top
КОМПЬЮТЕРНАЯ АКАДЕМИЯ

ОСНОВЫ ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ C++

Урок № 12

Указатели

Содержание

1. Статическое и динамическое выделение памяти

Статическая память — это область хранения всех глобальных и статических переменных. Переменные статической памяти объявляются лишь единожды и уничтожаются по завершении программы.

Динамическая память или память свободного хранения отличается от статической тем, что программа должна явным образом запросить память для элементов, хранимых в этой области, а затем освободить память, если она больше не нужна. При работе с динамической памятью программа может позволить себе, например, узнать количество элементов массива на этапе выполнения.

2. Указатели

Указатель — это переменная, содержащая адрес другой переменной. Указатели очень широко используются в языке C++. С помощью указателей вы можете обращаться к ячейкам оперативной памяти. Для этого вам нужно знать адрес ячейки.

Так как указатель содержит адрес объекта, это дает возможность «косвенного» доступа к этому объекту через указатель. Предположим, что x — переменная, например, типа int, а px — указатель, созданный неким еще не рассмотренным способом. Унарная операция & выдает адрес объекта, так что оператор:

```
px = &x;
```

присваивает адрес x переменной px; говорят, что px «указывает» на x. Операция & применима только к переменным и элементам массива, конструкции вида:

```
&(x-1) и &3
```

являются незаконными.

Унарная операция * рассматривает свой операнд, как адрес конечной цели, и обращается по этому адресу, чтобы извлечь содержимое. Следовательно, если y тоже имеет тип int, то:

```
y = *px;
```

присваивает y содержимое того, на что указывает px.

Так последовательность:

```
px = &x;
y = *px;
```

присваивает y то же самое значение, что и оператор:

```
y = x;
```

Переменные, участвующие в коде выше необходимо объявить:

```
int x, y;
int *px;
```

С описанием для x и y мы уже неоднократно встречались. Описание указателя:

```
int* px;
```

является новым для нас и говорит, что мы объявили указатель на ячейку памяти целого типа. Объявление указателя отличается от объявления переменной наличием * в объявлении. После объявления указатель ни на что не указывает. В нём случайное значение. Указатель может хранить только адрес ячейки. Напомним, что для присвоения адреса в указатель используется & (получение адреса), а для доступа к ячейке памяти необходимо применять * (разыменование)

Указатели могут входить в выражения. Например, если px указывает на целое x, то *px может появляться в любом контексте, где может встретиться x. Например:

```
y = *px + 1; // присваивает y значение,
             // на 1 большее значения x;
```

```
cout<< *px; // выводит текущее значение x;
d = sqrt((double) *px) // получает в d квадратный
                       // корень из x,

/* причем до передачи функции sqrt значение x
   преобразуется типу double
*/
```

В выражениях вида:

```
y = *px + 1;
```

Унарные операции * и & связаны со своим операндом более крепко, чем арифметические операции, так что такое выражение берет то значение, на которое указывает px, прибавляет 1 и присваивает результат переменной y. Мы вскоре вернемся к тому, что может означать выражение:

```
y = *(px + 1);
```

Вы можете использовать разыменование и в левой части присваиваний. Если px указывает на x, то:

```
 *px = 0;
```

полагает x равным нулю (так как использование * записывает значение в ту ячейку, на которую указывает px), а:

```
*px += 1;
```

увеличивает его на единицу, так как это преобразуется в

```
*px = *px + 1;
```

И наконец, так как указатели являются переменными, то с ними можно обращаться, как и с остальными

переменными. Если py — другой указатель на переменную типа int, то:

```
py = px;
```

копирует содержимое px в py, в результате чего py указывает на то же, что и px.

Рассмотрим изученные механизмы на примере:

```
#include <iostream>
using namespace std;

int main()
{
    int x = 10;
    int y = 5;

    int* px;
    int* py;


    //Записываем адрес переменной x в px
    px = &x;

    // Отображаем адрес x через указатель
    // и операцию взятия адреса
    cout << px << " " << &x << endl;

    // Отображаем значение x через переменную
    // и операцию разыменования
    // на экране 10 10
    cout << *px << " " << x << endl;

    // Изменяем значение переменной x
    // используем операцию разыменования
    *px = 99;
```

```
    /* Отображаем новое значение x через переменную
       и операцию разыменования
       на экране отобразится 99 99
     */
    cout << *px << " " << x << endl<<endl;

    //Записываем адрес переменной y в py
    py = &y;

    // Отображаем адрес y через указатель
    // и операцию взятия адреса
    cout << py << " " << &y << endl;

    // Отображаем значение y через переменную
    // и операцию разыменования
    // на экране 5 5
    cout << *py << " " << y << endl;

    // записываем значение адреса из px в py
    // теперь оба указатели указывают на x
    py = px;

    // Отображаем значение x через переменную
    // и два указателя
    // на экране 99 99 99
    cout << *px << " " <<*py<<" "<<x << endl << endl;

    return 0;
}
```

В этом примере мы использовали различные возможности по работе с указателями. Поэкспериментируйте с кодом этого примера, чтобы лучше понять механику указателей.

3. Указатели и массивы

В языке C++ существует сильная взаимосвязь между указателями и массивами, настолько сильная, что указатели и массивы действительно следует рассматривать одновременно. Любую операцию, которую можно выполнить с помощью индексов массива, можно произвести и с помощью указателей. Покажем, как это можно выполнить. Описание:

```
int a[10];
```

определяет массив размера 10, т.е. набор из 10 последовательных объектов, называемых a[0], a[1], ..., a[9]. Запись a[i] соответствует элементу массива через i позиций от начала. Если pa — указатель на целое, описанный как:

```
int *pa;
```

то присваивание:

```
pa = &a[0];
```

приводит к тому, что pa указывает на нулевой элемент массива a. Это означает, что pa содержит адрес элемента a[0].

Теперь присваивание:

```
x = *pa;
```

будет копировать содержимое a[0] в x (переменная целого типа).

Если pa указывает на некоторый определенный элемент массива a, то выражение pa+1 указывает на следующий элемент. Выражение pa-1 указывает на предыдущий

элемент. Выражение вида pa-i указывает на элемент, стоящий на i позиций до элемента, чей адрес находится в pa. Выражение pa+i на элемент, стоящий на i позиций после. Таким образом, если pa указывает на a[0], то:

```
*(pa+1)
```

ссылается на содержимое a[1], pa+i — адрес a[i], а *(pa+i) — содержимое a[i].

Эти замечания справедливы независимо от типа переменных в массиве a. Суть определения «добавления 1 к указателю», а также его распространения на всю арифметику указателей, состоит в том, что приращение масштабируется размером памяти, занимаемой объектом, на который указывает указатель. Таким образом, i в pa+i перед прибавлением умножается на размер объектов, на которые указывает pa.

Очевидно существует очень тесное соответствие между индексацией и арифметикой указателей. В действительности компилятор преобразует ссылку на массив в указатель на начало массива. В результате этого имя массива является указательным выражением. Отсюда вытекает несколько весьма полезных следствий. Так как имя массива является синонимом местоположения его нулевого элемента (фактически имя массива это указатель на его нулевой элемент), то присваивание:

```
pa = &a[0]
```

можно записать как pa = a.

Еще более удивительным, по крайней мере на первый взгляд, кажется тот факт, что ссылку на a[i] можно

записать в виде *(a+i). При анализировании выражения a[i] в языке C++ оно немедленно преобразуется к виду *(a+i); эти две формы совершенно эквивалентны. Если применить операцию & к обеим частям такого соотношения эквивалентности, то мы получим, что &a[i] и a+i тоже идентичны: a+i — адрес i-го элемента от начала a. С другой стороны, если pa является указателем, то в выражениях его можно использовать с индексом: pa[i] идентично *(pa+i). Подводим итог: любое выражение, включающее массивы и индексы, может быть записано через указатели и смещения и наоборот, причем даже в одном и том же утверждении.

Имеется одно различие между именем массива и указателем, которое необходимо иметь в виду. Указатель является переменной, так что операции pa=a и pa++ имеют смысл и компилируются. Однако, имя массива является константой, а не переменной: конструкции типа a=pa или a++, или p=&a будут незаконными.

Рассмотрим несколько примеров для закрепления материала:

```
#include <iostream>
using namespace std;

int main()
{
    const int size = 5;
    int arr[size] = { 33, 44, 7, 8, 9 };
/*  записываем адрес нулевого элемента массива
    в указатель
 */
    int* ptr = &arr[0];
```

```
    /*
        Показываем значение нулевого элемента массива
        через разыменование указателя.
        На экране 33
    */
    cout << *ptr << endl;

    /*
        Показываем значение первого элемента массива
        через разыменование указателя.
        Мы прибавляем смещение по адресу на один
        элемент после чего делаем разыменование.
        Адрес внутри указателя не меняется.
        На экране 44
    */
    cout << *(ptr + 1) << endl;

    /*
        Выполняем смещение на один элемент целого
        типа вперед и записываем новый адрес в
        указатель ptr.
        Фактически это операция выглядит так
        ptr = ptr + 1 * sizeof(int)
        Теперь в указателе содержится адрес первого
        элемента. Можно было также написать ptr++;
    */
    ptr = ptr + 1;

    /*
        Показываем значение первого элемента массива
        через разыменование указателя.
        На экране 44
    */
    cout << *ptr << endl;
    return 0;
}
```

В этом примере мы рассмотрели азы использования указателей для доступа к элементам массива: запись адреса массива в указатель, отображение значения, переход на следующий элемент массива.

Рассмотрим ещё один пример. В нём мы будем использовать указатель для отображения элементов массива и изменения их.

```
#include <iostream>
using namespace std;

int main()
{
    const int size = 5;
    int arr[size] = { 33, 44, 7, 8, 9 };

    // записываем адрес нулевого элемента массива
    // в указатель
    int* ptr = arr;

    /* отображаем весь массив через указатель
       на экране 33 44 7 8 9
    */
    for (int i = 0; i < size; i++) {
        cout << *(ptr + i)<<" ";
    }

    // изменяем значение первого элемента
    *(ptr + 1) = 55;

    // изменяем значение второго элемента
    *(ptr + 2) = 12;
    cout << endl << endl;

    /* отображаем весь массив через указатель
       на экране 33 55 12 8 9
    */
```

```
    for (int i = 0; i < size; i++) {
        cout << *(ptr + i) << " ";
    }

    return 0;
}
```

А теперь модифицируем этот пример. И применим синтаксис указателей к массиву. Напомним, имя массива — это адрес нулевого элемента. Однако, в отличии от значения адреса в указателе, его менять нельзя. Это константный указатель. Об этом мы поговорим позднее. Пример обращение к имени массива с помощью синтаксиса указателей:

```
#include <iostream>
using namespace std;

int main()
{
    const int size = 5;
    int arr[size] = { 33, 44, 7, 8, 9 };

    /* отображаем весь массив на экране
       33 44 7 8 9
    */
    for (int i = 0; i < size; i++) {
        cout << *(arr + i) << " ";
    }

    // изменяем значение первого элемента
    *(arr + 1) = 55;

    // изменяем значение второго элемента
    *(arr + 2) = 12;
```

```
    cout << endl << endl;
    /* отображаем весь массив на экране
       33 55 12 8 9
    */
    for (int i = 0; i < size; i++) {
        cout << *(arr + i) << " ";
    }

    return 0;
}
```

А можем ли мы использовать синтаксис массивов к указателю. Конечно! Рассмотрим на примере:

```
#include <iostream>
using namespace std;

int main()
{
    const int size = 5;
    int arr[size] = { 33, 44, 7, 8, 9 };
    int* ptr = arr;

    /* отображаем весь массив через указатель
       на экране
       33 44 7 8 9
    */
    for (int i = 0; i < size; i++) {
        cout << ptr[i] << " ";
    }

    // изменяем значение первого элемента
    ptr[1] = 55;

    // изменяем значение второго элемента
    ptr[2] = 12;
```

```
    cout << endl << endl;
    /* отображаем весь массив через указатель
       на экране
       33 55 12 8 9
    */
    for (int i = 0; i < size; i++) {
        cout << ptr[i] << " ";
    }

    return 0;
}
```

Мы уже знаем, что имя массива — это указатель (адрес) на нулевой элемент. Что же происходит, когда имя массива передается функции? Ответ очевиден: ей передается местоположение начала этого массива. Внутри вызванной функции такой аргумент является точно такой же переменной, как и любая другая, так что имя массива в качестве аргумента действительно является указателем, т.е. переменной, содержащей адрес.

Это значит, что мы можем применять арифметику указателей к имени массива и внутри функции. Например:

```
#include <iostream>
using namespace std;

void ShowArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        cout << *(arr + i) << " ";
    }
}
// подсчёт суммы элементов массива
int GetAmount(int* ptr, int size) {
    int sum = 0;
```

```
    for (int i = 0; i < size; i++) {
         sum += *(ptr + i);
    }
    return sum;
}

int main()
{
    const int size = 5;
    int arr[size] = { 33, 44, 7, 8, 9 };

    /* отображаем весь массив на экране
       33 44 7 8 9
    */
    ShowArray(arr, size);
    cout << endl << endl;

    // показываем сумму элементов массива
    // 101
    cout << "Amount of array elements: " <<
            GetAmount(arr, size)<<endl<<endl;

    /* отображаем весь массив на экране
       33 44 7 8 9
    */
    ShowArray(&arr[0], size);
    return 0;
}
```

Мы можем даже изменять указатель внутри функции. Например , функцию GetAmount можно было бы изменить так:

```
// подсчёт суммы элементов массива
int GetAmount(int* ptr, int size) {
    int sum = 0;
```

```
    for (int i = 0; i < size; i++,ptr++) {
         sum += *ptr;
    }
    return sum;
}
```

Операция увеличения ptr совершенно законна, поскольку эта переменная является указателем, ptr++ никак не влияет на оригинальный адрес массива, а только увеличивает локальную для функции GetAmount копию адреса.

Описания формальных параметров в определении функции в виде:

```
int m[];
```

и

```
int *m;
```

совершенно эквивалентны; какой вид описания следует предпочесть, определяется в значительной степени тем, какие выражения будут использованы при написании функции. Если функции передается имя массива, то в зависимости от того, что удобнее, можно полагать, что функция оперирует либо с массивом, либо с указателем, и действовать далее соответствующим образом. Можно даже использовать оба вида операций, если это кажется уместным и ясным.

4. Указатели — аргументы функций. Передача аргументов по указателю

Так как в C++ передача аргументов функциям осуществляется «по значению» (передаётся копия переменной, а не сама переменная), вызванная функция не имеет непосредственной возможности изменить переменную из вызывающей программы. Что же делать, если вам действительно надо изменить аргумент? Например, программа сортировки захотела бы поменять два нарушающих порядок элемента с помощью функции с именем swap. Для этого недостаточно написать:

```
swap(a, b);
```

определив функцию swap при этом следующим образом:

```
void swap(int x, int y)
{
    int temp;
    temp = x;
    x = y;
    y = temp;

}
```

Из-за вызова по значению swap не может воздействовать на аргументы a и b в вызывающей функции.

К счастью, все же имеется возможность получить желаемый эффект. Вызывающая программа передает указатели подлежащих изменению значений:

```
swap(&a, &b);
```

Так как операция & выдает адрес переменной, то &a является указателем на a. В самой swap аргументы описываются как указатели и доступ к фактическим операндам осуществляется через них:

```
void swap(int* px,int* py)
{
    int tmp;
    temp = *px;
    *px = *py;
    *py = temp;
}
```

Рассмотрим создания функции Swap с передачей по указателю:

```
#include <iostream>
using namespace std;

/*
    Передача переменных через указатель
    Внутри меняем значения переменных на которые
    указывают указатели
*/
void Swap(int* a, int* b) {
    int c = *a;
    *a = *b;
    *b = c;
}
```

```
int main()
{
    int av = 5, bv = 9;
    Swap(&av, &bv);
    // 9 5
    cout << av << " " << bv << endl;
    return 0;
}
```

5. Анализ использования NULL и nullptr

При создании указателя, если вы его не инициализируете, в него помещается случайное значение (его также называют мусорным значением). Наличие мусорного значения в указателе крайне опасно. Ведь это значение не имеет никакого смысла. Если мы попытаемся по нему обратиться последствия будут самые плачевные.

```
int*  ptr;
cout<<*ptr;
```

К счастью, Microsoft Visual Studio 2019 при попытке компиляции такой программы выдаст ошибку. Однако, любой другой компилятор C++ может повести себя иначе.

Поэтому лучше при создании указателя инициализировать его нейтральным адресом. Нужно показать в коде программы, что указатель пустой. Для этого используют нулевые указатели. Нулевой указатель показывает, что он никуда не указывает. Как показать, что указатель нулевой? Есть три пути.

Первый путь вписать значение 0 в указатель:

```
int* ptr = 0;
```

Потом в коде мы можем проверять, является ли указатель нулевым на текущий момент времени:

```
if (ptr != 0){
    …..
}
```

Такой путь создания нулевого указателя можно использовать, однако значение 0 это числовое значение и оно никак не отображает, что мы имеем дело с адресом.

Второй путь: мы можем воспользоваться макросом NULL, который пришел в C++ из языка C.

В случае C++ в Visual Studio вместо NULL подставляется 0.

```
int* ptr = NULL;

……

if(ptr != NULL){
     ……
}
```

Читая этот код понятнее, что имеешь дело с нулевым указателем. Однако, в C++ есть родной способ создания нулевого указателя. Для этого нужно использовать константу nullptr.

```
int* ptr = nullptr;

……

if(ptr != nullptr){
     ……
}
```

Именно этот способ является предпочтительным в C++ для создания нулевого указателя. Рассмотрим все три подхода в общем примере:

```
#include <iostream>
using namespace std;

int main()
{
    /*
        Указатель не инициализирован
        В нём случайное значение
    */
    int* ptr;
    // сейчас ptr нулевой указатель
    // так делать не рекомендуется
    ptr = 0;
    cout << ptr<< endl;
    // Наследие языка C
    // так делать не рекомендуется
    ptr = NULL;
    cout << ptr << endl;
    // современный способ C++
    // создания нулевого указателя
    ptr = nullptr;
    cout << ptr << endl;
    if (ptr == nullptr) {
        cout << "\n\nNull pointer was found!" << endl;
    }
    return 0;
}
```

Записывайте значение nullptr в указатель, чтобы не получить неприятные моменты при поиске багов в ваших программах.

6. Домашнее задание

1. Дан массив целых чисел. Воспользовавшись указателями, поменяйте местами элементы массива с четными и нечетными индексами (т.е. те элементы массива, которые стоят на четных местах, поменяйте с элементами, которые стоят на нечетных местах).
2. Даны два массива, упорядоченных по возрастанию: A[n] и B[m]. Сформируйте массив C[n+m], состоящий из элементов массивов A и B, упорядоченный по возрастанию. Используйте синтаксис указателей.
3. Даны два массива: A[n] и B[m]. Необходимо создать третий массив, в котором нужно собрать:

- Элементы обоих массивов;
- Общие элементы двух массивов;
- Элементы массива A, которые не включаются в B;
- Элементы массива B, которые не включаются в A;
- Элементы массивов A и B, которые не являются общими для них (то есть объединение результатов двух предыдущих вариантов).

Обязательно используйте синтаксис указателей для решения этой задачи.