1.5. Массивы.

Monday, March 23, 2020 18:05

Язык С предоставляет возможность работы со структурированными данными, которые называются массивами. В этом уроке мы рассмотрим:

- создание массивов (Setting up simple arrays)
- инициализацию массивов (Initializing arrays)
- работу со строковыми массивы (Working with character arrays)
- директива `const` (Using the const keyword)
- многомерные массивы (Implementing multidimensional arrays)
- динамические массивы (Creating variable-length arrays)

Предположим, у нас есть набор оценок `grades`, который нужно ввести в компьютер и произвести над ним некоторые операции, например, сортировать их по возрастанию, вычислить среднее значение и пр. В уроке 1.4 мы находили среднее значение и без массива, но такой подход гибкостью не отличается. Да и, например, сортировать будет неудобно, потому что придется объявлять отдельную переменную для каждой оценки. А если оценок тысячи? Замучаешься объявлять и описывать всё это. Тут в игру вступают массивы.

Объявление массива

По аналогии с математической записью, где i-ый элемент x записывают как x_i , в С переменная массива объявляется как

```
x[i]
```

То есть, выражение

```
grades[5]
```

(*англ. grades sub 5*) ссылается на 6й элемент массива `grades`.
прим.: это можно обойти с помощью арифметики указателей, но это уже смежная тема, и непосредственно к массивам не относится.
Первый элемент массива имеет индекс 0, так что `grades[0]` будет ссылаться на 1й элемент массива, а не на 0й (который и не может существовать).
Но часто для простоты восприятия его все же называют нулевым. Здесь я так и буду делать.
Можно также присваивать переменным значение элемента массива, например:

```
g = grades [50];
```

Ничто также не мешает вместо числа индекса массива вписать переменную:

```
g = grades[i];
```

То есть если `i = 7`, то переменной `g` будет присвоено значение 7го элемента массива `grades`.

Запись в массив производится по аналогии:

```
grades[100] = 95;
```

И, опять же, ничто не мешает вместо чисел использовать переменные:

```
grades[i] = g;
```

Можно даже использовать выражения:

```
next_value = sorted_data[(low + high) / 2];
```

Приведем простой пример работы с массивами. Цикл

```
1 for ( i = 0;  i < 100;  ++i )
2           sum += grades[i];
```

суммирует первые 100 элементов массива (с 0 по 99й).

Так же как и с обычными переменными, перед работой с массивом нужно его объявить. Синтаксис такой же, но нужно также указать максимальный размер массива, чтобы компилятор знал, сколько выделять памяти под этот массив. Строка

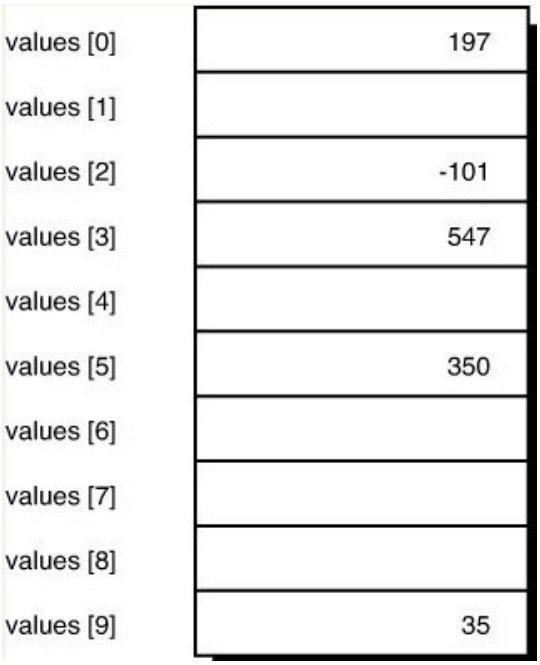
```
int values [10];
```

объявит массив `values`, содержащий максимум 10 целочисленных элементов. То есть обращаться к нему можно в диапазоне индексов от 0 до 9. С этим нужно соблюдать осторожность, потому что С не проверяет границы диапазона и если мы вдруг обратимся, например, к элементу 150, это вызовет непредвиденные результаты работы программы.

Попробуем записать что-нибудь в наш массив, который мы только что объявили:

```
values[0] = 197;
values[2] = -100;
values[5] = 350;
values[3] = values[0] + values[5];
values[9] = values[5] / 10;
--values[2];
```

После выполнения этих строк массив будет выглядет следующим образом:



Приведем нашу программу в нормальный вид и выведем в консоль весь массив целиком:

```
#include <stdio.h>

int main (void)
{
          int  values[10];
          int  index;

          values[0] = 197;
          values[2] = -100;
          values[5] = 350;
          values[3] = values[0] + values[5];
          values[9] = values[5] / 10;
          --values[2];

          for ( index = 0;  index < 10;  ++index )
                    printf ("values[%i] = %i\n", index, values[index]);

```

```
18          return 0;
19  }
```

программа выдаст:

```
values[0] = 197
values[1] = -2
values[2] = -101
values[3] = 547
values[4] = 4200992
values[5] = 350
values[6] = 4201094
values[7] = 4200992
values[8] = 108
values[9] = 35
```

Как видим, в тех элементах массива, над которыми мы не проводили никаких операций будет рандомный мусор, а точнее то, что было в адресах памяти, выделенных под эти элементы. При объявлении как массивов, так и переменных, выделяемый под них участок памяти не стирается, – там будут остатки данных от предыдущих программ, которых работали в этих участках.

Элементы массива в качестве счетчиков

Перейдем к более практическим примерам применения массивов. Допустим, мы собрали с 5000 человек оценки от 1 до 10 и хотим обработать эти данные и вывести в таблицу, сколько человек какую оценку поставили. Напишем программу, которая выполняет вышеописанную задачу. Оценки будем хранить в массиве `ratingCounters`.

```
1  #include <stdio.h>
2
3  int main (void)
4  {
5           int  ratingCounters[11], i, response;
6
7           for (i = 1; i <= 10; i++)
8                 ratingCounters[i] = 0; //initialize the array
9
10          printf ("Enter your responses\n");
11
12          for (i = 1; i <= 20; i++) {
13                scanf("%i", &response);
14
15                if (response < 1 || response > 10)
16                        printf("Bad response!\n");
17                else
18                        ++ratingCounters[response];
19          }
20
21          printf ("\n\nRating   Number of Responses\n");
22          printf ("------   -------------------\n");
23
24          for (i = 1; i <= 10; i++)
25                printf("%4i%14i\n", i, ratingCounters[i]);
26
27          return 0;
28 }
```

Мне поначалу было не совсем очевидно, что происходит в строке 18. Оказывается, все просто: оценка соответствует индексу массива, т.е., например, оценки 1 суммируются в `ratingCounters[1]`, оценки 5 в `ratingCounters[5]` и т.д.
Ну и стоит не забывать, что нужно объявить массив из 11 элементов, потому что счет начинается с

нулевого элемента, а нам нужны индексы с 1 по 10, т.е. с нулевым это будет 11 элементов.
Можно было написать

```
++ratingCounters[response - 1];
```

и объявить массив из 10, а не 11 элементов, но тогда код будет еще менее наглядным для новичка.

Генерирование чисел Фибоначчи

Вспомним, что числа Фибоначчи генерируются по формуле:

$$F_0 = 0, \qquad F_1 = 1, \qquad F_n = F_{n-1} + F_{n-2}$$

То есть, мы задаем первые два элемента вручную, а каждый следующий элемент массива будет равен сумме двух предыдущих элементов. Напишем программу, которая рассчитает список первых 15 чисел Фибоначчи:

```c
#include <stdio.h>

int main (void)
{
        int  Fibonacci[15], i;

        Fibonacci[0] = 0;
        Fibonacci[1] = 1;

        for (i = 2; i < 15; i++)
                Fibonacci[i] = Fibonacci[i-1] + Fibonacci[i-2];

        for (i = 0; i < 15; i++)
                printf(" %i\n", Fibonacci[i]);

        return 0;
}
```

Генерирование простых чисел через массивы

В уроке 1.4 и упражнении к нему мы говорили про то, что оптимальнее будет сгенерировать таблицу простых чисел не через операторы ветвления, а через массивы. Этим сейчас и займемся.
Суть оптимизации заключается в использовании свойства, что число будет являться простым, если оно не делится на простое число. Это выводится из того, что любое не простое число можно выразить через умножение простых чисел. Например, 20 = 2 × 2 × 5. То есть, нам достаточно проверять числа на делимость на все простые числа, которые меньше проверяемого числа. Для хранения списка найденных простых чисел будем использовать массив.
Еще можно использовать свойство, что в любом не простом числе один из множителей всегда будет меньше квадратного корня этого числа. То есть нам не обязательно проверять все множители, достаточно проверить только те, что меньше квадратного корня проверяемого числа (в целочисленном представлении, очевидно). Но сейчас мы не будем использовать это свойство.

```c
#include <stdio.h>
#include <stdbool.h>

int main (void)
{
        int  p, i, primes[50], primeIndex = 2;
        bool isPrime;

        primes[0] = 2;
```

```
10          primes[1] = 3;
11
12          for (p = 5; p <= 50; p = p + 2) {
13                  isPrime = true;
14
15                  for (i = 1; isPrime && p / primes[i] >= primes[i]; i++)
16                          if (p % primes[i] == 0)
17                                  isPrime = false;
18
19                  if (isPrime) {                  // if the number is prime
20                          primes[primeIndex] = p; // write it into the array
21                          primeIndex++;           // and increase the counter
22                  }
23          }
24
25          for (i = 0; i < primeIndex; i++)
26                  printf("%i ", primes[i]);
27
28          printf("\n");
29
30          return 0;
31  }
```

У многих, вероятно, возникнет вопрос, а где же проверка на `p < √primes[i]`? А нам не обязательно для этого вычислять квадратный корень, мы можем для этого использовать выражение

```
p / primes[i] >= primes[i]
```

Нетрудно показать, почему это выражение будет эквивалентно проверке корня числа, достаточно представить проверяемое число как два множителя, из которых берется корень. Можно также использовать выражение

```
p >= primes[i] * primes[i]
```

Такой подход часто быстрее, особенно во встраиваемых системах, потому что RISC архитектуры часто не могут выполнить операцию деления за один цикл, а умножать за один цикл могут. Но этот способ чреват переполнением переменной, нужно следить, чтоб квадрат `primes[i]` не вылез за допустимые пределы.

Инициализация массивов

Так же, как и для обычных переменных, можно задавать массивам начальные значения. Иначе, как мы видели раньше, они будут заполнены бессмысленным мусором. Значения можно разделять запятыми:

```
int  integers[5] = { 0, 1, 2, 3, 4 };
```

Здесь 0-й элемент будет равен 0, 1-й равен 1 и т.д.

Массив символов инициализируется похожим образом:

```
char  letters[5] = { 'a', 'b', 'c', 'd', 'e' };
```

При инициализации не обязательно записывать значение каждого элемента. Например, строка

```
float  sample_data[500] = { 100.0, 300.0, 500.5 };
```

запишет в первые три элемента массива `sample_data` числа 100.0; 300.0; 500.5, а во все остальные элементы будут автоматически записаны нули.

Если нужно просто обнулить массив, можно написать:

```
float  sample_data[500] = { 0 };
```

Ничто не мешает инициализировать массив не по порядку. Строка

```
float sample_data[500] = { [2] = 500.5, [1] = 300.0, [0] = 100.0 };
```

точно так же, как и предыдущая строка, запишет числа в первые три элемента массива, а остальные

элементы будут нулевыми.

Также можно в строку инициализации вставлять арифметические выражения:

```
int  x = 1233;
int  a[10] =  { [9] = x + 1, [2] = 3, [1] = 2, [0] = 1 };
```

Здесь 9-й элемент является функцией переменной `x`.
прим.: так нельзя делать при объявлении глобальных переменных, иначе компилятор выдаст ошибку `initializer element is not constant`. Нужно чтобы инициализация происходила внутри функции, например, `int main()`.

Напишем программу, которая инициализирует массив и через строку объявления переменной, и через цикл `for()`.

```
#include <stdio.h>

int main (void)
{
        int  i;
        int  array[10] = { 0, 1, 4, 9, 16 };

        for (i = 5; i <= 10; i++)
                array[i] = i * i;

        for (i = 0; i < 10; i++)
                printf("array[%i] = %i\n", i, array[i]);

        return 0;
}
```

Здесь первые 5 элементов мы задаем вручную, а последние 5 через цикл, который присваивает элементам значение i^2.

Символьные массивы

Напишем программу, в которой мы инициализируем и напечатаем символьный массив.

```
#include <stdio.h>

int main (void)
{
          char  word[] = { 'H', 'e', 'l', 'l', 'o', '!' };
          int   i;

          for ( i = 0;  i < 6;  ++i )
                    printf ("%c", word[i]);

          return 0;
}
```

Программа выдаст:

```
Hello!
```

Обратите внимание, что мы не задавали длину массива. Так можно делать, и компилятор автоматически определит длину массива, исходя из количества элементов при инициализации. То есть, если мы задали, например, 6 символов, массив будет размером в 6 элементов.
Также, если мы задаем только отдельные элементы массивов, то длина будет основана на максимальном указанном индексе элемента. Например, если мы объявим массив

```
float sample_data[] = { [0] = 1.0, [49] = 100.0, [99] = 200.0 };
```

то он будет размеров в 100 элементов, потому что максимальный указанный нами индекс – 99.

Преобразование систем счисления при помощи массивов

Напишем программу, которая конвертирует десятичное число в другую систему счисления, вплоть до шестнадцатиричной. Нужно будет спросить число и основаные системы счисления.
Опишем сначала алгоритм.

1. Находим остаток от деления на основание системы счисления:

   ```
   digit = number % base;
   ```

2. Делим преобразуемое число на основание системы счисления, отсекая дробную часть:

   ```
   number /= base;
   ```

3. Повторяем пункты 1 и 2, пока преобразуемое число не станет нулем.

Посмотрим, как будет выглядеть преобразование числа 10 в двоичную систему счисления:

Number	Number Modulo 2	Number / 2
10	0	5
5	1	2
2	0	1
1	1	0

То есть, с таким алгоритмом результат нужно читать наоборот: `1010`.

При написании программы для преобразования, нужно учитывать несколько вещей.
Первое – алгоритм дает перевернутое число и поэтому лучше не сразу выводить его в консоль, а занести его в массив и потом просто вывести массив задом наперед.
Второе – у нас максимальное основание системы счисления – 16. То есть нужно будет выводить не только числа, но и буквы от A до F. Здесь в игру вступает массив символов-констант, содержащий символы от 0 до F. Это нужно в первую очередь для того, чтобы если при конверсии в систему больше 10 алгоритм выдаст число больше 9, программа выдала букву, а не двузначную цифру.

Попробуем написать программу:

```
// Program to convert a positive integer to another base

#include <stdio.h>

int main (void)
{
          const char baseDigits[16] = {
                        '0', '1', '2', '3', '4', '5', '6', '7',
                        '8', '9', 'A', 'B', 'C', 'D', 'E', 'F' };
          int       convertedNumber[64];
          long int numberToConvert;
          int       nextDigit, base, index = 0;

          // get the number and the base

          printf ("Enter your number: ");
          scanf ("%ld", &numberToConvert);
          printf ("Base: ");
          scanf ("%i", &base);

          // convert to the indicated base

          do {
                    convertedNumber[index] = numberToConvert % base;
                    ++index;
```

```
                numberToConvert = numberToConvert / base;
        }
        while  ( numberToConvert != 0 );


        // display the results in reverse order

        printf ("Converted number = ");

        for (--index;  index >= 0;  --index ) {
                nextDigit = convertedNumber[index];
                printf ("%c", baseDigits[nextDigit]);
        }

        printf ("\n");
        return 0;
}
```

Здесь мы собираем массив `convertedNumber`, в котором каждый элемент соответствует выводимой цифре преобразованного числа. А затем выводим этот массив наоборот, вместо цифр подставляя соответствующие им символы из массива символов `baseDigits`.
Массив `convertedNumber` имеет размер в 64 элемента, что равно максимальному количеству цифр в переменной типа `long int`, в которой мы храним преобразуемое число. Мы взяли такой большой размер для избыточности, чтоб можно было преобразовывать широкий диапазон чисел.
Также здесь отсутствует проверка на то, что пользователь ввел правильные значения преобразуемого числа (неотрицательные) и основания системы счисления (от 2 до 16). В следующем уроке мы доведем программу до ума.

Директива `const`

Мы использовали эту директиву в программе выше при объявлении массива символов. Она нужна, чтобы указать компилятору, что над этим массивом (или переменной) не будут производиться никакие действия, т.е. фактически переменная превращается в константу. Это чаще всего бывает полезно для экономии оперативной памяти, потому что константы хранятся не в ОЗУ, а в ПЗУ. Если же мы попытаемся модифицировать константу, например:

```
const int a = 10;
a = a / 2;
```

то компилятор выдаст ошибку:

```
error: assignment of read-only variable 'a'
```

Также эта директива упрощает чтение кода, т.к. указывает на то, что мы не собираемся проводить в программе никакие операции над этими значениями. Что и происходит в нашей программе выше.

Многомерные массивы

Пока мы работали только с линейными одномерными массивами. Но язык С позволяет работать с массивами любой размерности. В этой главе мы рассмотрим работу с двумерными массивами.

Частым применением двумерных массивов являются матрицы. Приведем пример матрицы 4x5:

Column (j)	0	1	2	3	4
Row (i)					
0	10	5	-3	17	82
1	9	0	0	8	-7
2	32	20	1	0	14
3	0	0	8	7	6

По аналогии с математической записью $M_{i,j}$ в языке C обращение к элементу двумерного массива выглядит как

```
M[i][j]
```

где первое число – номер строки, а второе – номер столбца. Например, выражение

```
sum = M[0][2] + M[2][4];
```

прибавляет значение элемент 0-й строки, 2-го столбца (т.е. -3) к значению элемента 2-й строки, 4-го столбца (т.е. 14), что в итоге дает 11.

Двумерные массивы объявляются по аналогии с одномерными. Строка

```
int  M[4][5];
```

объявляет массив M, состоящий из 4 строк и 5 столбцов, где каждый элемент содержит целочисленное значение.

Инициализация тоже похожа на одномерные массивы:

```
int  M[4][5] = {
                    { 10,  5, -3, 17, 82 },
                    {  9,  0,  0,  8, -7 },
                    { 32, 20,  1,  0, 14 },
                    {  0,  0,  8,  7,  6 }
                };
```

Обратите внимание на расстановку запятых после закрывающих скобок. Кстати, можно и без скобок, в ряд:

```
int  M[4][5] = { 10, 5, -3, 17, 82, 9, 0, 0, 8, -7, 32,
                 20, 1, 0, 14, 0, 0, 8, 7, 6 };
```

Но это не так наглядно. Компилятор сам расставит элементы массива слева направо, как выше.

Как и с одномерными массивами, не обязательно указывать значения всех элементов массива. Выражение

```
int  M[4][5] = {
                    { 10,  5, -3 },
                    {  9,  0,  0 },
                    { 32, 20,  1 },
                    {  0,  0,  8 }
                };
```

инициализирует только первые три элемента каждой строки и столбца, а остальные элементы будут равны нулю. **Но теперь скобки нужно ставить**, иначе компилятор пойдет слева направо, переходя на следующую строку только после заполнения предыдущей и массив будет инициализирован не так, как нам хотелось бы (будут заполнены первые 2 строки и первые 2 элемента 3-й строки).

Можно заполнять и каждый элемент индивидуально:

```
int matrix[4][3] = {  [0][0] = 1, [1][1] = 5, [2][2] = 9 };
```

Не указанные нами элементы будут равны нулю.

Динамические массивы

прим.: стандарт C11 не требует обязательной поддержки компилятором динамических массивов. Проверяйте документацию к компилятору перед работой с ними.

Пока мы использовали массивы фиксированной длины; указывали их длину при их объявлении.
Но язык C также позволяет объявлять массивы переменной длины. Вернемся к программе расчета чисел Фибоначчи. В ней мы рассчитывали только первые 15 чисел. А если, скажем, нужно спросить у пользователя, сколько чисел считать?
Рассмотрим модифицированную программу, которая рассчитывает произвольное количество чисел

Фибоначчи, используя динамический массив.

```
 1 // Generate Fibonacci numbers using variable length arrays
 2
 3 #include <stdio.h>
 4
 5 int main (void)
 6 {
 7           int i, numFibs;
 8
 9           printf ("How many Fibonacci numbers do you want (between 1 and 75)? ");
10           scanf ("%i", &numFibs);
11
12           if (numFibs < 1 || numFibs > 75) {
13                     printf ("Bad number, sorry!\n");
14                     return 1;
15           }
16
17           unsigned long long int   Fibonacci[numFibs];
18
19           Fibonacci[0] = 0;           // by definition
20           Fibonacci[1] = 1;
21
22           for ( i = 2;  i < numFibs;  ++i )
23                     Fibonacci[i] = Fibonacci[i-2] + Fibonacci[i-1];
24
25           for ( i = 0;  i < numFibs;  ++i )
26                     printf ("%llu  ", Fibonacci[i]);
27
28           printf ("\n");
29
30           return 0;
31 }
```

Разберем нашу программу.

Переменная `numFibs` используется для хранения желаемого количества чисел Фибоначчи. Мы здесь еще реализовали проверку диапазона переменной, чтоб она могла находиться только в нужных нам пределах и реализовали завершение программы с выводом кода 1, в случае если введенное число не попадает в заданный нам диапазон.
То есть, как мы говорили в уроке 1.1, выражение

```
return 1;
```

завершает программу и возвращает число "1" и по этому числу мы можем определить, что именно привело к завершению программы; в нашем случае это будет ввод числа вне дозволенного диапазона.

После ввода числа мы видим выражение

```
unsigned long long int   Fibonacci[numFibs];
```

где мы объявляем массив `Fibonacci` длины `numFibs`. Этот массив называется динамическим, потому что его длина задается не константой, а переменной. Локальные переменные можно объявлять в любом месте функции и потому ничто нам не помешало объявить этот массив отдельно от остальных переменных. Но лучше таким не злоупотреблять, потому что гораздо удобнее читать код, когда все переменные объявляются в одном месте.
Элементы массива имеют тип `unsigned long long int`, потому что числа Фибоначчи всегда больше нуля и очень быстро становятся очень длинными, поэтому для гибкости мы выбрали тип переменной с наибольшим доступным положительным значением.

Остаток программы очевиден и не требует дополнительных пояснений.

Также упомянем, что можно использовать *динамическое выделение памяти* для работы с

динамическими массивами; т.е. такие функции как `malloc()` и `calloc()`.
Просто задавать длину массива переменной, как мы делаем сейчас, – плохая практика, хотя бы потому что такая конструкция компилируется в неоптимальный код. Но это мы разберем в будущих уроках.

Упражнения

2. Даже не знаю, что сказать.

```c
// Initialize array elements to 0 with for loop
#include <stdio.h>

int main (void)
{
          int  values[10];
          int  i;

          for (i = 0;  i < 10;  i++) {
                values[i] = 0;
                printf ("values[%i] = %i\n", i, values[i]);
          }

          return 0;
}
```

3. Достаточно было просто добавить бесконечный цикл и оператор выхода из него по вводу кода для выхода. Еще до кучи приукрасил внешний вид программы и сделал продвинутую защиту от ввода неверных данных.

```c
// Rating counter with "infinite" number of responses

#include <stdio.h>

int main (void)
{
     int  ratingCounters[11], i, response;

     for ( i = 0;  i <= 10;  ++i )
                    ratingCounters[i] = 0;

     while(1) {
        response = 0;  /* for some reason the program accepts
                          characters '+', '-', '*', '/'
                          and response retains its previous value
                       */

        fflush(stdin); /* clear the keyboard buffer, so
                          scanf() works properly if a
                          character is entered instead of number
                       */

        system("cls"); // clear the console
        printf ("Enter your response: ");
        scanf ("%i", &response);

        if (response == 999)  // if 999 is entered
            break;            // exit the loop and display results
        else if (response >= 1 && response <= 10 )
            ++ratingCounters[response];
        else {
            printf ("Bad response! Press any key to try again...");
            getch();          // wait for any key press
        }
     }

     printf ("\n\nRating Number of Responses\n");
     printf ("------ -------------------\n");
```

```
39
40      for ( i = 1;  i <= 10;  ++i )
41          printf ("%4i%14i\n", i, ratingCounters[i]);
42
43      return 0;
44  }
```

Исходник:

main

4. Сделал массив из 5 элементов, а не 10, чтоб проще было проверять работу программы.

```
1  // Calculating average of 5 float values
2
3  #include <stdio.h>
4
5  int main (void)
6  {
7            float values[5];
8            float acc = 0, average;
9            int i;
10
11           printf("Fill in the array.\n");
12
13           for (i = 0; i < 5; i++)
14                 scanf("%f", &values[i]);
15
16           for (i = 0; i < 5; i++)
17                 acc += values[i];
18
19           printf ("The average value is %3g", acc / 5);
20
21           return 0;
22 }
```

5. Программа выдаст:

```
1 1 2 4 8 16 32 64 128 256
```

Здесь каждый элемент (кроме первого) равен сумме всех предыдущих. Долго доходило.

6. Простое и интересное задание

```
1  // Fibonacci numbers generator using only 3 variables
2
3  #include <stdio.h>
4
5  int main (void)
6  {
7            int  f0, f1 = 0, f2 = 1, i;
8
9            printf("0\n1\n");
10
11           for (i = 2;  i < 15;  ++i) {
12                     f0 = f1 + f2;
13                     printf ("%i\n", f0);
14                     f1 = f2;
15                     f2 = f0;
16           }
17
18           return 0;
19 }
```

7. Решето Эратосфена. Мне кажется, я скоро и математику до кучи буду знать после этого курса.

```c
// Prime numbers generator using the Sieve of Eratosthenes

#include <stdio.h>

int main (void)
{
        int  n, i, j;

        printf("Upper margin to search for prime numbers: ");
        scanf("%i", &n);

        int list[n + 1];

        for (i = 0; i < (n + 1); i++)
                list[i] = 0;

        for (i = 2; i < n; i++) {
              for (j = 2; i * j <= n; j++)
                        list[i * j] = 1;
        }

        for (i = 2; i < n; i++)
              if (!list[i])
                printf("%i\n", i);

        return 0;
}
```

В задании допущена ошибка. Границы `j` описаны неверно: оно должно начинаться не с 1, а с 2. Что следует из определения простого числа, если внимательно подумать.
Сам алгоритм работает как минимум вдвое быстрее, чем предыдущие использованные нами алгоритмы.

8. См. предыдущее упражнение.

Источники:

1. Stephen Kochan – Programming in C (4th Edition); chapter 6
2. https://stackoverflow.com/questions/2589749/how-to-initialize-array-to-0-in-c
3. https://cboard.cprogramming.com/c-programming/143734-compiler-keeps-skipping-over-scanf-statement-please-help.html
4. https://www.tutorialspoint.com/clearing-input-buffer-in-c-cplusplus