x1 = 0, x2 = 0;
    SolveSquare(data.a, data.b, data.c, ...);
    if (x1 != data.x1) ... //ошибок не возникает
}
```

Но появилась опасность: теперь, когда передается указатель на структуру, можно взять и поменять значение какой либо из переменных структуры.
Для решения этой проблемы можно делать TestData неизменяемой структурой

```
double TestData{...}     объявлять
const double TestData{...}
```

Теперь придется обращаться

```
void TestOne(const TestData* data)
{
    double x1 = 0, x2 = 0;
    SolveSquare((*data).a, (*data).b, (*data).c, ...); //или data->a, data->b, data->c
    ...
}
```

Массивы из структур - ОЧЕНЬ полезная вещь:

```
TestData allData[Ntests] = {{a=0,b=0,c=0, ...}, ..., ...};
for(int i = 0; i < nTests; i++)
    TestOne(&allData[i])
```

- Перегрузка функций доступна не во всех языках! Поэтому это является не универсальным инструментом
- Рекомендуется в момент возможных появлений ошибок сразу расписыватьassert'ы (параллельно с расписыванием алгоритмы), чтобы не забыть о них. И только после окончания работы с алгоритмом возвращаться kassert'ам и убирать из них те, которые никогда не сработают или заменять их на if'ы
- try и catch - очень долгие операторы. В Google и некоторых других компаниях исключения вообще запрещены. Их не используют и заменяют на обычные if'ы
  1. Исключения очень долгие
  2. Они сильно зависят от версий
  3. В них очень легко совершить очень тонкие незаметные ошибки

  Хоть и профессионалы должны уметь писать код и с исключениями, и без них, но большинство из них склоняются к тому что исключения избыточные и ненужные
- В памяти двумерный массив все равно хранится как одномерный, то есть содержит, к примеру, пять шестерок чисел друг за другом
- Как называются переменные внутри структуры? Поля? Переменные? Ответ:
  поля структуры или *члены структуры*
- Почему дед назвал использование стрелочки секретной информацией, если в Прате написано об этой стрелочке?Ответ:
  Бывает такое, что в c++ стрелочка и звездочка (в данном применении) имеют разный смысл и выполняются по разному. А еще, на старых компьютерах стрелочка может работать медленнее чем звездочка. В связи с этим, стоит знать и использовать оба варианта написания
- Все что идет после переменной имеет приоритет выше чем все что до переменной ⇒ в записи &allData[i] программа сначала обращается по квадратным скобкам к i, и только потом применяет &
- Мораль по структурам: всегда используем const и всегда обращаемся к структурам через указатели
- Структуры в структурах используются, но может быть оверкилл

Заголовок

Подзаголовок

Обычный текст

```
Код
```

- Важно
- Вопрос
- Внезапные идеи для to-do
- Определение
- Что-то отстраненное от темы
- Выполнить

Разделение программы на файлы и раздельная компиляция

Все в одном файле:

main.cpp

```
#include ...
#include ...
int input(...)

int main()
{
    input();
}

int input()
{...}
```

1. Разделяем один файл на main.cpp и header.h

input.h

```
#include ...
#include ...
int input(...);
```

main.cpp

```
#include "input.h"
...

int main()
{
    input();
}
```

Это аналогично тому, если в main.cpp вместо строчки #include "input.h" был бы вставлен весь текст файла input.h

В итоге препроцессор (часть компилятора) формирует такой файл

stdio.h →
input.h →
main.cpp →

main.cpp

```
int printf(...);

int input();
int input()
{}

int main()
{
    input();
}
```

НО:
Функция input() будет компилироваться каждый раз при запуске программы.
Это долго.

2. Теперь разделяем header.h еще на два файла - в одном прототипы функции, во втором сами функции

input.h

```
int input();
```

input.cpp

```
#include "input.h"

int input()
{... }
```

Теперь при наличии еще одного .cpp файла можно скомпилировать его в машинный код (файл с расширением .o или ОБЪЕКТНЫЙ файл) и теперь использовать каждый раз при запуске.

g++ -c input.cpp →

input.o

```
...
//машинный код
...
```

main.cpp

```
#include <stdio.h>

#include "input.h"

int main()
{
    input();
}
```

Что сделает препроцессор? →

main.cpp

```
int printf(...)

int input();

int main()
{
    input()
}
```

g++ -c main.cpp →

main.o

```
...
//машинный код
...
```

Этот процесс называется линковкой

g++ input.o main.o →

исполняемый файл .exe

```
...
...
...
```

Получив объектные файлы, начальные cpp при отсутствии изменений больше не компилируются

-o [имя файла]
можно приписать в конец любой команды для обозначения имени выходного файла

- Где находится прототип мейна?
- Препроцессор, вставляя код на место #include, делает это в файле .cpp ? Или уже на следующем уровне? Ответ:
  Разные компиляторы по-разному. Некоторые создают временный файл .tmp и там собирают код, а потом удаляют. Некоторые компиляторы даже хранят это все в памяти
- Почему если в этом способе чистить объектные файлы то время компиляции получится больше? Ответ:
  Как минимум потому что компилятор вызывается несколько раз на каждый файл
- Нужно самому следить за изменениями, и в случае изменений отдельно перекомпилировать объектные файлы или как? Ответ:
  да, в общем случае нужно самому компилировать заново cpp-файлы и создавать объектные. однако существует специальные программы make, которые сами за этим следят. make-файлы появляются как раз из-за них
- Стандартная библиотека (архив) состоит из сотен объектных файлов
- В чем различие между .h и .cpp?
- g++ draft.txt -x.cpp //компилятор поймет через команду -x, что текстовый файл нужно читать как .cpp

День 5-ый. Массивы, передача массивов в функции. Введение в строки 28.08.2023

Заголовок

Подзаголовок

Обычный текст

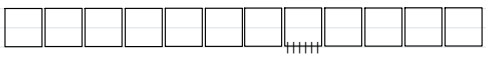
```
Код
```

- Важно
- Вопрос
- Внезапные идеи для to-do
- Определение
- Что-то отстраненное от темы
- Выполнить

Массивы, передача массивов через аргументы

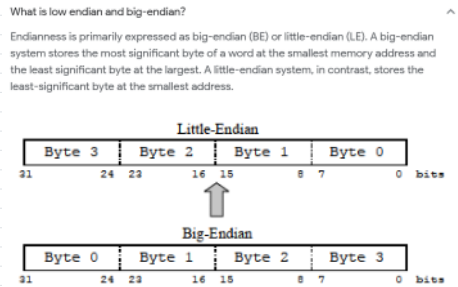
```
int data[10] = {1, 2, 3};
```

Оперативная память:



Хранение чисел:
1: 01 00 00 00
2: 02 00 00 00

Младшие разряды записываются слева, а не справа - это стандарт little (low) endian



Посчитаем сумму элементов массива

```
int sum = 0;
for (int i = 0; i < 10; i++)
    sum += data[i];
```

Добавим надежности с помощью assert'ов, чтобы не вылезти за границы массива

```
int sum = 0;
for (int i = 0; i < 10; i++)
{
    assert(0 <= i && i < 10);
    sum += data[i];
}
```

Избавимся от десятки

```
for (int i = 0; i < sizeof(data)/sizeof(data[0]); i++) ...
```

Но! sizeof не берет элемент, а просит у компилятора размер элемента. Обращение или доступ к элементу не происходит. Sizeof выполняется еще во время компиляции, поэтому такие записи являются корректными (но нежелательными):

```
sizeof(data)/sizeof(data[10])
sizeof(data)/sizeof(data[100])
sizeof(data)/sizeof(data[-100])
```

Цикл, идущий в обратную сторону:

```
for (int i = 9; i >= 0; i--) //важно следить за индексами, они должны находится в рамках [0; 10)
    sum += data[i];

for (int i = 10; i > 0; --i) //здесь тоже все правильно, потому что ДО входа в блок значение i понижается на единицу
    sum += data[i];
```

Наконец, давайте напишем функцию вычисляющую сумму элементов массива

```
int Sum (int data[10])
{
    int sum = 0;
    for (int i = 0; i < sizeof(data)/sizeof(data[0]); i++)
        sum += data[i];
    return sum;
}
```

Массивы передаются только с помощью указателей. А для того чтобы не иметь права изменять начальный массив, нужно добавлять const:

```
int Sum (const int data[10])
{
    int sum = 0;
    for (int i = 0; i < sizeof(data)/sizeof(data[0]); i++)
        sum += data[i];
    return sum;
}
```

это то же самое, что написать data[0] или data[-10] или data[] значение в квадратных скобках полностью игнорируется

важно: sizeof считает здесь не размер массива, а размер указателя. значение не то. здесь появляется ошибка.

Чтобы не писать каждый раз sizeof()/sizeof() можно это положить в define с параметрами

Обычно передают первый элемент массива и его длину

```
int Sum (const int data[], size_t size)
{
    int sum = 0;
    for (size_t i = 0; i < size); i++)
        sum += data[i];
    return sum;
}
```

В чем различие между int и size_t? Как хранится size_t?

Перепишем функцию еще раз:

```
int Sum (const int *data, size_t size)
{
    assert(data != NULL);
    int sum = 0;
    for (size_t i = 0; i < size); i++)
        sum += data[i];
    return sum;
}
```

data[i] это то же самое что *(data + i) или *(i + data) или i[data]

это ... арифметика, когда мы управляем не байтами, а элементами
программа сама умножает значение i на размер типа

из этого следует, что data[] это то же самое что *data

например, 1["DED"] равно числу 70
а 3["DED"] равно числу 0 (тк под индексом 3 нулевой символ)

Строки

```
char str[10] = {65, 66, 67}
             = {'A', 'B', 'C'}
             = "ABC"
```

Каверзные вопросы:

0 - число 0
65 - число 65
'A' - символ буквы A под ASCII номером 65
'0' - символ нуля под ASCII номером 48
' ' - символ пробелом под ASCII номером 32
'\0' - нулевой символ под ASCII номером 0
'' - две одиночных кавычки, между которыми ничего нет. это синтаксическая ошибка, тк такого "пустого" символа не существует
"ABC" - массив из четырех элементов типа char, то же самое что {'A', 'B', 'C', '\0'} или {65, 66, 67, 0}
"0" - массив из двух элементов, то же самое что {48, 0}
"" - массив из одного элемента, то же самое что {'\0'} или {0}
"AB\0C" - массив из пяти символов, то же самое что {'A', 'B', '\0', 'C', '\0'}, sizeof этой строки равен 5, длина этой строки равна 2
"\0" - массив из двух символов, то же самое что {'\0', '\0'}

Что если вернуть массив через return? Ответ:
Вернется не весь массив, а адрес его начала (нулевого элемента). При этом весь массив удалится и элементов не останется.
Указатель будет ссылаться на рандомный мусор.
Как вариант, можно загнать массив в структуру, но тогда она будет каждый раз копироваться заново.

Нулевой символ - это ноль терминатор, созданный чтобы не возникало проблем с определением длины строки
Если написать printf("Вася\0 дурак"), то выведется только Вася, тк \0 посчитается как символ конца строки

Стандартные функции для операций со строками

Большая часть функций находится в <strio.h>, но напишем их заново

Алгоритм копирования

```
char src[] = "ABC";
char dest[sizeof(src) + 5] = "xyz";

my_strcpy(dest, src)
```

```
char* my_strcpy(char dest[], const char src[])
{
    assert(src);
    assert(dest)
    if (src != dest}
    {
        size_t i;
        for (i = 0; src[i]; i++)
            dest[i] = src[i]
        dest[i] = '\0'
    }
    return dest; //возвращаем адрес получившейся строки
}
```

пока символ есть, мы его копируем

В вк есть видео "минус конспект"

Автоматически нулевой символ ставится при инициализации/присваивании с двойными кавычками

"Последовательность крепкости - не всегда крепкость" (с) Дед