

Основы программирования

Лабораторная работа 6

Тема: Работа с указателями

Цель лабораторной работы

Целями данной работы являются получение навыков использования указателей.

Краткие теоретические сведения

Указатели представляют собой объекты, значением которых служат адреса других объектов (переменных, констант, указателей и т.д.) или функций.

То есть, если по-простому, то **указатель** (*pointer*) – это переменная или константа, хранящая адрес другого объекта, обычно адрес другой переменной.

Переменная или константа, содержащая адрес ячейки памяти, должна быть объявлена как **указатель**. В общем виде объявление указателя записывается так:

```
тип_указателя *имя_указателя;
```

Тип указателя определяется **типом переменной**, на которую он будет ссылаться. Это может быть любой допустимый тип. Пример:

```
double *pointer1;
```

Операторы для работы с указателями

Оператор получения адреса

Оператор & (*амперсанта*) – унарный оператор, возвращающий **адрес** своего операнда. С помощью этого оператора можно получить адрес объекта и поместить его в указатель. Оператор **&** **нельзя** применить к литералам.

Например:

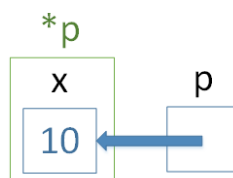
```
int x = 10; // целочисленная переменная
int *p;     // указатель на целочисленную переменную
p = &x;     // записываем адрес x в указатель p
```



Оператор получения значения по адресу

Оператор * (*оператор разыменования*) – это унарный оператор, возвращающий **значение** операнда, расположенного по указанному **адресу**. Например, если указатель *p* из предыдущего примера содержит адрес переменной *x*, то с помощью данного оператора можно получить значение переменной *x*:

```
int res = *p;
cout << res; // 10
```



Адресная арифметика

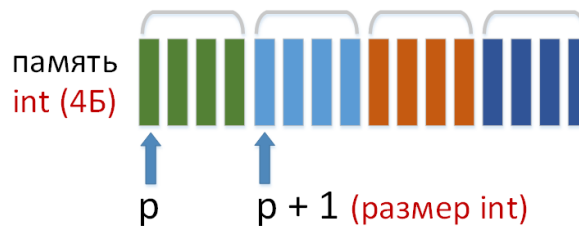
В языке C++ допустимы только некоторые **арифметические** операции над **указателями**:

Сложение и вычитание целого числа

Предположим, текущее значение указателя p типа `int` равно 2000. Также нам известно, что переменная типа `int` занимает в памяти 4 байта. Тогда после такой операции *сложения*:

$p = p + 1;$

указатель p принимает значение 2004, а не 2001. То есть, при увеличении на число 1 указатель p будет ссылаться на **следующее** целое число. При сложении происходит переход не на следующий байт в памяти, а на следующее значение переменной целого типа. Так как `int` занимает 4 байта, то и *сложение с 1* добавляет 4 байта к адресу.



При **сложении с целым числом** адрес указателя смещается на *указанное число* байт, умноженное на *размер типа данных* указателя.

Это же справедливо и для **операции вычитания**.

```
int n = 10;
int* ptr = &n;
// исходный адрес
cout << "address = " << ptr << " value = " << *ptr << endl;
ptr++;
// следующая переменная
cout << "address = " << ptr << " value = " << *ptr << endl;
ptr--;
// предыдущая (исходный адрес)
cout << "address = " << ptr << " value = " << *ptr << endl;
```

```
Консоль отладки Microsoft Visual Studio
address = 0073F9F8 value = 10
address = 0073F9FC value = -858993460
address = 0073F9F8 value = 10
```

Вычитание одного указателя из другого

Кроме *сложения* и *вычитания* указателя и целого, разрешена еще только одна операция адресной арифметики: можно **вычитать указатель из указателя**. В этом случае указатели должны быть одного типа.

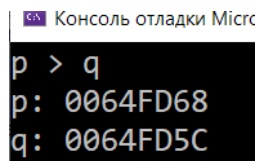
Благодаря этой операции можно определить количество объектов (типа данных указателя), расположенных между адресами указателей. То есть данная операция возвращает **целое число**.

Все остальные арифметические операции **запрещены** над указателями. Запрещено также и сложение двух указателей, так как из суммы адресов в памяти не получить никакую полезную информацию. Такая операция *не имеет смысла*.

Сравнение указателей

Для использования **операций сравнения** на указателях оба операнда должны быть указателями **одного типа** или значениями **нулевого указателя** типа *NULL* и *nullptr*:

```
int n = 1, m = 2;
int* p = &n;
int* q = &m;
if (p < q)
    cout << "p < q" << endl;
else
    cout << "p > q" << endl;
cout << "p: " << p << endl;
cout << "q: " << q << endl;
```



```
Консоль отладки Micr
p > q
p: 0064FD68
q: 0064FD5C
```

Однако сравнение указателей может оказаться полезным, только тогда, когда два указателя ссылаются на общий объект, например, на *массив*.

Указатели и массивы

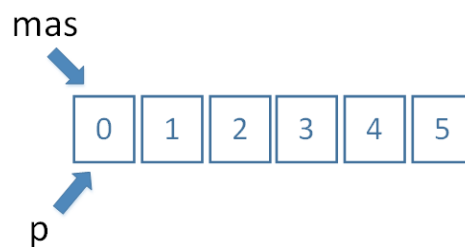
Указатели и **массивы** тесно связаны между собой. *Имя* массива, по сути, является указателем на *первый* элемент массива, только изменять этот указатель нельзя. В этом случае можно сказать, что массив очень похож на *константный указатель*.

Допустим, у нас есть **массив** и **указатель**:

```
int mas[6], *p;
```

Мы хотим, чтобы указатель ссылался на первый элемент массива. Это можно сделать:

```
p = &mas[0]; // через адрес первого элемента
p = mas;    // через имя массива
```



Для обращения к другим элементам массива, помимо индексов, также можно использовать **адресную арифметику**.

К примеру, зная адрес первого элемента можно обратиться к *третьему* и *шестому* элементам, **прибавив** соответствующее число:

```
*p      ~ mas[0]
*(p+2)  ~ mas[2]
*(p+5)  ~ mas[5]
```

Вычитание одного указателя из другого также имеет смысл при работе с массивами.

Допустим, у нас имеется два указателя, которые перемещаются по массиву:

```
int mas[6];
int *p1 = &mas[2], *p2 = &mas[5];
```

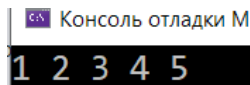
В любой момент мы можем определить, на сколько элементов указатели смещены относительно друг друга:

```
int res = p2 - p1;
```

Массивы в параметрах функции

Для принятия массива в функцию можно использовать **параметр-указатель**. Пример:

```
void show(int *m, size_t size)
{
    for(int i = 0; i < size; i++)
        cout << m[i] << " ";
    cout << endl;
}
int main() {
    int nums[] = {1, 2, 3, 4, 5};
    show(nums, 5);
    return 0;
}
```



Консоль отладки М
1 2 3 4 5

Определение длины массива аргумента функции через указатели

До этого мы уже рассматривали способы передачи длины массива через отдельное числовое значение, обозначающее количество элементов в массиве. Рассмотрим еще один способ на основе **указателей**.

Чтобы должным образом определять конец массива, перебирать элементы массива, необходимо использовать специальный маркер, который бы сигнализировал об окончании массива. Таким маркером может выступить указатель на **конец массива**.

Можно вручную вычислить адрес конца массива, а можно использовать встроенные библиотечные функции **std::begin()** и **std::end()**:

```
int nums[] = { 1, 2, 3, 4, 5 };
int *begin = std::begin(nums); // указатель на начало массива
int *end = std::end(nums);     // указатель на конец массива
```

Причем функция *end()* возвращает указатель не на последний элемент, а адрес, следующий за последним элементом в массиве.

Для перебирания элементов массива в этом случае удобно использовать указатель. Изначально этот указатель будет равен началу массива, а далее с помощью цикла мы будем смещать его на один элемент дальше по массиву. Когда этот указатель сравняется с указателем конца массива, это значит, что весь массив пройден, на этом цикл завершится.

```
void show(int* begin, int* end) {
    for (int* p = begin; p != end; p++) {
        cout << *p << " ";
    }
}
int main() {
    int nums[] = { 10, 12, 13, 14, 15 };
```

```

int* begin = std::begin(nums);
int* end = std::end(nums);
show(begin, end);
return 0;
}

```

Консоль отладки Microsoft
10 12 13 14 15

Ссылки

Ссылки – особый тип данных, являющийся скрытой формой указателя, который при использовании автоматически разыменовывается. Размер ссылки в памяти полностью совпадает с указателем. Ссылка считается псевдонимом (вторым именем) объекта, на который ссылается.

Общая форма:

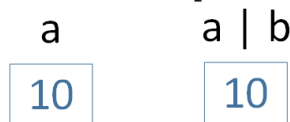
```
тип_ссылки &имя_ссылки = имя_объекта;
```

При объявлении ссылка должна быть проинициализирована именем объекта, на который она будет ссылаться. Тип данных, на который указывает ссылка, может быть любым, но должен совпадать с объектом, на который ссылается. Пример:

```

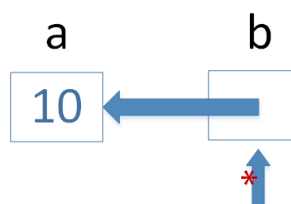
int a = 10;
int &b = a; //теперь b – псевдоним (второе имя) переменной a

```



Объявив ссылку, мы создаем объект, который, как указатель, ссылается на какое-то значение, но, в отличие от указателя, постоянно привязан к этому значению. И его нельзя перепривязать к другому значению. В этом плане ссылка очень похожа на *константный указатель*. Только в отличие от указателя из ссылки нельзя получить адрес того объекта, на который она ссылается, так как при обращении к ссылке она *автоматически разыменовывается*:

```
cout << b << endl; // 10
```



Основное назначение **указателя** – это организация динамических объектов, то есть размер, которых может меняться, тогда, как **ссылки** предназначены для организации прямого доступа к тому, или иному объекту.

Задание

Разработать программу для каждой задачи из **общего** и **индивидуального** задания.

Для решения задач необходимо использовать **указатели**:

- Если требуется передать массив в функцию, то передавайте его через **параметр-указатель**.
- Для работы с массивами вместо индексов используйте **АДРЕСНУЮ АРИФМЕТИКУ**.
- В **индивидуальном** задании вместо передачи числа-размера массива используйте указатель на **конец массива** (можно использовать стандартные функции `std::begin()` и `std::end()`).

Все пользовательские функции должны **только выполнять вычисление и возвращать результат** в главную функцию (если в задаче на сказано иного). Все результаты работы программ необходимо выводить на экран в функции `main()`.

Методические указания

Выполните **общие** и **индивидуальные** задания.

При выполнении **общих** заданий не требуется отчет, но все еще нужна защита с объяснением кода программы.

При выполнении **индивидуальных** заданий необходимо выполнить следующие этапы **для каждой задачи отдельно**:

- 1) словесная постановка задачи;
- 2) анализ задачи и формальная постановка задачи;
- 3) проектирование (разработка алгоритма) – лучше всего в графической форме (блок-схема), но можно и в словесной форме или псевдокод;
- 4) реализация (кодирование, отладка);
- 5) тестирование.

Результаты выполнения индивидуальных заданий оформить в виде отчета.

Общие задания

- 1) Написать функцию, которая получает **указатель** на массив, его размер и аргумент через **параметр-указатель**. Вычисляет **сумму** и **произведение** его элементов.

Сумма возвращается самой функцией, а **произведение** – с помощью *параметра-указателя*.

- 2) Напишите функцию, которая массив и два дополнительных аргумента. Первый аргумент – *число*, второй – **ссылка на число**.

Нужно найти количество элементов массива **меньших**, чем первый аргумент, и сохранить их в **аргументе-ссылке**. Также нужно найти количество элементов массива **больших**, чем первый аргумент, и вернуть в качестве результата функции.

Варианты индивидуальных заданий

Вариант 1

Написать *функцию-процедуру*, которая принимает в качестве аргументов вещественный массив и переменную. Заменяет все элементы массива на элементы, возведенные в "-1" степень. Считает количество элементов, которые стали больше по значению после возведения в "-1" степень. Сохраняет это количество в переменную-аргумент.

Вариант 2

Написать функцию, которая принимает в качестве аргументов целочисленный массив и переменную. Она находит минимальный элемент массива и сохраняет его в переменную-аргумент. Возвращает функция максимальный элемент массива.

Вариант 3

Написать функцию, которая принимает в качестве аргументов вещественный массив и переменную. Она находит среднее значение элементов массива и сохраняет его в переменную-аргумент. Возвращает количество элементов, меньших, чем среднее значение.

Вариант 4

Написать *функцию-процедуру*, которая принимает в качестве аргументов целочисленный массив и переменную. Она находит сумму элементов массива, которые больше **среднего значения** по массиву, и сохраняет его в переменную-аргумент.

Вариант 5

Написать функцию, которая принимает в качестве аргументов целочисленный массив и переменную. Находит сумму четных и сумму нечетных элементов. В переменную-аргумент сохраняет максимальную из сумм. Возвращает **минимальную** из этих сумм.

Вариант 6

Написать *функцию-процедуру*, которая принимает в качестве аргументов целочисленный массив и переменную. Она находит минимальный элемент и считает количество элементов массива меньших, чем квадрат найденного элемента. Сохраняет это количество в переменную-аргумент.

Вариант 7

Написать *функцию-процедуру*, которая принимает в качестве аргументов вещественный массив и переменную. Находит номер элемента, который ближе всех по значению к среднему арифметическому элементов массива и сохраняет его в переменной-аргументе.

Вариант 8

Написать функцию, которая принимает в качестве аргументов целочисленный массив и две переменные. Находит два элемента, разница между которыми минимальная. Сохраняет номера этих элементов в переменных-аргументах. Возвращает эту разницу.

Вариант 9

Написать *функцию-процедуру*, которая принимает в качестве аргументов целочисленный массив и переменную. Она находит сумму элементов массива, которые меньше среднего значения по массиву, и сохраняет его в переменную-аргумент.

Вариант 10

Написать *функцию-процедуру*, которая принимает в качестве аргументов целочисленный массив и три переменные. Находит максимальный и минимальный элементы и сохраняет их значения в двух переменных-аргументах. В третьей переменной сохраняет разницу между этими элементами.

Вариант 11

Написать функцию, которая принимает в качестве аргументов вещественный массив и переменную. Находит сумму всех элементов и сохраняет ее в переменную-аргумент. Возвращает произведение всех элементов, не равных 0.

Вариант 12

Написать *функцию-процедуру*, которая принимает в качестве аргументов целочисленный массив и две переменные. Находит номера первого и последнего элемента, равного 0. Сохраняет эти номера в переменные-аргументы.

Вариант 13

Написать *функцию-процедуру*, которая принимает в качестве аргументов вещественный массив и переменную. Она заменяет каждое значение элемента массива его квадратом и сохраняет в переменную-аргумент сумму измененных элементов массива.