& ← гопник

& ← передает значение номера ячейки

* ← баскетболист, рандомный котик меняет значение на рандомные

* ← используем адрес чтобы поместить туда значение

Ср 21.08.24

Конспект

1) Конспект должен быть понятен другим

2) Сокращения / обозначения

int main()

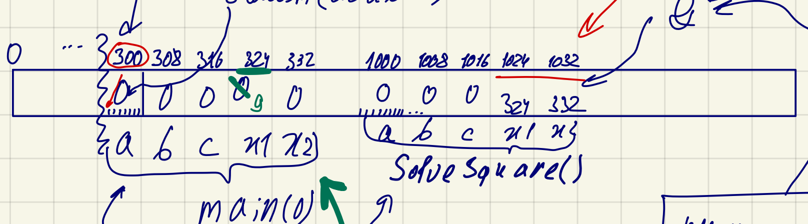
```
int main()
{
double a=0, b=0, c=0;
double x1=0, x2=0;
  ...



SolveSquare (a,b,c,&x1,&x2);
....
int SolveSquare(double a, double b, double c, double *x1, double *x2)
{
------
   *x1 = (-b - sqrt(discr)) / (2*a);
// "*" обращение к ячейке с индексом равным x1

}
```

!!!

побочная часть кода

300 (адрес только первой ячейки)

8 байт (double)



зона памяти для переменных

9

для 64х
в 32х int занимает
другое кол-во ячеек

копирование адреса

нужно для того чтобы когда SolveSquare() окончится значения не уничтожились, а записались в main()
в 324 и 332

* Определение типов (double, char, int) происходит по запросу адресов от процессора.
* Можно сделать супер архитектуру на 1 байте = 32; 64 бита)
* Легче и дешевле это сделать через эмуляторы.
* При обращении к нулевой области памяти или чужой области запрос будет перехвачен и вылезет предупреждение.
* Разрядность определяет максимальное кол-во опер. пам. (2^{32}; 2^{64} ...)
* Возвращаемое значение - либо регистр, либо при больших числах компилятор заимствует доп ячейку.

Функция тестирования

```
int  RunVasiaRun()
{
        double  x1=0,  x2=0;
        int nRoots = SolveSquare(1,0,-4,&x1,&x2);
        if (nRoots != 2 || x1 != -2 || x2 != +2)
        {
                printf("ERROR Test1: a=%lg, b=%lg, c=%lg, x1=%lg, x2=%lg,
                 nRoots=%d\n" "Expected: x1=%lg, x2=%lg, nRoots=%d\n",
                 1, 0, -4, x1, x2, nRoots,
                 -2, +2, 2)
        }
}
```

Версия с параметрами

функция что-то должна в итоге вернуть

```
int RunVasiaRun(int nTest, double a, double b, double c, double x1expected,
double x2expected, int nRootsExpected )
{
        double  x1=0,  x2=0;
        int nRoots = SolveSquare(a,b,c,&x1,&x2);
        if (nRoots != nRootsExpected || x1 != x1Expected || x2 != x2Expected)
        {
                printf("ERROR Test%d: a=%lg, b=%lg, c=%lg, x1=%lg, x2=%lg,
                 nRoots=%d\n" "Expected: x1=%lg, x2=%lg, nRoots=%d\n",
                 nTest, a, b, c, x1, x2, nRoots,
                 x1Expected, x2Expected, nRootsExpected);
        }
}
```

дописать возвращение

Создадим функцию параметров для тестов

```
RunVasiaRun(1, 1, 0, -4, -2, +2, 2)
```
← пример

Чт 22.08.24

Нам необходим такой тип данных который может сохранять данные разных типов

Массивы

Нужны для обработки большого кол-ва данных одного типа
Например пересчитать варианты в цикле, с структурой это невозможно (у каждого элемента свое имя)

Struct

1) группа

2) сохранение имен

3) разные типы данных

```
include ....

struct SP        <- данные тестирования
{
        double a,b,c
        double x1ref, x2ref
        int nRoots;
}

int main()
{
        SP test1 = {1, 0, -4, -2, +2, 2};        <- инициализация (происходит один раз)
        SP test2 = {.a=1, .b=0, .c=-4, .x1ref=-2, .x2ref=+2 ...};
        test2.x1ref = test1.x2ref;  <- замена переменных (можно ∞)
     ① RT(test1);
     ② RT(&test2);
}

① void RT(SP data) <- группа data типа SP
{
        ....
        ....
        SolveSquare(data.a, data.b, data.c, ...)  <- принятие параметров без изменения SolveSquare на структурную функцию
        ....
}
```

```
② void RT (SP* data)
{
        ..... SolveSquare((*data).a,   data->b,
```

краткая запись аналогичная (*).

скобки необходимые иначе первым будет применен оператор " . "

* если структура мала выгоднее передавать ее полностью

Пн 23.08.24

* Структура - именованная группа элементов разных типов данных
* Массив - нумерованная группа элементов одного типа данных

Начальная версия:

```
void AllTests ()
{
        uTest ( .... )  ←  тесты в заданных значениях
        uTest ( .... )
        ....            ←  циклическую проверку с разными тестами сделать невозможно
}
```

Промежуточная версия:

```
void AllTests ()
{
        for (i=0; i<nTest; i++)
        {
                uTest ( .... )  ←  проверка на одном тесте (в таком нет смысла)
                ....            ←  возможна проверка на верное выполнение
        }
}
```

Конечная версия:

```
void AllTests ()
{
        const int nTests = 10;
        double a[nTests] = {1, 1, 1, ....};  ←  инициализация массива
        double b[nTests] = {2, 0, 0, ....};      (ограничений по кол-ву раз нет)
        ......

        for (i=0; i<nTest; i++)
        {
                uTest (i, a[i], b[i] ...)  ←  происходит вызов из массивов
                                                 (тесты разные)
                ....                       ←  возможна проверка на верное выполнение
        }
}
```

Массив Структур

```
struct  param_solution_expected
{
          int  n tests;
          double  a, b, c;
          double  x1,  x2;
          int  n roots;
};

void  AllTests ()
{
          const  int  nTest  = 10;
          struct param_solution_expected  data [nTests] = {{ 1, 2, ..... },
                                                            { ... },
                                                            { ... },
                                                             ....
                                                                          };
          for (i=0; i<nTest; i++)
          {
                    uTest (data[i]);
                    ....
          }
}
```

передача одного struct из массива

закрытие массива