6. УКАЗАТЕЛИ

Оглавление

Одно из самых больших преимуществ языка C++ в том, что он позволяет писать высокоуровневые приложения, абстрагируясь от машинного уровня, и в то же время работать, по мере необходимости, близко к аппаратным средствам. Действительно, язык C++ позволяет контролировать производительность приложения на уровне байтов и битов. Понимание работы **указателей** и **ссылок** — один из этапов на пути к способности писать программы, эффективно использующие системные ресурсы.

§6.1 Понятие указателя

Указатель – это переменная, значением которого является адрес области в памяти. Точно также, как переменная типа `int` используется для хранения значения целочисленного типа, переменная указателя используется для хранения адреса области в памяти. Визуализация принципа работы указателя представлена на рисунке 6.1.

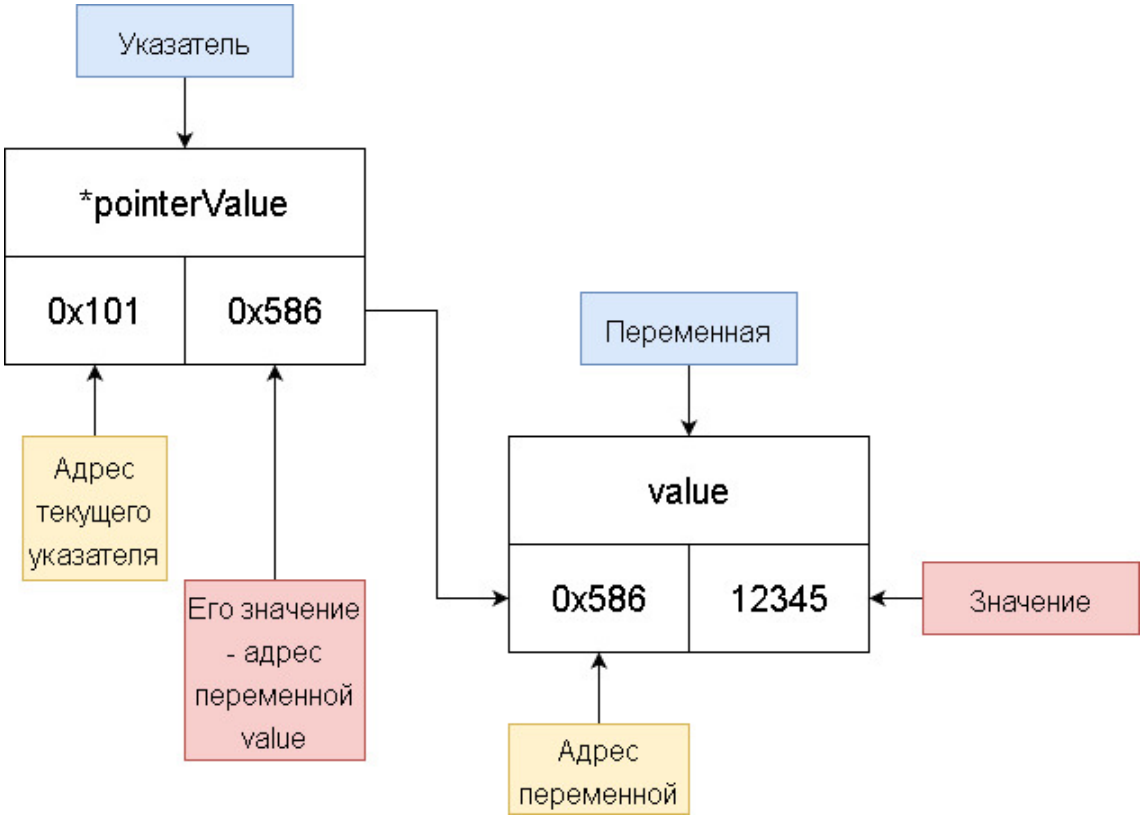

Рисунок 6.1 – Визуализация указателя

Указатель, как и любая переменная, занимает пространство в памяти (в случае рисунка 6.1 – это 0x101). Но особенным является то, что содержащееся в нем значение интерпретируется как адрес области памяти (в данном случае 0x586). Следовательно, указатель — это специальная переменная, которая указывает на область в памяти.

§6.2 Объявление указателя

Поскольку указатель является переменной, его следует объявить. Обычно вы объявляете, что указатель указывает на значение определенного типа (например, типа `int`). Это значит, что содержавшийся в указателе адрес указывает на область в памяти, содержащую целое число. Можно также определить указатель на блок памяти (называемый также пустым указателем (void pointer)).

Указатель должен быть объявлен, как и все остальные переменные:

```
ТипУказателя *ИмяПеременнойУказателя;
```

Отличие заключается только в том, что перед именем ставится символ звёздочки *. Визуально указатели отличаются от переменных только одним символом. При объявлении указателей компилятор выделяет несколько байт памяти, в зависимости от типа данных отводимых для хранения некоторой информации в памяти.

Как и в случае с большинством переменных, если не инициализировать указатель, он будет содержать случайное значение. Во избежание обращения к случайной области памяти, указатель инициализируют значением `NULL`. Наличие значения `NULL` можно проверить, и оно не может быть адресом области памяти:

```
ТипУказателя *ИмяПеременнойУказателя = NULL; // инициализирующее значение
```

Таким образом, объявление указателя на целое число было бы таким:

```
int *pointerValue = NULL;
```

Указатель – это переменная специального типа для хранения адресов памяти. Для получения адреса переменной используется **оператор ссылки &**. Если `value` – переменная, то применение оператора ссылки с именем переменной `&value` возвращает адрес в памяти, где хранится ее значение.

Так, если вы объявили переменную, используя хорошо известный синтаксис инициализации переменной:

```
int value = 12345;
```

то оператор `&value` вернет адрес области в памяти, куда помещается значение (12345).

Оператор ссылки (referencing operator) (&) называется также **оператором обращения к адресу** (address-of operator).

Для того, чтобы сохранить адрес этой переменной в указателе, следует объявить указатель на тот же тип и инициализировать его, используя оператор обращения к адресу (&).

```
Тип *Указатель = &ИмяПеременной;
```

Указатель на тип `int`, хранящий адрес значения ранее объявленной целочисленной переменной `value`, будет объявляться следующим образом:

```
int *pointerValue = &value;
```

Листинг 6.1 демонстрирует применение указателя для хранения адреса, полученного с использованием оператора обращения к адресу (&).

Листинг 6.1.

```
1  #include <iostream>
2  using namespace std;
3  int main()
```

```
4  {
5  int Age = 20;
6  int* plnteger = &Age;
7  cout << "Integer Age is at: " << plnteger << endl;
8  return 0;
9  }
```

Результат выполнения программы:

Integer Age is at: 0x0045FE00

В данном примере в строке 6 объявляется указатель на целочисленный тип, инициализированный результатом операции обращения к адресу переменной `Age`, объявленной в строке 5. Далее отображается значение указателя – адрес в памяти, где хранится содержимое переменной `Age`.

Стоит заметить, что адрес переменной может изменяться при каждом запуске приложения на том же компьютере.

Для того, чтобы записать или прочитать данные, содержащиеся в области памяти, адрес которой хранит указатель, используется **оператор обращения к значению (оператор разыменования) (*)**. По существу, если есть допустимый указатель `pData`, использование оператора `*pData` позволяет получить доступ к значению, хранящемуся по адресу, содержащемуся в указателе.

Использование оператора разыменования (*) показано в листинге 6.2.

Листинг 6.2.

```
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5  int Age = 20;
6  int DogsAge = 9;
7  cout << "Integer Age = " << Age << endl;
8  cout << "Integer DogsAge = " << DogsAge << endl;
9  int *plnteger = &Age;
10 cout << "plnteger points to Age" << endl;
11 // Отображение значения указателя
12 cout << "plnteger = " << plnteger << endl;
13 // Отображение значения в указанной области
14 cout << "*plnteger = " *plnteger << endl;
15 plnteger = &DogsAge;
16 cout << "plnteger points to DogsAge now" << endl;
```

```
17  cout << "plnteger = " << plnteger << endl;
18  cout << "*plnteger = " << *plnteger << endl;
19  return 0;
20  }
```

Результат выполнения программы:

Integer Age = 20

Integer DogsAge = 9

plnteger points to Age

plnteger = 0x0025F788

*plnteger = 20

plnteger points to DogsAge now

plnteger = 0x0025F77C

*plnteger = 9

Кроме изменения адреса, хранимого в указателе (строка 15), здесь используется также оператор обращения к значению (*) с тем же указателем `plnteger` для отображения значения двух адресов (строка 14, 18).

В обеих этих строках осуществляется доступ к целочисленному значению, на которое указывает указатель `plnteger`, с использованием оператора обращения к значению (*). Поскольку адрес, содержавшийся в указателе `plnteger`, изменяется в строке 15, тот же указатель после этого позволяет обратиться к переменной `DogsAge` и отобразить значение 9.

Указатель в приведенном выше примере использовался для чтения (получения) значения из области памяти, на которую указывает указатель. В листинге 6.3 показано, что происходит, когда оператор `*plnteger` используется как l-value, т.е. для присвоения значения, а не для его чтения.

Листинг 6.3.

```
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5  int DogsAge = 20;
6  cout << "Initialized DogsAge = " << DogsAge << endl;
7  int *pAge = &DogsAge;
```

```
8   cout << "pAge points to Dog'sAge" << endl;
9   cout << "Enter an age for your dog: ";
10  // сохранить ввод в области памяти, на которую указывает pAge
11  cin >> *pAge;
12  // Отобразить адрес, по которому хранится возраст
13  cout << "Input stored using pAge at " << pAge << endl;
14  cout << "Integer DogsAge = " << DogsAge << endl;
15  return 0;
16  }
```

Результат выполнения программы:

Initialized DogsAge = 20

pAge points to DogsAge

Enter an age for your dog: 10

Input stored using pAge at 0x0025FA18

Integer DogsAge = 10

Ключевой этап здесь в строке 11, где введенное пользователем целое число сохраняется в области, на которую указывает указатель `pAge`. Обратите внимание, несмотря на то, что введенное число было сохранено при помощи указателя `pAge`, строка 14 отображает это же значение при помощи переменной `DogsAge`. Это связано с тем, что указатель `pAge` указывает на переменную `DogsAge`, как было инициализировано в строке 7. Любое изменение в области памяти, где хранится значение переменной `DogsAge`, и на которую указывает указатель `pAge`, отражается на обоих.

§6.3 Арифметика указателей

Указатели могут участвовать в арифметических операциях (сложение, вычитание, инкремент, декремент). Однако сами операции производятся немного иначе, чем с числами. И многое здесь зависит от типа указателя.

К указателю можно прибавлять целое число, и также можно вычитать из указателя целое число. Кроме того, можно вычитать из одного указателя другой указатель.

Рассмотрим вначале операции инкремента и декремента и для этого возьмем указатель на объект типа `int`:

Листинг 6.4.

```
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5  int n = 10;
6  int *ptr = &n;
7  cout << "address=" << ptr << "\tvalue=" << *ptr << endl;
8  ptr++;
9  cout << "address=" << ptr << "\tvalue=" << *ptr << endl;
10 ptr--;
11 cout << "address=" << ptr << "\tvalue=" << *ptr << endl;
12 return 0;
13 }
```

Операция инкремента ++ увеличивает значение на единицу. В случае с указателем увеличение на единицу будет означать увеличение адреса, который хранится в указателе, на размер типа указателя. То есть в данном случае указатель на тип `int`, а размер объектов `int` в большинстве архитектур равен 4 байтам. Поэтому увеличение указателя типа `int` на единицу означает увеличение значение указателя на 4.

Результат работы программы показан на рисунке 8.2.

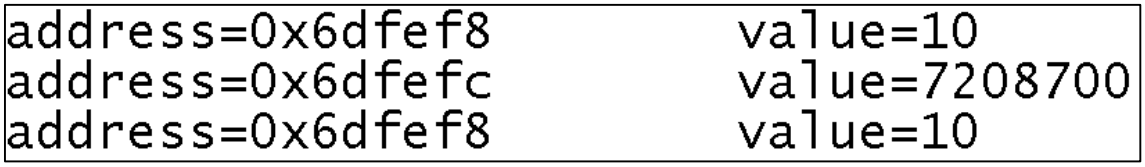

Рисунок 8.2

Здесь видно, что после инкремента значение указателя увеличилось на 4: с `0x6dfef8` до `0x6dfefc`. А после декремента, то есть уменьшения на единицу, указатель получил предыдущий адрес в памяти.

Фактически увеличение на единицу означает, что мы хотим перейти к следующему объекту в памяти, который находится за текущим и на который указывает указатель. А уменьшение на единицу означает переход назад к предыдущему объекту в памяти.

После изменения адреса мы можем получить значение, которое находится по новому адресу, однако это значение может быть неопределенным, как показано в случае выше.

В случае с указателем типа `int` увеличение/уменьшение на единицу означает изменение адреса на 4. Аналогично, для указателя типа `short` эти операции изменяли бы адрес на 2, а для указателя типа `char` на 1.

В отличие от сложения операция вычитания может применять не только к указателю и целому числу, но и к двум указателям одного типа:

Листинг 6.5.

```
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5  double d3[5] = { 1.0, 2.0, 3.0, 4.0, 5.0 };
6  cout << "&d3[4] = " << &d3[4] << endl;
7  cout << "&d3[1] = " << &d3[1] << endl;
8  int l = &d3[4] - &d3[1];
9  cout << "Length = " << l << endl;
10 }
```

Результат работы программы показан на рисунке 8.3.

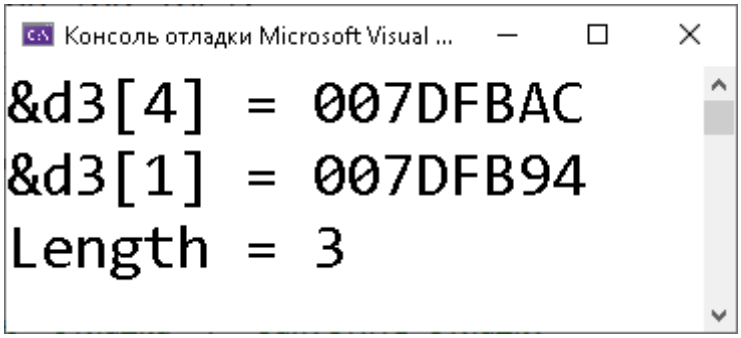

Рисунок 8.3

Результатом разности двух указателей является **«расстояние»** между ними.

Расстояние – это разность значений указателей, деленная на размер типа в байтах. Так, разность указателей на третий и нулевой элементы массива равна трем, а на третий и девятый — шести. Суммирование двух указателей не допускается.

Вычитание двух указателей определяет, сколько переменных данного типа размещается между указанными ячейками. Эти операции применимы только к указателям одного типа и имеют смысл в основном со структурными типами данных, например, с массивами.

Например, в случае выше адрес из первого указателя на 12 больше, чем адрес из второго указателя (0x7DFBAC + 4*3 = 0x7DFB94). Так как размер одного объекта `int`

равен 4 байтам, то расстояние между указателями будет равно (0x7DFBAC - 0x7DFB94)/4 = 3.

При работе с указателями надо отличать операции с самим указателем и операции со значением по адресу, на который указывает указатель.

Листинг 6.6.

```
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5  int a = 10;
6  int *pa = &a;
7  int b = *pa + 20;    // операция со значением, на который указывает указатель
8  pa++;                // операция с самим указателем
9  cout << "b: " << b << endl;  // 30
10 return 0;
11 }
```

То есть в данном случае через операцию разыменования `*pa` получаем значение, на которое указывает указатель `pa`, то есть число 10, и выполняем операцию сложения. То есть в данном случае обычная операция сложения между двумя числами, так как выражение `*pa` представляет число.

Но в то же время есть особенности, в частности, с операциями инкремента и декремента. Дело в том, что операции *, ++ и -- имеют одинаковый приоритет и при размещении рядом выполняются справа налево.

Например, выполним постфиксный инкремент:

Листинг 6.7.

```
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5  int a = 10;
6  int *pa = &a;
7  cout << "pa: address=" << pa << "\tvalue=" << *pa << endl;
8  int b = *pa++;      // инкремент адреса указателя
9  cout << "b: value=" << b << endl;
10 cout << "pa: address=" << pa << "\tvalue=" << *pa << endl;
11 }
```

В выражении `b = *pa++;` сначала к указателю присваивается единица (то есть к адресу добавляется 4, так как указатель типа `int`). Затем так как инкремент постфиксный, с помощью операции разыменования возвращается значение, которое было до инкремента - то есть число 10. И это число 10 присваивается переменной `b`. Результат работы программы показан на рисунке 8.4.

```
pa: address=0x6dfef4    value=10
b: value=10
pa: address=0x6dfef8    value=10
```

Рисунок 8.4

Изменим выражение:

```
int b = (*pa)++;
```

Скобки изменяют порядок операций. Здесь сначала выполняется операция разыменования и получение значения, затем это значение увеличивается на 1. Теперь по адресу в указателе находится число 11. И затем так как инкремент постфиксный, переменная `b` получает значение, которое было до инкремента, то есть опять число 10. Таким образом, в отличие от предыдущего случая все операции производятся над значением по адресу, который хранит указатель, но не над самим указателем. И, следовательно, изменится результат работы (см. Рисунок 8.5):

```
pa: address=0x6dfef4    value=10
b: value=10
pa: address=0x6dfef4    value=11
```

Рисунок 8.5

Аналогично будет с префиксным инкрементом:

```
int b = ++*pa;
```

В данном случае сначала с помощью операции разыменования получаем значение по адресу из указателя `pa`, к этому значению прибавляется единица. То есть теперь значение по адресу, который хранится в указателе, равно 11. Затем результат операции присваивается переменной b.

Изменим выражение:

```
int b = *++pa;
```

Теперь сначала изменяет адрес в указателе, затем мы получаем по этому адресу значение и присваиваем его переменной b. Полученное значение в этом случае может быть неопределенным.

§6.4 Динамическое распределение памяти

Когда вы пишете программу, содержащее такое объявление массива, как

```
int Numbers[100]; // статический массив для 100 целых чисел
```

у вашей программы есть две проблемы.

1. Так вы фактически ограничиваете емкость своей программы, поскольку она не сможет хранить больше 100 чисел;

2. Вы ухудшаете производительность системы в случае, когда храниться должно только 1 число, а пространство все равно резервируется для 100 чисел.

Причиной этих проблем является статическое, фиксированное резервирование памяти для массива компилятором, как уже обсуждалось ранее.

Чтобы программа могла оптимально использовать ресурсы памяти, в зависимости от потребностей пользователя, необходимо использовать **динамическое распределение памяти**. Это позволит при необходимости резервировать больше памяти и освобождать ее, когда необходимости в ней больше нет. Язык C++ предоставляет два оператора, new и delete, позволяющие подробно контролировать использование памяти в вашем приложении. Указатели, являющиеся переменными, хранящими адреса памяти, играют критически важную роль в эффективном динамическом распределении памяти.

Оператор new используется для **резервирования** новых блоков памяти. Чаще всего используется версия оператора new, возвращающая указатель на затребованную область память в случае успеха, и передающая исключение в противном случае. При использовании оператора new необходимо указать тип данных, для которого резервируется память:

```
Тип* Указатель = new Тип;
// запрос памяти для одного элемента
```

Вы можете также определить количество элементов, для которых хотите зарезервировать память (если нужно резервировать память для нескольких элементов):

```
Тип* Указатель = new Тип [КолЭлементов];
```

```
//запрос элементов для КолЭлементов
```

Таким образом, если необходимо разместить в памяти целые числа, используйте следующий код:

```
int* pNumber = new int;
// получить указатель на целое число
int* pNumbers = new int[10];
// получить указатель на блок из 10 целых чисел
```

Каждая область памяти, зарезервированная оператором new, должна быть в конечном счете **освобождена** (очищена) соответствующим оператором `delete`:

```
Тип* Указатель = new Тип;
delete Указатель;
// освобождение памяти, зарезервированной ранее для одного
экземпляра Типа
```

Это правило применимо также при запросе памяти для нескольких элементов:

```
Тип* Указатель = new Тип [КолЭлементов];
delete[] Указатель;
// освободить зарезервированный ранее блок
```

Если не освободить зарезервированную память после прекращения ее использования, то она так и останется зарезервированной для вашего приложения даже после его завершения. Это, в свою очередь, сократит объем системной памяти, доступной для использования другими приложениями, а возможно, даже замедлит выполнение вашего приложения. Это называется **утечкой памяти**, и ее следует избегать любой ценой.

В листинге 6.8 показано динамическое распределение и освобождение памяти.

Листинг 6.8.

```
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5  // Запрос области памяти для int
6  int* pAge = new int;
7  // Использование распределенной память для хранения числа
8  cout << "Enter your dog's age: ";
```

```
9   cin >> *pAge;
10  // использование оператора косвенного доступа * для обращения к значению
11  cout << "Age " << *pAge << " is stored at " << pAge << endl;
12  delete pAge; // освобождение памяти
13  return 0;
14  }
```

Результат выполнения программы:

Enter your dog's age: 9

Age 9 is stored at 0x00338120

Строка 6 демонстрирует использование оператора `new` для запроса области под целое число, в которой планируется хранить введенный пользователем возраст собаки. Обратите внимание, что оператор `new` возвращает указатель, вот почему осуществляется присвоение. Введенный пользователем возраст сохраняется в этой только что зарезервированной области памяти, а оператор `cin` обращается к ней в строке 9, используя оператор разыменования *. Строка 11 отображает значение возраста, снова используя оператор обращения к значению (*), а также отображает адрес области памяти, где оно хранится. Содержавшийся в указателе `pAge` адрес в строке 11 остается все тем же, который был возвращен оператором `new` в строке 6, — он с тех пор не изменился.

Следует заметить, что оператор `delete` не может быть вызван ни для какого адреса, содержащегося в указателе, кроме тех, и только тех, которые были возвращены оператором `new` и еще не были освобождены оператором `delete`.

Обратите внимание, что при резервировании памяти для диапазона элементов с использованием оператора `new[]` вы освобождаете ее, используя оператор `delete[]`, как показано в листинге 6.9.

Листинг 6.9.

```
1  #include <iostream>
2  #include <string>
3  using namespace std;
4  int main()
5  {
6  cout << "Enter your name: ";
7  string Name;
8  cin >> Name;
```

```
9  // Добавить 1 к резервируемому объему памяти для завершающего нулевого символа
10 int CharsToAllocate = Name.length() + 1;
11 // запрос памяти для содержания копии ввода
12 char* CopyOfName = new char [CharsToAllocate];
13 // strcpy копирует из строки с завершающим нулевым символом
14 strcpy(CopyOfName, Name.c_str());
15 // Отобразить скопированную строку
16 cout << "Dynamically allocated buffer contains: " << CopyOfName << endl;
17 delete[] CopyOfName;
18 return 0;
19 }
```

Результат выполнения программы:

Enter your name: Mingaliev

Dynamically allocated buffer contains: Mingaliev

Самыми важными являются строки 12 и 17, где используются операторы `new` и `delete` соответственно. По сравнению с листингом 6.8, где резервировалось место только для одного элемента, здесь резервируется блок памяти для нескольких элементов. Такому резервированию массива элементов должно соответствовать освобождение с использованием оператора `delete[]`, чтобы освободить память по завершении ее использования. Количество резервируемых символов вычисляется в строке 10 как на единицу большее, чем количество символов, введенных пользователем, чтобы разместить завершающий символ NULL, который важен в строках стиля C. Фактическое копирование осуществляется в строке 14 методом `strcpy`, который использует метод `c_str()` строки `Name`, предоставляемый классом `std::string`, чтобы скопировать ее в символьный буфер `CopyOfName`.

В листинге 6.10 показан результат инкремента указателей или добавления смещений к ним.

Листинг 6.10.

```
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5  cout << "How many integers you wish to enter? ";
6  int InputNums = 0;
```

```
7   cin >> InputNums;
8   int* pNumbers = new int[InputNums];
9   int* pCopy = pNumbers;
10  cout << "Successfully allocated memory for " << InputNums << " integers" << endl;
11  for(int Index = 0; Index < InputNums; ++Index)
12  {
13  cout << "Enter number " << Index << ": " ;
14  cin >> *(pNumbers + Index);
15  }
16  cout << "Displaying all numbers input: " << endl;
17  for(int Index = 0, int* pCopy = pNumbers; Index < InputNums; ++Index)
18  cout << *(pCopy++) « " ";
19  cout << endl;
20  delete[] pNumbers;
21  return 0;
22  }
```

Результат выполнения программы:

How many integers you wish to enter? 2

Successfully allocated memory for 2 integers

Enter number 0: 789

Enter number 1: 575

Displaying all numbers input:

789 575

Программа запрашивает у пользователя количество целых чисел, которые он хочет ввести в систему, прежде чем резервировать память для них в строке 8. Обратите внимание, как мы сохраняем резервную копию этого адреса в строке 9, которая используется впоследствии при освобождении этого блока памяти оператором `delete` в строке 20. Эта программа демонстрирует преимущество использования указателей и динамического распределения памяти перед статическим массивом. Когда пользователь желает хранить меньше чисел, данное приложение использует меньше памяти; когда чисел больше, он резервирует больше памяти, но никогда не растрачивает ее впустую. Благодаря динамическому распределению нет никакого верхнего предела для количества хранимых чисел, если только они полностью не исчерпают системные ресурсы. Строки 11-15 содержат цикл `for`, где пользователя просят ввести числа,

которые затем, в строке 14, сохраняют их последовательно в памяти. Именно здесь отсчитываемое от нуля значение смещения (`Index`) добавляется к указателю, заставляя вставлять введенное пользователем значение в соответствующую область памяти, не перезаписывая предыдущее значение. Другими словами, выражение (`pNumber + Index`) возвращает указатель на целое число в отсчитываемой от нуля индексной области в памяти (т.е. индекс 1 принадлежит второму числу), а, следовательно, оператор обращения к значению `*(pNumber + Index)` и является тем выражением, которое оператор `cin` использует для доступа к значению по отсчитываемому от нуля индексу. Цикл `for` в строках 17 и 18 подобным образом отображает эти значения, сохраненные предыдущим циклом. Создавая копию в указателе `pCopy` и увеличивая ее содержимое в строке 18, чтобы отобразить значение, цикл `for` использует несколько выражений инициализации.

Причина создания копии в строке 9 в том, что цикл изменяет указатель, используемый в операторе инкремента (++). Исходный указатель, возвращенный оператором `new`, должен храниться неповрежденным для использования в операторе `delete[]` (строка 20), где должен быть использован адрес, возвращенный оператором `new`, а не любое произвольное значение.

§6.5 Использование ключевого слова const с указателями

Известно, что объявление переменной как `const` фактически гарантирует, что значение переменной фиксируется после ее инициализации. Значение такой переменной не может быть изменено, и она не может использоваться как l-value.

Указатели — это тоже переменные, а, следовательно, ключевое слово `const` также вполне уместно для них. Однако указатели — это особый вид переменной, которая содержит адрес области памяти и позволяет модифицировать совокупность данных в памяти. Таким образом, когда дело доходит до указателей и констант, возможны следующие комбинации.

Первая комбинация. Данные, на которые указывает указатель, являются постоянными и не могут быть изменены, но сам адрес, содержащийся в указателе, вполне может быть изменен (т.е. указатель может указывать и на другое место):

```
int HoursInDay = 24;
```

```
const int *pInteger = &HoursInDay; // OK!
int MonthsInYear = 12;
pInteger = &MonthsInYear; // OK!
*pInteger =13; // Ошибка компиляции: нельзя изменять данные
int *pAnotherPointerToInt = pInteger; // Ошибка компиляции: нельзя
присвоить константу не константе
```

Вторая комбинация. Содержащийся в указателе адрес является постоянным и не может быть изменен, однако данные, на которые он указывает, вполне могут быть изменены:

```
int DaysInMonth = 30;
int *const pDaysInMonth = &DaysInMonth;
*pDaysInMonth = 31; // OK! Значение может быть изменено
int DaysInLunarMonth = 28;
pDaysInMonth = &DaysInLunarMonth; // Ошибка компиляции: нельзя
изменить адрес!
```

Третья комбинация. И содержащийся в указателе адрес, и значение, на которое он указывает, являются константами и не могут быть изменены (самый ограничивающий вариант):

```
int HoursInDay = 24;
// указатель может указать только на HoursInDay
const int* const pHoursInDay = &HoursInDay;
*pHoursInDay = 25; // Ошибка компиляции: нельзя изменить значение,
на которое указывает указатель
int DaysInMonth = 30;
pHoursInDay = &DaysInMonth; // Ошибка компиляции: нельзя изменить
// значение указателя
```

Эти разнообразные формы константности особенно полезны при передаче указателей функциям. Параметры функций следует объявлять, обеспечивая максимально возможный (ограничивающий) уровень константности, чтобы гарантировать невозможность функции изменить значение, на которое указывает

указатель, если это не предполагалось. Это облегчает сопровождение функций, особенно со временем и с учетом возможных изменений, внесенных в программу.

§6.6 Указатели и указатели

Указатель на указатель – это указатель, который содержит адрес другого указателя. То есть это переменная, адресом которого является адрес другого указателя.

Обычный указатель типа int объявляется с использованием одной звёздочки:

```
int *ptr; // указатель типа int, одна звёздочка
```

Указатель на указатель типа int объявляется с использованием двух звёздочек:

```
int **ptrptr; // указатель на указатель типа int
```

Указатель на указатель работает подобно обычному указателю: вы можете его разыменовать для получения значения, на которое он указывает. И, поскольку этим значением является другой указатель, для получения исходного значения вам потребуется выполнить разыменование ещё раз. Их следует выполнять последовательно:

Листинг 6.11.

```
1  #include <iostream>
2  using namespace std;
3  int main()
4  {
5  int value = 7;
6  int *ptr = &value;
7  cout << *ptr << endl; // разыменовываем указатель, чтобы получить значение типа int
8  int **ptrptr = &ptr;
9  cout << **ptrptr << endl;
10 return 0;
11 }
```

Результат выполнения программы:

```
7
7
```

Обратите внимание, вы не можете инициализировать указатель на указатель напрямую значением.

```
int value = 7
int **ptrptr = &&value // нельзя
```

Это связано с тем, что оператор адреса (&) требует l-value, но `&value` — это r-value.

Однако указателю на указатель можно задать значение `null`:

```
int **ptrptr = NULL;
```

Указатели на указатели имеют несколько применений. Наиболее используемым является динамическое выделение **массива указателей**:

```
int **array = new int*[20]
// выделяем массив из 20 указателей типа int
```

Это тот же обычный динамически выделенный массив, за исключением того, что элементами являются указатели на тип `int`, а не значения типа `int`.

Динамическое выделение двумерного массива немного отличается. Сначала мы выделяем массив указателей (как в примере выше), а затем перебираем каждый элемент массива указателей и выделяем динамический массив для каждого элемента этого массива. Итого, наш динамический двумерный массив — это динамический одномерный массив динамических одномерных массивов.

```
int **array = new int*[15]
// выделяем массив из 15 указателей типа int – это наши строки
for (int count = 0; count < 15; ++count)
array[count] = new int[7] // а это наши столбцы
```

Доступ к элементам массива выполняется как обычно:

```
array[8][3] = 4 // это то же самое, что и (array[8])[3] = 4
```

Этим методом, поскольку каждый столбец массива динамически выделяется независимо, можно сделать динамически выделенные двумерные массивы, которые не являются прямоугольными. Например, мы можем создать массив треугольной формы:

```
int **array = new int*[15]; // выделяем массив из 15 указателей
типа int – это наши строки
for (int count = 0; count < 15; ++count)
array[count] = new int[count+1]; // а это наши столбцы
```

В примере, приведенном выше, array[0] — это массив длиной 1, а array[1] — массив длиной 2 и т.д.

Для освобождения памяти динамически выделенного двумерного массива (который создавался с помощью этого способа) также потребуется цикл:

```
for (int count = 0; count < 15; ++count)
    delete[] array[count];
delete[] array; // это следует выполнять в конце
```

Обратите внимание, мы удаляем массив в порядке, противоположном его созданию. Если мы удалим массив перед удалением элементов массива, то нам придется получать доступ к освобожденной памяти для удаления элементов массива. А это, в свою очередь, приведет к неожиданным результатам.

Поскольку процесс выделения и освобождения двумерных массивов является несколько запутанным (можно легко наделать ошибок), то часто проще «сплющить» двумерный массив в одномерный массив:

```
// Вместо следующего:
int **array = new int*[15]; // выделяем массив из 15 указателей
    типа int – это наши строки
for (int count = 0; count < 15; ++count)
    array[count] = new int[7]; // а это наши столбцы
// Делаем следующее:
int *array = new int[105]; // двумерный массив 15x7 "сплющенный" в
    одномерный массив
```

Простая математика используется для конвертации индексов строки и столбца прямоугольного двумерного массива в один индекс одномерного массива:

```
int getSingleIndex(int row, int col, int numberOfColumnsInArray)
    { return (row * numberOfColumnsInArray) + col; }
// Присваиваем array[9,4] значение 3, используя наш "сплющенный"
    массив
array[getSingleIndex(9, 4, 5)] = 3;
```

Также можно объявить указатель на указатель на указатель:

```
int ***ptrx3;
```

Или сделать еще большую вложенность, если захотите. Однако на практике такие указатели редко используются.

§6.7 Передача указателей в функции

Указатели — это эффективный способ передачи функции областей памяти, содержащих значения и способных содержать результат. При использовании указателей с функциями важно гарантировать, что вызываемой функции позволено изменять только те параметры, которые вы хотите изменить, но не другие. Например, функции, вычисляющей площадь круга по радиусу, переданному как указатель, нельзя позволить изменять радиус. Вот где пригодятся константные указатели, позволяющие эффективно контролировать то, что функции позволено изменять, а что — нет.

В листинге 6.12 представлен пример использования ключевого слова `const` при вычислении площади круга, когда радиус и число Пи передаются как указатели.

Листинг 6.12.

```
1  #include <iostream>
2  using namespace std;
3  void CalcArea(const double* const pPi, const double* const pRadius, double* const pArea)
4  {
5  if (pPi && pRadius && pArea)
6  *pArea = (*pPi) * (*pRadius) * (*pRadius);
7  }
8  int main()
9  {
10 const double Pi = 3.1416;
11 cout << "Enter radius of circle: ";
12 double Radius = 0;
13 cin >> Radius;
14 double Area = 0;
15 CalcArea (&Pi, &Radius, &Area);
16 cout << "Area is = " << Area << endl;
17 return 0;
18 }
```

Строка 3 демонстрирует две формы константности, где оба указателя, `pRadius` и `pPi`, передаются как “константные указатели на константные данные”, так что ни адрес указателе, ни данные, на которые он указывает, не могут быть изменены. Указатель

pArea, вполне очевидно, — это параметр, предназначенный для хранения вывода. Значение указателя (адрес) не может быть изменено, но данные, на которые он указывает, могут. Строка 5 демонстрирует проверку на допустимость переданных функции указателей перед их использованием. Было бы нежелательно, чтобы функция вычисляла площадь, если вызывающая сторона по неосторожности передаст пустой указатель как любой из этих трех параметров, поскольку это привело бы к аварийному отказу приложения.

§6.8 Ссылочная переменная

Ссылка (reference) — это псевдоним переменной. При объявлении ссылки ее необходимо инициализировать переменной. Таким образом, ссылочная переменная — это только другой способ доступа к данным, хранимым в переменной.

Для объявления ссылки используется оператор ссылки (&), как в следующем примере:

```
VarType Original = Value;
VarType& ReferenceVariable = Original;
```

Ссылки позволяют работать с областью памяти, которой они инициализируются. Это делает ссылки особенно полезными при создании функций.

Ссылки позволяют работать с областью памяти, которой они инициализируются. Это делает ссылки особенно полезными при создании функций. Типичная функция объявляется так:

```
ТипВозвращаемогоЗначения СделатьНечто (Тип Параметр);
```

Функция СделатьНечто () вызывается так:

```
ТипВозвращаемогоЗначения Результат = СделатьНечто (аргумент); //
вызов функции
```

Приведенный выше код привел бы к копированию аргумента в параметр, который затем использовался бы функцией СделатьНечто(). Этап копирования может быть весьма продолжительным, если рассматриваемый аргумент занимает много памяти. Аналогично, когда функция СделатьНечто() возвращает значение, оно копируется в Результат. Было бы хорошо, если бы удалось избежать или устранить этапы копирования, позволив функции воздействовать непосредственно на данные в стеке вызывающей стороны. Ссылки позволяют сделать именно это.

Версия функции без этапа копирования выглядит следующим образом:

```
ТипВозвращаемогоЗначения СделатьНечто (Тип& Параметр);
// обратите внимание на ссылку &
```

Эта функция была бы вызвана так:

```
ТипВозвращаемогоЗначения Результат = СделатьНечто(аргумент);
```

Поскольку аргумент передается по ссылке, параметр не будет копией аргумента, а скорее псевдонимом последнего. Кроме того, функция, которая получает параметр как ссылку, может возвращать значение, используя ссылочные параметры. Рассмотрим листинг 6.13, чтобы понять, как функции могут использовать ссылки вместо возвращаемых значений.

Листинг 6.13.

```
1  #include <iostream>
2  using namespace std;
3  void ReturnSquare(int& Number)
4  {
5  Number *= Number;
6  }
7  int main ()
8  {
9  cout << "Enter a number you wish to square: ";
10 int Number = 0;
11 cin >> Number;
12 ReturnSquare(Number);
13 cout << "Square is: " << Number << endl;
14 return 0;
15 }
```

Результат выполнения программы:

Enter a number you wish to square: 5

Square is: 25

Функция, вычисляющая квадрат числа, находится на строках 3-6. Обратите внимание на то, что она получает возводимое в квадрат число по ссылке и возвращает результат таким же образом. Если бы вы забыли отметить параметр `Number` как ссылку (&), то результат не достигнет вызывающей функции `main()`, поскольку функция `ReturnSquare()` выполнит свои действия с локальной копией переменной `Number`, которая будет удалена при выходе из функции. Используя ссылки, вы позволяете

функции ReturnSquare() работать в том же пространстве адресов, где определена переменная Number в функции main(). Таким образом, результат доступен в функции main() даже после того, как функция ReturnSquare() закончила работу.

Возможны ситуации, когда ссылке не позволено изменять значение исходной переменной. При объявлении таких ссылок используют ключевое слово const:

```
int Original = 30;
const int& ConstRef = Original;
ConstRef = 40; // Недопустимо: ConstRef не может изменить значение в Original
int& Ref2 = ConstRef; // Недопустимо: Ref2 не const
const int& ConstRef2 = ConstRef; // OK
```

При передаче аргументов по ссылке всегда используйте константные ссылки, если вам не нужно, чтобы функция изменяла значения аргументов.

Иногда нам может понадобиться, чтобы функция возвращала сразу несколько значений. Однако оператор return позволяет функции иметь только одно возвращаемое значение. Одним из способов возврата сразу нескольких значений является использование ссылок в качестве параметров:

Листинг 6.14.

```
1  #include <iostream>
2  #include <math.h> // для sin() и cos()
3  using namespace std;
4  void getSinCos(double degrees, double &sinOut, double &cosOut)
5  {
6      const double pi = 3.14159265358979323846; // значение Пи
7      double radians = degrees * pi / 180.0;
8      sinOut = sin(radians);
9      cosOut = cos(radians);
10 }
11 int main()
12 {
13     double sin(0.0);
14     double cos(0.0);
15     getSinCos(30.0, sin, cos);
16     cout << "The sin is " << sin << '\n';
17     cout << "The cos is " << cos << '\n';
18     return 0;
```

```
}
```

Эта функция принимает один параметр (передача по значению) в качестве входных данных и «возвращает» два параметра (передача по ссылке) в качестве выходных данных. Параметры, которые используются только для возврата значений обратно в caller, называются параметрами вывода. Они дают понять caller-у, что значения исходных переменных, переданных в функцию, не столь значительны, так как мы ожидаем, что эти переменные будут перезаписаны.

Давайте рассмотрим это детально. Во-первых, в функции `main()` мы создаем локальные переменные sin и cos. Они передаются в функцию `getSinCos()` по ссылке (а не по значению). Это означает, что функция `getSinCos()` имеет прямой доступ к исходным значениям переменных `sin` и `cos`, а не к их копиям. Функция `getSinCos()`, соответственно, присваивает новые значения переменным `sin` и `cos` (через ссылки `sinOut` и `cosOut`), перезаписывая их старые значения. Затем `main()` выводит эти обновленные значения.

Если бы `sin` и `cos` были переданы по значению, а не по ссылке, то функция `getSinCos()` изменила бы копии `sin` и `cos`, а не исходные значения и эти изменения уничтожились бы в конце функции — переменные вышли бы из локальной области видимости. Но, поскольку `sin` и `cos` передавались по ссылке, любые изменения, внесенные в `sin` или `cos` (через ссылки), сохраняются и за пределами функции `getSinCos()`. Таким образом, мы можем использовать этот механизм для возврата сразу нескольких значений обратно в caller.

Неконстантные ссылки могут ссылаться только на неконстантные l-values (например, на неконстантные переменные), поэтому параметр-ссылка не может принять аргумент, который является константным l-value или r-value (например, литералом или результатом выражения).