1.6. Пользовательские функции. Локальные и глобальные переменные.

Sunday, March 29, 2020 21:00

Функции используются практически в любой программе на С. Например, мы в каждом уроке пользовались функцией `printf()`, также часто пользуемся функцией `scanf()`. В этом уроке мы разберем, как создавать собственные, пользовательские функции, которые могут сильно упростить работу с кодом за счет разбиения задачи на маленькие подзадачи.
План урока:

- основные свойства функций
- локальные, глобальные, автоматические и статические переменные (*local, global, automatic, and static variables*)
- использование одно- и многомерных массивов с функциями
- возвращение данных функциями
- использование функций для структурного программирования сверху вниз (*top-down programming*)
- вызов функций внутри других функций, рекурсивные функции

Понятие функции

Вспомним наш первый урок, где мы написали программу, выводящую фразу Hello world!

```c
#include <stdio.h>

int main (void)
{
        printf ("Hello world!\n");

        return 0;
}
```

Теперь напишем пользовательскую функцию, выполняющую то же действие:

```c
void printMessage (void)
{
        printf ("Hello world!\n");
}
```

Первая строка состоит из 3-4 аргументов, которые (в порядке слева направо) задают свойства функции:

1. Кто может вызывать функцию (пока что мы этот аргумент не используем, мы придем к нему позже).
2. Тип значения, которое возвращает функция (например, `int main (void)` возвращает тип `int`).
3. Имя функции
4. Аргументы, которые принимает функция (`void`, если не принимает никаких аргументов).

Собственно, наша функция не принимает в себя никаких значений и никакие значения не возвращает. Поэтому мы можем обойтись без оператора `return`. Если мы в **конце функции** запишем

```c
return;
```

это никак не повлияет на работу функции, если она возвращает `void`. Если этот оператор будет в **каком-либо другом месте функции**, функция завершится после его выполнения, то есть *оператор `return` можно использовать для завершения функции, возвращающей `void`.*
Так-то любая функция завершается после оператора `return`, так как он дает функции команду прекратить работу и вернуть результат своей работы, но если функция возвращает что-то кроме `void`, то обязательно нужно будет что-то вывести, – пустой `return` недопустим.

Как и с названиями переменных, название функции должно отражать ее суть, это упростит чтение кода. Мы назвали функцию `printMessage`, что она, собственно, и делает.

Также стоит не забывать, что функция `int main (void)` обязательно должна присутствовать в коде, переименовывать её нельзя, потому что именно с этой функции начинается выполнение программы.

Используем теперь нашу пользовательскую функцию в законченной программе:

```
#include <stdio.h>

void printMessage (void)
{
        printf ("Hello world!\n");
}

int main (void)
{
        printMessage ();

        return 0;
}
```

То есть, теперь у нас в программе есть две функции – `printMessage()` и `main()`. Когда мы вызываем функцию `printMessage ()`, программа перепрыгивает на участок кода, где была описана функция и выполняет тело функции. Пустые скобки означают то, что мы не передаем функции никаких аргументов. После выполнения тела функции возобновляется выполнение основного кода в `main()`.

Можно вызывать функцию несколько раз подряд:

```
        printMessage ();
        printMessage ();
```

или вызывать в цикле:

```
        for ( i = 1;  i <= 5;  ++i )
                printMessage ();
```

Таким образом, при помощи функций мы можем разбить сверхзадачу на маленькие подзадачи, реализовать их как функции и потом их использовать при работе с основной программой.

Аргументы и локальные переменные

Когда мы вызываем функцию `printf()`, мы передаем ей один или несколько аргументов; первый аргумент – отображаемая строка, остальные – значения операторов вывода данных.
Создадим свою функцию, которая будет принимать аргументы. В уроке 1.3 мы кучей разных методов считали треугольные числа. Снова напишем программу, которая посчитает треугольное число для введенного пользователем числа, но в этот раз используем для этого функцию.

```
// Function to calculate the nth triangular number

#include <stdio.h>

void calculateTriangularNumber (int n)
{
        int  i, triangularNumber = 0;

        for ( i = 1;  i <= n;  ++i )
                triangularNumber += i;

        printf ("Triangular number %i is %i\n", n, triangularNumber);
}

int main (void)
{
        calculateTriangularNumber (10);
        calculateTriangularNumber (20);
        calculateTriangularNumber (50);

```

```
21          return 0;
22  }
```

Здесь мы объявили функцию с названием `calculateTriangularNumber()`, которая принимает на вход один аргумент типа `int` и внутри функции он хранится в переменной `n`. Функция не возвращает никаких значений (`void`). Вместо этого она печатает в консоль.

Можно объявлять несколько функций, возвращающих один и тот же тип данных:

```
    void print_pun(void), print_count (int n);
```

Можно даже объявить несколько функций, возвращающих один и тот же тип данных и принимающих одни и те же параметры:

```
    double x, y, average(double a, double b);
```

Но так делать нежелательно, потому что такой код труднее читать.

Автоматические локальные переменные

Также внутри функции мы объявляем две переменные,– `i` и `triangularNumber`. Они не будут доступны вне функции и автоматически создаются каждый раз, когда мы вызываем функцию. Поэтому они называются *локальными автоматическими переменными*. Если мы задаем начальное значение переменной при объявлении, оно будет ей присвоено при каждом вызове функции. На самом деле, точнее будет при их объявлении использовать ключевое слово auto:

```
   auto int  i, triangularNumber = 0;
```

Но, так как компилятор C обычно автоматически считает локальными автоматическими все переменные внутри функции, можно обойтись и без этого слова.

Напишем программу для расчета наибольшего общего делителя, в этот раз использовав функцию, принимающую два аргумента.

```
 1  /* Function to find the greatest common divisor
 2          of two nonnegative integer values             */
 3
 4  #include <stdio.h>
 5
 6  void gcd (int u, int v)
 7  {
 8          int  temp;
 9
10          printf ("The gcd of %i and %i is ", u, v);
11
12          while ( v != 0 ) {
13                  temp = u % v;
14                  u = v;
15                  v = temp;
16          }
17
18          printf ("%i\n", u);
19  }
20
21
22  int main (void)
23  {
24          gcd (150, 35);
25          gcd (1026, 405);
26          gcd (83, 240);
27
28          return 0;
29  }
```

Возвращение значений функциями

Наши предыдущие функции только выводят итоговые результаты своей работы в терминал. Но бывает, что нам не нужно, чтоб функция печатала в терминал, а передала результаты своей работы, например, основной программе. В каждой нашей программе мы пользовались оператором `return`:

```
return expression;
```

Это выражение заставляет функцию по своему завершению выдать значение *`expression`* в вызвавшее ее выражение. Но одного `return` недостаточно, чтоб функция правильно вернула значение. Нужно также при объявлении функции не забыть *указать тип данных, которые она будет возвращать*. Это указывается *перед* названием функции. Мы опять же встречались с этим, когда объявляли `main()`: ключевое слово `int` указывает на то, что функция возвращает данные типа `int`. Кстати, если не указать тип данных, компилятор автоматически использует тип `int`. Но лучше не ленитесь указывать тип данных, это упрощает чтение кода.
Если мы укажем, что функция не возвращает никаких данных (`void`), то при попытке получить значение функции, компилятор выдаст ошибку. Вспомним первую программу в этом уроке и покажем, как делать не надо. Подобная строка приведет к ошибке компиляции, так как функция ничего не возвращает:

```
number = calculateTriangularNumber (20);
```

Давайте перепишем программу для расчета наибольшего общего делителя, чтобы функция расчета не напрямую печатала результат, а передавала рассчитаное значение в `main()` для печати.

```
 1 /* Function to find the greatest common divisor of two
 2      nonnegative integer values and to return the result    */
 3
 4 #include <stdio.h>
 5
 6 int  gcd (int u, int v)
 7 {
 8           int  temp;
 9
10           while ( v != 0 ) {
11                   temp = u % v;
12                   u = v;
13                   v = temp;
14           }
15
16           return u;
17 }
18
19 int main (void)
20 {
21           int  result;
22
23           result = gcd (150, 35);
24           printf ("The gcd of 150 and 35 is %i\n", result);
25
26           result = gcd (1026, 405);
27           printf ("The gcd of 1026 and 405 is %i\n", result);
28
29           printf ("The gcd of 83 and 240 is %i\n", gcd (83, 240));
30
31           return 0;
32 }
```

Перед завершением выполнения функции `gcd()` выполняется выражение

```
return u;
```

и после этого продолжается выполнение кода в `main()` и переменной `result` присваивается значение возвращенного функцией значения `u`.
Функции в С может возвращать только одно значение, – как мы описали выше. В отличие от некоторых других языков программирования, язык С не различает подпрограммы(процедуры) и функции. Нам доступны только функции, которые могут либо вернуть одно значение, либо ничего не вернуть.

Вспомним еще одну программу из предыдущих уроков, – где мы брали число по модулю.
Перепишем ее так, чтоб она для обработки входного числа использовала функцию.

```
 1 // Function to calculate the absolute value
 2
 3 #include <stdio.h>
 4
 5 float  absoluteValue (float x)
 6 {
 7         if ( x < 0 )
 8             x = -x;
 9
10         return x;
11 }
12
13 int main (void)
14 {
15         float  f1 = -15.5;
16         int    i1 = 716;
17         float  result;
18
19         result = absoluteValue ( (float) i1 ) + absoluteValue (f1);
20         printf ("result = %.2f\n", result);
21         printf ("f1 = %.2f\n", f1);
22
23         return 0;
24 }
25
```

Этот пример показывает еще одно удобное свойство функций: они не изменяют входные аргументы. В одном из упражнений в уроке 1.5 нам пришлось вводить буферную переменную, чтобы не модифицировать исходные данные. Функция же не работает со входными аргументами напрямую, а помещает их в свои локальные переменные и работает уже с ними. Поэтому мы можем не заморачиваться с буферными переменными.

Также тут показана передача аргумента с типом данных не совпадающего с тем, что принимает функция. Нужно не забывать его преобразовывать в указанный в функции тип (строка 19).

Функция **exit**

Из функций обычно выходят через `return`. Помимо этого можно использовать функцию `exit` из библиотеки `<stdlib.h>` и она завершит **программу** с указанным кодом выхода, как будто мы выполнили функцию `return` в `main()`. То есть, аргумент, передаваемый в `exit`, имеет тот же смысл, что возвращаемое значение в `main()`: он указывает на статус завершения **программы**. Чтобы указать на нормальное завершение программы, нужно передать функции 0:

```
exit(0);   /* normal termination */
```

Но так не совсем понятно, что это значит. Лучше так:

```
exit(EXIT_SUCCESS);   /* normal termination */
```

На завершение с ошибкой можно указать при помощи `EXIT_FAILURE`:

```
exit(EXIT_FAILURE);   /* abnormal termination */
```

`EXIT_SUCCESS` и `EXIT_FAILURE` – это макросы, определенные в `<stdlib.h>`. Их численные значения могут отличаться в разных версиях библиотеки, но обычно это 0 и 1 соответственно.

Разница между `return` и `exit` в том, что `exit` приводит к завершению программы вне зависимости от того, где она была вызвана. В случае с `return` завершение программы происходит только при его вызове в `main()`.

Функции, вызывающие функции, вызывающие функции, вызывающие функции...

В наши времена есть куча программ и приложений, в которых можно посчитать квадратный корень. Но раньше на большинстве калькуляторов не было такой функции и это действие часто проводилось вручную. Погрузимся в ретро-хардкор и напишем программу для расчета квадратного корня. Итерационный метод Ньютона-Рафсона (метод касательных) неплохо подойдет для нашей задачи.

Мы начинаем с попытки "угадать" квадратный корень, выбрав некоторое стартовое число `guess`. Чем ближе число к решению, тем меньше итераций нам будет нужно. Угадывать у программ получается хуже, чем у людей, так что пусть нашим стартовым числом будет 1.

Число `x`, из которого извлекается корень, делится на `guess`, и результат деления прибавляется к `guess`. Затем этот промежуточный результат делится на 2. Это и будет новым значением `guess` и цикл повторяется снова, уже для нового значения `guess`.

Но так как наш метод выполняет бесконечное приближение к истинному значению, то нам нужно определиться, когда прекратить расчет, введя параметр, задающий точность – `epsilon (ε)`. Как только мы достигаем заданной им точности (т.е. количества цифр после запятой), цикл завершается.

Опишем алгоритм:

1. Установить `guess = 1`
2. Если $|guess^2 - x| < \varepsilon$, завершить цикл и вернуть значение `guess`
3. Установить `guess = ((x / guess) + guess) / 2`

Во втором шаге мы берем разность `guess` и `x` по модулю, потому что приближение к истинному значению может происходить с любой стороны (т.е. нам может понадобиться как прибавлять, так и вычитать от значения `guess`).

Теперь напишем программу.

```
 1 // Calculating square root of a number using Newton-Raphson method
 2
 3 #include <stdio.h>
 4
 5 // Function to calculate the absolute value of a number
 6 float abs (float x)
 7 {
 8         if (x < 0)
 9                 x = -x;
10         return (x);
11 }
12
13 // Function to compute the square root of a number
14 float squareRoot (float x)
15 {
16         const float epsilon = .00001;
17         float       guess   = 1.0;
18
19         while (abs(guess * guess - x) >= epsilon)
20                 guess = ((x / guess) + guess) / 2.0;
21
22         return guess;
23 }
24
```

```c
int main (void)
{
        float number = 0;

        printf ("Enter your number: ");
        scanf ("%f", &number);

        printf ("sqrt(%g) = %g\n", number, squareRoot(number));

        return 0;
}
```

Как видим, мы можем вызвать одну функцию из другой функции. Также обе функции имеют аргумент с именем `x` и это не приводит к ошибкам, потому что это локальные и таким образом не зависящие друг от друга переменные. То же самое касается и других локальных переменных внутри функции.
На самом деле, есть способы обращаться к локальным переменным извне функции, но до этого мы дойдем, когда будем изучать указатели.

В зависимости от системы, на которой работает програма (точнее, от особенностей её работы с числами с плавающей точкой), последние цифры результата могут различаться.

Константу `epsilon` можно изменять в соответствии с желаемой точностью вычислений.
Чем она меньше, тем точнее.

Прототипы функций. Объявление принимаемого и возвращаемого типа данных.

Как мы уже говорили, если мы не зададим функции возвращаемый тип данных, компилятор автоматически использует тип `int`. Но только этого недостаточно. Тип данных обязательно должен быть задан **до** того, как функция будет вызвана. Рассмотрим программу, которую мы только что написали. Если бы мы задали функцию `squareRoot()` перед `abs()`, то при попытке вызвать функцию `abs()`, она бы выдала ошибочное значение. Но компиляторы обычно не пропускают такие ошибки и как минимум выдают предупреждение вроде "implicit declaration of function".

Чтобы у нас была возможность объявить функцию `abs()` после функции `squareRoot()`, нужно перед объявлением `squareRoot()` объявить прототип функции `abs()`

```c
float  abs (float);
```

Можно также задать название переменной входного аргумента, но это делать не обязательно, т.к. компилятор все равно его проигнорирует. Поэтому ему даже не обязательно совпадать с названием, указанным при описании функции.

```c
float  abs (float  x);
```

Если же функция должна принимать разное количество аргументов (как `printf()`, например), то нужно сделать *указатель* на первый аргумент:

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

Можно звездочку ставить и после названия, суть не изменится:

```c
format*
```

Мы в каждой программе подключали библиотеку `stdio.h`, можно туда заглянуть и увидеть, что там есть прототип функции `printf()`, который выглядит именно так.

При написании пользовательских библиотек желательно всегда пользоваться прототипами, чтобы избегать ошибок. Да и прототипы служат удобным списком присутствующих в библиотеке функций. Также прототипы можно объявлять только внутри функций, которые их вызывают, так тоже будет проще понять, к каким другим функциям ваша функция обращается.

Кстати, при написании прототипа допустимо, чтобы названия входных параметров функций **не совпадали** с их названиями в оригинальной функции. Названия вообще **могут отсутствовать** так-то, можно просто через запятую писать типы аргументов. Главное чтобы совпадало название функции, типы

и количество переменных. Иногда названия опускаются для исключения случаев, когда название макроса может совпадать с названием аргумента функции, чтобы избежать нежелательной подстановки.

Проверка аргументов функций

Если мы попытаемся взять квадратный корень из отрицательного числа, нам придется иметь дело с комплексными числами. А алгоритм Ньютона-Рафсона зависнет в бесконечном цикле, так как значение `guess` не будет приближаться к истинному ответу и заданная точность никогда не будет достигнута.

Но мы ведь должны писать хорошие программы, которые не будут зависать, правда?
Можно пойти по самому очевидному пути и перед каждым вызовом функции проверять знак передаваемого в функцию аргумента, но у такого решения есть минусы. Например, у нас тысячи строк кода и десятки мест, где мы вызываем эту функцию. Забудем где-то реализовать проверку и будем потом рвать волосы, ища ошибку.

Намного удобнее будет реализовать проверку аргумента в самой функции. Таким образом, мы можем вызывать функцию как угодно и где угодно, не боясь подать в нее неправильный аргумент. Давайте перепишем программу для расчета квадратного корня, но в этот раз при вводе отрицательного числа функция будет выдавать ошибку.

прим.: кстати, функция `sqrt()` из библиотеки `math.h` тоже выдает ошибку, если ей передать отрицательное значение. Возвращает она разные значения, зависит от конкретной реализации библиотеки. Но обычно это не численное значение, т.е. при попытке его вывести, вместо числа будет `NaN` (not a number)

```c
// Calculating square root of a number using Newton-Raphson method

#include <stdio.h>

// Function to calculate the absolute value of a number
float abs (float x)
{
        if (x < 0)
                x = -x;
        return (x);
}

// Function to compute the square root of a number
float squareRoot (float x)
{
        const float epsilon = .00001;
        float       guess   = 1.0;
        float  abs (float  x);

        if (x < 0)
        {
                printf("Error: Negative argument to squareRoot.\n");
                return -1.0;
        }

        while (abs(guess * guess - x) >= epsilon)
                guess = ((x / guess) + guess) / 2.0;

        return guess;
}

int main (void)
{
        float number = 0;

        printf ("Enter your number: ");
        scanf ("%f", &number);

```

```
39          printf ("sqrt(%g) = %g\n", number, squareRoot(number));
40
41          return 0;
42  }
```

Введем отрицательное число и посмотрим, что произойдет:

```
Enter your number: -12
Error: Negative argument to squareRoot.
sqrt(-12) = -1
```

Обратите также внимание, что в функцию `squareRoot()` также добавился прототип функции `abs()`

```
float  abs (float  x);
```

Так что теперь мы можем не бояться насчет порядка описания функций.

Также, как мы уже говорили в первой главе урока, в функции может быть несколько операторов `return`, после выполнения которого функция завершится. Мы сделали две возможных точки выхода из функции с возвратом соответствующих значений: –1, если входной аргумент меньше нуля и рассчитаное значение квадратного корня входного числа, если оно положительное.

Функции и массивы

Точно так же, как мы передаем функции переменные и константы, мы можем передать ей в качестве аргумента элемент массива или даже весь массив целиком. Например, если нам нужно взять квадратный корень из элемента `array[i]` и записать результат в переменную `sqrt_result`, мы напишем:

```
sq_root_result = squareRoot (array[i]);
```

То есть, для работы с элементами массива ничего внутри функции `squareRoot()` менять не нужно.

Но если нужно передать весь массив целиком, то правила немного меняются. Чтобы функция могла принять массив, нужно при ее объявлении указать, каких размеров массив она должна принимать:

```
1  int  minimum (int  values[100])
2  {
3           ...
4           return minValue;
5  }
```

Примерно так будет выглядеть объявление функции, которая, например, находит минимальное значение массива из 100 чисел. А вызывается она точно так же, как обычные функции:

```
minimum (gradeScores)
```

При передаче такой функции только одного элемента массива программа скомпилируется, но в итоге закроется с ошибкой, когда он будет передан функции. Компилятор при этом выдал предупреждение:
`warning: passing argument 1 of 'minimum' makes pointer from integer without a cast`

Давайте теперь оформим это все в законченную программу.

```
1  #include <stdio.h>
2
3  // Function to find the minimum value in an array
4  int  minimum (int  values[10])
5  {
6          int  minValue, i;
7
8          minValue = values[0];
9
10         for (i = 1;  i < 10;  i++)
```

```
                if ( values[i] < minValue )
                      minValue = values[i];

        return minValue;
}

int main (void)
{
        int  scores[10], i, minScore;
        int  minimum (int  values[10]);

        printf ("Enter 10 scores:\n");

        for (i = 0;  i < 10;  i++)
                scanf ("%i", &scores[i]);

        minScore = minimum (scores);
        printf ("\nMinimum score is %i.\n", minScore);

        return 0;
}
```

Обратите внимание, что в начале `main()` мы объявили прототип функции `minimum()`. Здесь это не обязательно было делать, потому что сама функция была объявлена до её вызова, но лучше всегда объявлять прототипы используемых функций просто для того, чтоб избегать потенциальных ошибок.

Также массив `scores` не обязательно должен быть размера 10. Но следует помнить, что если мы подадим в функцию массив из, скажем, 5 элементов, то функция все равно будет обрабатывать 10 элементов, а не 5; а на месте незаполненных 5 элементов будет мусор. Если же мы подадим в функцию больше элементов, чем она может принять, лишние элементы будут проигнорированы.

Если нужно передавать в функцию разное количество элементов, нужно ее объявить несколько иначе: не указывать точное количество элементов массива, а передавать их количество во втором аргументе.

Изменим нашу функцию, чтобы она принимала массивы с разным количеством элементов.

```
#include <stdio.h>

// Function to find the minimum value in an array
int  minimum (int  values[], int  numberOfElements)
{
        int  minValue, i;

        minValue = values[0];

        for (i = 1;  i < numberOfElements;  i++)
                if ( values[i] < minValue )
                        minValue = values[i];

        return minValue;
}

int main (void)
{
        int  array1[5] = { 157, -28, -37, 26, 10 };
        int  array2[7] = { 12, 45, 1, 10, 5, 3, 22 };
        int  minimum (int  values[], int  numberOfElements);

        printf ("array1 minimum: %i\n", minimum (array1, 5));
        printf ("array2 minimum: %i\n", minimum (array2, 7));

        return 0;
}
```

Здесь мы по очереди передали в функцию массивы из 5 и 7 элементов и она корректно их обработала.

Особенности работы операторов присвоения при работе с массивами в функции

Когда мы передаем функции массив в качестве аргумента, он не передается туда в привычном виде, как обычная переменная. Вместо этого передается адрес памяти, то есть информация о том, где находится указанный массив. **Поэтому, в отличие от переменной, массив не дублируется в локальную переменную функции, а функция работает с переданным массивом напрямую**, изменяя его, если мы описали его изменение. Очень важно учитывать эту особенность, чтобы потом не удивляться, почему мы передали в функцию массив и она его изменила.

Рассмотрим на примере.

```c
#include <stdio.h>

void  multiplyBy2 (float  array[], int  n)
{
        int  i;

        for (i = 0;  i < n;  i++)
                array[i] *= 2;
}

int main (void)
{

        float  floatVals[4] = { 1.2f, -3.7f, 6.2f, 8.55f };
        int    i;
        void   multiplyBy2 (float  array[], int  n);

        multiplyBy2 (floatVals, 4);

        for (i = 0;  i < 4;  i++)
                printf ("%.2f   ", floatVals[i]);

        printf ("\n");

        return 0;
}
```

Программа выдаст:

```
2.40   -7.40   12.40   17.10
```

Как видим, функция `multiplyBy2()` не возвращает вообще никаких значений. Но, несмотря на это, переданный ей массив `floatVals` она успешно изменила.

Сортировка массивов

Для дальнейшей демонстрации того, как функция может изменять значения элементов передаваемых в нее массивов, напишем функцию, которая будет сортировать числа в массиве. Алгоритмы сортировки широко распространены и внимание к ним очень высоко. Так как этот курс не про хитрые алгоритмы сортировки, а про язык С, будем решать задачу в лоб.

Сортировку *n* элементов массива по возрастанию можно произвести последовательным сравнением каждого элемента. Начинаем со сравнения первого элемента со вторым. Если первый больше второго, меняем их местами. Затем сравниваем первый элемент с третьим. Опять же, если он больше третьего, меняем их местами. Когда закончим проверять первый элемент, начинаем сравнивать второй элемент с третьим, потом в второй с четвертым… И так далее, до конца.

Опишем это в виде алгоритма:

1. i = 0
2. j = i + 1
3. Если a[i] > a[j], поменять эти значения местами
4. j = j + 1. Если j < n, перейти к п.3
5. i = i + 1. Если i < n - 1, перейти к п.2

И напишем программу.

```c
#include <stdio.h>

// Function to sort numbers of array in ascending order
void  sort (int  array[], int  n)
{
        int  i, j, temp;

        for (i = 0;  i < n - 1;  i++)
                for (j = i + 1; j < n; j++)
                        if (array[i] > array[j]) {
                                temp = array[i];
                                array[i] = array[j];
                                array[j] = temp;
                        }
}

int main (void)
{
        int    i;
        int  numbers[16] = { 34, -5, 6, 0, 12, 100, 56, 22,
                             44, -3, -9, 12, 17, 22, 6, 11 };
        void sort (int  array[], int  n);

        sort (numbers, 16);

        for (i = 0;  i < 16;  i++)
                printf ("%i\n", numbers[i]);

        printf ("\n");

        return 0;
}
```

Алгоритм далёк от оптимального, но работает. Более оптимальные алгоритмы сортировки мы рассмотрим в курсе алгоритмики.

В библиотеке `stdlib.h` есть функция `qsort()`, которой можно пользоваться для сортировки. Но для работы с ней нужно понимать, как работают указатели в C. Так что её мы пока не рассматриваем. Но на практике рекомендуется не изобретать велосипед и писать собственную функцию сортировки только если функционала `qsort()` недостаточно.

Многомерные массивы

В функцию можно передавать не только одномерные массивы, но и многомерные. То есть, выражение

```
squareRoot (matrix[i][j]);
```

вызовет функцию `squareRoot()`, передав ей значение, хранящееся в `matrix[i][j]`.

Можно передать и весь массив целиком, просто написав его название (как мы делали с одномерными массивами). Выглядеть будет так:

```
scalarMultiply (measured_values, constant);
```

Функция `scalarMultiply()` может, например, умножать все элементы массива `measured_values` на некоторое число `constant`.

Присваивание функциями значений переданным в них многомерных массивах происходит точно так же, как и с одномерными: их содержание не дублируется в локальную переменную функции, а работа идет напрямую с переданным массивом, т.е. тоже передается не сам массив, а его адрес в памяти.

Когда мы объявляли одномерный массив как аргумент функции, нам не обязательно было указывать количество элементов, мы могли поставить пустые скобки и передавать сколько угодно элементов.
С многомерными массивами немного не так. Мы можем не указывать фиксированное количество строк, но обязательно должны указать количество столбцов. То есть, **правильно** писать:

```
int  array_values[100][50]
```

или

```
int  array_values[][50]
```

А **неправильно** будет:

```
int  array_values[100][]
```

или

```
int  array_values[][]
```

Для закрепления материала напишем программу, которая будет умножать двумерный массив на некоторое скалярное значение.

```
#include <stdio.h>

int main (void)
{

        void  scalarMultiply (int  matrix[3][5], int  scalar);
        void  displayMatrix (int  matrix[3][5]);
        int   sampleMatrix[3][5] =
                    {
                            {  7, 16, 55, 13, 12 },
                            { 12, 10, 52,  0,  7 },
                            { -2,  1,  2,  4,  9 }
                    };

        printf ("Original matrix:\n");
        displayMatrix (sampleMatrix);

        scalarMultiply (sampleMatrix, 2);

        printf ("\nMultiplied by 2:\n");
        displayMatrix (sampleMatrix);

        scalarMultiply (sampleMatrix, -1);

        printf ("\nThen multiplied by -1:\n");
        displayMatrix (sampleMatrix);

```

```
28          return 0;
29  }
30
31  void scalarMultiply (int matrix[3][5], int scalar)
32  {
33          int row, column;
34
35          for (row = 0; row < 3; row++)
36                  for (column = 0; column < 5; column++)
37                  matrix[row][column] *= scalar;
38  }
39
40  void displayMatrix (int matrix[3][5])
41  {
42          int row, column;
43
44          for (row = 0; row < 3; row++) {
45                  for (column = 0; column < 5; column++)
46                          printf("%5i", matrix[row][column]);
47
48                  printf("\n");
49          }
50  }
```

Здесь мы также объявили функции после `main()`. Лично мне такой подход кажется более удобным, но нужно не забывать про прототипы, в этом случае они обязательны для правильной работы программы.

Функции и многомерные массивы переменной длины

Да, функции умеют с ними работать. Перепишем нашу предыдущую программу, чтоб она принимала массивы с любым количеством строк и столбцов.

```
1  #include <stdio.h>
2
3  int main (void)
4  {
5
6      void  scalarMultiply (int nRows, int nCols, int matrix[nRows][nCols], int scalar);
7      void  displayMatrix (int nRows, int nCols, int matrix[nRows][nCols]);
8      int   sampleMatrix[3][5] =
9                  {
10                         {  7, 16, 55, 13, 12 },
11                         { 12, 10, 52,  0,  7 },
12                         { -2,  1,  2,  4,  9 }
13                 };
14
15     printf ("Original matrix:\n");
16     displayMatrix (3, 5, sampleMatrix);
17
18     scalarMultiply (3, 5, sampleMatrix, 2);
19
20     printf ("\nMultiplied by 2:\n");
21     displayMatrix (3, 5, sampleMatrix);
22
23     scalarMultiply (3, 5, sampleMatrix, -1);
24
25     printf ("\nThen multiplied by -1:\n");
26     displayMatrix (3, 5, sampleMatrix);
27
28     return 0;
29 }
```

```
30
31 void scalarMultiply (int nRows, int nCols, int matrix[nRows][nCols], int scalar)
32 {
33     int row, column;
34
35     for (row = 0; row < nRows; row++)
36             for (column = 0; column < nCols; column++)
37             matrix[row][column] *= scalar;
38 }
39
40 void displayMatrix (int nRows, int nCols, int matrix[nRows][nCols])
41 {
42     int row, column;
43
44     for (row = 0; row < nRows; row++) {
45             for (column = 0; column < nCols; column++)
46                 printf("%5i", matrix[row][column]);
47
48             printf("\n");
49     }
50 }
```

Важно, чтобы количество строк (`nRows`) и столбцов (`nCols`) массива matrix были объявлены как аргументы функции до объявления самой матрицы, иначе, если мы попытаемся сначала объявить массив `matrix[nRows][nCols]`, то компилятор не поймет, кто такие `nRows` и `nCols` на месте строк и столбцов. То есть, вот так **работать не будет**:

```
void  scalarMultiply (int matrix[nRows][nCols], int nRows, int nCols, int  scalar)
```

и компилятор выдаст ошибку.

Глобальные переменные

Глава получилась очень большая, давайте немного подытожим выученное и добавим немного нового.

Вспомним программу из урока 1.5 (Массивы), которая преобразовывала систему счисления для введенного десятичного числа и реализуем её в виде функции. Перед тем, как приступить к написанию, нужно разбить алгоритм на функциональные сегменты. Фактически мы это уже там сделали, достаточно почитать комментарии в коде: программа состоит из трех этапов – получение числа и желаемой системы счисления от пользователя, конверсия числа в указанную систему счисления и отображение результата.

Три этапа – три функции. Первую функцию, которую мы будем вызывать, назовем `getNumberAndBase()`. Она будет спрашивать у пользователя число и систему счисления. Помимо основного функционала, добавим немного плюшек. Изначально у нас нет защиты от ввода некорректных данных и в функции мы реализуем эту защиту.
Если пользователь введет значение основания системы счисления меньше 2 или больше 16, функция выведет соответствующую ошибку и установит основание равным 10 (то есть, программа никак не преобразует число, ибо оно изначально десятичное). Также можно попросить пользователя ввести основание заново, но это мы разберем в упражнениях к этой главе.

Вторую функцию назовем `convertNumber()`. Эта функция будет брать введенное пользователем число и основание системы счисления и преобразовывать систему счисления числа в заданную пользователем, записывая результаты своей работы в массив `convertedNumber`.

Третья и последняя функция – `displayConvertedNumber()`. Она считывает цифры из массива `convertedNumber` и показывает их пользователю в правильном порядке и формате. Внутри функции также нужно будет описать массив констант baseDigits, из которого будут браться соответствующие каждому числу цифры и буквы (A - F для 10 - 16).

Три функции будут между собой обмениваться информацией при помощи *глобальных переменных*. Как мы уже говорили в этом уроке, основное свойство локальных переменных – то, что доступ к их значению

может быть получен только той функцией, в которой эти переменные были объявлены.
Как подсказывает их название, у глобальных переменных нет такого ограничения. Доступ к глобальным переменным есть у всех функций в программе.

Отличие глобальных переменных в том, что они объявляются вне функций. Это указывает на саму их суть – они не принадлежат никакой функции, они сами по себе. Так что любая функция может считывать и модифицировать их значение без всяких преград.

Напишем программу для конвертации системы счисления числа с использованием описанных выше трех функций и также используем глобальные переменные.

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

#include <stdio.h>

int      convertedNumber[64];
long int numberToConvert;
int      base, index = 0;

int main (void)
{
void  getNumberAndBase (void), convertNumber (void),
      displayConvertedNumber (void);

getNumberAndBase();
convertNumber();
displayConvertedNumber();

return 0;
}

// Function to get the number and the base
void getNumberAndBase(void) {
        printf ("Number to be converted? ");
        scanf ("%ld", &numberToConvert);
        printf ("Base? ");
        scanf ("%i", &base);

        if (base < 2 || base > 16) {
                printf("Error: base must be between 2 and 16.\n");
                base = 10;
        }
}

// Function to convert to the indicated base
void convertNumber (void) {
        do {
                convertedNumber[index] = numberToConvert % base;
                index++;
                numberToConvert = numberToConvert / base;
        }
        while  ( numberToConvert != 0 );
}

// Function display the results in reverse order
void displayConvertedNumber (void) {
        const char baseDigits[16] = {'0', '1', '2', '3', '4', '5', '6', '7',
                                     '8', '9', 'A', 'B', 'C', 'D', 'E', 'F' };
        int nextDigit;

        printf ("Converted number = ");

```

```
52          for (index--;  index >= 0;  index--) {
53                      nextDigit = convertedNumber[index];
54                      printf ("%c", baseDigits[nextDigit]);
55          }
56
57          printf ("\n");
58  }
```

Забавно, что программа заработала правильно несмотря на то, что я в `main()` забыл написать `return 0` и не добавил прототипы трех используемых там функций. Компилятор выдал предупреждения вроде `implicit declaration`, но, видимо, скорректировал мои ошибки.

Теперь, когда мы разбили нашу задачу на три четко определенных функции, достаточно посмотреть в `main()`, чтобы понять, что и как делает программа. А благодаря тому, что названия функций четко отражают их суть, не нужно проверять код функции, чтоб понять, что она делает.

Тут еще, кстати, видно, что не обязательно указывать тип каждой функции отдельно, т.е. если у нас, например, три функции, возвращающие `void`, мы можем их написать через запятую. Как переменные.

Изначально я очень феерично описал глобальные переменные, как будто они очень удобные и ими стоит постоянно пользоваться. Это не совсем так. Например, глобальные переменные снижают гибкость использования функций, например, когда нужно использовать написанную функцию в другой программе, нужно убедиться, что используемые функцией глобальные переменные вообще существуют (т.е. были объявлены). Глобальные переменные хоть и удобны для обмена данными между функциями, но злоупотреблять ими не стоит, лучше по возможности пользоваться локальными переменными.

Например, у нас получилось, что работоспособность функции `convertNumber()` зависит от аж четырех глобальных переменных: `numberToConvert, base, index и convertedNumber[]`. Правильнее было бы написать функцию так, чтобы она принимала переменные в качестве аргумента, а не использовала исключительно глобальные переменные.

Программы с большим количеством глобальных переменных также труднее читать. Для упрощения отличения локальных переменных от глобальных часто ставят перед их названием букву g:

```
int       gCurrentMove;
float     gDigit[64];
```

И, например, если читающий код увидит выражение

```
nextMove = gCurrentMove + 1;
```

то сразу поймет, что `nextMove` – локальная переменная, а `gCurrentMove` – глобальная, и ему не нужно будет пролистывать код до участка, где объявлены переменные, чтоб узнать тип переменной.

Еще одно важное замечание по поводу глобальных переменных. Они всегда автоматически обнуляются при старте программы, в отличие от локальных переменных. То есть, например, если мы объявим массив

```
int  gData[100];
```

то нам не нужно его обнулять, он сразу будет нулевым.

Автоматические и статические переменные

Когда мы объявляем локальные переменные внутри функции, им присваивается *автоматический тип продолжительности*. Этот тип также можно явно задать ключевым словом `auto`, но если этого слова нет, компилятор по умолчанию создает автоматическую переменную.

Суть автоматических переменных в том, что они создаются *каждый раз* при вызове функции и *прекращают существовать* при завершении функции и значение, записанное в них, тоже пропадает. Другими словами, значение автоматической переменной *гарантированно не существует* при повторном вызове функции.

Если при объявлении переменной мы используем ключевое слово `static`, то зададим переменной

статический тип продолжительности. То есть, такая переменная будет сохранять свое значение после выхода из функции.

Также статические переменные иначе инициализируются: не каждый раз при вызове функции, а при старте программы, т.е. еще до того, как функция будет вызвана. Если же мы явно не задали начальное значение переменным, они будут автоматически обнулены, в отличие от автоматических переменных, у которых нет начального значения (т.е. если их не инициализировать, в них будет что-то рандомное).

Синтаксис объявления статических переменных не отличается от объявления автоматических:

```
1 void auto_static (void)
2 {
3     static int  staticVar = 100;
4             . . .
5 }
```

Зато немного отличается логика работы. Здесь переменной `staticVar` будет задано значение 100 только при старте программы, то есть при повторном вызове функции переменная сохранит свое значение, оставшееся после предыдущего ее вызова.

Продемонстрируем различия между автоматической и статической переменной на простом примере:

```
1  // Program to illustrate static and automatic variables
2
3  #include <stdio.h>
4
5  void  auto_static (void)
6  {
7          int         autoVar = 1;
8          static int  staticVar = 1;
9
10         printf ("automatic = %i, static = %i\n", autoVar, staticVar);
11
12         ++autoVar;
13         ++staticVar;
14 }
15
16
17 int main (void)
18 {
19         int   i;
20         void  auto_static (void);
21
22         for ( i = 0;  i < 5;  ++i )
23                     auto_static ();
24
25         return 0;
26 }
```

Программа выдаст:

```
automatic = 1, static = 1
automatic = 1, static = 2
automatic = 1, static = 3
automatic = 1, static = 4
automatic = 1, static = 5
```

Здесь в функции `auto_static()` объявлено две локальных переменных: автоматическая переменная `autoVar` и статическая `staticVar`. Сама функция печатает текущее значение переменных, а затем

увеличивает их значение на 1.

В `main()` реализован цикл, который вызывает функцию 5 раз подряд. Из выданного программой результата видно, что `autoVar` при каждом вызове функции сбрасывает свое значение до 1, а `staticVar` сохраняет свое предыдущее значение после выхода из программы.

Выбор между статической и автоматической переменной зависит от назначения этой переменной. Статические переменные используются, если нужно, чтоб переменная сохраняла свое значение от вызова к вызову (например, если нужно посчитать, сколько раз была вызвана функция). Также использование статических переменных ускоряет работу функции, если переменную нужно инициализировать только один раз. Представьте себе огромный массив автоматического типа: при каждом вызове функции программа будет его инициализировать, а пользователь при этом сильно злиться, что программа тормозит. Оптимальнее будет сделать задать такому массиву статический тип продолжительности и не тратить каждый раз время на ненужную повторную инициализацию.

Если же нужно, чтобы переменная возвращалась к исходному значению при каждом вызове функции, оптимальнее будет использовать автоматические переменные.

Ключевое слово `static` в применении к функциям

Помимо переменных, слово `static` можно применять и к функциям:

```
static void static_function ();
```

Статическая функция может быть вызвана только **внутри** того файла, в котором она была объявлена. Это иногда полезно при работе с несколькими файлами. Обратите внимание, что при рекурсивном вызове функции все вызванные функции будут обращаться к **одной и той же** переменной, т.е. ее копии **не будут создаваться**, как это происходит с автоматическими переменными.

Ключевое слово `static` в применении к массивам

Стандарт C99 позволяет использовать слово `static` при объявлении массива в качестве параметра функции. В примере ниже `static` указывает компилятору, что массив **гарантированно** будет состоять **как минимум** из 3 элементов:

```
int sum_array ( int a[static 3], int n)
{
     ...
}
```

Иногда это позволяет компилятору оптимизировать код, например он может предзагрузить элементы массива до того как они будут нужны в функции. В случае с многомерными массивами слово `static` можно применять только к первому измерению.

Рекурсивные функции

Рекурсивные функции используются для компактного и эффективного решения многих задач. В основном они используются там, где задача решается через последовательное применение одного и того же решения к подмножествам(??) задачи, например как в случае вычисления факториала. Также рекурсивные функции применяются в алгоритмах поиска и сортировки.

Рассмотрим рекурсивные функции на примере вычисления факториала числа. Он равен произведению всех чисел от 1 до n:

$$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$$

Очевидно, что $6! = 6 \times 5! = 720$.

Из этих двух выражений можем выразить формулу для факториала в общем виде:

`n! = n × (n - 1)!`

То есть, значение n! имеет *рекурсивное определение*, потому что значение итогового факториала основано на значении предыдущих факториалов.

Напишем программу, которая выведет факториалы от `0!` до `10!`:

```c
#include <stdio.h>

int main (void)
{
        unsigned int i;
        unsigned long int factorial (unsigned int n);

        for (i = 0; i < 11; i++)
                printf("%2u! = %lu\n", i, factorial(i));

        return 0;
}

// Recursive function to calculate the factorial of a positive integer

unsigned long int factorial (unsigned int n)
{
        unsigned long int result;

        if (n == 0)
                result = 1;
        else
                result = n * factorial(n - 1);

        return result;
}
```

Здесь мы задали аргументу функции тип `unsigned int`, потому что факториалы можно брать только из положительных целых чисел и мы можем удвоить доступный нам диапазон чисел, отказавшись от отрицательных значений. Переменная счетчика также сделана типом `unsigned int` для совместимости с типом аргумента фунции.

Тот факт, что мы вызываем функцию `factorial()` из самой себя, делает ее рекурсивной. Когда, например, мы рассчитываем `factorial (3)`, аргумент n равен 3. Так как n ≠ 0, выполняется строка

```c
result = n * factorial(n - 1);
```

то есть рассчитывается значение

```c
result = 3 * factorial (2);
```

Получается, что `factorial(3)` вызывает функцию `factorial(2)` и ждет, пока она вернет значение. А `factorial(2)` в свою очередь вызывает `factorial(1)`. И в конце `factorial(1)` вызывает `factorial(0)`, которая возвращает 1, потому что выполняется условие `n == 0`.
Затем рекурсия "сворачивается" обратно к изначальной функции, потому что функции начинают в обратном порядке возвращать свои значения.

Рассмотрим этот процесс на схеме.

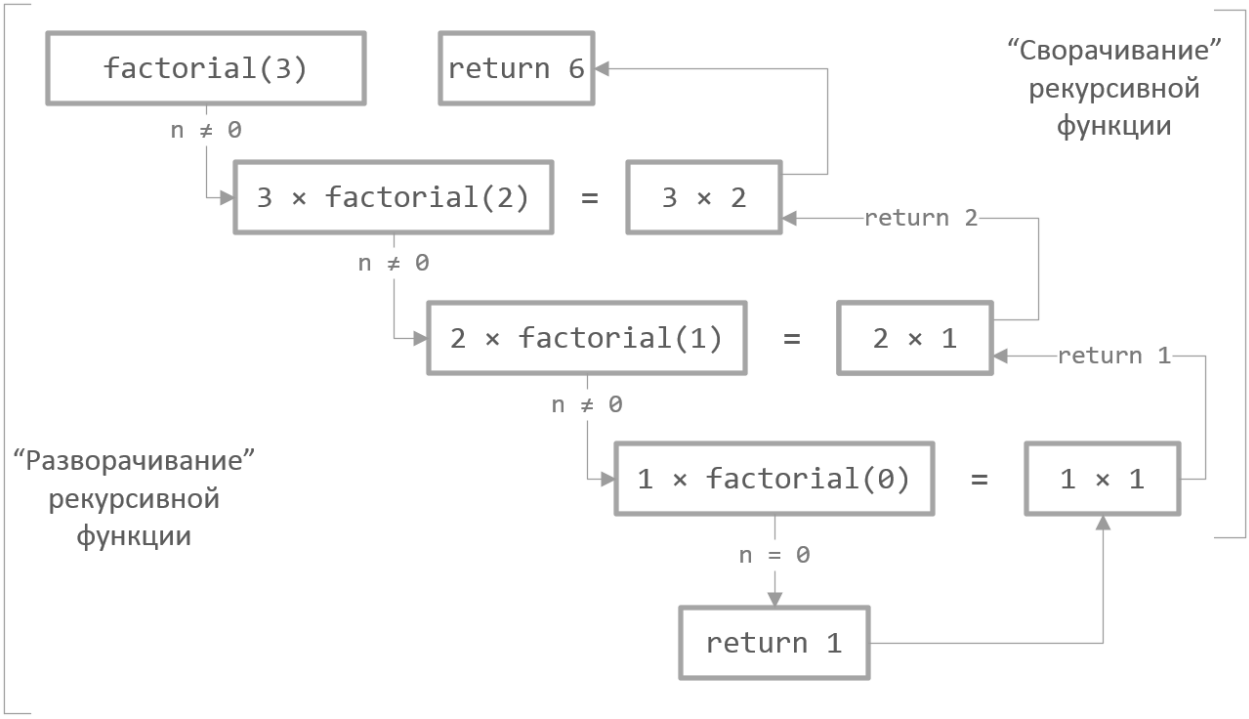

Схема работы рекурсивной функции

Мы видим, что сначала рекурсивная функция "разворачивается", создавая несколько отдельных индивидуальных функций (со своими индивидуальными локальными переменными), и этот процесс продолжается до тех пор, пока не появится функция, которая наконец не будет вызывать следующую функцию, а вернет какое-нибудь значение (что в нашем случае делает `factorial(0)`).
После этого рекурсия начинает "сворачиваться", потому что ждущие результата функции наконец начинают его получать и могут в свою очередь передать результаты своей работы вызвавшей их функции и завершить свою работу.

Упражнения

2. *Modify Program 7.4 so the value of triangularNumber is returned by the function.*

```c
// Function to calculate the nth triangular number

#include <stdio.h>

int calculateTriangularNumber (int n)
{
        int  i, triangularNumber = 0;

        for ( i = 1;  i <= n;  ++i )
                triangularNumber += i;

        return triangularNumber;
}

int main (void)
{
        int result;
        int n = 10;

        result = calculateTriangularNumber (n);

        printf ("Triangular number %i is %i\n", n, result);

```

```
24          return 0;
25  }
```

3. *Modify Program 7.8 so that the value of epsilon is passed as an argument to the function. Try experimenting with different values of epsilon to see the effect that it has on the value of the square root.*
4. *Modify Program 7.8 so that the value of guess is printed each time through the while loop. Notice how quickly the value of guess converges to the square root.*
5. *The criteria used for termination of the loop in the squareRoot() function of Program 7.8 is not suitable for use when computing the square root of very large or very small numbers. Rather than comparing the difference between the value of x and the value of guess2, the program should compare the ratio of the two values to 1.*
6. *Modify Program 7.8 so that the squareRoot() function accepts a double precision argument and returns the result as a double precision value.*

```c
#include <stdio.h>

// Function to calculate the absolute value of a number

double absoluteValue (double x)
{
    if ( x < 0 )
            x = -x;
    return (x);
}

// Function to compute the square root of a number

double squareRoot (float x, double epsilon)
{
    double guess = 1.0;

    while  ( absoluteValue ((x / (guess * guess)) - 1) >= epsilon ) {
            guess = ( x / guess + guess ) / 2.0;
            printf ("guess = %.16g\n", guess);
    }

    return guess;
}

int main (void)
{
    const double precision = 1e-15;
    float number;

    printf("Enter number: ");
    scanf ("%f", &number);

    printf ("\nsquareRoot(%g) = %.16g\n", number, squareRoot(number, precision));

    return 0;
}
```

В коде выполнены задания 3 - 6.

7. *Write a function that raises an integer to a positive integer power. Call the function `x_to_the_n()` taking two integer arguments x and n. Have the function return a `Long int`.*

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

```c

// Function to calculate x^n

long int x_to_the_n (long int x, int n)
{
        int  i;
        int buf;
        long int prev_x;

        buf = x;

        for ( i = 1;  i < n;  i++ ) {
                prev_x = x; //store previous value to check the overflow
                x *= buf;

                if (prev_x != 0 && x / prev_x != buf) {
                        printf("\nError: integer overflow!");
                        return -1;
                }
        }

        return x;
}

int main (void)
{
        long int x;
        int n;

        scanf("%li", &x);
        scanf("%i", &n);

        printf ("\n%li^%i = %li\n", x, n, x_to_the_n(x, n));

        return 0;
}
```

Забавный опыт с переполнением знаковой переменной. Беззнаковый тип `unsigned int` не переполняется, кстати. См. [2] и [3] для подробностей и методов проверки на переполнение.

8. *Write a program to solve a quadratic equation. The program should allow the user to enter the values for a, b, and c. If the discriminant is less than zero, a message should be displayed that the roots are imaginary; otherwise, the program should then proceed to calculate and display the two roots of the equation.*

```c
#include <stdio.h>

// Function to calculate the absolute value of a number

double absoluteValue (double x)
{
        if ( x < 0 )
                x = -x;
        return (x);
}

// Function to compute the square root of a number

double squareRoot (float x, double epsilon)
{
        double guess = 1.0;

        while  ( absoluteValue ((x / (guess * guess)) - 1) >= epsilon ) {
                guess = ( x / guess + guess ) / 2.0;
```

```
        }

        return guess;
}

int main (void)
{
        const double precision = 1e-15;

        float a, b, c, discr;
        double x1, x2;

        printf("    ax^2 + bx + c = 0\n\n");
        printf("Enter equation coefficients\na = ");
        scanf ("%g", &a);
        printf("b = ", &b);
        scanf ("%g", &b);
        printf("c = ", &c);
        scanf ("%g", &c);

        discr = b * b - 4 * a * c;

        if (discr < 0)
                printf("Error: equation roots are imaginary!");
        else
                x1 = (-b + squareRoot (discr, precision)) / (2 * a);
                x2 = (-b - squareRoot (discr, precision)) / (2 * a);

                printf ("\nx1 = %g\nx2 = %g\n", x1, x2);

        return 0;
}
```

9. *Write a function lcm() that takes two integer arguments and returns their lcm. The lcm() function should calculate the least common multiple by calling the gcd() function from Program 7.6.*

```
#include <stdio.h>

// function to find the greatest common divisor

int  gcd (int u, int v)
{
        int  temp;

        while ( v != 0 ) {
                temp = u % v;
                u = v;
                v = temp;
        }

        return u;
}

// function to find the least common multiple

float lcm (int u, int v)
{
        float lcm;

        if (u < 0 || v < 0) {
                printf ("Error: negative arguments into lcm.\n");
                return -1;
        }
        else
```

```
29                  lcm = (u * v) / gcd(u, v);
30
31          return lcm;
32  }
33
34  int main (void)
35  {
36          printf ("The lcm of 15 and 10 is %g.\n", lcm (15, 10));
37
38          return 0;
39  }
```

10. *Write a function prime() that returns 1 if its argument is a prime number and returns 0 otherwise.*

```
1  #include <stdio.h>
2
3  // function to check if number is prime
4
5  int isPrime (int num)
6  {
7          int isPrime = 1;
8          int i;
9
10         if (num <= 1)
11                 return 0;
12         else
13                 for (i = 2; i < num; i++)
14                         if (num % i == 0)
15                                 isPrime = 0;
16
17         return isPrime;
18 }
19
20 int main (void)
21 {
22         int num;
23
24         printf ("Enter the number to test: ");
25         scanf ("%i", &num);
26
27         printf("\n%i\n", isPrime(num));
28
29         return 0;
30 }
```

11. *Write a function called arraySum() that takes two arguments: an integer array and the number of elements in the array. Have the function return as its result the sum of the elements in the array.*

```
1  #include <stdio.h>
2
3  // function to find the sum of all elements in array
4
5  int arraySum (int array[], int elemNum)
6  {
7          int i, sum = 0;
8
9          for (i = 0; i < elemNum; i++)
10                 sum += array[i];
11
12         return sum;
13 }
14
15 int main (void)
```

```c
{
        int array[10];
        int i;

        for (i = 0; i < 10; i++)
                array[i] = 2*i;

        printf ("Sum of elements = %i\n", arraySum(array, 10));

        return 0;
}
```

12. *A matrix M with i rows, j columns can be transposed into a matrix N having j rows and i columns by simply setting the value of $N_{a,b}$ equal to the value of $M_{b,a}$ for all relevant values of a and b.*
Using variable-length arrays, write the transposeMatrix() function that takes the number of rows and columns as arguments, and to transpose the matrix of the specified dimensions. Also write a main() routine to test the function.

```c
#include <stdio.h>

// function to find the sum of all elements in array

void transposeMatrix (int height, int width,
                      int M[][50], int N[][50])
{
        int i, j;

        for (i = 0; i < height; i++)
                for (j = 0; j < width; j++)
                        N[j][i] = M[i][j];
}

int main (void)
{
        static int n = 5;
        static int m = 10;

        int M[n][50];
        int N[m][50];

        int i, j;

        for (i = 0; i < n; i++) {
                printf("\n");

                for (j = 0; j < m; j++) {
                        M[i][j] = 2*i + j;

                        printf ("%-2i  ", M[i][j]);
                }
        }

        transposeMatrix(n, m, M, N);

        printf("\n\nTransposed matrix: \n\n");

        for (i = 0; i < m; i++) {
                printf("\n");

                for (j = 0; j < n; j++) {
                        printf ("%-2i  ", N[i][j]);
                }
```

```
45          }
46
47          printf("\n");
48
49          return 0;
50  }
```

Не нравится мне эта задача. Синтаксис разве что закрепить.
Ширина массива ведь не может быть переменной длины и получается, что размер второго массива ограничен размерами первого и делать ему переменную длину нет смысла. А если второй массив фиксированного размера, то смысл делать первый массив переменного размера?

13. *Modify the sort() function from Program 7.12 to take a third argument indicating whether the array is to be sorted in ascending or descending order. Then modify the sort() algorithm to correctly sort the array into the indicated order.*

```
1  void  sort (int  a[], int  n, int direction)
2  {
3      int  i, j, temp;
4
5      for ( i = 0;  i < n - 1;  ++i )
6              for ( j = i + 1;  j < n;  ++j )
7                      if ((direction == 1 && (a[i] > a[j])) ||
8                          (direction == -1 && (a[i] < a[j]))) {
9                              temp = a[i];
10                             a[i] = a[j];
11                             a[j] = temp;
12                     }
13 }
```

Весь код целиком в приложенных исходниках.

15. *Modify Program 7.14 so that the user is asked again to type in the value of the base if an invalid base is entered. The modified program should continue to ask for the value of the base until a valid response is given.*

```
1  void  getNumberAndBase (void)
2  {
3      printf ("Number to be converted? ");
4      scanf ("%li", &numberToConvert);
5
6      printf ("Base? ");
7      scanf ("%i", &base);
8
9      if  ( base < 2  ||  base > 16 ) {
10         printf ("Bad base - must be between 2 and 16\n");
11
12         getch();
13         system("cls");
14         getNumberAndBase ();
15     }
16 }
```

В задании опечатка, там имеется в виду программа 7.15. Для задания достаточно было добавить повторный вызов функции getNumberAndBase (), но решил сделать красивее, со сбросом командной строки. Поговаривают, что через system() нехорошо сбрасывать консоль [4], а getch() – функция из сильно устаревшей библиотеки conio, но по-другому пока не сумел.

16. *Modify Program 7.14 so that the user can convert any number of integers. Make provision for the program*

to terminate when a zero is typed in as the value of the number to be converted.

```
int main (void)
{
    void  getNumberAndBase (void), convertNumber (void),
          displayConvertedNumber (void);

    while (1) {
          getNumberAndBase ();
          if (numberToConvert == 0)
              break;

          convertNumber ();
          displayConvertedNumber ();

          getch();
          system("cls");
    }

    return 0;
}
```

Источники:

1. Stephen Kochan – Programming in C (4th Edition); chapter 7
2. https://stackoverflow.com/questions/1815367/catch-and-compute-overflow-during-multiplication-of-two-large-integers
3. https://stackoverflow.com/questions/199333/how-do-i-detect-unsigned-integer-multiply-overflow
4. http://www.cplusplus.com/articles/4z18T05o/