Лекция 3

Массивы, векторы

Современное решение:

```
#include <vector>
using namespace std;
vector<int> numbers(5);
```

Более старая форма:

```
int numbers[5];
```

В языке C++ существует соглашение: элементы в векторе нумеруются, начиная с 0.

`numbers[0] = 111;` присвоение значения 111 первому элементу вектора.

`int i = numbers[2];` значение в третьем элементе вектора присваиваем целочисленной переменной i.

`numbers[1] = numbers[4];` значение пятого элемента скопировано во второй.

Вычисление суммы всех значений, хранящихся в векторе:

```
vector<int> numbers(5);

int sum = 0;

for(int i = 0; i < 5; i++)

    sum += numbers[i];
```

Инициализация вектора:

```
vector<int> int_vector = {0,1,2,3,4};
```

Вектор нулевой длины:

```
vector<int> int_vector;
```

Можно использовать векторы разных типов:

```
vector<float> float_vec(10);

vector<char> surname(20);

vector<bool> votes(100);
```

Векторы могут содержать элементы более сложной структуры

Элементы вектора являются векторами (имеет более одного измерения, массив) :

```
vector<vector<float>>temp(31, vector<float>(24));
```

Пример 3-мерного массива (вектор вектора векторов)

Отель, состоящий из трех зданий, по 15 этажей. На каждом этаже по 20 комнат. Нам нужен массив, который может собирать и обрабатывать информацию о количестве гостей, зарегистрированных в каждом номере.

```
vector<vector<vector<int>>>;

vector<vector<vector<int>>>                guests(3,
vector<vector<int>>(15, vector<int>(20)));

vector<vector<vector<int>>>                guests(3,
vector<vector<int>>(15, vector<int>(20)));

vector<vector<vector<int>>>                guests(3,
vector<vector<int>>(15, vector<int>(20)));
```

Забронировать номер для 2-х человек на во 2 корпусе на 10 этаже, комната 14:

```
 guests[1][9][13] = 2;
```

Освободить 2-ую комнату на 5 этаже в 1-ом здании:

```
  guests[0][4][1] = 0;
```

Есть ли свободные места на на 15-ом этаже 3-его здания:

```
int room;

int vacancy = 0;

for (room = 0; room < 20; room++)

  if (guests[2][14][room] == 0)

    vacancy++;
```

Преимущества векторов: матрицы и массивы старого стиля жестки. Созданные однажды, они существуют неизменными до конца своего

существования. Их нельзя ни расширить, чтобы вместить больше данных, ни сжать, чтобы освободить ненужную память. Современный вектор может сделать все это сам.

Использование функции. Нахождение квадратного корня из переменной типа float:

```
float value = 144;

float root = sqrt(value);  // 12.
```

Методы. У каждого вектора есть метод size, который предоставляет текущий размер вектора объекта.

Вызов метода:

```
vector<int> vect(10);

int current_size = vect.size(); // 10
```

Метод вызывается изнутри вектора!

Современные векторы (по сравнению со старыми решениями) оснащены множеством методов, что делает их удобными и гибкими.

При старом объявлении массива простого способа определить количество элементов вектора не существует - метод size() не существует и не может быть вызван.

Одно из возможных решений, которое можно использовать вместо этого, - определить размер всего вектора и разделить его на размер одного элемента.

Современные векторы могут быть свободно расширены - старые не могут. Метод с именем push_back(value) способен увеличить размер вектора на единицу и поместить новое значение в конец вектора.

Пример: установили начальный размер вектора = 0 и поместили три значения, изменив размер вектора на 3.

Тип возвращаемого значения должен быть совместим с типом элементов вектора!

Есть много возможностей, которые предоставляют новые векторы стиля, а старые - нет.

Указатели

При объявлении переменной - она занимает часть компьютерной памяти.

Важно знать, какое значение хранится в переменной.

Где хранится это значение?

Эта характеристика данных (атрибут) - адрес.

Каждая переменная “живет” по своему адресу.

Важное различие:

- значение переменной - это то, что хранит переменная;
- адрес переменной - это информация о том, где находится эта переменная.

Указатели используются для хранения информации о местоположении (адресе) данных.

Объявление обычных типов данных и типа указателя:

```
int i;  int *p;
```

Синтаксис языка C++ гарантирует, что всегда можно определить, для какого значения используется * в любом конкретном контексте.

Объявление означает, что p является указателем и будет использоваться для хранения информации о местоположении данных типа int.

Указатели всегда используются для указания на конкретные данные, указанные в объявлении.

Можно использовать аморфные указатели, которые можно использовать для указания на любые данные любого типа.

Каков тип переменной p? Есть несколько ответов, идентичных по смыслу:

1) p - это переменная типа “указатель на int”

2) p - переменная типа int * (* - это не умножение).

Указателю можно присвоить значение с помощью оператора =.

Какие значения разрешено присваивать указателям?

Использование литерала - это не вариант: p = 148324;

Исключение: p = 0;

Это - нулевой указатель, null (лат., nullus – нет), ему присвоено нулевое значение.

Присвоение нулевого значения переменной указателя иногда приводит к ошибкам и недоразумениям, существует неписаное соглашение о том, что разработчики будут избегать присвоения нулевых указателей.

Они должны следовать следующему: `p = nullptr;`

Символ `nullptr` является воплощением пустого (нулевого) указателя. Он выглядит как переменная, но нельзя изменить его значение, его можно было назвать constant. `nullptr` должен быть назначен только указателям.

Более старые стандарты C++ использовали другой (производный от языка C) способ обозначения указателей null - это было сделано словом (фактически макросом, т.е. символом, неявно замененным компилятором на целое число ноль) NULL.

Можно найти следы этой практики в более старом коде C++.

Еще: использование символа NULL требует включения файла заголовка с именем cstring или любого другого файла заголовка, который включает в себя саму cstring (один из них - iostream).

Можно присвоить указателю значение, указывающее на любую уже существующую переменную.

Для этого - оператор &, оператор ссылки (reference).

Это унарный префиксный оператор.

Оператор находит адрес своего аргумента.

Если назначить nullptr указателю, то указатель p не указывает ни на переменную i, ни на какую-либо другую переменную.

Объявление: `p = &i;`

Переменная p укажет на место, где в памяти хранится переменная i.

Указатель (который не является нулевым) можно разыменовать (dereference).

Разыменование - это операция, с помощью которой переменная-указатель становится синонимом значения, на которое она указывает.

Это может быть переменная и выражение, которое выдает указатель.

Объявление переменных типа int (ivar) и типа int * (ptr): int ivar, *ptr;

```
ivar = 2;   // присвоение значения 2 переменной ivar

ptr = &ivar;     // указатель ptr  указывает на переменную ivar
```

Получение значения, на которое указывает указатель - разыменование (dereferencer).

```
*ptr     // значение, которое хранится в том месте, на которое указывает указатель
```

Вызов `cout << *ptr;` отобразит 2 на экране (ptr указывает на ivar , а ivar = 2).

Если написать: *ptr = 4 значение указателя не изменится.

Изменится значение, на которое указывает указатель. Это важное различие.

Объявление указателя `ANY_TYPE *pointer;` означает, что:

- переменная pointer имеет тип `ANY_TYPE*`
- * выражение имеет тип `ANY_TYPE*`

Разыменование указателей nullptr строго запрещено и приводит к проблемам.

Основные операции с указателями:

* - взятие содержимого по адресу (*i - содержимое переменной с адресом i)

& - взятие адреса (&a - адрес переменной a).

Описание:

тип *имя_указателя;

При описании указателя задается тип значения, на которое он указывает.

Примеры описаний:

```
int *i, j, *pointj;
```

```
int v1, *pointv1=&v1, *p=(int*)200;

int i=1, num=40;

ptr=&i;

ptr=&num;

ptr=&num;  val=num;

val=*ptr;
```

нельзя:

```
p=&(c+2);

p=&'A';
```

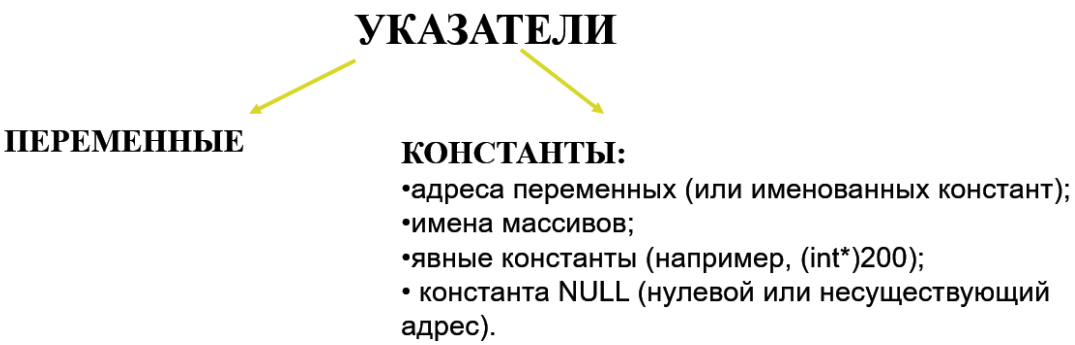

Рис. 3.2. Типы указателей

Нельзя брать содержимое от константы без приведения типа:

*200; - нельзя

(int)200; - можно

Нельзя брать адрес явной константы:

&200; - нельзя

Нельзя определять адрес выражения.

Оператор sizeof

Оператор sizeof выглядит как переменная.

Это оператор с унарным префиксом и с максимально возможным приоритетом. Другое отличие: типичный оператор требует значения в качестве аргумента и обычно изменяет значение определенным образом.

Оператор ожидает, что его аргумент является литералом, переменной, или выражением, заключенным в круглые скобки, или именем типа.

Оператор предоставляет информацию о том, сколько байтов памяти занимает (или может занимать) его аргумент.

Название объясняет цель: sizeof

sizeof – это не только оператор, но и ключевое слово (keyword).

```
i = sizeof(char); // переменной i будет присвоено значение1 т.к. значения char всегда занимают один байт.
```

Можно добиться того же эффекта:

```
char c;

int i = sizeof c;
```

Нельзя использовать ( ), когда аргумент является литералом или значением, но вы должны использовать их, когда аргумент является типом.

Переменная i = 8 (64-разрядные адреса), i= 4 (32-разрядные адреса).

```
int *ptr = nullptr;

int i = sizeof ptr;
```

Переменная j всегда будет иметь значение 8, так как double занимают 64 бита по правилам IEEE-754.

```
int j = sizeof 3.1415;
```

Пример. int k = sizeof k; Значения типа int занимают 4 байта в большинстве современных компиляторов/компьютеров, но этого нельзя гарантировать для всех ПК. Запустите программу на вашем компьютере. Узнаете, как ваш ПК используют память.

```
int main()
{
  cout << "This computing environment uses:" << endl;
  cout << sizeof(char) << " byte for chars" << endl;
  cout << sizeof(short int) << " bytes for shorts" << endl;
  cout << sizeof(int) << " bytes for ints" << endl;
  cout << sizeof(long int) << " bytes for longs" << endl;
  cout << sizeof(float) << " bytes for floats" << endl;
  cout << sizeof(double) << " bytes for doubles" << endl;
  cout << sizeof(bool) << " byte for bools" << endl;
  cout << sizeof(int *) << " bytes for pointers" << endl;
}
```

Вывод:

1 byte for chars

2 bytes for shorts

4 bytes for ints

8 bytes for longs

4 bytes for floats

8 bytes for doubles

1 byte for bools

8 bytes for pointers

Арифметика указателей

Арифметика указателей позволяет выполнять только следующие операции:

- добавление целого значения к указателю, дающее указатель (ptr + int → ptr);
- вычитание целого значения из указателя, дающее указатель (ptr – int → ptr);
- вычитание указателя из указателя, дающее целое число (ptr – ptr → int);
- сравнение двух указателей на равенство или неравенство (дает значение типа int либо true , либо false) (ptr = = ptr → int; ptr!= ptr → int).

Любые другие операции либо запрещены, либо бессмысленны

Результат арифметической операции над указателями зависит не только от значения операндов, но и от типа, с которым связан указатель. ptr=ptr+k, Û p увеличивается на k*sizeof (тип)

Что общего у указателей и векторов?

Метод вектора с именем data() возвращает указатель, указывающий на первый элемент вектора.

```
vector<int> vect {1, 2, 3};

int *ptr = vect.data();
```

Объявили трехэлементный вектор типа int (с именем vect) и указатель на int (с именем ptr), изначально установленный в значение, связанное с самым первым элементом вектора.

Того же эффекта можно достичь, применив оператор & к vect[0].

```
int *ptr = &vect[0];
```

Код выводит значение 1, это доказывает, что ptr и ptr2 указывают на одно и то же местоположение в памяти - элемент первого вектора:

```
#include <iostream>

#include <vector>

using namespace std;

int main()

{

  std::vector vect {1, 2, 3};

  int *ptr = vect.data();

  int *ptr2 = &vect[0];

  cout << (ptr == ptr2) << endl;

}
```

Что общего у указателей и массивов?

Имя массива без индексов - это всегда синоним указателя, указывающего на первый элемент массива.

Или: имя одномерного массива является указателем-константой, равной адресу начала массива, т. е. адресу элемента с индексом 0 (первого элемента).

```
int *ptr, arr[3];    //объявили указатель на int и трехэлементный массив типа int.

arr == &arr[0];   // всегда верно
```

Устанавливают ptr на одно и то же значение:

```
int * ptr, arr[3];
```

```
ptr = &arr[0];

ptr = arr;

int a[10];

&a[0]    // эквивалентно a

a[0]     // эквивалентно *a

&a[i]    // эквивалентно a+i (i=0,1,...9)

a[i]     // эквивалентно *(a+i)
```

Имя двумерного массива является указателем-константой на начало (элемент с индексом 0) массива указателей-констант, i-й элемент этого массива - указатель -константа на начало (элемент с индексом 0) i-й строки двумерного массива.

Пример: `int b[5][8];`

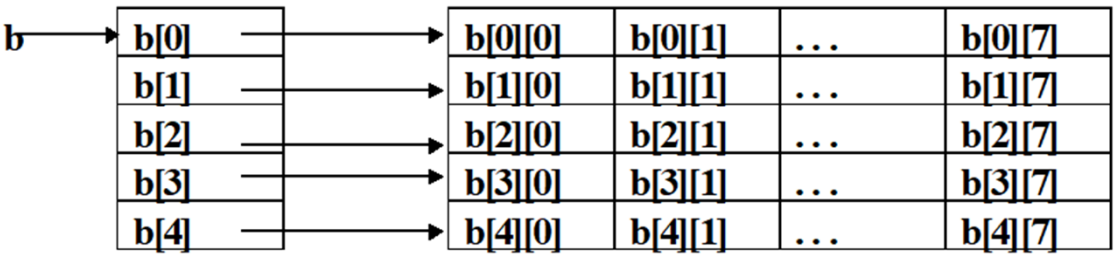

Рис. 3.3. Пример двухмерного массива

Пример. ptr1 указывает на первый элемент array. После следующего назначения ptr2 также указывает на первый элемент массива.

```
vector<int> array {1, 2, 3};
     int *ptr1 = array.data();
     int *ptr2;
     int i;
}
```

```
int *ptr1, *ptr2, array[3],
i;
ptr1 = array;
```

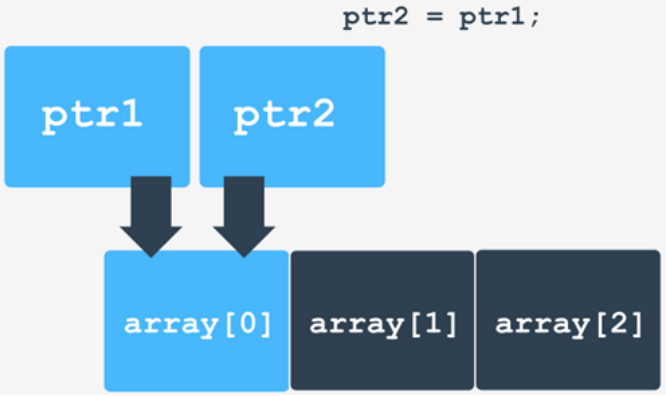

Рис. 3.4. Два указателя указывают на одни и те же данные

Операторы выполняют одну и ту же операцию - они добавляют 1 к ptr2 (рисунок):

• учитывается, на какой тип указывает указатель - в примере это int;

• определил, сколько байтов памяти занимает тип (автоматически используется оператор sizeof), в данном случае это sizeof (int);

• значение, которое хотим добавить к указателю, умножается на заданный размер;

• адрес, который хранится в указателе, увеличивается на результирующий продукт.

По сути, указатель сам перемещается к следующему значению int в памяти.

Если мы добавим 2 вместо 1, то ptr2 будет увеличен на (2 * sizeof (int)), ptr2 пройдет через два значения int и укажет на третий элемент массива (array[2]).

Сравнение указателей и вычитание

ptr1 == ptr2 // ложь

ptr1 != ptr2 // истина, так как адреса, на которые указывают указатели, разные

Вычитание (вычитание дает тип int):

- учитывается: тип, на который указывают указатели (int); это означает, что оба указателя должны указывать на один и тот же тип; компилятор проверит это;

- адреса, сохраненные в указателях, вычитаются;

- результат вычитания делится на размер типа, на который указывают указатели.

Конечный результат говорит, сколько переменных данного типа (int) помещается между адресами, хранящимися в указателях. В данном случае это1, она будет присвоена i.

```
i = ptr2 - ptr1;
```

Результат будет > 0, если ptr2 указывает на память, расположенную после ptr1; в противном случае он будет < 0.

Сходство между векторами и указателями

Указатели могут вести себя как векторы и наоборот.

Пусть указатель ptr1 указывает на второй элемент массива vector. Хотим:

1) разыменовать его и получить значение, находящееся по адресу, который содержит указатель: int value = *ptr1;

2) разыменовать значение, расположенное на один элемент после указания текущего указателя. Но не хотим, чтобы указатель был изменен:

value = *(ptr1 + 1); круглые скобки в выражении обязательны.

value = *ptr1 + 1; значение, указанное ptr1 , увеличилось на 1

Это потому, что *, работающий в качестве разыменователя, имеет более высокий приоритет, чем +.

Язык C++ предполагает, что операция `*(pointer + offset)` является синонимом `pointer[offset]`. Это означает, что наше предыдущее задание может быть переписано как:

```
value = ptr1[1];
```

В этом сходство с поведением вектора.

Использование отрицательных индексов также возможно, но только для указателей, для векторов – нет.