# 1 Указатели

Начнем со следующей задачи. Предположим, в программе имеются 3 целые переменные x, y, z. Задача состоит в том, чтобы в одном месте программы выбрать одну из них, а в другом месте — присвоить выбранной переменной значение 1.

Самое простое решение этой задачи таково. Мы заводим еще одну целую переменную, например t, и договариваемся о том, что она может иметь значения от 1 до 3, причем ее значению 1 соответствует выбор x, значению 2 — y и 3 — z. В том месте программы, где нужно выбирать одну из трех переменных, мы присваиваем t соответствующее значение; например, выбор переменной y будет выглядеть как «t=2;». Ну а в том месте, где выбранной переменной нужно присвоить 1, будет стоять следующий фрагмент:

```
switch(t)
{
case 1: x:=1; break;
case 2: y:=1; break;
case 3: z:=1;
}
```

Это решение при всей своей простоте имеет ряд недостатков. Отметим два из них. Первый состоит в том, что указанная конструкция усложняется с ростом числа переменных, из которых мы хотим выбрать одну. Второй связан с тем, что нам нужно помнить, какое значение специальной переменной какую переменную означает.

Для устранения этих недостатков и было изобретено понятие указателя. По-прежнему нужна дополнительная переменная, но теперь она имеет не целый тип, а некоторый специальный тип, который называется «тип указателя на целую переменную» (если мы хотим выбирать одну из целых переменных; если же нам нужно выбрать одну из вещественных переменных, нужен уже другой тип — тип указателя на вещественную переменную). Переменная-указатель объявляется почти так же, как и обычная переменная, но с добавлением символа «*» перед именем переменной. Итак, специальная переменная (называемая указателем) заводится так:

```
int *t;
```

В этом определении * относится к p, а не к int. Это означает, что если я хочу в одной декларации определить несколько указателей, * должна ставиться перед каждым именем:

```
int *p, *q;
```

Как и любую переменную, указатель можно инициализировать при объявлении.

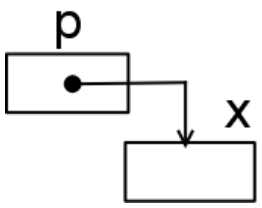
Теперь, чтобы выбрать одну из целых переменных (а этот способ позволяет выбирать любую из целых переменных, определенных в программе), например x, нужно написать:

```
t = &x;
```

В том же месте, где нужно присвоить выбранной переменной некоторое значение, например, 1, нужно написать:

```
*t = 1;
```

В дальнейшем, если переменная-указатель выбирает некоторую другую переменную, мы будем говорить, что указатель указывает на выбираемую переменную. Тот факт, что указатель p указывает на переменную x, обычно изображается следующим образом:




В C++ также имеется некоторый специальный тип указателя «неизвестно на что». Такие указатели определяются с ключевым словом void:

```
void *p;
```

Такому указателю можно присвоить любой указатель (т. е. указатель на любой тип), но пользоваться им непосредственно нельзя — сначала нужно явно привести его к типу указателя на конкретный тип, написав перед ним (тип *) (см. операцию преобразования типов).

## 1.1 Реализация указателя.

Теперь скажем пару слов о реализации понятия указателя. Как известно, память компьютера состоит из пронумерованных от 0 и далее ячеек, каждая из которых может хранить небольшое целое число (более конкретно, от 0 до 255). Номер каждой такой ячейки называется ее адресом, он представляет собой обычное целое число. Так вот, в переменной типа указателя (независимо от того, на переменные какого типа он указывает) хранится просто адрес первой ячейки, которую занимает та переменная, на которую он указывает. Если целая переменная x занимает ячейки с адресами от 300 до 303, то после присваивания t = &x в переменной t окажется число 300.

Однако, компилятор различает типы указателей по типу переменных, на которые они указывают; в частности, int* и double* (т. е. указатели на целые и указатели на вещественные переменные) считаются разными типами, и нельзя переменной одного типа указателя присвоить переменную другого типа указателя.

Первая из рассмотренных в предыдущем разделе операций & называется операцией взятия адреса. Ее можно применять только к таким выражениям, которые могут стоять в левой части оператора присваивания, например, именам переменных или элементам массивов. Ее результатом будет указатель соответствующего типа, выбирающий (или указывающий на) ту переменную, к которой применяется эта операция.

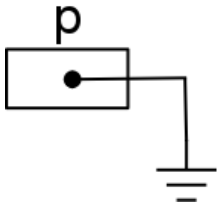
Вторая операция, *, называется операцией разыменования. Она может применяться только к указателям и дает ту переменную, на которую этот указатель указывает. Результат этой операции может встречаться как в левой части оператора присваивания, что означает присваивание выбранной указателем переменной, так и в правой, что означает использование значения указываемой переменной.

Таким образом, если p — указатель на целую переменную, то *p обозначает ту переменную, на которую указывает p. К нулевому указателю эту операцию применять нельзя, однако эту ошибку замечает обычно не компилятор, а операционная система во время выполнения программы. К указателю неизвестно на что эту операцию тоже применять нельзя; если известно, на что на самом деле он указывает, нужно воспользоваться операцией преобразования типов для превращения его в обычный указатель. Например, если указатель p определен при помощи void *p; но в определенном месте программы известно, что он указывает на целое число, до этого целого числа можно добраться, написав *(int *)p. При этом, разумеется, компилятор никак не может проверить, что этот указатель действительно указывает на переменную типа целое число, а не вещественное; он не может проверить даже того, что этот указатель вообще указывает на какую-либо переменную, а не на пустую или даже недоступную программе область памяти.

Операцию взятия адреса можно применять обычно к так называемым l-величинам, т. е. к тем выражениям, которые могут стоять в левой части операции присваивания. Примерами таких выражений могут служить переменные, элементы массивов, поля структурных переменных. Исключением являются битовые поля — что это такое, будет объясняться позже. Имеются из этого правила исключения и в другую сторону: операцию взятия адреса можно применять к поименованным константам.

## 1.2 Указатель ни на что.

Оказалось очень удобным иметь некоторое специальное значение указателя, которое говорит о том, что указатель не указывает вообще ни на какую переменную. Такое значение указателя обозначается ключевым словом nullptr (или, в старом стандарте, просто целым числом 0) и может быть присвоено указателю любого типа.Тот факт, что указатель p не указывает ни на что, обычно изображается следующим образом:

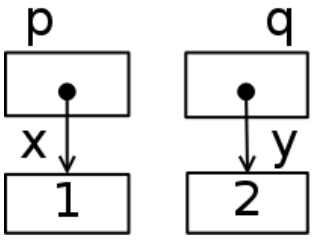



## 1.3 Присваивание указателей.

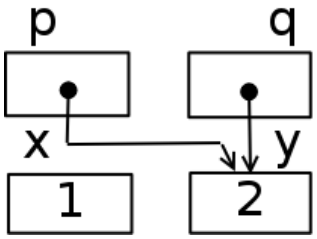
Указатели, как и переменные других типов, можно присваивать друг другу, передавать в процедуры и функции в качестве параметров, возвращать в качестве результатов и сравнивать на равенство или отсутствие равенства. Равными считаются указатели, выбирающие одну и ту же переменную, т. е. такие, в которых хранится один и тот же адрес.

Присваивание указателей и присваивание переменных, на которые они указывают — совсем не одно и то же.

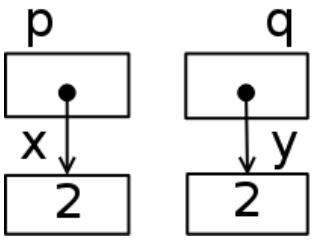
Пусть у нас есть 2 указателя p и q, каждый из которых указывает на свою переменную (предположим, что эти переменные, а следовательно, и указатели на них имеют одинаковый тип), эта ситуация изображена на следующем рисунке:




После «p=q;» оба указателя начнут указывать на одну и ту же переменную (на которую раньше указывал только q); при этом значения обоих переменных не изменятся (см. следующий рисунок).




Если же выполняется оператор «*p=*q;», то содержимое указателей не меняется, они по-прежнему указывают на те же переменные, что и раньше, но содержимое той переменной, на которую указывает q, записывается в ту переменную, на которую указывает p (см. следующий рисунок).




## 1.4 Висячие указатели.

Уместно в связи с указателями сделать одно предупреждение. Если в программе имеется функция, возвращающая указатель, то нельзя из этой функции возвращать указатель на локальную переменную. Это связано с тем, что после выхода из функции все ее локальные переменные исчезают, и память, которую они занимали, освобождается. Таким образом, возвращенный из функции указатель будет указывать на освобожденную память; такие указатели называются висячими. Использование висячих указателей приводит к неверным результатам работы программы или к ее аварийной остановке.

## 1.5 Динамические переменные и массивы.

Обычные локальные переменные возникают в момент начала работы подпрограммы, живут пока она работает и исчезают по окончании ее работы. Однако иногда требуется в подпрограмме завести переменную, которая сохранилась бы и при выходе из подпрограммы. Если число таких переменных известно заранее (в момент компиляции программы), можно воспользоваться статическими переменными. Часто это не так, и в этом случае можно воспользоваться механизмом динамических переменных.

Динамическая переменная хранится в особой области памяти, называемой кучей. Особенность этой области памяти в том, что, в отличие от стека, где хранятся обычные локальные переменные и где уничтожать их можно только в порядке, обратном порядку их создания, в куче можно уничтожать переменные и освобождать занимаемую ими память в любом порядке. Платой за такое удобство является то, что выделение и освобождение памяти в куче происходит намного медленнее, чем в стеке.

Для того, чтобы завести динамическую переменную, нужно написать new тип Этот оператор заводит динамическую переменную указанного типа и возвращает указатель на нее. Если p — переменная типа int*, то после выполнения оператора p = new int; указатель p будет указывать на вновь созданную динамическую переменную целого типа.

Динамические переменные не имеют имен, и доступ к ним возможен только через указатели на них. Если в результате ошибок в программе возникают динамические переменные, на которые никакой указатель не указывает (т. е. до которых невозможно добраться), это явление называется утечкой памяти, а сами недоступные динамические переменные называются мусором.

Еще одно отличие динамических переменных от обычных локальных состоит в том, что если они перестали использоваться, их надо удалять вручную вызовом операции delete, параметром которой является указатель на удаляемую переменную.

В связи с динамическими переменными, хотелось бы сказать пару слов и о динамических массивах. Если нужно завести динамический массив, это делается так:

```
указатель_на_элемент = new тип_элемента[число_элементов];
```

Такой фрагмент присвоит указателю адрес первого элемента массива. Удаляются динамические массивы так:

```
delete [] указатель_на_первый_элемент_удаляемого_массива;
```

Динамический массив хорош тем, что число его элементов может быть любым выражением, возвращающим целое число. Здесь нет того ограничения, как для обычных массивов, что размер массива должен быть известен на момент компиляции.

Здесь надо еще заметить, что в C++ нельзя менять размер массива на ходу. Если нам нужно расширить наш динамический массив, нужно явно завести массив большего размера, переписать в него элементы старого массива и наконец так же явно уничтожить старый массив. Правда, в стандартной библиотеке C++ имеется шаблонный класс vector (будет обсуждаться позже), который позволяет менять размер, но внутри него это реализовано так же, как обсуждалось выше. Это, в частности, означает, что если у Вас был указатель на какой-то элемент, после добавления в vector новых элементов этот указатель может перестать указывать на правильный элемент (т. е. он по-прежнему будет указывать на элемент старого массива, который будет уничтожен в связи с переездом массива на другое место в памяти).

## 1.6 Синонимы типов: операция typedef

Ключевое слово typedef может применяться и к декларациям указателей. Например, допустим, мы хотим создать синоним ULIPTR для типа unsigned long int *. Для этого сначала объявим переменную такого типа с именем ULIPTR:

```
unsigned long int *ULIPTR;
```

Далее, чтобы превратить объявление переменной в объявление синонима ее типа, добавим перед этой декларацией ключевое слово typedef:

```
typedef unsigned long int *ULIPTR;
```

После такой декларации мы можем использовать ULIPTR для определения переменных, массивов, указателей и т. д., так же, как и стандартные типы вроде int или double. Например, что реально определяет следующая декларация:

```
ULIPTR *pptr;
```

При этом конструкция typedef не создает новых типов, а именно определяет их синонимы. Реально это означает, что разные переменные, определенные с помощью разных синонимов одного и того же типа, считаются имеющими один и тот же тип; например, их запросто можно присвоить одну другой.

## 1.7 Передача и возврат указателей из функций

Как и любую другую переменную, указатель можно передать в качестве параметра; его можно также вернуть из функции в качестве результата. Причем и то, и другое может быть как по значению, так и по ссылке. Передача указателя в качестве параметра по значению имеет тот же синтаксис, что и параметра любого другого типа: просто в списке параметров ставится соответствующая декларация. Если нужно передать указатель по ссылке, как и в случае с другими типами, нужно добавить в декларацию формального параметра символ & непосредственно перед именем параметра. Например, следующая функция получает по ссылке параметр p типа указатель на целое число:

```
void f(int *&p);
```

Если функция возвращает указатель по значению, перед ее именем ставится *:

```
int *f();
```

А если по ссылке, то еще и &:

```
int *&f();
```

**Задача.**

Написать функцию, принимающую 4 указателя на вещественные переменные и заменяющую значение второй из них суммой значений первых двух, значение третьей — суммой значений (до преобразования) первых трех, значение последней — суммой значений всех четырех.

## 1.8 Арифметика указателей

В C++ разрешены некоторые арифметические операции с указателями. Однако, все такие операции неявно предполагают, что все участвующие в них указатели (как исходные и промежуточные результаты, так и окончательные) указывают на элементы одного и того же массива. Однако, компилятор никак не проверяет истинность этого предположения и соблюдение его всецело на совести программиста.

Если указатель p указывает на элемент массива с индексом i, то p+j, где j — целое число, будет указывать на элемент того же массива с индексом i+j. При этом совершенно неважно, положительно j или отрицательно, лишь бы окончательный индекс лежал в допустимых пределах; аналогичным образом, p-j указывает на элемент с индексом i-j. Также, если указатели p и q указывают на элементы с индексами i и j, то выражение p-q имеет целый результат и равно i-j (опять же неважно, положительно это значение, равно нулю или отрицательно).

Этим, однако, арифметика указателей и ограничивается. Несмотря на то, что указатели, с точки зрения внутреннего устройства, это просто адреса в памяти (целые числа), тем не менее их нельзя складывать или умножать друг на друга или даже на целые числа. Если такая нужда возникает (и Вы знаете, что делаете), можно явно превратить указатель в целое число при помощи операции преобразования типов, написав перед ним (int). Затем можно проделать необходимые манипуляции с полученным целым числом, и наконец, снова превратить теперь уже целое число обратно в указатель при помощи той же операции преобразования типов, написав перед ним (тип *), где тип — тип той переменной, в указатель на которую Вы хотите превратить полученное целое число.

Заметим здесь еще следующее. Во-первых, в большинстве выражений, в которых участвует имя массива, оно понимается как указатель на первый элемент этого массива (с индексом 0). Во-вторых, можно индексировать указатели так же, как и массивы: запись p[i] означает то же самое, что и *(p+i). Однако, это работает только для одномерных массивов: имя двумерного массива понимается как указатель на первую строку (одномерный массив), и не приводится к типу «указатель на указатель». Также, из этого правила имеются очевидные исключения: в выражении sizeof M, если M — массив (неважно, одномерный или многомерный), вычисляется размер массива M, а не указателя.

Напоследок скажем только, что арифметика указателей не работает для указателей неизвестно на что (типа void *).

**Задача.**

Написать функцию, принимающую массив целых чисел, его длину и указатель на какой-то элемент этого массива. Под шагом понимается сдвиг указателя на число элементов, равное значению указываемого элемента массива. Функция должна возвращать указатель на элемент массива-параметра, полученный при помощи 10 шагов из исходного указателя; при этом, если на каком-то шаге результат выходит за пределы массива, нужно возвращать последнее значение указателя, еще указывающее в пределах массива-параметра.

## 1.9 Ввод-вывод указателей

Ввести с клавиатуры указатель в C++ нельзя, но вывести на экран можно. При этом выводится содержащийся в нем адрес в шестнадцатеричной системе счисления, состоящий всегда из восьми цифр (возможно, с ведущими нулями).

Так ведут себя все указатели, кроме указателей на символ типа char *. Почему эти указатели выводятся по-другому, будет сказано позже. Если, однако, нам нужно вывести такой указатель именно как указатель, т. е. адрес в шестнадцатеричной системе счисления, то нужно явно привести его к типу void *, написав перед ним (void *).

**Задача.**

Объявить последовательно переменные типов int, char, bool, short, long, float, double, long double, int. Вывести их адреса; сравнить разности адресов с размерами переменных, выдаваемыми операцией sizeof.

## 1.10 Преобразование типов указателей

Мы уже говорили о том, что можно преобразовать тип указателя в целое число и обратно. Оказывается, принудительно можно преобразовать любой тип указателя в любой другой. Это приведет к тому, что содержимое памяти по этому адресу начнет трактоваться по-другому. Опреация преобразования типов указателей никак не меняет содержимого памяти по данному адресу; она лишь меняет способ трактовки этого содержимого, который тоже может быть важен. Например, и число типа int, и число типа float оба занимают по 4 байта. Но способ представления конкретных чисел, например 5 для int и 5.0 для float, существенно отличается один от другого. Поэтому следующий фрагмент выдаст совсем не 5.0, а нечто другое (что именно?):

```
int x = 5;
cout<<*(float*)&x<<endl;
```

Однако, такое преобразование переменных из одного типа в другой, сохраняющее не значение переменной, а содержимое памяти по данному адресу, иногда бывает важно. Например, таким образом можно превратить какое-либо значение в набор байтов, чтобы потом обрабатывать эти байты по-отдельности, скажем, для записи значения в файл в двоичном виде.

Кроме обсуждаемого выше явного преобразования типов указателей имеется и неявное, которое происходит автоматически, без всяких усилий программиста. Это имеет место в следующих случаях:

1) Целое значение 0 (явно выписанное, и только оно) преобразуется в указатель любого типа, и означает указатель, никуда не указывающий.

2) Любой указатель может быть преобразован в тип void *, т. е. любой указатель можно присвоить переменной типа void * (обратное преобразование возможно только явно).

3) Любой указатель на один тип может быть преобразован в указатель на другой тип, который является синонимом первого, созданным при помощи конструкции typedef.

4) Наконец, при наследовании указатель на объект производного класса может быть преобразован в указатель на объект базового класса (обратное преобразование, как и в п. 2, возможно только явно).

**Задачи.**

1. Процессор называется Big Endian, если целые числа, занимающие несколько байтов, хранятся по следующему правилу: младший байт записан по наибольшему адресу, и Little Endian, если младший байт хранится по наименьшему адресу; если же порядок записи байтов в составе целого числа не совпадает ни с одним из упомянутых, назовем это Mixed Endian. Определить тип процессора, на котором мы работаем.

2. Написать макроопределение, выдающее значение последнего байта той переменной, на которую указывает параметр (это, естественно, указатель — больше о нем ничего не известно).

## 1.11 Использование ключевого слова const с указателями

Как и при определении любой переменной, мы можем пользоваться ключевым словом const для указания того, что данная переменная не может меняться после ее определения. Однако, с указателем связаны две переменные: одна — сам указатель, и другая — та, на которую он должен указывать.

Оказывается, в C++ можно раздельно управлять константностью каждой из этих переменных.

Если мы хотим, чтобы константным был сам указатель, т. е. чтобы самому указателю после определения нельзя было ничего присвоить, мы должны поставить ключевое слово const непосредственно перед именем указателя. Как и для любых константных объектов, такой указатель обязательно должен быть инициализирован:

```
int x;
int * const p = &x;
```

Однако, это никак не влияет на константность той переменной, на которую указывает p, и мы запросто можем написать *p = 2; (тогда 2 попадет в переменную x).

Независимо от константности самого указателя мы можем потребовать константность той переменной, на которую он должен указывать. Для этого ключевое слово const добавляется перед декларацией:

```
const int *p;
```

На сей раз мы имеем право менять сам указатель, например p=0; но не имеем права менять ту переменную, на которую он указывает (выражение *p=2 запрещено). При этом важно помнить два момента:

1) Во-первых, такая константность означает только, что переменная, на которую указывает такой указатель, не может меняться через этот указатель; она вовсе не означает отсутствие других способов менять эту переменную. Например, если у нас имеются декларации

```
int x;
const int *p = &x;
```

то выражение *p=1 является противозаконным, а x=1, означающее по смыслу то же самое, — нет. Все дело в том, меняется ли переменная через этот указатель.

2) Во-вторых, если у нас имеется поименованная константа, то на нее в принципе может указывать только такой указатель. В противном случае, если на константу будет указывать обычный указатель, у нас появится возможность, по крайней мере теоретически, изменить значение этой константы через этот указатель, что запрещено. Это компилятор проверяет, и в случае обнаружения такой ошибки выдает соответствующее сообщение. Причем, если даже такая ошибка встречается в составном операторе, стоящем под if, компилятор все равно выдаст ошибку, независимо от того, выполняется ли условие этого if хотя бы когда-нибудь или нет. Это происходит потому, что ответить на такого рода вопросы (выполняется ли когда-нибудь конкретный фрагмент кода или нет) компилятор не может в принципе — соответствующая задача, как принято говорить, алгоритмически неразрешима, т. е. нет алгоритма, который бы отвечал на такой вопрос, глядя на текст программы. Таким образом, следующий набор деклараций ошибочен:

```
const int x = 57;
int *p = &x;
```

Напомним еще раз, что оба вышеизложенных способа указать константное поведение указателя совершенно независимы один от другого, так что всего существует 4 способа использования ключевого слова const при определении указателя.

Напоследок скажем здесь о так называемом расконстанчивании. Иногда у нас может иметься указатель на константу, и нам очень хочется сделать из него указатель на обычный (неконстантный) объект. Для этого существует

специальное преобразование типов указателей такого вида: const_cast<тип обычного указателя>(указатель на константу).

**Задача.**

1. Объявить константный указатель на константу типа float.
2. Объявить тип указателя на long int.
3. Объявить указатель на double.
4. Объявить указатель на константу типа short int.
5. Объявить тип константного указателя на константу типа float.
6. Объявить тип указателя на константу типа char.
7. Объявить константный указатель на double.
8. Объявить тип константного указателя на unsigned int.

## 1.12 Массивы указателей и указатели на массивы

В языке C++ можно построить массив из элементов любого типа, в том числе и указателей. Массив из 20 указателей на целое число можно определить так:

```
int *p[20];
```

Здесь уместно сказать о том, как вообще устроены декларации в C++. Основной принцип здесь такой: переменная описывается так же, как используется. Это означает, что последнюю декларацию надо читать так: p — это нечто, к чему можно применить операцию индексации (квадратные скобки, их приоритет выше, чем у *), затем операцию разыменования *, и в итоге получится int. Отсюда, очевидно, следует, что p — массив указателей на int.

С подобными вложенными декларациями мы уже встречались, когда говорили о функциях, возвращающих указатель:

```
int *f();
```

Эту декларацию надо на самом деле читать так: если к f применить операцию вызова функции (круглые скобки, их приоритет тоже выше, чем у *), затем операцию разыменования *, то в итоге получится int. Отсюда, очевидно, следует, что f — функция, возвращающая указатель на int.

Если у нас есть декларация переменной, то тип этой переменной получается выкидыванием ее имени из декларации. Например, тип переменной-указателя на целое число будет записываться как int*. Мы уже встречались с такой записью в связи с использованием операции преобразования типов. Тип массива из 20 указателей на int будет записываться так: int *[20].

Указатель на массив встречается гораздо реже, чем массив указателей, однако его тоже можно определить в программе (в основном, для возврата из функции многомерных массивов). Для этого нужно всего лишь изменить порядок операций индексации и разыменования в декларации, для чего поставить скобки:

```
int (*p)[20];
```

Тип такой переменной будет уже int (*)[20], и скобки здесь очень существенны.

Можно также определить тип указателя на указатель, поставив две *:

```
int **p;
```

Указатели на указатели используются для организации динамических двумерных массивов (будет чуть позже), а также для работы с разными ссылочными структурами данных, такими как спсики и деревья (использование указателей на указатели позволяет убрать разбор большого числа случаев, за счет чего текст программы сокращается).

## 1.13 Двумерные динамические массивы

Иногда возникает потребность иметь двумерный массив, оба размера которого определяются только на этапе выполнения программы. К решению этой задачи имеются два разных подхода.

Первый подход состоит в том, чтобы завести одномерный массив по количеству необходимых элементов, а затем пересчитывать пары индексов в двумерном массиве в один индекс в одномерном. Если нам нужно завести таким образом двумерный массив m×n, где m и n — переменные, сначала мы пишем int *p = new int[m*n]; а для доступа к элементу i-й строки и j-го столбца надо написать p[i*n+j]. Когда массив больше не нужен, необходимо освободить занимаемую им память, написав delete [] p;

Если такой синтаксис доступа к массиву нас не устраивает, есть и второй подход к решению этой задачи, при котором доступ осуществляется с использованием традиционного синтаксиса. Однако, этот способ гораздо сложнее. Пусть по-прежнему надо завести массив m×n, где m и n — переменные.

Тогда в рамках второго подхода нужно сначала завести динамический массив указателей на строки:

```
int **p = new int *[m];
```

Далее, нужно завести сами строки:

```
for(int k = 0; k < m; k++)
    p[k] = new int[n];
```

Доступ к такому массиву, как уже говорилось, будет иметь традиционный вид: p[i][j]. Для уничтожения такого массива нужно проделать следующее:

```
for(int k = 0; k < m; k++)
    delete [] p[k];
delete [] p;
```

Этот подход позволяет иметь даже строки разной длины, что удобно для хранения так называемых разреженных матриц (т. е. таких, среди элементов которых много нулей), например, треугольных матриц.

**Задача.**

Написать функцию, заводящую верхне-треугольную матрицу заданного порядка (он определяется во время выполнения программы; хранить в памяти, естественно, нужно только возможно ненулевые элементы, заведомые нули ниже главной диагонали хранить незачем) в соответствии со вторым подходом к двумерным динамическим массивам (возвращает int**). Написать также функцию, выдающую ссылку на заданный элемент этой матрицы по указателю на такую матрицу (типа int**) и номерам строки и столбца (номера в исходной квадратной матрице; если соответствующий элемент лежит не ниже главной диагонали, нужно выдать ссылку на ту переменную в составе двумерного динамического массива, в которой этот элемент хранится, иначе выдается ссылка на локальную статическую переменную, которая обнуляется при каждом вызове этой функции).

## 1.14 Указатели на функции

В C++ можно иметь переменные, значениями которых будут функции. Типы таких переменных называются указателями на функции.

Указатель на функцию заводится следующим образом:

```
void (*p)();
```

Разумеется, тип результата этой функции и список параметров могут быть любыми (для примера я взял простейший случай).

Для работы с этим указателем ему нужно присвоить какое-нибудь значение. Если у нас есть функция с тем же типом результата и набором типов параметров, что и указатель, мы можем присвоить ему указатель на эту функцию:

```
void f();
...
p = &f;
```

Более того, знак & в этом выражении можно опустить и написать просто p = f;

Указателям на функции можно присваивать даже перегруженные функции; компилятор понимает, какая из них нам нужна, по типу указателя на функцию (берется та версия перегруженной функции, которая соответствует определению указателя по значению и числу и типам параметров).

После того, как указателю на функцию присвоено значение, можно вызвать ту функцию, на которую он указывает, написав (*p)(); или даже проще p();

Понятие указателя на функцию используется очень часто, здесь мы приведем только два варианта его использования:

1) Создание массивов функций. Очень часто имеется набор одинаковых по типу возвращаемого значения и типам параметров функций, из которых нужно вызвать одну по номеру. Можно, конечно, написать большой оператор выбора и в зависимости от номера вызвать одну из функций, но проще построить массив из указателей на функции и затем проиндексировать его номером той функции, которая нам нужна:

```
void f1();
void f2();
void f3();
void (*f[3])() = { f1, f2, f3 };
...
int main()
{
...
   f[i](); //вызов
...
}
```

2) Передача функций в качестве параметров другим функциям:

```
double integral(double a, double b, double (*f)(double))
{
    int n = 100;
    double sum = 0;
    for(int i = 1; i<=n; i++)
        sum += f(a+(i-0.5)*(b-a)/n);
    return sum*(b-a)/n;
}
...
int main()
{
...
   cout<<integral(0,1,sin)<<endl; //вызов
...
}
```

В этом примере участвует функция integral, которая приближенно вычисляет определенный интеграл от функции, переданной ей в качестве параметра. В теле функции main эта функция используется для вычисления $\int_a^b \sin(x)dx$.

C++ даже позволяет вернуть указатель на функцию из другой функции в качестве результата, однако это в любом случае должна быть одна из функций, определенных в программе; создать свою собственную функцию в процессе работы программы невозможно.

**Задача.**

Написать функцию, принимающую в качестве параметров две функции (каждая из них имеет два параметра типа char и возвращает char), вычисляющие операции $\triangle$ и $\bigtriangledown$, а также массив M из 80 элементов типа char, и вычисляющую выражение (((M[0] $\triangle$ M[1]) $\bigtriangledown$ M[2]) $\triangle$ M[3]) $\triangle$ ... $\triangle$ (((M[76] $\triangle$ M[77]) $\bigtriangledown$ M[78]) $\triangle$ M[79]).

## 1.15 Безымянные функции

Часто бывает так, что функция, передаваемая в качестве параметра другой функции, больше нигде не используется. Тогда, вместо того, чтобы определять эту функцию традиционным образом, можно воспользоваться так называемой безымянной функцией. Безымянная функция состоит из следующих элементов:

1) Спецификации захвата переменных, описывающей, на какие внешние по отношению к данной функции переменные и как может ссылаться ее тело.

Спецификация захвата переменных выглядит как список элементов захвата через запятую в квадратных скобках. Каждый элемент захвата состоит из символа, определяющего тип захвата («=» — по значению, «&» — по ссылке) и имени переменной. Можно также в начале списка указать тип захвата по умолчанию, т. е. для всех внешних переменных, не перечисленных явно.

2) Описания параметров и результата безымянной функции — в скобках идет список параметров, затем «− >» и тип результата.

3) Наконец, тела функции в фигурных скобках.

С использованием безымянной функции предыдущий пример (п. 2 из предыдущего подраздела) может быть переписан так:

```
template<class F>
double integral(double a, double b, F f)
{
    int n = 100;
    double sum = 0;
    for(int i = 1; i<=n; i++)
        sum += f(a+(i-0.5)*(b-a)/n);
    return sum*(b-a)/n;
}
...
int main()
{
...      //вызов
   cout<<integral(0, 1, [] (double x)->double { return sin(x); })<<endl;
...
}
```

Конечно, для простого синуса все эти накрутки кажутся излишними, однако этот пример легко расширяется для, скажем, функции $\sin^3(x) + 1$:

```
int main()
{
...      //вызов
   cout<<integral(0, 1,
       [] (double x)->double { return sin(x)*sin(x)*sin(x)+1; })<<endl;
...
}
```

На достаточно умных компиляторах безымянная функция без захвата внешних переменных (с пустыми квадратными скобками) может быть автоматически преобразована в обычный указатель на функцию, так что модификация той функции, куда она передается, в шаблон не требуется.

На менее умных компиляторах, равно как и для функций с захватом переменных, также может потребоваться указание явного типа для безымянной функции. Для этого в программе нужно подключить файл functional, и воспользоваться шаблоном function с параметром-типом функции с нужными типами параметров и типом результата:

```
#include <functional>

using namespace std;

double integral(double a, double b, function<double(double)> f)
{
    int n = 100;
    double sum = 0;
    for(int i = 1; i<=n; i++)
        sum += f(a+(i-0.5)*(b-a)/n);
    return sum*(b-a)/n;
}
...
int main()
{
...      //вызов
   cout<<integral(0, 1, [] (double x)->double { return sin(x); })<<endl;
...
}
```

## 1.16 Подстановки параметров

В C++ есть также весьма удобное средство для подстановки конкретных значений в функции, выдающее новые функции. Средство это называется bind. Первый его параметр — функция, куда мы подставляем параметры, и затем идут сами подставляемые параметры. При этом можно использовать так называемые поля подстановки (placeholders), выглядящие как _номер (они соответствуют параметрам той функции, которая получается в результате). При этом если точный тип результата не важен, можно использовать auto, а если важен — function<нужный_тип_функции> (см. предыдущий подраздел).

## 1.17 Строки языка C

В этом разделе обсуждаются средства обработки строк, доставшиеся языку C++ в наследство от C. В языке C строка — это просто указатель на ее первый символ. Такой указатель имеет тип char*. Именно такой тип имеют строковые константы, заключенные в двойные кавычки. При этом, для того, чтобы библиотечные функции правильно работали со строками, всякая строка должна кончаться специальным символом с кодом 0 (не путать с цифрой '0', которая имеет код 48).

Завести строку можно двумя способами. Можно определить переменную типа char*. Такой вариант годится, если этот указатель будет указывать только на строковые константы, на обычные или динамические массивы символов, определенные или созданные где-то в другом месте. В этом случае память выделяется только под указатель, но не под те символы, из которых состоит строка. Если же нам нужно иметь то место, куда будут записаны символы, принадлежащие только этой строке, мы должны объявить массив символов достаточной для хранения строки длины.

Прежде чем описывать библиотечные функции обработки строк, обсудим сначала ряд полезных библиотечных макроопределений для анализа символов. Они находятся в заголовочном файле cctype.

Часто в программе возникает необходимость узнать, к какой категории относится символ, код которого записан в переменную — является ли этот символ буквой, цифрой, управляющим символом или чем-то еще. Для этого в стандартной библиотеке имеются полезные макроопределения, проверяющие такого рода условия. Например, макроопределение isaplha проверяет, является ли его параметр буквой. По сравнению с явно выписанным условием ('A'<=c

&& c<='Z') || ('a'<=c && c<='z') вызов макроопределения isalpha(c) имеет ряд преимуществ: запись существенно короче, код переносимый (будет работать везде, где реализация C++ соответствует стандарту, не зависимо от кодировки символов), и даже быстрее работает, поскольку такие макроопределения проверяют символы по таблице, ничего не вычисляя.

Однако, есть и определенные предостережения касательно использования таких макроопределений. Например, в качестве условия нужно использовать именно isalpha(c), но никак не isalpha(c)==true. Дело в том, что в случае положительного ответа возвращается число, отличное от нуля, но совсем не 1. Если мы используем в условии isalpha(c), то любое ненулевое число воспринимается условным оператором как истина, и все работает. Если же используется второй вариант, результат такого условия (почти) всегда будет ложь. Кроме проверки на букву имеются и много других подобного рода проверок, например isdigit проверяет, что его аргумент содержит код цифры.

Теперь перейдем к функциям для работы со строками. Их прототипы находятся в файле cstring. Мы обсудим только важнейшие из них, и первая из них — strlen — возвращает длину строки (завершающий нулевой символ не учитывается). Например, strlen("Hello") будет 5.

Следующая функция strcpy копирует строку. Она имеет два параметра — куда копировать (должен указывать на то место в памяти, которое будет содержать копию) и что копировать (должен указывать на первый символ исходной строки). При этом забота о выделении памяти под копию всецело ложится на плечи программиста. Например, следующий фрагмент содержит сразу две ошибки:

```
char *p;
strcpy(p, "Hello");
```

Во-первых, указателю не присвоено начальное значение, и во-вторых, не выделена память под копию строки. В итоге произойдет попытка копировать строку в случайное место в памяти, и скорее всего, операционная система поймает Вас за руку.

Менее тривиальное предостережение касается перекрывающихся строк, т. е. того случая, когда области памяти, содержащие исходную строку и ее копию, имеют непустое пересечение, если рассматривать их как множества ячеек. В этом случае стандарт не определяет результат этой (равно как и других) функции обработки строк, т. е. этот результат вполне может отличаться от Ваших ожиданий. Пример такой ситуации приведен ниже:

```
char p[50] = "Hello";
strcpy(p+3, p);
```

Следующая функция strcat приписывает одну строку в конец другой. Она имеет два параметра — строка, в конец которой надо дописывать и что дописывать. Сюда также относятся те предостережения, которые были у предыдущей функции.

Следующая функция strcmp сравнивает две строки в лексикографическом (т. е. как в словаре) порядке. Она возвращает целое число, которое отрицательно, если первая строка меньше второй, равно нулю, если строки совпадают, и положительно, если первая строка больше.

У всех вышеперечисленных функций есть разновидность с буквой n в середине (strncpy и т. д.). Такие функции имеют еще один параметр-целое число, и обрабатывают не больше указанного этим числом количества символов. Например, если вызвана strncmp с этим параметром, равным 10, и первые 10 символов у обеих строк совпадают, то она вернет 0, даже если ее параметры как строки были различны (после первых 10 символов).

Кроме того, имеются функции для поиска символов и подстрок в строке. Функция strchr принимает в качестве параметров строку и символ, и возвращает указатель на первое вхождение данного символа в строку-параметр; если таких вхождений нет, возвращается 0. Аналогично, strrchr ищет последнее вхождение символа в строку. Функция strstr принимает две строки и ищет указатель на первое вхождение строки, заданной вторым параметром, в строку, заданную первым.

Вывести строку на экран можно стандартным образом, написав cout«s, где вместо s нужно написать имя указателя на char или массива символов. Именно из-за того, что вывод указателя на символ воспринимается как вывод строки, мы не можем непосредственно вывести адрес, содержащийся в таком указателе.

Ввести строчку тоже можно стандартным способом, написав cin»s. Однако нужно позаботиться о том, чтобы эта строка влезла в отводимую под нее память, для чего обычно используется манипулятор setw с параметром, равным размеру этой области памяти: char s[50]; cin»setw(50)»s; (setw учитывает необходимость размещения завершающего нулевого символа). Правда, чтобы setw работал, нужно подключить заголовочный файл iomanip.

При вводе строк таким способом есть еще одна деталь: пробелы считаются разделителями, поэтому так считывается только строка до первого пробела. Если мы хотим считывать строку, содержащую пробелы (например, до символа перевода строки), нужно пользоваться следующей конструкцией:

```
cin.getline(buf, num);
```

Здесь buf — указатель на то место в памяти, куда надо считывать символы строки, а num — размер этой области памяти в байтах. Такая конструкция считывает строку до символа перевода строки, записывает завершающий 0 и извлекает из потока cin завершающий символ перевода строки (который, впрочем, не попадает в считанную строку).

Последнее предостережение касается смешивания разных стилей ввода. Например, если написать:

```
int x; char buf[50];
cin>>x;
cin.getline(buf, 50);
```

и при вводе завершить число символом перевода строки (т. е. клавишей Enter), то в buf будет считана пустая строка. Это происходит потому, что при считывании целого числа завершающий его символ перевода строки не будет удален из потока, и при считывании строки он будет первым считанным символом. Для исправления ситуации нужно удалить символ перевода строки из потока, воспользовавшись манипулятором ws, удаляющим из потока пробельные символы:

```
int x; char buf[50];
cin>>x>>ws;
cin.getline(buf, 50);
```

**Задачи.**

1. Написать функцию, принимающую в качестве параметров строку и символ и возвращающую число символов между первым и вторым вхождениями символа-параметра в строку-параметр (без использования операторов цикла).

2. Написать функцию, принимающую в качестве параметров две строки и возвращающую копию первого параметра, из которой удалены все вхождения второго параметра.

## 1.18 Параметры функции main

До сих пор, когда мы писали программы на C++, мы не указывали параметры функции main. Оказывается, функция main может иметь параметры, которые используются для передачи ей параметров командной строки.

В операционных системах семейства Windows командная строка существует, но в явном виде используется достаточно редко. Однако, на самом деле всякая программа, которая запускается, на самом деле запускается из командной строки, и эту командную строку можно найти и изменить (как это делается в Windows, выходит за рамки нашего курса).

Очень редко встречаются программы, имеющие раз и навсегда запрограммированное поведение. Как правило, поведение программы можно менять и настраивать для конкретных нужд, и не последнюю роль в этом играют параметры командной строки.

Всякая командная строка делится на слова. Обычно разделителями слов являются пробелы, но иногда возникает необходимость включить пробелы в состав некоторых слов; такие слова обычно заключаются в кавычки.

Так или иначе, первое слово в командной строке — это имя команды. Все остальные слова называются параметрами. Например, команда копирования файлов в Windows называется copy. Однако для работы она требует и параметры — имя того файла, который нужно копировать, и имя копии.

Для того, чтобы писать полноценные консольные приложения (т. е. программы с текстовым интерфейсом, запускающиеся из командной строки), нужно уметь получать доступ к параметрам той командной строки, при помощи которой была запущена Ваша программа. Именно для этой цели и служат параметры функции main.

Если нам нужно иметь доступ к параметрам командной строки, можно определить функцию main с двумя параметрами. Первый из них имеет тип int и значением его является количество параметров командной строки, т. е. количество тех слов, на которые разбивалась командная строка. Поскольку имя команды должно присутствовать всегда, это количество никогда не бывает меньше единицы.

Второй параметр — это массив указателей на символ неопределенной длины (на самом деле, размер этого массива равен значению первого параметра функции main). В этом массиве передаются сами параметры как строки языка C. Первый из этих параметров всегда содержит имя команды, а далее следуют параметры по порядку, как они встречаются в командной строке.

В качестве примера рассмотрим программу, которая просто печатает все параметры командной строки, каждый на своей строчке:

```
int main(int argc, char *argv[])
{
    for(int i = 0; i<argc; i++)
        cout<<argv[i]<<endl;
    return EXIT_SUCCESS;
}
```

Чтобы убедиться, что эта программа работает, нужно запустить ее из командной строки с параметрами. Для этого можно запустить Far, перейти в папку, содержащую решение (solution) — это будет внешняя из папок, имя которой совпадает с именем Вашего проекта, и затем в подпапку Debug. Там (после сборки Вашего проекта) будет лежать готовая программа. Ее имя совпадает с именем проекта, а расширение, очевидно, «.exe». Теперь нужно нажать ctrl-o, чтобы отключить панели, и набрать в командной строке имя программы и за ним через пробелы несколько параметров, после чего нажать клавишу Enter.

**Задача.**

Написать программу, принимающую в качестве параметра командной строки целое число и печатающую сумму его цифр.