# 1 Задания к лабораторным работам

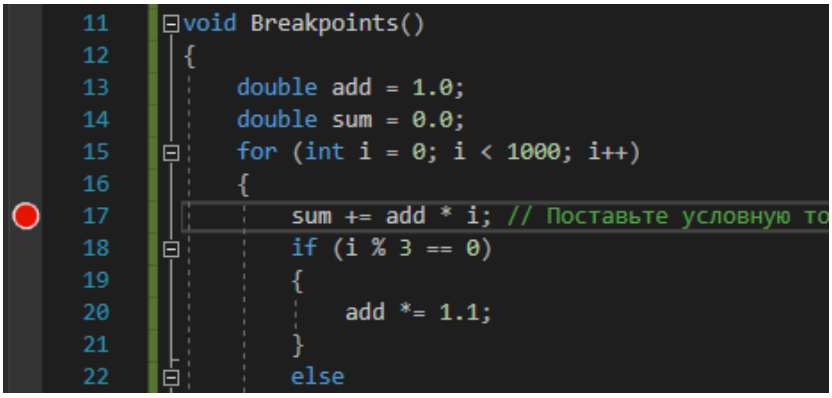
## 1.1 Лабораторная работа #1 Базовые элементы процедурного программирования

*Прежде чем переходить к изучению объектно-ориентированного программирования, следует повторить ряд программных техник, таких как функции, указатели и динамическая память, обработка исключений и другие. Решите ряд задач для подготовки, а затем выполните проверочную работу.*

### 1.1.1 Алгоритмы и отладка

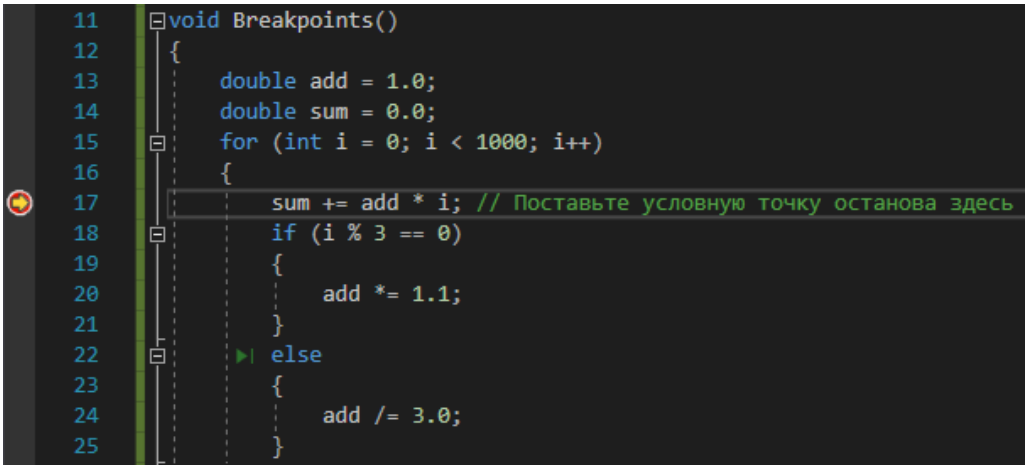
*Современные программы состоят из сложных алгоритмов, с большим количеством функций, вызывающих друг друга, и передающих друг другу значения. Увы, чтение кода далеко не всегда позволяет понять, как работает программа, или почему во время выполнения программы вылетает ошибка. Для анализа поведения программ во время их работы в средах разработки добавлены* ***точки останова*** *(breakpoints).*

**Точка останова** – это инструмент в среде разработки, позволяющий временно приостановить выполнение программы в заданной строчке кода.




**Для установки точки останова (красный кружок) необходимо кликнуть мышью слева от требуемой строчки кода**

Теперь при запуске программы в режиме отладки (Start Debugging, F5) программа прервется как только её выполнение дойдет до отмеченной строчки кода.




**Желтая стрелка во время отладки указывает,
какая строка должна выполниться далее**

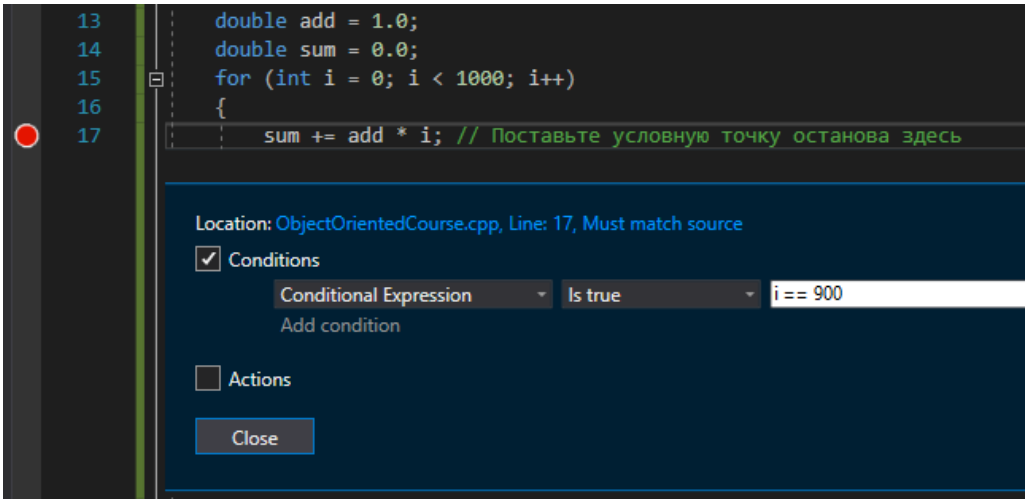
Во время работы точки останова разработчик может:

- посмотреть значения переменных, наведя на них курсор мыши;
- выполнить одну строку кода, нажав F10;
- продолжить работу программы, нажав F5 (до завершения программы или срабатывания следующей точки останова);
- прервать выполнение программы, нажав Shift + F5.

Нужно помнить, что точки останова будут работать только в режиме отладки (Start Debugging, F5). Если же запускать программу без отладки (Start Without Debugging, Shift + F5), точки останова срабатывать не будут. Также точка останова не будет срабатывать, если строка кода, в которой она установлена, не выполнится. Например, если вы, как разработчик, поставили точку останова внутри условия if, которое никогда не будет истинным.

Точка останова будет срабатывать каждый раз, когда программа достигнет отмеченной строки кода. Например, если точка останова стоит в цикле, то она сработает столько раз, сколько итераций в цикле. Другой пример, если точка останова стоит в функции, которая вызывается несколько раз, то точка останова внутри функции сработает столько раз, сколько вызовется данная функция.

Иногда нужно узнать значение тех или иных переменных внутри цикла или функции, которые часто вызываются. Однако, если цикл выполняет, скажем, 1000 итераций, а нас интересуют значения переменных на 900 итерации, то останавливать и запускать точку останова 900 раз может стать долгим занятием. Чтобы не тратить время, точке останова можно указать условие, при котором она должна сработать. Для этого надо кликнуть правой кнопкой по установленной точке останова и выбрать пункт Conditions. В появившемся окне надо прописать условие срабатывания точки останова:




**Точке останова можно задать условие, при котором она сработает.**
**Например, при таком условии точка сработает, когда будет достигнута 900-я итерация**

В программе можно установить любое количество точек останова. Чтобы удалить точку останова, достаточно кликнуть по ней левой кнопкой мыши.

Точки останова – важнейший инструмент разработчика. Без него сложно изучать работу программы или искать ошибки.

1.1.1.1 Перепишите и скомпилируйте данный пример. Поставьте точку останова внутри цикла. Запишите, какие значения принимает переменная `sum` на каждой итерации:

```
#include <iostream>

using namespace std;

void Breakpoints()
{
   double add = 1.0;
   double sum = 0.0;
   for (int i = 0; i < 10; i++)
   {
      sum += add * i;
      add *= 1.1;
   }
   cout << "Total sum is " << sum << endl;
}

int main()
{
   Breakpoints();
```

```
}
```

1.1.1.2 Перепишите и скомпилируйте данный пример. Поставьте точку останова в строке внутри цикла, где указан комментарий. Укажите точке останова условие, когда `i` будет равен 777. Запишите, какое значение принимает переменная `sum` на 777 итерации:

```
#include <iostream>

using namespace std;

void Breakpoints()
{
   double add = 1.0;
   double sum = 0.0;
   for (int i = 0; i < 1000; i++)
   {
      sum += add * i; // Поставьте условную точку останова здесь
      if (i % 3 == 0)
      {
         add *= 1.1;
      }
      else
      {
         add /= 3.0;
      }
   }
   cout << "Total sum is " << sum << endl;
}

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

### 1.1.2 Массивы

1.1.2.1 Напишите в функции `main()` блок кода. В блоке кода создайте массив целых чисел на 10 элементов. Проинициализируйте массив значениями непосредственно в коде программы. Выведите массив на экран с помощью цикла `for`. Напишите код, который выполняет сортировку целочисленного массива по возрастанию (можно написать любой алгоритм сортировки, например, полного перебора или метод пузырька). Выведите отсортированный массив на экран.

Пример работы кода:

```
Source array is:
12 21 119 -80 300 75 81 -8 47 31

Sorted array is:
-80 -8 12 21 31 47 75 81 119 300
```

1.1.2.2 Напишите в функции `main()` блок кода. В блоке кода создайте массив вещественных чисел на 12 элементов. Проинициализируйте массив значениями в программном коде. Выведите массив на экран с помощью цикла `for`. Считайте вещественное значение `searchingValue` с клавиатуры. Затем подсчитайте количество элементов массива, значение которых больше или равно введенному пользователем значению `searchingValue`.

Пример работы кода:

```
Source array is:
12.0 21.5 119.2 -80.7 300.0 75.5 81.2 8.1 47.3 31.2 85.3 100.1

Enter searching value: 80.0
Elements of array more than 80.0 is: 5
```

1.1.2.3 Напишите в функции `main()` блок кода. В блоке кода создайте символьный массив на 8 элементов. Проинициализируйте массив значениями с клавиатуры с помощью цикла `for`. Выведите исходный массив на экран. Затем, с помощью цикла `for` переберите массив и выведите все символы, которые являются буквами от `'a'` до `'z'`.

Пример работы кода:

```
Enter array of 8 chars
a[0]: a
a[1]: 1
a[2]: 5
a[3]: @
a[4]: f
a[5]: %
a[6]: z
a[7]: u
```

```
Your array is:
a 1 5 @ f % z u
All letters in your array:
a f z u
```

### 1.1.3 Функции

1.1.3.1 Напишите функцию `double GetPower(double base, int exponent)`, которая возводит значение `base` в степень `exponent`. Рассчитанное значение передается в качестве возвращаемого значения.

1.1.3.2 Вызовите функцию `GetPower()`, написанную в предыдущей задаче, несколько раз в функции `main()` для разных значений. В функции `main` напишите код, который выводит значение степени на экран. Пример вывода:

```
2.0 ^ 5 = 32
3.0 ^ 4 = 81
-2.0 ^ 5 = -32
```

1.1.3.3 Напишите функцию `void DemoGetPower(double base, int exponent)`, которая вызывает функцию `GetPower()` и выводит её результат на экран в формате `base ^ exponent = result` (аналогично предыдущей задаче). Функция `main()` должна несколько раз вызывать функцию `DemoGetPower()`. Пример работы программы:

```
2.0 ^ 5 = 32
3.0 ^ 4 = 81
-2.0 ^ 5 = -32
```

Фактически, программа должна делать то же самое, что и в предыдущей задаче. Однако, работа с консолью и вызов функции `GetPower()` должны быть вынесены в отдельную функцию, что сократит реализацию функции `main()` и уменьшит дублирование кода.

1.1.3.4 Напишите функцию `void RoundToTens(int& value)`, в которой переменная `value` передается в функцию по ссылке и округляет значение переменной до десятков (например, 14 округляет до 10, 191 до 190, 27 до 30). Для этого используйте операции деления без остатка / и взятие остатка %. Например, возьмите остаток деления `value` на 10, и, если остаток меньше 5, значит нужно присвоить в `value` новое значение, равное старому значению, разделенному без остатка на 10, и снова умноженному на 10. Таким

образом, число будет округлено в меньшую сторону до десятков. Как округлить число в большую сторону при остатке равному и большему 5, придумайте самостоятельно.

В функции `main()` создайте целочисленную переменную a и поместите в неё некоторое значение для округления. Выведите его на экран. Затем вызовите функцию `RoundTo-Tens()` с этой переменной, и выведите её новое значение на экран. Если функция реализована правильно, то значение должно измениться. Аналогично вызовите функцию `RoundToTens()` несколько раз. Пример:

```
For 14 rounded value is 10
For 191 rounded value is 190
For 27 rounded value is 30
```

### 1.1.4 Адреса и указатели

1.1.4.1 В функции main() напишите следующий код:

```
int main()
{
    int a = 5;
    int b = 4;
    cout << "Address of a: " << &a << endl;
    cout << "Address of b: " << &b << endl;

    double c = 13.5;
    cout << "Address of c: " << &c << endl;

    bool d = true;
    cout << "Address of d: " << &d << endl;
}
```

Программа выводит на экран адреса разных переменных. Для получения адреса используется операция взятия адреса `&`. Запустите программу и убедитесь:

- все переменные имеют разные адреса;
- адреса переменных разных типов (int, double, char) имеют одинаковый формат в виде восьми 16-ричных значений.
- при каждом запуске программы адреса переменных будут разные – операционная система каждый раз выделяет новый участок памяти для исполняемой программы.

Пример вывода программы:

```
Address of a: 010FFCE0
```

```
Address of b: 010FFCD4
Address of c: 010FFCC4
Address of d: 010FFCBB
```

1.1.4.2 В функции main() напишите следующий код:

```
int main()
{
   int a[10] = {1, 2, 7, -1, 5, 3, -1, 7, 1, 6};
   cout << "Size of int type: "<< sizeof(int) << endl;
   for (int i = 0; i < 10; i++)
   {
      cout << "Address of a["<< i << "]: " << &a[i] << endl;
   }

   cout << endl;
   cout << "Size of double type: " << sizeof(double) << endl;
   double b[10] = { 1.0, 2.0, 7.0, -1.0, 5.0, 3.5, -1.8, 7.2, 1.9, 6.2
};
   for (int i = 0; i < 10; i++)
   {
      cout << "Address of b[" << i << "]: " << &b[i] << endl;
   }
}
```

Программа последовательно выводит на экран адреса всех ячеек созданных массивов. Запустите программу и убедитесь:

- все ячейки массива имеют разные адреса;
- разница между адресами ячеек целочисленного массива равна 4 (ровно столько байт требуется для хранения одного целого числа);
- разница между адресами ячеек вещественного массива равна 8 (ровно столько байт требуется для хранения одного вещественного числа).

Пример вывода программы:

```
Size of int type: 4
Address of a[0]: 006FFD20
Address of a[1]: 006FFD24
Address of a[2]: 006FFD28
Address of a[3]: 006FFD2C
Address of a[4]: 006FFD30
Address of a[5]: 006FFD34
Address of a[6]: 006FFD38
Address of a[7]: 006FFD3C
Address of a[8]: 006FFD40
```

```
Address of a[9]: 006FFD44

Size of double type: 8
Address of b[0]: 006FFCBC
Address of b[1]: 006FFCC4
Address of b[2]: 006FFCCC
Address of b[3]: 006FFCD4
Address of b[4]: 006FFCDC
Address of b[5]: 006FFCE4
Address of b[6]: 006FFCEC
Address of b[7]: 006FFCF4
Address of b[8]: 006FFCFC
Address of b[9]: 006FFD04
```

1.1.4.3 В функции main() напишите следующий код:

```
int main()
{
    int a = 5;
    int& b = a;

    cout << "Address of a: " << &a << endl;
    cout << "Address of b: " << &b << endl;

    cout << endl;
    b = 7;
    cout << "Value of a: " << a << endl;
}
```

Программа демонстрирует работу ссылок. Ссылка – это переменная, которая хранит адрес другой переменной, и этот адрес нельзя изменить после инициализации ссылки. Объявление ссылки задается в формате:

```
type& reference = name;
```

, где type – это тип данных, на который будет ссылаться ссылка, reference – имя переменной-ссылки, name – имя переменной, на которую будет ссылаться ссылка. После инициализации ссылка будет иметь тот же адрес, что и исходная переменная. Поэтому ссылки также называют синонимами переменных.

Обратите внимание на использование `&`: когда знак амперсанда стоит после типа данных – это объявление ссылки. Когда же знак амперсанда стоит перед именем уже созданной переменной – это операция взятия адреса.

Запустите программу и убедитесь:

- адрес ссылки `b` совпадает с адресом переменой `a`;
- после присвоения значения в переменную b, также меняется значение переменной a. Это объясняется тем, что обе переменные связаны с одним и тем же участком оперативной памяти и меняют одно и то же значение.

Пример вывода программы:

```
Address of a: 00AFFDFC
Address of b: 00AFFDFC

Value of a: 7
```

1.1.4.4 В функции main() напишите следующий код:

```
void Foo(double a)
{
    cout << "Address of a in Foo(): " << &a << endl;
    cout << "Value of a in Foo(): " << a << endl;

    a = 15.0;
    cout << "New value of a in Foo(): " << a << endl;
}

int main()
{
    double a = 5.0;
    cout << "Address of a in main(): " << &a << endl;
    cout << "Value of a in main(): " << a << endl;
    cout << endl;

    Foo(a);

    cout << endl;
    cout << "Value of a in main(): " << a << endl;
}
```

Программа демонстрирует передачу переменных в функцию по значению. Пример показывает, что при передаче переменных в функцию, значение исходной переменной копируется во внутреннюю переменную функции. Так как переменные имеют разные адреса, то изменение значения переменной внутри функции никак не влияет на изменение значения переменной вне функции. Таким образом, передача переменных в функцию по значению является безопасной для исходных переменных.

Запустите программу и убедитесь:

- адрес переменной `a` внутри функции `Foo()` отличается от адреса переменной `a` в функции `main()`;
- значение переменной `a` внутри функции `Foo()` изначально равняется исходной переменной – значение передано в функцию правильно;
- изменение переменной `a` внутри функции `Foo()` никак не влияет на значение переменной `a` в `main()`.

Пример вывода программы:

```
Address of a in main(): 00D3F814
Value of a in main(): 5

Address of a in Foo(): 00D3F73C
Value of a in Foo(): 5
New value of a in Foo(): 15

Value of a in main(): 5
```

1.1.4.5 В функции main() напишите следующий код:

```
void Foo(double& a)
{
    cout << "Address of a in Foo(): " << &a << endl;
    cout << "Value of a in Foo(): " << a << endl;

    a = 15.0;
    cout << "New value of a in Foo(): " << a << endl;
}

int main()
{
    double a = 5.0;
    cout << "Address of a in main(): " << &a << endl;
    cout << "Value of a in main(): " << a << endl;
    cout << endl;

    Foo(a);

    cout << endl;
    cout << "Value of a in main(): " << a << endl;
}
```

Программа демонстрирует передачу переменных в функцию по ссылке. Пример показывает, что при передаче ссылок в функцию, внутренняя переменная имеет тот же адрес,

что и исходная переменная. Следовательно, изменение значения переменной внутри функции приведет к изменению значения исходной переменной. Таким образом, передача переменных в функцию по ссылке можно использовать в тех случаях, когда нужно изменить значение исходной переменной.

Запустите программу и убедитесь:

- адрес переменной `a` внутри функции `Foo()` совпадает с адресом переменной `a` в функции `main()`;
- значение переменной `a` внутри функции `Foo()` изначально равняется исходной переменной – значение передано в функцию правильно;
- изменение переменной `a` внутри функции `Foo()` изменяет значение исходной переменной `a` в `main()`.

Пример вывода программы:

```
Address of a in main(): 00D3F814
Value of a in main(): 5

Address of a in Foo(): 00D3F814
Value of a in Foo(): 5
New value of a in Foo(): 15

Value of a in main(): 15
```

1.1.4.6 В функции main() напишите следующий код:

```
int main()
{
    int a = 5;
    int* pointer = &a;

    cout << "Address of a: " << &a << endl;
    cout << "Address in pointer: " << pointer << endl;
    cout << "Address of pointer: " << &pointer << endl;

    cout << endl;
    *pointer = 7;
    cout << "Value in a: " << a << endl;
    cout << "Value by pointer address: " << *pointer << endl;
}
```

Программа показывает специальный тип данных - указатели. В предыдущих примерах было показано, что адреса любых типов данных имеют одинаковый формат. Адреса в

оперативной памяти имеют собственный диапазон допустимых значений, с адресами можно совершать различные операции – фактически, адреса являются самостоятельным типом данных. Как адреса могут принимать значения, логично, что эти значения можно хранить в переменных. Для работы с адресами предназначен специальный тип данных - **указатели**.

Запустите программу и убедитесь:

- адрес в указателе `pointer` совпадает с адресом переменной `a`;
- сам указатель, как и любая другая переменная, имеет собственный адрес, отличный от переменной `a`;
- операция разыменования `*`, применяемая к указателю позволяет изменить значение исходной переменой a, или получить значение переменной, например, для вывода на экран.

Пример вывода программы:

```
Address of a: 00B9FCF4
Address in pointer: 00B9FCF4
Address of pointer: 00B9FCE8

Value in a: 7
Value by pointer address: 7
```

1.1.4.7 Ответьте на вопрос: как в исходном коде отличить операцию разыменования, объявление указателя и операцию умножения? Все три операции используют символ *, поэтому важно уметь отличать их в исходном коде.

1.1.4.8 В функции main() напишите следующий код:

```
void Foo(double* a)
{
    cout << "Address in pointer: " << a << endl;
    cout << "Address of pointer: " << &a << endl;
    cout << "Value in pointer address: " << *a << endl;

    *a = 15.0;
    cout << "New value in pointer address: " << *a << endl;
}

int main()
{
    double value = 5.0;
    double* pointer = &value;
    cout << "Address of value in main(): " << &value << endl;
    cout << "Address in pointer in main(): " << pointer << endl;
```

```
    cout << "Address of pointer in main(): " << &pointer << endl;
    cout << "Value of a in main(): " << value << endl;
    cout << endl;

    Foo(pointer);

    cout << endl;
    cout << "Value of a in main(): " << value << endl;
}
```

Программа демонстрирует передачу переменных в функцию по указателю. При передаче переменной по указателю, значение адреса из исходного указателя копируется в указатель функции `a`. Таким образом, оба указателя хранят адрес одной и той же переменной `value`. Следовательно, если внутри функции с помощью разыменования поменять значение через указатель, то значение исходной переменной `value` будет изменено после завершения функции. Передача переменной по указателю – это еще один способ передачи аргументов в функцию, наравне с передачей по ссылке, позволяющий изменить значение исходной переменной.

Пример вывода программы:

```
Address of value in main(): 004FFA64
Address in pointer in main(): 004FFA64
Address of pointer in main(): 004FFA58
Value of a in main(): 5

Address in pointer: 004FFA64
Address of pointer: 004FF984
Value in pointer address: 5
New value in pointer address: 15

Value of a in main(): 15
```

Также обратите внимание, что адреса самих указателей внутри функции и вне её отличаются, хотя хранят один адрес. Это значит, что, если внутри функции вы присвоите в указатель `a` адрес другой переменной, в исходном указателе `pointer` адрес на `value` останется без изменений. Возможность изменения адреса, хранимого в указателе, отличает передачу аргументов в функцию по указателю от передачи по ссылке, где изменить адрес в ссылке невозможно.

Запустите программу и убедитесь:

- адрес в указателе `a` внутри функции `Foo()` совпадает с адресом переменной `value` в функции `main()`;

- значение переменной при разыменовании указателя `a` внутри функции `Foo()` изначально равняется исходной переменной `value` – значение передано в функцию правильно;
- изменение значения переменной при разыменовании указателя `a` внутри функции `Foo()` изменяет значение исходной переменной `value` в `main()`;
- адреса самих указателей `a` и `pointer` отличаются друг от друга.

### 1.1.5 Динамическая память

1.1.5.1 Выделите память под массив вещественных чисел и проинициализируйте его значениями из кода программы. Выведите значения массива на экран. После освободите память.

Пример работы программы:

```
Array of double:
1.0 15.0 -8.2 -3.5 12.6 38.4 -0.5 4.5
```

1.1.5.2 Выделите память под массив булевых чисел и проинициализируйте его значениями из кода программы. Выведите значения массива на экран. После освободите память.

Пример работы программы:

```
Array of bool:
true false true true false true false false
```

1.1.5.3 Запросите пользователя ввести положительное целое число n. Выделите память под массив символов (количество элементов массива равно n) и проинициализируйте его значениями с клавиатуры. Выведите значения массива на экран. После освободите память.

Пример работы программы:

```
Enter char array size: 10
Enter a[0]: a
Enter a[1]: 5
Enter a[2]: m
Enter a[3]: i
Enter a[4]: %
Enter a[5]: !
Enter a[6]: s
```

```
Enter a[7]: p
Enter a[8]: *
Enter a[9]: 9

Your char array is:
a 5 m I % ! s p * 9
```

1.1.5.4 Выделите память под массив из 10 вещественных чисел. Из кода программы задайте значения массиву и выведите его на экран. Передайте массив в ранее написанную функцию по сортировке массива. Выведите отсортированный массив на экран. После освободите память.

Пример работы кода:

```
Array of double:
1.0 15.0 -8.2 -3.5 12.6 38.4 -0.5 4.5 16.7 4.5

Sorted array of double:
-8.2 -3.5 -0.5 1.0 4.5 4.5 12.6 15.0 16.7 38.4
```

1.1.5.5 Выделите память под массив из 10 целых чисел. Из кода программы задайте значения массиву. Передайте массив в ранее написанную функцию по поиску индекса элемента массива. Выведите на экран исходный массив и индекс найденного элемента.

Пример работы кода:

```
Int array:
1 15 -8 -3 12 38 0 4 16 4

Enter searching value: 12

Index of searching value 12 is: 4
```

1.1.5.6 Выделите память под массив из 15 символов. Из кода программы задайте значения массиву. Передайте массив в ранее написанную функцию, подсчитывающую количество букв (`'a'`- `'z'`) среди символов массива. Выведите на экран исходный массив и количество подсчитанных символов. После освободите память.

Пример работы кода:

```
Char array is:
```

```
a 5 m i % ! s p * 9 f ^ ; q k

Letters in array:
a m i s p f q k
```

1.1.5.7 Напишите функцию `int* MakeRandomArray(int arraySize)`, которая выделяет память под массив целых чисел заданного размера `arraySize`. После выделения памяти функция инициализирует массив случайными значениями от 0 до 100 и возвращает указатель на массив. В функции `main()` вызовите функцию 3 раза для создания массивов размером 5, 8, 13. Выведите созданные массивы на экран. Освободите память от созданных массивов.

Пример работы программы:

```
Random array of 5:
25 66 82 54 19

Random array of 8:
8 90 55 42 47 12 63 27

Random array of 13:
66 85 100 27 53 78 78 43 22 34 95 11 61
```

1.1.5.8 Скажите, есть ли утечка памяти в следующем примере кода. Если да, то как нужно исправить пример для устранения утечки? Перепишите данный пример в проект в среде разработки и, при необходимости, исправьте его. Убедитесь, что код работает верно.

```
int* ReadArray(int count)
{
   int* values = new int[count];
   for (int i = 0; i < count; i++)
   {
      cin >> values[i];
   }
   return values;
}

int CountPositiveValues(int* values, int count)
{
   int result = 0;
   for (int i = 0; i < count; i++)
```

```
   {
      if (values[i] > 0)
      {
         result++;
      }
   }
   return result;
}

int main()
{
   int count = 15;
   int* values = ReadArray(count);
   cout << "Count is: " << CountPositiveValues(values, count) << endl;

    count = 20;
   values = ReadArray(count);
   cout << "Count is: " << CountPositiveValues(values, count) << endl;

   delete[] values;
}
```

### 1.1.6 Теоретические вопросы

1) Как установить точку останова в Visual Studio? Как сделать точку останова с условием?
2) Чем отличаются конфигурации сборки «Отладка» (Debug) и «Выпуск» (Release)?
3) Что такое массив? Как расположены элементы массива в оперативной памяти?
4) Как объявить массив в Си++? Как проинициализировать массив?
5) Как объявить функцию в Си++?
6) Чем отличаются передача аргументов в функцию по значению и по ссылке?
7) Чем отличаются передача аргументов в функцию по указателю и по ссылке?
8) Что представляет из себя адрес в оперативной памяти? Будет ли формат адреса отличаться для 32-битных и 64-битных архитектур?
9) Определите по формату адреса переменных, какое количество ячеек памяти может быть проиндексировано, а также к какому количеству оперативной памяти можно обратиться из программы?
10) Как правильно передавать массив в функцию?
11) Можно ли передать в функцию указатель по ссылке?
12) Что такое куча (динамическая память)?
13) Что такое сегмент кода и сегмент данных?
14) Какие ошибки могут возникнуть при работе с динамической памятью?