Модуль 2. Массивы, строки, указатели. Основные алгоритмы для работы с массивами данных

Тема 2.3. Ссылки. Указатели

Указатели и ссылки. Инициализация указателей и ссылок. Операции с указателями. Арифметика указателей. Указатели и константы. Указатели и массивы. Динамические массивы.

Под переменной мы понимаем именованную область памяти. Например, при объявлении целочисленной переменной i

```
int i;
```

в оперативной памяти выделяется место размером 4 байта.

```
i = 0;
```

При присваивании переменной значения оно записывается в область памяти, выделенную для переменной.

Пока мы ни разу не говорили об «адресе» переменной. Правильнее будет сказать, о физическом адресе той области оперативной памяти, куда записывается значение переменной. Делается это с помощью указателей. Под **указателем** подразумевают переменную, значением которой является адрес памяти. Таким образом, если значением обычной переменной является то, что записано в определенной области памяти, то значением переменной-указателя является адрес области памяти. Если указатель содержит в качестве значения адрес памяти с данными, то говорят, что указатель ссылается на эти данные.

Поясним сказанное на простом примере: на улице города каждый дом имеет свой номер и каждая квартира в доме также имеет номер. Говорят, что человек, живущий в определенной квартире, проживает по определенному адресу. Т е адрес является указателем на место жительства человека. Если предположить, что человек переехал и в ту же квартиру поселился другой житель, можно сказать, что наша квартира-переменная получила новое значение-жильца.

Работая в программе с переменными, мы фактически даем инструкции на предмет того, что делать со значениями этих переменных.

Существует две операции, которые приходится часто выполнять при работе с указателями. Во-первых, это определение адреса ячейки по ее имени (перед ее именем указать &) и, во-вторых, определение значения, записанного по указанному адресу (перед соответствующим указателем ставим оператор *).

Указатели - это переменные, значениями которых являются *адреса* других переменных. При объявлении указателя используется символ звездочка (*).

```
int* p; // p – указатель на переменную типа int
int d = 3; // d - переменная типа int
p = &d; // В указатель p скопирован адрес переменной d
```

Существуют следующие способы инициализации указателя:

1. Присваивание указателю адреса существующего объекта:
   a) с помощью операции получения адреса:
      int a = 5; // целая переменная
      int* p = &a; // в указатель записывается адрес a
      int* p (&a); // то же самое другим способом
   b) с помощью значения другого инициализированного указателя: int* r = p;
   c) с помощью имени массива или функции, которые трактуются как адрес
   int b[103] // массив

```
int* t = b;                    // присваивание адреса начала массива
```

2. Присваивание указателю адреса области памяти в явном виде:
char* vp = (char *)0xB8000000;
Здесь 0xB8000000 — шестнадцатеричная константа, (char *) — операция приведения типа: константа преобразуется к типу «указатель на char».

3. Присваивание пустого значения:

```
int* a= NULL;
int* b = 0;
```

В первой строке используется константа NULL, определенная в некоторых заголовочных файлах С как указатель, равный нулю. Рекомендуется использовать просто 0, так как это значение типа int будет правильно преобразовано стандартными способами в соответствии с контекстом. Поскольку гарантируется, что объектов с нулевым адресом нет, пустой указатель можно использовать для проверки, ссылается указатель на конкретный объект или нет.
4. Выделение участка динамической памяти и присваивание ее адреса указателю:
с помощью операции new:

```
int* n = new int;
```

операция new выполняет выделение достаточного для размещения величины типа int участка динамической памяти и записывает адрес начала этого участка в переменную n. Память под саму переменную n (размера, достаточного для размещения указателя) выделяется на этапе компиляции.
Более подробно это вариант рассмотри ниже.

Операции с указателями

1) К указателям можно применять унарный оператор *, возвращающий то значение, на которое ссылается данный указатель (операция разадресации, или разыменования). Важно помнить, что при этом * размещается перед указателем.

Рассмотрим для понимания сказанного ***пример 1***:

```
#include <iostream>
#include <clocale>

using namespace std;
int main()
{
	setlocale(LC_ALL, "Russian");
	int* p; // p – указатель на переменную типа int
	int d = 3; // d - переменная типа int
	p = &d; // В указатель p скопирован адрес переменной d

	cout << "Значение переменной d = " << d << endl;
	cout << "указатель p = " << p << endl;
	cout << "Разадресованный указатель *p = " << *p << endl;
	return 0;
}
```

2) **Арифметические операции.**
С указателями можно выполнять следующие операции: сложение с константой, вычитание, инкремент и декремент. Они автоматически учитывают размер типа величин, адресуемых указателями. Эти операции применимы только к указателям одного типа
и имеют смысл в основном при работе со структурами данных, последовательно

размещенными в памяти, например, с массивами.

Инкремент перемещает указатель к следующему элементу массива, *декремент* — к предыдущему. Фактически значение указателя изменяется на величину sizeof (тип). Если указатель на определенный тип увеличивается или уменьшается на константу, его значение изменяется на величину этой константы, умноженную на размер объекта данного типа, например:

```
short * p = new short [5];
p++; // значение p увеличивается на 2
long * q = new long [5];
q++; // значение q увеличивается на 4
```

Разность двух указателей — это разность их значений, деленная на размер типа в байтах (в применении к массивам разность указателей, например, на третий и шестой элементы равна 3). Суммирование двух указателей не допускается.
При записи выражений с указателями следует обращать внимание на приоритеты операций. В качестве примера рассмотрим последовательность действий, заданную в операторе

```
*p++ = 10;
```

Операции разадресации и инкремента имеют одинаковый приоритет и выполняются справа налево, но, поскольку инкремент постфиксный, он выполняется после выполнения операции присваивания. Таким образом, сначала по адресу, записанному в указателе p, будет записано значение 10, а затем указатель будет увеличен на количество байт, соответствующее его типу. То же самое можно записать подробнее:

```
*p = 10;
 p++;
```

Выражение (*p)++. напротив, инкрементирует значение, на которое ссылается указатель.
Унарная операция получения адреса & применима к величинам, имеющим имя и размещенным в оперативной памяти. Таким образом, нельзя получить адрес скалярного выражения, неименованной константы или регистровой переменной.

Ссылки

Ссылка представляет собой ***синоним имени,*** указанного при инициализации ссылки. Ссылку можно рассматривать как указатель, который всегда разыменовывается. Формат объявления ссылки:

тип & имя;

где тип — это тип величины, на которую указывает ссылка, & — оператор ссылки, означающий, что следующее за ним ***имя*** является именем переменной ссылочного типа, например:

```
int a;
int& b = a;                    // ссылка b - альтернативное имя для a;
const char&  c = ' \ n ' :     // ссылка на константу
```

Запомните следующие правила.

- Переменная-ссылка должна явно инициализироваться при ее описании, кроме случаев, когда она является параметром функции (этот случай будет рассмотрен в 3 модуле), extern или ссылается на поле данных класса (этот случай будет рассмотрен в 4 модуле).
- После инициализации ссылке не может быть присвоена другая переменная.
- Тип ссылки должен совпадать с типом величины, на которую она ссылается.
- Не разрешается определять указатели на ссылки, создавать массивы ссылок и

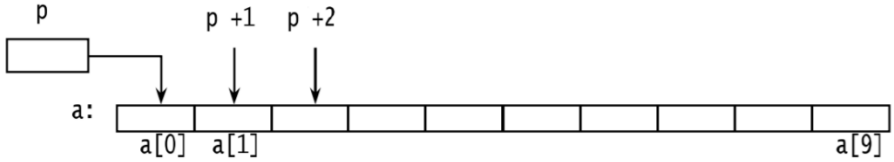
ссылки на ссылки.
Ссылки применяются чаще всего в качестве параметров функций и типов возвращаемых функциями значений. Ссылки позволяют использовать в функциях переменные, передаваемые по адресу, без операции разадресации, что улучшает читаемость программы.
Ссылка, в отличие от указателя, не занимает дополнительного пространства в памяти и является просто другим именем величины. Операция над ссылкой приводит к изменению величины, на которую она ссылается.

Указатели и массивы

Рассмотрим простой ***пример 2*** по работе с одномерными массивами:

```
#include <iostream>
using namespace std;
int main()
{
    int a[10];
    for (int i = 0; i < 10; i++) {
        a[i] = rand()% 10;
        cout << a[i] << " ";
    }
    cout << "\n";
    return 0;
}
```

Ячейки памяти, в которые сохраняются значения элементов массива, размещены рядом, olyf за другой. Проверки на предмет выхода за пределы массива в C++ нет, неверно указанный индекс элемента массива приводит к тому, что выполняется обращение к одной из смежных ячеек за пределами массива.
Еще одна особенность C++ связана с тем, что имя массива (без индексов) является **указателем** на первый элемент массива. Например, если массив создается командой `int a[10]`, то имя массива a является указателем (адресом) на первый элемент массива `a[0]`. Адрес этого элемента можно получить и так: `&a[0]`



Зная адрес первого элемента и количество элементов в массиве, получаем доступ ко всему массиву. Хотя нам привычнее осуществлять доступ через имя массива и индекс элемента, арифметические операции с адресами выполняются быстрее, по сравнению с индексированием массива.

Пример 2:

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main() {
	setlocale(LC_ALL, "Russian");
	int a[10], *p;
	p = a;
	cout <<  "Значение элемента массива a[i] выводится как *(p + i)\n";
	for (int i = 0; i < 10; i++) {
		p[i] = 10 - i;
		cout << *(p + i) << " ";
	}
	cout << "\n";
	cout << "\nЗначение элемента массива a[i] выводится без использования указателя *p\n";
	for (int i = 0; i < 10; i++) {
		*(a + i) = 10 - i;
		cout << a[i] << " ";
```

```
	}
	cout << "\n";
	return 0;
}
```

Динамическое выделение памяти

Рассмотрим подробнее четвертый способ инициализации указателя. Динамический массив от статического отличается в первую очередь тем, что на момент компиляции размер динамического массива неизвестен, в отличие от массивов статических, для которых размер должен быть известен уже при компиляции.

Динамические массивы реализуются посредством операторов динамического распределения памяти. В C++ для выделения области памяти используется оператор `new`, а для освобождения выделенной ранее памяти используют оператор `delete`. В частности, оператор `new` выделяет область памяти и возвращает в качестве значения указатель на первую ячейку выделенной памяти.

Итак, оператор `new` выделяет память под объект во время выполнения программы.

```
double *pd; // Указатель на double
pd = new double;
```

Выделяется память под переменную типа `double`, адрес которой присваивается `pd`. После оператора new указывается тип создаваемого объекта.

Оператор `delete` освобождает память, выделенную ранее оператором `new`

```
delete pd;
```

Теперь указатель pd можно использовать для других целей, а память, освобожденная оператором `delete`, может быть повторно использована под объекты, создаваемые оператором `new`.

Динамические массивы

Практически так же, как для обычных переменных, выделяется память для массивов. Главное отличие в синтаксисе вызова оператора `new` состоит в том, что после имени базового типа переменной указывается размер массива. Например, массив для 10 целых чисел можно создать так:

```
int* a = new int[10];
```

При освобождении памяти, выделенной под массив, после оператора `delete` перед именем указателя на массив указывается оператор `[]`.

```
delete[] a;
```

Таким образом, для выделения памяти под массив используют синтаксис вида

```
тип_массива* указатель;
указатель = new тип_массива[размер];
```

Для освобождения памяти, выделенной под массив, используют команду вида

```
delete[] указатель;
```

Рассмотрим ***пример 3:***

```
#include <iostream>
using namespace std;
int main() {
	int *p, n;
	cout << "Enter n = ";
	cin >> n;
	p = new int[n];
	for (int i = 0; i < n; i++) {
		p[i] = 2 * i + 1;
		cout << p[i] << " ";
	}
	delete[] p;
	cout << endl;
	return 0;
}
```

```
Enter n = 5
1 3 5 7 9
```

При создании одномерного динамического массива возвращается адрес первого его элемента и выделяется введенное пользователем n - количество ячеек памяти.

Двумерные динамические массивы

Двумерные динамические массивы создаются по тому же принципу, что и одномерные: сначала создаются динамические одномерные массивы **строк** двумерного массива, а адреса первых ячеек этих одномерных массивов заносятся в еще один динамический одномерный массив.
Рассмотрим на ***примере 4*** возможность создания двумерных динамических массивов:

```
#include <iostream>
using namespace std;
int main() {
	setlocale(LC_ALL, "Russian");
	int** p;
	int n, m, i, j;
	cout << "Введите количество строк: ";
	cin >> n;
	cout << "Введите количество столбцов: ";
	cin >> m;
	p = new int* [n]; // переменная p является указателем на указатель n
	for (i = 0; i < n; i++) {
		p[i] = new int[m]; //объявляется массив указателей на m целых чисел
		for (j = 0; j < m; j++) {
			p[i][j] = (i * m + j) % 10;
			cout << p[i][j] << " ";
		}
		cout << endl;
	}
	for (i = 0; i < n; i++)
		delete[] p[i];
	delete[] p;
	return 0;
}
```

```
Введите количество строк: 3
Введите количество столбцов: 4
0 1 2 3
4 5 6 7
8 9 0 1
```

Командой `p[i] = new int[m]` во внешнем цикле `for` элементу `p[i]` массива p вместе с выделением места под одномерный целочисленный массив размера m присваивается ссылка на первый элемент этого массива.
Если просто воспользоваться командой `delete[] p`, будет освобождено место в памяти, занятое массивом p адресов первых ячеек одномерных массивов. Сами одномерные массивы при этом останутся в памяти. Поэтому перед «уничтожением» массива p освобождаем память, выделенную для всех одномерных массивов. И лишь затем освобождается место, занятое «главным» массивом адресов p.