С.С. Михалкович

Язык C++.
Указатели и ссылки

Методические указания

для студентов механико-математического факультета

1. Введение

Указатели (*pointers*) и *ссылки* (*references*) в том или ином виде присутствуют в большинстве языков программирования. При этом значение слова «ссылка» может иметь несколько различный смысл. Чтобы лучше понять значения этих слов, приведем несколько примеров из повседневной жизни.

Когда мы произносим имя человека, то *ссылаемся* на определенного человека. Можно сказать, что имя человека является ссылкой на него. Ссылкой на этого же человека является и фраза «владелец дома по адресу …». Ссылкой на дом является адрес этого дома, его название (например, «Дом книги») или его однозначная характеристика (например, «красный дом в конце квартала»). В роли указателя может выступать табличка с именем объекта и инструкцией по его нахождению.

Дадим более строгие определения. Будем называть *ссылкой (в широком смысле)* некоторое имя или фразу, однозначно идентифицирующие объект. Очевидно, несколько ссылок могут обозначать один и тот же объект, представляя его различные имена.

Указателем (в широком смысле) назовем объект, хранящий информацию о местонахождении другого объекта. По существу, указатель представляет собой объект, хранящий ссылку на другой объект. Указатель в процессе своего существования может изменить значение содержащейся в нем ссылки, указывая тем самым на другой объект.

В языках программирования в роли ссылки чаще всего выступает имя объекта или адрес этого объекта. Указатель же – это объект, хранящий адрес другого объекта. Подчеркнем, что, в отличие от терминологии, принятой в объектно-ориентированных языках программирования, везде в данных методических указаниях под *объектом* понимается область памяти, имеющая определенный тип. Очень близким по смыслу является понятие *переменной* – области памяти, имеющей тип и имя. Таким образом, согласно нашей терминологии, переменная – это объект, имеющий имя.

2. Указатели в C++

Указатель в C++ – это переменная, хранящая адрес некоторого объекта. Если объект имеет тип `T`, то указатель на него описывается следующим образом:

```
T* p;
```

При объявлении нескольких указателей символ `*` обязателен перед именем каждой переменной. Так, в объявлении

```
double* p1, p2, *p3;
```

переменная `p2` принадлежит к типу `double`.

Записи `T* p` и `T *p` равноценны. Запись `T* p` обычно читается как «`p` принадлежит к типу *указатель на `T`* », а запись `T *p` – как «`p` является указателем на объект типа `T`».

Указателю можно присвоить либо значение 0, либо адрес объекта, используя унарный оператор взятия адреса &. Например:

```
int i,a[10];
int* pi=&i,*pa,*pb=0;
pa=&a[9];
```

Здесь указатель pi инициализируется при объявлении адресом переменной i, указатель pb – нулем, а указатель pa инициализируется присваиванием. Присваивание указателю нулевого значения означает, что он не указывает ни на один объект.

Массив указателей и указатель на массив объявляются следующим образом:

```
int* b[10];         // массив указателей
int (*paa)[10]=&a; // указатель на массив
```

Можно также объявить указатель на указатель:

```
char c;
char* pc=&c;
char** ppc=&pc;
```

Указатели, инициализированные значением 0, называются *нулевыми* и ни на что не указывают.

Бывают случаи, когда нас интересует просто значение адреса, а не тип указываемого объекта. С этой целью в язык C++ введены указатели void*, способные хранить адрес любого объекта. В отличие от *типизированных* указателей, рассмотренных выше, указатели void* называются *нетипизированными*.

3. Оператор разыменования *

Для доступа к объекту через указатель используется оператор *разыменования* *, осуществляющий так называемую *косвенную адресацию*: *p означает «объект, на который указывает p». По существу, *p представляет собой ссылку (в широком смысле) на объект. Например, если действуют описания предыдущего пункта, то в результате выполнения следующего фрагмента

```
*pi=5;
(*paa)[9]=77;
**ppc='a';
```

переменной i будет присвоено значение 5, переменной a[9] – значение 77, а переменной c – значение 'a'. Отметим, что во второй строке скобки обязательны, так как оператор [] имеет более высокий приоритет, чем оператор *.

Если p является указателем на структуру s с полем a, то оператор -> в записи p->a используется для сокращения записи (*p).a. Например:

```
struct point {
  double x,y;
};
```

```
point t, *pt=&t;
pt->x=pt->y=1;
```

Попытка разыменования нулевого указателя приводит к ошибке при выполнении программы, попытка же разыменования указателя `void*` вызовет ошибку на этапе компиляции.

Одной из распространенных ошибок является разыменование *неинициализированных указателей*. Например, в любом из следующих случаев результат работы программы непредсказуем:

```
char* s;
*s='a'; // ошибка!
cin>>s; // ошибка!
```

Отметим, что в последнем случае, где выводится строка, на которую указывает `s`, разыменование присутствует неявно и происходит внутри оператора ввода из потока.

4. Указатели и преобразование типов

Указатель на один тип нельзя присвоить указателю на другой тип без явного преобразования типов. Исключение составляет указатель `void*`, который трактуется как указатель на некоторый участок памяти. Он называется *родовым* указателем и может получать в качестве значения указатель на любой другой тип без явного преобразования. Например:

```
int i;
void* pi=&i;
```

Напомним, что указатель `void*` нельзя разыменовывать. Каким же образом получить доступ к переменной `i` через указатель `pi`? Для этого необходимо использовать оператор приведения типа `static_cast`. Он преобразует объект к типу указателя, записанному в угловых скобках:

```
int* pi1=static_cast<int*>pi;
```

В некоторых старых компиляторах оператор `static_cast` отсутствует, поэтому приходится использовать приведение типа в старом стиле:

```
int* pi1=(int*)pi;
```

Отметим, что явные преобразования типов в большинстве случаев являются потенциально опасными и должны применяться с крайней осторожностью. Так, в следующем примере

```
double d=*static_cast<double*>pi;
```

содержимое переменной `d` непредсказуемо.

Именно потенциальная опасность операторов приведения типа привела к необходимости систематизации ситуаций, в которых используется приведение, В новой редакции C++ старый способ приведения заменен на четыре новых, различающихся по степени безопасности. Среди них уже рассмотренный нами оператор `static_cast`, оператор снятия константности `const_cast` (он будет рассмотрен в пункте 5), а также операторы `reinterpret_cast` (для

преобразования принципиально различных типов) и dynamic_cast (для преобразования полиморфных типов, связанных иерархией наследования).

Свойство указателей void* хранить данные разнородных типов используется при создании так называемых родовых (generic) массивов, т. е. массивов, хранящих разнотипные объекты. Например:

```
int i=5;
char c='u';
double d[2]={3,6};
void* generic[]={&i,&c,&d};
cout<<*static_cast<int*>generic[0]<<' '
    <<*static_cast<char*>generic[1]<<' '
    <<(static_cast<double*>generic[2])[1];
```

5. Указатели и спецификатор const

Спецификатор const нужен, как известно, для запрещения доступа на запись. Комбинирование спецификатора const с указателями порождает несколько новых типов:

```
int i=3,j=4;
const int n=5;
int *pi;
const int* pi1; // указатель на константу
int const* pi11; // тоже указатель на константу
int* const pi2=&j; // константный указатель
const int* const pi3=&n; // константный указатель
  // на константу
```

Обычный указатель pi может менять в процессе работы программы как свое значение, так и значение объектов, на которые он указывает. Его нельзя инициализировать адресом константного объекта, поскольку в противном случае можно было бы модифицировать значение константы косвенно:

```
pi=&i;  // верно
pi=&n;  // ошибка!
pi=pi1; // ошибка!
pi=pi2; // верно
```

Указатель на константу pi1 в процессе работы программы может получать адреса как константных, так и неконстантных объектов. Однако, изменять их значения указатель на константу не может:

```
pi1=&n;
cout<<*pi1; // верно
pi1=&j;
*pi1=7; // ошибка!
```

Константный указатель pi2, напротив, должен (как и любая константа) инициализироваться при объявлении. Менять свое значение в процессе работы он не может:

```
*pi2=7; // верно
pi2=&j; // ошибка!
```

Наконец, *константный указатель на константу* `pi3` не может модифицировать ни свое значение, ни значение указываемого им объекта. На практике такой указатель используется крайне редко.

6. Оператор `const_cast` и снятие константности

В некоторых немногочисленных случаях необходимо уметь модифицировать значение константного объекта. Например:

```
int i=3;
const int* pic=&i;
int* pi=pic; // ошибка!
```

Несмотря на то, что мы заведомо знаем, что `pi` указывает на неконстантный объект, изменить значение этого объекта мы не можем. В этом случае требуется явное приведение типа с помощью оператора приведения типа `const_cast`, «снимающего» константность:

```
int* pi=const_cast<int*>(pic); // верно
```

Теперь данные, адрес которых хранится в переменной `pic`, можно косвенно изменить через указатель `pi`:

```
*pi=4;
```

То же самое можно сделать, используя `const_cast` в левой части оператора присваивания:

```
*const_cast<int*>(pic)=4;
```

Вновь подчеркнем потенциальную опасность операторов явного приведения типа. В следующем примере

```
const int n=5;
int* pi=const_cast<int*>(&n);
*pi=7;
```

изменяется значение настоящей константы, что недопустимо!

7. Указатели и передача параметров в функции

С помощью указателей можно организовать передачу параметров в функции по ссылке. Например, следующая функция меняет местами значения, адресуемые указателями:

```
void swap(int* pa, int* pb)
{
  int t=*pa; *pa=*pb; *pb=t;
}
```

В функциях стандартной библиотеки можно встретить множество примеров передачи параметров с использованием указателей. Такова, например, функция `memcpy`, предназначенная для копирования данных из одной области

памяти в другую. Она имеет следующий прототип, объявленный в заголовочном файле <string.h> (или по стандарту 1998 г. в <cstring>):

```
void *memcpy(void *dest, const void *src, size_t n);
```

и обеспечивает копирование n байтов данных из src в dest. Параметр src объявлен указателем на константу, что свидетельствует о том, что данные, на которые он указывает, не могут быть изменены функцией memcpy. Поскольку в качестве типа фигурирует void*, то вместо src и dest при вызове функции memcpy можно подставлять указатель на любой тип. Наконец, стандартный тип size_t используется для указания того, что данная переменная хранит размер и представляет собой беззнаковое целое (обычно unsigned int или unsigned long).

Возвращение функцией указателя на локальную переменную является грубой ошибкой. Например, в ситуации

```
int* f()
{
  int i=5;
  return &i;
}
```

переменная i разрушается после выхода из функции, поэтому результат работы программы непредсказуем.

8. Арифметические действия с указателями

Над указателями можно совершать ряд арифметических действий. При этом предполагается, что если указатель p относится к типу T*, то p указывает на элемент некоторого массива типа T. Тогда p+1 является указателем на следующий элемент этого массива, а p-1 – указателем на предыдущий элемент. Аналогично определяются выражения p+n, n+p и p-n, а также действия p++, p--, ++p, --p, p+=n, p-=n, где n – целое число. Важно отметить, что арифметические действия с указателями *выполняются в единицах того типа, к которому относится указатель*. То есть p+n, преобразованное к целому типу, содержит на sizeof(T)*n большее значение, чем p.

Из равенства p+n==p1 следует, что p1-p==n. Именно так вводится оператор разности двух указателей: его значением является целое, равное количеству элементов массива от p до p1. Отметим, что это – единственный случай в языке, когда результат бинарного оператора с операндами одного типа принадлежит к принципиально другому типу.

Сумма двух указателей не имеет смысла и поэтому не определена. Не определены также арифметические действия над нетипизированными указателями void*.

Наконец, все указатели, в том числе и нетипизированные, можно сравнивать, используя операторы отношения >, <, >=, <=, ==, !=.

Поясним сказанное примерами.

Пример 1. Сумма элементов массива.

```
int a[10],s;
for (int* p=&a[0]; p<=&a[9]; p++)
  s+=*p;
```

Пример 2. Инвертирование массива.

```
for (int* p=&a[0],*q=&a[9]; p<q; p++,q--)
  swap(p,q); // см.п.6
```

9. Указатели и массивы

Указатели и массивы тесно взаимосвязаны. *Имя массива может быть неявно преобразовано к константному указателю на первый элемент этого массива*. Так, `&a[0]` равноценно `a`. Вообще, верна формула

```
&a[n] == a+n
```

то есть адрес `n`-того элемента массива есть увеличенный на `n` элементов указатель на начало массива. Разыменовывая левую и правую части, получаем основную формулу, связывающую массивы и указатели:

`a[n] == *(a+n)` (*)

Данная формула, несмотря на простоту, требует нескольких пояснений. Во-первых, компилятор любую запись вида `a[n]` интерпретирует как `*(a+n)`. Во-вторых, формула (*) поясняет, почему в C++ массивы индексируются с нуля и почему нет контроля выхода за границы диапазона. Наконец, используя (*), мы можем записать следующую цепочку равенств:

```
a[n] == *(a+n) == *(n+a) == n[a]
```

Таким образом, элемент массива `a` с индексом `2` можно обозначить не только как `a[2]`, но и как `2[a]` (проверьте!).

Из связи массивов и указателей вытекает способ передачи массивов в функции – с помощью указателя на первый элемент. Более того, следующие прототипы функций полностью эквивалентны:

```
void print(int a[10], size_t n);
void print(int a[], size_t n);
void print(int *a, size_t n);
```

В частности, нетрудно проверить, что `sizeof(a)` внутри функции `print()` во всех трех случаях совпадает с `sizeof(int*)`. Таким образом, *внутри функции теряется информация о размере массива*, поэтому размер необходимо передавать явно как еще один параметр. Заметим также, что массив в языке C++ нельзя передать по значению (как в языке Паскаль): изменение элемента массива внутри функции `print()` всегда приводит к изменению фактического параметра-массива. Это же замечание справедливо для строк `char*`, являющихся указателями на символьные массивы.

Подчеркнем еще раз, что указатель, к которому преобразуется имя массива, – константный. В частности, это означает, что *имени массива нельзя присвоить значение*:

```
int a[10],b[10];
a=b; // ошибка!
```

Но если запрещено присваивание, то как скопировать один массив в другой? Для этого, помимо тривиального поэлементного копирования, существуют следующие способы.

Способ 1. Воспользоваться функцией `memcpy` (см. п.6):

```
memcpy(a,b,10*sizeof int);
```

Способ 2. Воспользоваться функцией `copy(first,last,out)`, объявленной в заголовочном файле `<algorithm>`, которая копирует *диапазон* значений массива между указателями `first` и `last` в массив `out`:

```
copy(a,a+10,b);
```

Диапазоном называется множество элементов между двумя указателями `first` и `last` на элементы некоторого массива, причем, `*first` входит в диапазон, а `*last` не входит. Следуя математической нотации, будем обозначать указанный диапазон как `[first,last)`. Таким образом, диапазон `[a,a+10)` описывает все элементы массива `a`. Если `first==last`, то говорят, что диапазон пуст.

Замечание. Если `first` и `last` ошибочно указывают на элементы различных массивов, то возникает недиагностируемая ошибка выхода за границы диапазона, которая при `first>last` приводит к зацикливанию.

10. Идиома *p++

Идиомой называют устойчивую конструкцию, воспринимаемую и используемую как единое целое. Конструкция `*p++`, где `p` – указатель, очень часто используется при низкоуровневой работе с указателями. Рассмотрим ее подробнее.

Операторы `*` и `++` имеют одинаковый приоритет, поэтому порядок выполнения определяется ассоциативностью. Оба этих оператора являются унарными, поэтому они ассоциируются справа налево. Таким образом, в записи `*p++` скобки неявно расставлены следующим образом: `*(p++)`. Но оператор `++` постфиксный, поэтому результатом выражения `*p++` будет значение `*p` до увеличения указателя `p`, после чего указатель `p` будет увеличен на `1`. В результате в записи `*p++` совмещаются два действия: возвращение значения текущего элемента массива и переход к следующему элементу (напомним, что вне массивов арифметические действия с указателями не имеют смысла).

Приведем ряд примеров.

Пример 1. Сложение двух векторов.

```
int a[10],b[10],c[10];
for (int *p=a,*q=b,*r=c; p<a+10; )
  *r++=*p+++*q++;
```

Пример 2. Копирование массивов целых.

```
void copy(const int* first, const int* last, int* out)
```

```
{
  while (first!=last)
    *out++=*first++;
}
```

Пример 3. Поиск элемента `v` в диапазоне `[first,last)`.

```
bool find(const int* first, const int* last, int v)
{
  while (first!=last)
    if (*first++==v) return true;
  return false;
}
```

11. С-строки

В C++ имеется два типа строк: встроенный тип, унаследованный от языка C (строки данного типа мы будем называть С-строками), и класс `string` из стандартной библиотеки C++. Класс `string` появился в стандарте языка в августе 1998 г. и может быть не реализован в устаревших компиляторах. Мы рассмотрим лишь С-строки, поскольку они тесно связаны с указателями и используются для получения эффективного кода.

С-строка – это массив символов, оканчивающийся символом с кодом 0, или *нулевым символом* (`'\0'`). К строковым константам нулевой символ добавляется автоматически:

```
sizeof("LoveC++")==8
```

Доступ к строке обычно осуществляется с помощью указателя `char*`, поэтому, как правило, тип `char*` ассоциируется именно со строкой. Так, если переменная `s` имеет тип `char*`, то при выполнении оператора

```
cout<<s;
```

в поток вывода записываются символы, на которые указывает `s`, до тех пор, пока не будет встречен нулевой символ `'\0'`.

Всюду далее, если не оговорено противное, под строкой будем понимать именно С-строку.

11.1. Описание и инициализация строк

Для строки резервируется массив символов:

```
char s[20];
```

Длина массива должна быть достаточной для хранения всех возможных строковых значений, которые могут встретиться при работе с данной строковой переменной (напомним, что контроль выхода за границы массива в C++ отсутствует).

При описании строка может быть инициализирована строковой константой:

```
char s1[20]="String";
char s2[5]="Hello"; // ошибка: не отведено место
```

```
                                    // под завершающий '\0'
```

Если требуется описать именованную строковую константу, то используются следующие варианты инициализации:

```
const char s3[]="Hello";
const char* s4="Good Bye";
```

Под массив `s3` при этом отводится память из шести элементов типа `char`. Заметим, что, в отличие от `s3`, указатель `s4` может менять свое значение в процессе работы.

Можно также встретить аналогичную инициализацию без `const`:

```
char s3[]="Hello";
char* s4="Good Bye";
```

Она не вполне корректна, поскольку позволяет модифицировать константные по смыслу данные, однако ее можно встретить в реальных программах.

11.2. Ввод строк

Оператор >> позволяет ввести слово:

```
char word[10];
cin>>word;
```

При этом в потоке ввода вначале пропускаются все символы-разделители (пробелы, символы перехода на новую строку и символы табуляции), затем в переменную `word` считываются символы до символа-разделителя и дописывается нулевой символ. Основная ошибка здесь – это попытка ввести больше символов, чем вмещает в себя массив символов `word`. Например, при вводе строки `"  abracadabra  "` два лидирующих пробела будут пропущены, в массив `word` будут записаны символы `"abracadabr"`, а символы `'a'` и `'\0'` попадут в следующие за `word` ячейки памяти. Сообщение об ошибке при этом, как правило, не возникнет.

Для решения проблемы выхода за границы массива-строки в приведенном выше примере следует использовать манипулятор `setw`, устанавливающий максимальную ширину поля ввода:

```
#include <iomanip.h>
...
cin>>setw(10)>>word;
```

В этом случае в массив `word` попадут символы `"abracadab"` плюс завершающий нулевой символ, а символ `'a'` останется в потоке ввода. Заметим, что использование манипулятора `setw` влияет только на непосредственно следующую за ней операцию ввода.

Если требуется ввести не отдельное слово, а строку целиком, то используется функция-член `getline` класса `istream`, к которому принадлежит поток `cin`:

```
cin.getline(word,10);
```

В этой ситуации ввод будет осуществляться до символа перехода на новую строку '\n', но максимально будет считано 9 символов, после чего к строке word будет добавлен '\0'.

Функция getline имеет также третий параметр – символ-разделитель, до которого осуществляется считывание. По умолчанию он равен '\n', но может быть явно изменен. В следующем примере считывание осуществляется до символа '!', но не более 9 символов:

```
cin.getline(word,10,'!');
```

Отметим, что сам символ-разделитель удаляется из потока ввода.

11.3. Копирование строк

Поскольку строка – это массив символов, то копирование строк невозможно осуществить с помощью оператора присваивания. Действительно, если имеются следующие описания

```
char s1[10]="Hello",s2[10],*s3;
```

то присваивание s1=s2 вызовет ошибку компиляции, поскольку имя массива s1 является константным указателем. Присваивание же s3=s1 вполне законно, но оно не копирует данные из одной строки в другую, а лишь инициализирует указатель s3 адресом начала строки s1. В этом случае любое изменение строки через указатель s3 меняет исходную строку s1.

Поскольку строка завершается нулевым символом, ее копирование имеет специфику. Рассмотрим вначале несколько способов копирования без привлечения библиотечных функций.

Следующий цикл производит посимвольное копирование из строки s1 в строку s2 до того момента, как в строке s1 встретится нулевой символ:

```
int i;
for (i=0; s1[i]; i++)
  s2[i]=s1[i];
s2[i]=0;
```

После цикла нулевой символ дописывается в конец строки s2 (число 0 неявно преобразуется в символ '\0').

Посимвольное копирование можно также осуществить, используя указатели:

```
char *p=s1,*q=s2;
while (*p)
  *q++=*p++;
*q=0;
```

Если *p становится равным нулю, то достигнут конец строки s1, и цикл завершается. Завершающий нулевой символ также приходится дописывать «вручную». Обратите внимание, что после цикла длина строки s1 может быть вычислена как p-s1.

Наконец, учитывая то, что оператор присваивания возвращает значение левой части после присваивания, мы можем записать алгоритм копирования максимально компактно:

```
char *p=s1,*q=s2;
while (*q++=*p++);
```

На последней итерации цикла `*p` становится равным нулю, это значение присваивается `*q` и возвращается как результат оператора присваивания, что и приводит к завершению цикла.

11.4. Стандартные функции работы со строками

Функции для манипулирования С-строками объявлены в заголовочном файле <string.h> (или по стандарту 1998 г. в <cstring>). Наиболее часто используются функции `strlen`, `strcpy`, `strcmp`, `strcat`.

Функция `strcpy` имеет следующий прототип:

```
char *strcpy(char *p, const char *q);
```

Она копирует строку `q` в строку `p`, включая нулевой символ, и возвращает указатель на строку `p`. При этом выход за границы массива `p` не контролируется:

```
char s1[6]="Hello", s2[20]="Good bye";
strcpy(s2,s1); // верно
strcpy(s1,s2); // ошибка!
```

Возможные реализации функции `strcpy` приведены в предыдущем пункте.

Для определения длины строки служит функция `strlen` с прототипом

```
size_t *strlen(const char *p);
```

Использовать ее предельно просто:

```
char s3[20]="abracadabra";
size_t sz=strlen(s3); //  sz==11
```

Очевидно, что для вычисления длины строки функция `strlen` должна просканировать строку до конца. Возможная реализация `strlen` выглядит так:

```
size_t *strlen(const char *p)
{
  size_t len;
  while (*p++) len++;
  return len;
}
```

Для добавления одной строки к другой используется функция `strcat` с прототипом:

```
char *strcat(char *p, const char *q);
```

Данная функция добавляет содержимое строки `q` в конец строки `p` и возвращает полученную строку. Например:

```
char s[20]; s1[]="love";
strcpy(s,"I ");
```

```
strcat(s,s1);
strcat(s,” C++”); // s==”I love C++”
```

Возможная реализация strcat приведена ниже:

```
char *strcat(char *p, const char *q)
{
  while(*p) p++;
  while (*p++=*q++);
  return p;
}
```

Отметим, что если длина исходной строки известна заранее, то вместо strcat можно воспользоваться strcpy:

```
strcpy(s,”I ”);
strcpy(s+2,s1);
strcpy(s+7,” C++”);
```

Такой алгоритм эффективнее, так как строка, к которой происходит добавление, не сканируется в поисках завершающего нуля.

Функция strcmp предназначена для лексикографического сравнения строк. Она имеет прототип

```
int strcmp(const char *p, const char *q);
```

и возвращает 0 если строки совпадают, положительное число, если первая строка больше второй, и отрицательное – если вторая больше первой. Более точно: производится посимвольное сравнение строк до обнаружения пары различных символов или до конца одной из строк и возвращается разность кодов первых не совпавших символов или 0.

Возможная реализация strcmp имеет вид:

```
int strcmp(const char *p, const char *q)
{
  while(*p==*q && *p)
  {
    p++; q++;
  }
  return *p-*q;
}
```

В заключение данного пункта приведем менее употребительные функции работы с С-строками:

```
char *strncpy(char *p, const char *q, int n);
```

– то же, что и strcpy, но копируется максимум n символов.

```
char *strncat(char *p, const char *q, int n);
```

– то же, что и strcat, но добавляется максимум n символов.

```
int strncmp(const char *p, const char *q, int n);
```

– то же, что и strcmp, но сравнивается максимум n символов.

```
char *strchr(const char *p, char c);
```

возвращает адрес первого вхождения символа c в строку p (или 0, если символ не найден).

```
char *strrchr(const char *p, char c);
```

возвращает адрес последнего вхождения символа c в строку p (или 0, если символ не найден).

```
char *strstr(const char *p, const char *q);
```

возвращает адрес первого вхождения подстроки q в строку p (или 0, если подстрока не найдена).

```
char *strpbrk(const char *p, const char *q);
```

возвращает адрес первого вхождения в строку p какого-либо символа из строки q (или 0, если совпадений не обнаружено).

```
size_t strspn(const char *p, const char *q);
```

возвращает число начальных символов в строке p, которые не совпадают ни с одним из символов из строки q.

```
size_t strcspn(const char *p, const char *q);
```

возвращает число начальных символов в строке p, которые совпадают с одним из символов из строки q.

```
char *strtok(char *p, const char *q);
```

– последовательность вызовов функции strtok разбивает строку p на лексемы, разделенные символами из строки q. При первом вызове в качестве p передается указатель на строку, которую надо разбить на лексемы, при всех последующих – нулевой указатель. При этом значение указателя, с которого должен начинаться поиск следующей лексемы, сохраняется в некоторой системной переменной. Функция strtok возвращает указатель на найденную лексему или 0, если лексем больше нет. Она также модифицирует исходную строку, вставляя нулевой символ после найденной лексемы. Далее в пункте 10.6 будет дан пример использования функции strtok.

11.5. Задача о перестановке слов: реализация без библиотечных функций

Пусть дана строка слов, разделенных одним или несколькими пробелами. В начале и в конце строки могут находиться также лидирующие и завершающие пробелы. Требуется удалить лишние пробелы, поменять местами четные и нечетные слова и записать результат в новую строку.

Существует множество алгоритмов решения этой задачи, отличающихся друг от друга по универсальности, сложности, скорости, компактности и понятности кода. Будем стремиться найти самый быстрый алгоритм.

Если для реализации алгоритма мы проходим по исходной строке один раз, то алгоритм будем называть однопроходным, если два – двухпроходным и т. д. Наиболее наглядным в данной задаче представляется двухпроходный алгоритм: при первом проходе запоминаются адреса начал слов, а при втором формируется новая строка с попутным удалением лишних пробелов. Однако при

более детальном анализе оказывается, что при первом проходе можно запоминать адрес очередного нечетного слова, затем искать следующее за ним четное слово и копировать его в результирующую строку, после чего копировать нечетное слово, адрес которого мы запомнили. Такой алгоритм, где четные слова проходятся один раз, а нечетные – два, естественно назвать *полуторапроходным*.

Отметим, что приведенный вначале «очевидный» двухпроходный алгоритм имеет еще один недостаток: адреса начал слов придется хранить в некоторой структуре данных (массив, очередь, список); при этом возникает ряд непростых вопросов: как ее инициализировать и кто будет ответственен за ее удаление. Полуторапроходный алгоритм не требует введения дополнительной структуры данных: в нем запоминается адрес лишь одного слова.

В нашем алгоритме несколько раз будут встречаться следующие циклы:

```
while (*p==' ') p++;        // пропуск пробелов
while (*p!=' ' && *p) p++; // пропуск слова
while (*p!=' ' && *p) *ps++=*p++; //копирование слова
```

Условие `*p!=' ' && *p` читается как “пока `*p` не пробел и не конец строки”.

Пусть `p` – исходная строка, `ps` – результирующая. Приведем одну из возможных реализаций алгоритма.

```
void swap_even_odd(const char* p, char* ps)
{
 char* p1;
 while (*p==' ') p++; // пропустить лидирующие пробелы
 do
 {
  p1=p; // запомнить адрес нечетного слова
  while (*p!=' ' && *p) p++;//пропустить нечетное слово
  while (*p==' ') p++;     //пропустить пробелы
  if (*p) // если есть четное слово
  {
    while (*p!=' ' && *p)
      *ps++=*p++;       // копировать четное слово
    *ps++=' ';          // добавить пробел
  }
  while (*p1!=' ' && *p1)
    *ps++=*p1++;        // копировать нечетное слово
  *ps++=' ';            // добавить пробел
  while (*p==' ') p++; // пропустить пробелы
 } while (*p);
 *--ps=0; // удалить лишний пробел и добавить 0
}
```

Заметим, что алгоритм корректно работает в случае пустой строки и строки, состоящей из одних пробелов.

11.6. Задача о перестановке слов: реализация с библиотечными функциями

Для решения задачи о перестановке слов удобнее всего воспользоваться функцией `strtok`, разбивающей строку на лексемы. Преимуществом здесь является то, что в качестве множества разделителей здесь можно указать не только пробел, но и набор других символов (например, `" ,.-;:!?"`).

Перед добавлением второго и последующих слов в результирующую строку будем добавлять пробел (функция `add_word`).

```
void add_word(const char* word, char* ps)
{
  if (*ps) strcat(ps," ");
  strcat(ps,word);
}
void swap_even_odd(char* p,char* ps,const char* delim)
{
  // делаем результирующую строку пустой
  *ps=0;
  // ищем первое нечетное слово
  char* r=strtok(p,delim);
  while (r)
  {
    char* r1=strtok(0,delim); // ищем четное слово
    if (r1) add_word(r1,ps); // добавляем его
    // добавляем предыдущее нечетное слово
    add_word(r,ps);
    // ищем очередное нечетное слово
    r=strtok(0,delim);
  }
}
```

Отметим, что функция `add_word` крайне неэффективна: для добавления слова к результирующей строке она вынуждена всякий раз сканировать результирующую строку в поисках ее конца. Для решения проблемы эффективности можно воспользоваться либо строковыми потоками (`ostrstream`), либо оператором += класса `string`, которые эффективно добавляют данные в конец строки. Однако обсуждение этих способов выходит за рамки настоящих методических указаний.

12. Динамическое распределение памяти

По способу хранения все объекты в C++ делятся на *статические, автоматические* и *динамические*. Статические объекты создаются в так называемой *статической памяти* и хранятся в ней до окончания работы программы. К статическим относятся все глобальные объекты, а также локальные объекты, описанные с модификатором `static`. Локальные объекты без модификатора `static` и аргументы функций называются автоматическими и хранятся в *ав-*

томатической памяти (называемой также *программным стеком*). Программный стек создается в момент начала работы программы. При входе в блок в стеке выделяется память для всех автоматических объектов этого блока, при выходе из блока эта память освобождается. В отличие от статических, автоматические объекты не сохраняют свои значения между повторными входами в блок, в частности, между вызовами функций.

Существует третий вид памяти: это *динамическая память*, или *куча* (от англ. *heap*). Она выделяется и освобождается специальным диспетчером динамической памяти по определенному запросу. Доступ к динамическому объекту (т. е. объекту, созданному в динамической памяти) осуществляется только с помощью указателя, хранящего его адрес. Для создания динамического объекта необходимо воспользоваться оператором `new`, для удаления – оператором `delete`. Например, следующий код

```
int* p;
p=new int; // или p=new(int)
...
delete p;
```

создает в динамической памяти объект целого типа. Оператору `new` передается тип создаваемого объекта, результатом же выражения `new int` будет адрес ячейки целого типа, выделенной диспетчером динамической памяти, который присваивается указателю `p`. После использования динамическая память явно возвращается системе оператором `delete`, при этом указатель `p` *не обнуляется*.

По умолчанию после создания динамическая переменная содержит неопределенное значение. Начальное значение динамической переменной можно задать следующим образом:

```
p=new int(5); // или p=new(int)(5);
```

12.1. Динамические массивы

При распределении массива в динамической памяти необходимо указать тип массива в качестве параметра оператора `new`:

```
p=new int[10]; // или p=new(int[10]);
```

Никакого способа задать начальные значения для элементов массива при этом не существует.

В отличие от статических и автоматических массивов, *размер динамического массива можно задавать во время выполнения программы*:

```
int n;
cin>>n;
int* p=new int[n];
```

В силу формулы (*) из п.8 обращаться к элементам такого массива можно через указатель, используя обычный `[]`-синтаксис, поскольку `p[2]==*(p+2)`.

Замечание. Если `p` – указатель типа `T*`, где `T` – класс, то в результате выполнения оператора

```
T* p=new T[10];
```

для каждого элемента массива вызывается конструктор по умолчанию (т. е. конструктор без параметров). Если же конструктор по умолчанию отсутствует, то возникает ошибка при компиляции.

Для освобождения динамической памяти, занимаемой массивом, используется модифицированная форма оператора `delete`:

```
delete[] p;
```

Если `p` – указатель на встроенный тип, то квадратные скобки можно опускать, если же `p` – указатель на объект некоторого класса `T`, то скобки опускать нельзя. При освобождении динамической памяти с помощью `delete[] p` для каждого элемента массива вызовется деструктор.

12.2. Исчерпание динамической памяти

Если при выполнении оператора `new` диспетчер динамической памяти не может выделить требуемый объем памяти, то генерируется исключение `bad_alloc`. Для его обработки пользуются стандартным блоком `try-catch`:

```
try {
  for (;;) // бесконечный цикл
    new char[10000];
}
catch (bad_alloc) {
  cout<<"Недостаточно динамической памяти/n";
}
```

Замечание. В старых версиях C++ при невозможности выделить память оператор `new` возвращал `0`:

```
double* pd=new double[10000];
if (!pd) ... // память не выделена
```

12.3. Ошибки при работе с динамической памятью

Использование динамической памяти обладает высокой гибкостью, поскольку позволяет программисту контролировать процесс ее выделения и освобождения. Однако, это может привести к определенным ошибкам при программировании. Рассмотрим наиболее часто встречающиеся.

Ошибка 1. Неинициализированные указатели.

Эта ошибка уже упоминалась в пункте 2. При работе с динамической памятью она возникает, когда предпринимается попытка использовать объект, для которого забыли выделить динамическую память:

```
int* p;
*p=5; // ошибка!
```

Ошибка 2. Висячие указатели.

Эта ошибка возникает при попытке использовать объект после освобождения динамической памяти:

```
int* p=new int;
delete p;
*p=5; // ошибка!
```

Указатель p, содержащий адрес динамической переменной, уже удаленной из памяти, называется *висячим указателем*. Последствия работы программы с такой ошибкой – непредсказуемые, как и при работе с неинициализированными указателями.

Ошибка 3. Память, выделенная динамически, не возвращается системе в конце работы программы.

Данная ошибка не имеет фатальных последствий, так как при завершении работы программы диспетчер динамической памяти C++ автоматически возвращает системе всю затребованную программой динамическую память. Однако, подобное неаккуратное программирование неизбежно скажется при реализации больших проектов (например, если программа будет переделана в подпрограмму, которая вызывается в цикле – см. ошибку 4). Поэтому настоятельно рекомендуется придерживаться следующего основного правила: *в начале и в конце работы программы количество свободной динамической памяти должно быть одинаково*.

Ошибка 4. Утечка памяти.

В приведенном фрагменте программы

```
int* p=new int, a=5;
p=&a;
```

динамическая память, отведенная оператором new, после выполнения последнего оператора более не связана ни с одним указателем и поэтому становится недоступной. В частности, она не может быть возвращена системе оператором delete. Такая динамическая память часто называется мусором, а данное явление называется *утечкой памяти*. В отличие от предыдущей ошибки, динамическая память необратимо теряется не в конце, а в середине программы. Если утечка памяти постоянно происходит в цикле, то это может привести к исчерпанию динамической памяти.

Другая часто встречаемая причина утечки памяти – вызов оператора new внутри функции без заключительного вызова delete:

```
void f()
{
  char* p=new char[1000];
  ...
  // delete отсутствует
}
```

Динамическая память контролируется локальной переменной. При выходе из функции локальная переменная перестает существовать и контроль над выделенной памятью утрачивается. Вызов функции f() в большом цикле приводит к быстрому исчерпанию свободной динамической памяти.

Ошибка 5. Попытка возвратить память, не распределенную ранее.

Подобна ошибка возникает, например, в следующем фрагменте кода

```
int* p;
...
delete p; // ошибка!
```

Если переменная `p` является автоматической, то она может содержать произвольное значение. Попытка вызвать оператор `delete` для такого неинициализированного указателя неизбежно приведет к ошибке. Отметим, что если переменная `p` является статической, и следовательно, инициализируется по умолчанию нулевым значением, то ошибка не проявляется: `delete p` просто ничего не делает.

Ошибка 6. Попытка повторно освободить динамическую память.

В следующем фрагменте кода

```
int* p=new int;
delete p;
delete p; // ошибка!
```

предпринята ошибочная попытка дважды освободить одну и ту же область динамической памяти. Вероятность совершения подобной ошибки возрастает, если два указателя содержат адрес одного и того же участка динамической памяти («слуга двух господ»). Само по себе наличие двух указателей-«господ» не является ошибкой; ошибка возникает, когда каждый указатель пытается освободить данный участок динамической памяти:

```
int* p=new int, *q=p;
delete p;
delete q; // ошибка!
```

Ошибка 7. Попытка освобождения нединамической памяти.

При выполнении приведенного фрагмента

```
int a, *p=&a;
delete p; // ошибка!
```

возникнет ошибка независимо от того, является переменная `a` статической или автоматической.

Как правило, большинство ошибок начинающий программист совершает именно при работе с динамической памятью. К сожалению, такие ошибки часто не диагностируются, сказываются на работе программы не сразу и поэтому находятся с трудом. Основная рекомендация, позволяющая уменьшить количество таких ошибок, – аккуратное и вдумчивое программирование. Помните: искать ошибку подобного рода значительно сложнее и дольше, чем писать аккуратные и понятные программы!

Замечание. Многие современные языки автоматизируют процесс работы с динамической памятью. Так, в языке программирования Java имеется оператор `new`, но отсутствует оператор `delete`. При нехватке динамической памяти специальный *сборщик мусора* возвращает в нее те захваченные ранее участки, которые уже не контролируются ни одной переменной. Это позволяет избежать всех описанных выше ошибок, кроме первой и второй: попытка использовать

переменную, под которую не выделена динамическая память, приводит к исключению.

13. Шаблон auto_ptr

В стандартной библиотеке C++ стандарта 1998 года имеется шаблонный класс auto_ptr (описанный в заголовочном файле memory), экземпляр которого инициализируется указателем и может быть использован как указатель. Кроме того, динамический объект, на который он указывает, будет неявно удален в конце своей области видимости при вызове деструктора auto_ptr. Например:

```
{
  auto_ptr<int*> a(new int[10]);
  a[9]=5;
  ...
} // неявный вызов delete
```

Указатели в стиле auto_ptr часто используются для быстрого решения проблемы утечки памяти в функциях.

14. Выделение динамической памяти в функции

Количество динамической памяти в начале и в конце работы функции в большинстве случаев должно быть одинаковым. Функции, выделяющие динамическую память и не освобождающие ее впоследствии, имеют особый статус. Они, по существу, конструируют некоторый объект и возлагают ответственность за его удаление на внешнее окружение. Использовать их необходимо с особой осторожностью. Основное неудобство состоит в том, что программист, использующий подобные функции, должен знать и вынужден помнить, что память, распределенную такой функцией, необходимо вернуть назад системе. Таким образом, использование подобных функций провоцирует совершение ошибок 3, 4 и 6.

Рассмотрим следующий пример. Предположим, что мы создали функцию strclone, конструирующую дубликат строки в динамической памяти:

```
char* strclone(const char* s)
{
  char* p=new char[strlen(s)+1];
  strcpy(p,s);
  return p;
}
```

Ее более короткий вариант имеет вид (разберитесь!):

```
char* strclone(const char* s)
{ return strcpy(new char[strlen(s)+1],s); }
```

После

```
char s[10]="Original";
char* s1=strclone(s);
```

ответственность за вызов оператора `delete s1` возлагается на программиста, использующего `strclone`.

Отметим, что в языках со сборщиком мусора (Java, Smalltalk) подобных проблем не возникает, и возвращение функцией объекта, созданного с помощью `new`, является здесь, в отличие от C++, обычным делом.

15. Двумерные массивы

Многомерные массивы в C++ конструируются из одномерных. Рассмотрим их создание и использование на примере двумерных массивов.

Описание

```
int a[3][4];
```

вводит массив из трех элементов, каждый из которых имеет тип `int[4]`, т. е. представляет собой одномерный массив из четырех целых. Если первый индекс двумерного массива трактовать как номер строки матрицы, то можно сказать, что двумерный массив хранится в оперативной памяти построчно: вначале первая строка `a[0]`, затем вторая – `a[1]` и затем третья – `a[2]` (рис.1).

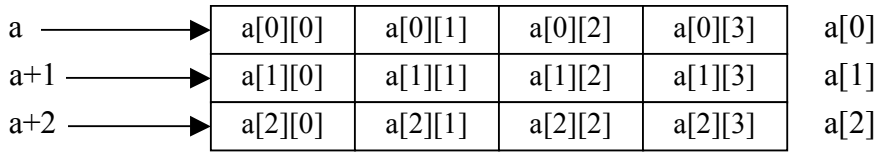

Рис. 1. Двумерный массив: распределение памяти.

Расшифруем выражение `a[1][2]`, воспользовавшись формулой (*) из п.8:

```
a[1][2] == *(*(a+1)+2)
```

Имя массива `a` можно трактовать как указатель на начало одномерного массива с элементами типа `int[4]`, т. е. на его первый элемент `a[0]`. Поскольку арифметика указателей чувствительна к типу, то запись `a+1` трактуется как указатель на второй элемент одномерного массива типа `int[4]`, т. е. как указатель на `a[1]`. Разыменовывая указатель `a+1`, получаем `*(a+1)`, или `a[1]`, представляющий собой одномерный массив элементов типа `int`. Имя этого массива `a[1]` преобразуется к указателю на его первый элемент, т. е. на элемент `a[1][0]`. Если к этому указателю на тип `int` прибавить 2, то `a[1]+2` будет указывать на элемент `a[1][2]`. После разыменования указателя `a[1]+2` мы и получим элемент `a[1][2]`:

```
*(*(a+1)+2) == *(a[1]+2) == a[1][2]
```

Данная расшифровка понадобится нам в п.16, где объясняется, как передавать двумерные массивы в функции.

Задание. Считая, что переменная `a` описана как `a[3][4]`, расшифруйте выражения: `*a`, `**a`, `*a+1`, `(*a)[2]`, `*(a[2])`, `*a[2]`, `*2[a]`, `*1[a+1]`, `&a[0][0]`, `2[a][3]`.

Указание. Оператор индексирования `[]` имеет больший приоритет, чем оператор разыменования `*` и оператор взятия адреса `&`.

16. Массивы указателей на массивы

Двумерные массивы можно создавать также с помощью массивов указателей, инициализируя их элементы адресами одномерных массивов. Например:

```
int b0[4]={1,2,3,4},
    b1[4]={5,6,7,8},
    b2[4]={9,0,1,2};
int* a[3]={b0,b1,b2};
```

В этом случае выражение `a[1][2]` расшифровывается как `*(a[1]+2)`, что в свою очередь есть `*(b1+2)`, или `b1[2]`. Аналогичного эффекта можно добиться, инициализируя массив `a` динамическими одномерными массивами:

```
int* a[3];
for (int i=0; i<3; i++)
  a[i]=new int[4];
```

Сам массив указателей также можно создать в динамической памяти. Он будет контролироваться указателем на тип `int*`, т. е. переменной типа `int**`. В результате мы получим двумерный динамический массив, размерности которого можно задавать при выделении памяти в процессе работы программы:

```
int **a,n,m;
n=3; m=4;
a=new int*[n];
for (int i=0; i<n; i++)
  a[i]=new int[m];
```

При освобождении памяти, занимаемой таким массивом, надо действовать в обратном порядке, вначале освобождая строки, а затем – сам одномерный массив указателей.

```
for (int i=0; i<n; i++)
  delete[] a[i];
delete[] a;
```

К элементам нашего двумерного динамического массива можно обращаться обычным образом: `a[1][2]`. Расшифруем последнее выражение, воспользовавшись изображением полученной в памяти динамической структуры (рис. 2):

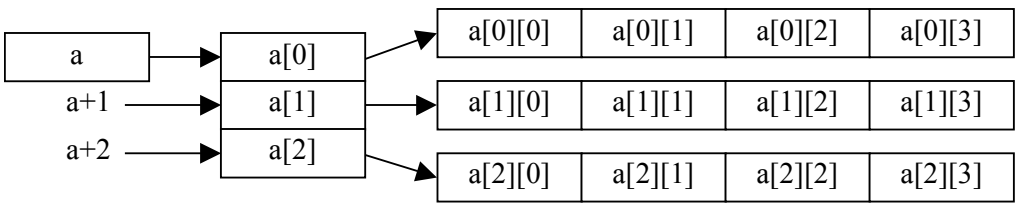

Рис. 2. Массив указателей на массивы. Распределение памяти.

По формуле (*) `a[1][2]==*(*(a+1)+2)`. Но, в отличие от обычного двумерного массива, `a` является указателем не на `int[4]`, а на `int*`. Поэтому `a+1` указывает на следующий элемент типа `int*` в одномерном массиве `a`, т. е. на `a[1]`. Наконец, поскольку `a[1]` имеет тип `int*`, то `a[1]+2` указывает на элемент `a[1][2]`.

Отметим, что, в отличие от обычного двумерного массива, строки нашего динамического массива не обязательно располагаются в памяти последовательно. Именно благодаря этому структура двумерного динамического массива является чрезвычайно гибкой. В частности, для перестановки его строк достаточно поменять местами указатели в одномерном массиве:

```
int* v=a[1]; a[1]=a[2]; a[2]=v;
```

Двумерный динамический массив позволяет также хранить строки разной длины. Например, для создания нижнетреугольной матрицы можно использовать следующий фрагмент:

```
a=new int*[n];
for (int i=0; i<n; i++)
  a[i]=new int[i+1];
```

Одномерный массив указателей может также хранить С-строки, отводя под них столько места, сколько они занимают. Инициализация такого массива при вводе строк из стандартного потока `cin` приводится ниже:

```
char* s[10];
char buf[80];
for (int i=0; i<10; i++)
{
  cin.getline(buf,80);
  s[i]=new char(strlen(buf)+1);
  strcpy(s[i],buf);
}
```

Отметим, что массив С-строк также относится к двумерным динамическим массивам со строками переменной длины. В частности, в алгоритме сортировки при перестановке строк нужно менять местами лишь указатели.

Задание. Реализуйте алгоритм пузырьковой сортировки для массива строк.

17. Двумерные массивы в качестве параметров функций

В этом пункте будут даны ответы на следующие вопросы:

- Как передавать двумерные массивы в функции?
- Имеются ли отличия в способах передачи обычных и динамических двумерных массивов?
- Можно ли передавать в одну и ту же функцию двумерные массивы разных размеров?

Рассмотрим простейший случай:

```
void print(int a[3][4])
{
  for (int i=0; i<3; i++)
  {
    for (int j=0; j<4; j++)
      cout<<a[i][j];
    cout<<endl;
  }
}
int b[3][4];
...
print(b);
```

Подобная функция работает лишь для массивов 3 × 4. Вспоминая, что при передаче в функцию информация о размере одномерного массива утрачивается, мы можем модифицировать предыдущий пример для передачи массивов, имеющих переменный первый размер:

```
void print(int a[][4], int n)
{
  for (int i=0; i<n; i++)
    ...
}
int b[7][4];
...
print(b);
```

Отметим, что информация о втором размере двумерного массива не утрачивается и, более того, существенно необходима для интерпретации внутри функции записи `a[i]`: для вычисления значения выражения `a+i` к содержимому ячейки `a` во внутреннем представлении добавляется величина `i*sizeof(int[4])`. Именно по этой причине никакого способа передавать в функции обычные массивы с переменной второй размерностью *не существует*.

При передаче в функцию динамического двумерного массива подобных проблем не возникает:

```
void print_dyn(int** a, int n, int m)
{
  for (int i=0; i<n; i++)
  {
    for (int j=0; j<m; j++)
      cout<<a[i][j];
    cout<<endl;
  }
}
int** b=new int*[5];
for (int i=0; i<5; i++)
  b[i]=new int[6];
```

```
...
print_dyn(b,5,6);
```

Здесь при трактовке записи `a[i][j]` внутри функции информация о втором размере массива не используется, что позволяет передавать в функцию `print_dyn` динамические массивы произвольных размеров. Отметим, что передавать обычные двумерные массивы в функцию `print_dyn` нельзя.

18. Ссылки в C++

Ссылка в C++ – это альтернативное имя (псевдоним) объекта. Это определение ỳже общего определения ссылки, данного во введении (напомним, что ссылка в широком смысле – это любое имя объекта). Запись `T&` обозначает ссылку на объект типа `T`. Например, в следующем фрагменте

```
int i=1;
int& r=i;
```

объявляется ссылка на переменную `i`, в результате чего как `r`, так и `i` ссылаются на один и тот же объект целого типа. Теперь значение переменной `i` можно изменить через ссылку `r`:

```
r=2; // i получит значение 2
r++; // i получит значение 3
```

В момент объявления ссылка обязательно должна инициализироваться объектом соответствующего типа. После объявления повторная инициализация ссылки невозможна.

Обратите внимание, что для обозначения ссылочного типа используется тот же символ `&`, что и для оператора взятия адреса. Что именно имеется в виду, ясно из контекста: знак `&` после имени типа используется для объявления ссылки, в остальных случаях он обозначает оператор взятия адреса. Так, в результате выполнения оператора

```
cout<<&r;
```

выводится адрес объекта, представляемого ссылкой `r`.

В ситуациях, когда основное имя объекта является составным, ссылка позволяет его сократить:

```
int a[3][4];
int& last=a[2][3];
```

Заметим также, что могут существовать объекты, вообще не имеющие собственного имени; ссылка для них выступает в роли единственного имени и предоставляет единственный способ доступа:

```
int& rd=*new int; // ссылка на безымянный участок
                  // динамической памяти
rd=5;
delete &rd;
```

Ссылку можно представлять себе как константный указатель, который всегда разыменован. Однако необходимо помнить, что, в отличие от указателя,

память под ссылку не выделяется, т. е. сама ссылка не является объектом! Именно поэтому не имеет смысла объявление указателя на ссылку, ссылки на ссылку или массива ссылок:

```
int&& r2=r; // ошибка!
int&* r3=r; // ошибка!
int& a[3];  // ошибка!
```

Ссылка же на указатель является распространенной практикой:

```
int *p;
int*& rp=p;
rp++;
```

19. Ссылки и спецификатор `const`

Как и в случае указателей, `const T&` означает ссылку на константу. В отличие от обычной ссылки `T&`, ее можно инициализировать константным объектом:

```
int i=1;
const int c=5;
int& rv=c;  // ошибка!
int& rv1=5; // ошибка!
const int& rc=c;  // верно
const int& rc1=5; // верно
const int& rc2=i; // верно
```

Отметим, что некоторые компиляторы рассматривают третью и четвертую строки лишь потенциально ошибочными, выдавая вместо сообщения об ошибке предупреждающее сообщение.

Константная ссылка `int& const` смысла не имеет.

Для снятия константности связанных со ссылкой данных следует использовать оператор `const_cast` (см. п. 5):

```
int i=1;
const int& ri=i;
int& ri1=const_cast<int&>(ri);
```

20. Ссылка на объект другого типа

Ссылка на константу `const T&` может инициализироваться выражением или объектом другого типа `T1` (разумеется, если возможно неявное преобразование из `T1` в `T`). В этом случае:

1) если возможно и если необходимо, осуществляется неявное преобразование правой части к типу `T`;
2) создается временная (*temporary*) переменная типа `T`, которая инициализируется полученным значением;
3) ссылка связывается с этой временной переменной.

Например, в ситуации

```
int i=2;
```

```
const double& d=i;
```

произойдет следующее: поскольку возможно неявное преобразование `int` в `double`, то будет создана временная переменная типа `double`, она будет проинициализирована значением `i`, после чего ссылка `d` свяжется с этой переменной. Таким образом, последняя строка кода эквивалентна следующей:

```
const double& d=(double)i;
```

Отметим, что если обычные временные переменные разрушаются сразу после своего использования, то временные переменные, контролируемые ссылками, существуют, пока не закончится время жизни ссылки. Отметим также, что некоторые компиляторы выдают диагностическое сообщение вида «*Temporary used to initialize 'd'*», напоминая о создании временной переменной. Необходимость инициализировать ссылку объектом другого типа возникает, как мы увидим далее, при передаче параметров в функцию.

В отличие от ссылки на константу, ссылка на переменную другого типа интенсивно провоцирует ошибки. Поэтому, хотя стандарт языка этого явно не запрещает, следующий код

```
int i=2;
double& d1=i;
```

многие компиляторы считают ошибочным. Некоторые же компиляторы допускают подобный код, выдавая лишь уже известное предупреждение «*Temporary used to initialize 'd1'*», напоминающее о возможной ошибке. Понятно, что создание временной переменной скорее всего приведет к проблемам: в результате изменения `d1` будет меняться значение именно этой временной переменной, а переменная `i` останется неизменной. Как правило, это не то, чего ожидает программист. Поэтому договоримся считать подобный код ошибочным в любом случае.

21. Ссылки и передача параметров в функции

Ссылки могут использоваться в качестве параметров функции, которая изменяет значение передаваемого ей объекта. Рассмотрим реализацию функции `swap`, приведенной в п.6, с использованием ссылок:

```
void swap(int& a, int& b)
  { int t=a; a=b; b=t; }
...
int i=3,j=4;
swap(i,j);
```

При вызове `swap(i,j)` ссылки `a` и `b` внутри функции становятся новыми именами для переменных `i` и `j`, в результате чего работа происходит именно с самими переменными `i` и `j`, а не с их копиями.

Отметим также, что попытка вызвать функцию `swap` с параметрами другого типа неизбежно приведет к ошибке, детально рассмотренной в конце предыдущего пункта:

```
unsigned u1=5,u2=6;
```

```
swap(u1,u2); // ошибка!
```

Внутренний механизм передачи параметров-ссылок тот же, что и для указателей: в программный стек копируется не значение передаваемого объекта, а указатель на него, и ссылка внутри функции выполняет роль этого указателя, который всегда разыменован.

В следующем примере в качестве параметра используется ссылка на указатель:

```
void next_space(char*& p)
{
  while (*p!=' ' && *p)
    p++;
}
```

Как следует из названия, функция сканирует строку, на которую указывает `p`, в поисках пробела. В результате своей работы она возвращает в `p` либо адрес первого пробела, либо адрес конца строки.

Ссылки на константы широко используются для передачи больших параметров, чтобы избежать копирования значений этих параметров в стек. Например, в функции

```
double det(const Matrix& A) {...}
```

параметр `A` передается как ссылка на константу, что подчеркивает, с одной стороны, его неизменность внутри функции, а с другой – то, что он имеет большой размер и поэтому передается как указатель.

Отметим, что ссылка на константу позволяет безболезненно указывать в качестве фактического параметра объект другого типа: при вызове совершается неявное преобразование к типу формального параметра, соответствующее значение записывается во временную переменную, после чего ссылка связывается с этой безымянной временной переменной. Например, в ситуации

```
void f(const double& d);
...
f(3);
```

будет создана временная переменная типа `double`, в нее будет записано значение `3` (принадлежащее изначально к типу `int`), после чего ссылка `d` внутри функции будет являться псевдонимом этой временной переменной. Напомним, что такая временная переменная будет существовать все время жизни ссылки, т. е. до выхода из функции `f`.

22. Ссылки в качестве возвращаемого значения функции

Функция может возвращать значение типа ссылка. Рассмотрим пример реализации так называемого ассоциативного массива, т. е. массива, индексируемого элементами произвольного типа. Например, для ведения учета товаров на складе хотелось бы использовать код вида:

```
count("монитор")=5;
count("мышь")+=3;
```

```
count("принтер")++;
cout<<count("клавиатура");
```

Характерная особенность этого кода – использование вызова функции в левой части оператора присваивания. Следовательно, функция `count` должна возвращать ссылку на ячейку памяти целого типа.

В простейшем случае ассоциативный массив можно реализовать с помощью обычного массива пар элементов `(name,value)`. При вызове `count(s)` в этом массиве пар ищется пара `(name0,value0)`, в которой `name0` совпадает с `s`, после чего функция возвращает ссылку на второй элемент пары – `value0`. Если же такая пара не найдена, то она создается, добавляется к массиву пар, `value0` инициализируется нулем, а ссылка на эту ячейку возвращается функцией:

```
struct Pair
{
  char* name;
  int value;
};

const max_pairs=100;
Pair pairs[max_pairs];

int n=0;

int& count(char* s)
{
  for (int i=0; i<n; i++)
    if (strcmp(s,pairs[i].name)==0)
      return pairs[i].value;
  pairs[i].name=new char[strlen(s)+1];
  strcpy(pairs[i].name,s);
  pairs[i].value=0;
  return pairs[i].value;
}
```

Отметим, что при выполнении оператора `count("мышь")+=3;` пара `("мышь",value)` может не существовать, в этом случае в конце массива `pairs` создается пара `("мышь",0)`, ссылка на вторую ячейку возвращается функцией, после чего содержимое этой ячейки увеличивается на 3.

Замечание. Недостаток подобной реализации – наличие глобальных переменных `pairs` и `n` для каждого ассоциативного массива в программе. Подобный недостаток можно ликвидировать лишь инкапсуляцией всех данных и функций ассоциативного массива в класс, однако рассмотрение такой возможности выходит за рамки данных методических указаний.

В заключение данного пункта отметим, что, как и для указателей, возвращение ссылки на локальную переменную является грубой ошибкой. Возвраще-

ние же ссылки на динамически распределенную память возможно, но со всеми оговорками, сделанными в п.13.

23. Указатели и ссылки на функции

В программировании нередко возникают ситуации, когда функции необходимо передавать в качестве параметров других функций. С этой целью в языке C++ используются ссылки и указатели на функции. Указатели на функции могут использоваться также для изменения какого-либо действия в ходе выполнения программы.

Начнем с примера. Пусть имеется функция

```
double add(double a, double b)
{ return a+b; }
```

Указатель на такую функцию объявляется следующим образом:

```
double (*pf)(double a, double b);
```

Затем он может быть инициализирован адресом этой функции, после чего функция `add` может быть вызвана косвенно через указатель:

```
pf=&add;
cout<<(*pf)(2,3);
```

Разыменование указателя на функцию при помощи оператора `*` не является обязательным. Также не является обязательным использование оператора `&` для получения адреса функции. Таким образом, предыдущий код можно упростить:

```
pf=add;
cout<<pf(2,3);
```

Для повторного использования типа указателя на функцию удобно присвоить ему имя при помощи директивы `typedef`:

```
typedef double(*FUN)(double a, double b);
```

После этого указатель на функцию можно описать следующим образом:

```
FUN pf=add;
```

Указатели на функции имеют все преимущества обычных указателей. В процессе работы указатель может менять свое значение, обеспечивая тем самым различное поведение программы в зависимости от действий пользователя. Можно также объявить массив указателей на функции:

```
double add(double a, double b);
double sub(double a, double b);
double mul(double a, double b);
double div(double a, double b);

FUN f[4]={add,sub,mul,div};
int n;
double a,b;
cin>>n>>a>>b;
```

```
cout<<f[n](a,b);
```

Ссылка на функцию объявляется аналогично:

```
double (&rf)(double a, double b)=add;
```

Однако, поскольку ссылка, в отличие от указателя, не может в процессе работы поменять функцию, с которой связана, диапазон применения ссылок на функции ограничивается передачей параметров-функций.

Разумеется, указатель или ссылка на функцию может инициализироваться только функциями того же типа, т. е. имеющими те же количество и типы формальных параметров, а также тип возвращаемого значения, что и тип указателя (ссылки).

Замечание. Так как `inline`-функции не являются функциями в собственном смысле этого слова и не имеют адреса, использовать указатели или ссылки на `inline`-функции запрещено.

Для иллюстрации применения указателей и ссылок на функции рассмотрим два примера.

Пример 1. Применение функции, передаваемой в качестве параметра, ко всем элементам диапазона (о понятии диапазона см. п. 8):

```
typedef void (&FUN)(double&);
void for_each(double* a, double* b, FUN f)
{
  while(a!=b)
    f(*a++);
}
```

Пример 2. Сортировка массива с элементами произвольного типа и критерием сравнения, передаваемым в качестве параметра.

В данном примере реализована функция `ssort`, сортирующая методом пузырька массив данных произвольного типа. Количество элементов задается параметром `n`, размер каждого элемента – параметром `sz`, функция сравнения – параметром `cmp`. Поскольку типы элементов заранее неизвестны, указатель на первый элемент передается как `void*`. Внутри функции `ssort` он преобразуется к типу `char*` для возможности работы с адресной арифметикой. Функция `memswap` меняет местами две области памяти размера `sz`. В функцию сравнения передаются два указателя `void*` на сравниваемые элементы массива. Каждая конкретная функция сравнения вначале преобразует эти указатели к нужному типу и затем осуществляет собственно сравнение элементов. В программе, демонстрирующей варианты применения функция `ssort`, реализована сортировка целых чисел (по возрастанию и убыванию), текстовых строк, а также структур (по нескольким полям).

Далее приводится полный текст программы и результаты вывода.

```
typedef int (*CMP)(const void*,const void*);

inline void swap(char& a, char& b)
```

```
{
  char temp=a;
  a=b;
  b=temp;
}

void memswap(char* a, char* b, size_t sz) {
  for (int k=0; k<sz; k++)
    swap(*a++,*b++);
}

void ssort(void* base, size_t n, size_t sz, CMP cmp)
{
  for (int i=1; i<n; i++)
   for (int j=n-1; j>=i; j--)
   {
     char* bj=(char*)base + j*sz;
     if (cmp(bj,bj-sz))
       memswap(bj,bj-sz,sz);
   }
}

struct database
{
  char* name;
  int age;
};

int less_int(const void* p,const void* q)
{ return *(int*)p<*(int*)q; }

int greater_int(const void* p,const void* q)
{ return *(int*)p>*(int*)q; }

int less_str(const void* p,const void* q)
{ return strcmp(*(char**)p,*(char**)q)<0; }

int less_age(const void* p,const void* q)
{ return ((database*)p)->age<((database*)q)->age; }

int less_name(const void* p,const void* q)
{ return strcmp(((database*)p)->name,
                ((database*)q)->name)<0; }

void print (int* mas, int n)
{
  for (int i=0; i<n; i++)
    cout<<mas[i]<<"  ";
  cout<<endl;
```

```
}

void print (char** mas, int n)
{
  for (int i=0; i<n; i++)
    cout<<mas[i]<<"  ";
  cout<<endl;
}

void print (database* mas, int n)
{
  for (int i=0; i<n; i++)
    cout<<mas[i].name<<"  "<<mas[i].age<<endl;
}

const n=10, m=5;
int mas[n]={1,5,2,6,3,7,12,-1,6,-3};
char* strmas[n]={"adg","dfgj","jk","asg","gjh",
          "sdh","hj","sd","kfj","sdadgh"};
database d[m] ={{"Petrov",32},{"Ivanov",24},
                {"Kozlov",21},{"Oslov",20},
                {"Popov",18}};

void main()
{
  ssort(mas,n,sizeof(int),less_int);
  print(mas,n);
  ssort(mas,n,sizeof(int),greater_int);
  print(mas,n);
  ssort(strmas,n,sizeof(char*),less_str);
  print(strmas,n);
  cout<<endl;
  ssort(d,m,sizeof(database),less_age);
  print(d,m);
  cout<<endl;
  ssort(d,m,sizeof(database),less_name);
  print(d,m);
}
```

Результаты вывода:

```
-3  -1  1  2  3  5  6  6  7  12
12  7  6  6  5  3  2  1  -1  -3
adg  asg  dfgj  gjh  hj  jk  kfj  sd  sdadgh  sdh

Popov  18
Oslov  20
Kozlov  21
Ivanov  24
```

```
Petrov  32
Ivanov  24
Kozlov  21
Oslov  20
Petrov  32
Popov  18
```

24. Указания к выполнению заданий

В следующем пункте приведен ряд заданий на указатели. Дадим несколько общих указаний к их выполнению.

Все задания рассчитаны на последовательный перебор элементов массива, поэтому там, где это возможно, вместо доступа по индексу `p[i]` следует использовать более эффективные конструкции вида `*p++`, `*p--`.

Алгоритм выполнения задания должен содержаться в функции, которой передаются все необходимые (входные и выходные) параметры. При этом ввод начальных данных и вывод итогового результата следует осуществлять в основной программе.

Если функция, выполняющая поставленную задачу, имеет больше 20–25 строк кода или состоит из нескольких логических частей, то ее следует разбить на несколько функций. В качестве «кандидатов» на вспомогательные функции выступают, например, заполнение массива слов или поиск начала следующего слова.

Задание должно быть оформлено в виде проекта. Функции, выполняющие поставленную задачу, следует записать в отдельный .cpp-файл и включить его в проект наряду с .cpp-файлом, содержащим функцию `main()`. Файл, содержащий реализацию функций, следует сопроводить соответствующим заголовочным файлом, в который необходимо включить заголовки необходимых функций, описания структур и глобальных констант.

В ряде заданий на строки следует сформировать массив указателей на начала слов (или массив копий слов). Для функции, выполняющей эту подзадачу, рекомендуется следующий заголовок:

```
void words(char* s, char** begs, int& n);
```

Здесь `s` – исходная строка, `begs` – массив указателей на начала слов (или массив копий слов), `n` – количество слов (выходной параметр).

Наиболее тонким является ответ на вопрос, где распределять память под массив `begs`. Основная проблема состоит в том, что изначально не известно количество слов в строке. Поэтому, если требуется максимальная эффективность, и дополнительный проход по строке для подсчета количества слов неприемлем, то память под массив указателей `begs` рекомендуется распределять вне функции. Поскольку заранее не известна заполненность этого массива, он должен иметь достаточный размер, заведомо превосходящий количество слов в строке `s`. Если же время работы алгоритма существенно превосходит время однократного прохода по строке (например, требуется сортировка массива слов),

то перед заполнением массива `begs` следует выполнить проход по строке для подсчета количества слов. После этого память под массив указателей распределяется внутри функции. В любом случае если требуется хранить дубликаты слов, то выделение памяти под них производится внутри функции `words`. В конце работы алгоритма вся затребованная динамическая память должна быть возвращена системе, для чего рекомендуется предусмотреть специальную функцию.

Несомненно, для эффективного решения проблемы с подсчетом количества слов в качестве `begs` следует использовать экземпляр класса `vector` или класса `list` стандартной библиотеки. Однако приведенные задания ориентированы на закрепление навыков работы с указателями, поэтому использование указанных стандартных классов запрещено.

25. Задания

25.1. Задания на строки

1. Вводится строка слов, разделенных пробелами (возможны лишние пробелы в начале и в конце строки и между словами). Циклически сдвинуть все слова влево на k слов, удалив при этом лишние пробелы (k заведомо меньше количества слов).
2. Вводится строка слов, разделенных пробелами (возможны лишние пробелы в начале и в конце строки и между словами). Циклически сдвинуть все слова вправо на k слов, удалив при этом лишние пробелы (k заведомо меньше количества слов).
3. Вводится строка слов, разделенных пробелами (возможны лишние пробелы в начале и в конце строки и между словами). Скопировать в новую строку два самых коротких слова исходной строки. Алгоритм просмотра исходной строки должен быть *однопроходным*.
4. Вводится строка слов, разделенных пробелами (возможны лишние пробелы в начале, в конце строки и между словами). Сформировать новую строку, в которой содержатся все слова-перевертыши (палиндромы) исходной строки. Алгоритм просмотра исходной строки должен быть *полуторапроходным* (полпрохода на проверку того, является ли слово перевертышем).
5. Вводится строка слов, разделенных пробелами (возможны лишние пробелы в начале и в конце строки и между словами), а также целочисленный массив перестановок слов. По данной строке и массиву перестановок сформировать новую строку, удалив при этом лишние пробелы. Например, если задана строка ” aa bbb c dd eeee” и массив перестановок 5 2 4 3 1, то итоговая строка должна иметь вид: ”eeee bbb dd c aa”.
   Указание: вначале сформировать массив указателей на начала слов.
6. Вводится строка слов, разделенных пробелами (возможны лишние пробелы в начале и в конце строки и между словами). Сформировать строку, в которой слова из исходной строки упорядочены по алфавиту, удалив при этом лишние пробелы.

Указание: для сравнения строк можно воспользоваться библиотечной функцией `strcmp(s,s1)`.

7. Вводится строка слов, разделенных пробелами (возможны лишние пробелы в начале, в конце строки и между словами). Сформировать строку, в которой слова из исходной строки упорядочены по длине (а при равной длине порядок их следования остается таким же, как и в исходной строке), удалив при этом лишние пробелы.
8. Вводится строка слов, разделенных пробелами (возможны лишние пробелы в начале, в конце строки и между словами). Сформировать строку, в которой слова из исходной строки упорядочены по количеству гласных (а при равном количестве гласных порядок их следования остается таким же, как и в исходной строке), удалив при этом лишние пробелы.
9. Вводится строка слов, разделенных пробелами (возможны лишние пробелы в начале и в конце строки и между словами). Сформировать строку, в которой удалены лишние пробелы и повторявшиеся ранее слова. Порядок слов не менять.
10. Вводится строка слов, разделенных пробелами (возможны лишние пробелы в начале и в конце строки и между словами). Сформировать строку, в которой слова упорядочены по повторяемости. Дубликаты слов следует удалить. При одинаковой повторяемости первым должно следовать слово, первое вхождение которого встречается раньше в исходной строке.
11. Вводится строка слов, разделенных пробелами (возможны лишние пробелы в начале и в конце строки и между словами). Выдать таблицу слов и количество их повторений в строке. Дубликаты слов не выдавать.
12. Вводится строка. Заменить в ней все цифры их словесными обозначениями: ”0” на ”zero”, ”1” на ”one”, ”2” на ”two” и т.д.
13. Удалить из строки `s` каждое вхождение подстроки `s1`.
14. Заменить в строке `s` каждое вхождение подстроки `s1` на подстроку `s2`.
15. Даны две строки. Найти индексы (их может быть несколько) и длину самого длинного одинакового участка в обоих массивах. Например, в строках “abracadabra” и “sobrat” самым длинным одинаковым участком будет “bra”.
16. Дан массив строк. Сформировать по нему массив подстрок, удовлетворяющих условию, передаваемому в качестве параметра (условие должно представлять собой функцию, принимающую параметр `char*` и возвращающую значение логического типа).
17. Реализовать функцию

```
char *mystrstr(const char *p, const char *q);
```

возвращающую первое вхождение подстроки `q` в строку `p` (или 0, если подстрока не найдена).

18. Реализовать функцию

```
char *mystrpbrk(const char *p, const char *q);
```

возвращающую указатель на первое вхождение в строку `p` какого-либо символа из строки `q` (или 0, если совпадений не обнаружено).

19. Реализовать функцию

```
size_t mystrspn(const char *p, const char *q);
```

возвращающую число начальных символов в строке `p`, которые не совпадают ни с одним из символов из строки `q`.

20. Реализовать функцию

```
size_t mystrcspn(const char *p, const char *q);
```

возвращающую число начальных символов в строке `p`, которые совпадают с одним из символов из строки `q`.

21. Реализовать функцию

```
char* mystrtok(const char *p, const char *q, char* t);
```

пропускающую символы разделителей, хранящихся в строке q, считывающую первую лексему в строке p в строку t (до следующего символа разделителя или до конца строки) и возвращающую указатель на первый непросмотренный символ.

25.2. Задания на массивы

22. Дан массив целых. Оформить функцию `count_if`, вычисляющую количество элементов в массиве, удовлетворяющих данному условию, передаваемому в качестве параметра (условие должно представлять собой функцию, принимающую параметр типа `int` и возвращающую значение логического типа).
23. Дан массив целых. Заполнить его, передавая в качестве параметра функцию, задающую алгоритм генерации следующего значения, вида `int f()`. Для генерации данная функция может запоминать значения, сгенерированные на предыдущем шаге, либо в глобальных переменных, либо в статических локальных переменных.
24. Дан массив целых, отсортированный по возрастанию. Удалить из него дубликаты.
25. Дан массив целых. Составить функцию `remove_if`, удаляющую из него все элементы, удовлетворяющие условию, передаваемому в качестве параметра.
26. Дан массив чисел и число `a`. Переставить элементы, меньшие `a`, в начало, меняя их местами с предыдущими. Порядок элементов, меньших `a`, а также порядок элементов, больших `a`, не менять.
27. Дан массив целых. Найти в нем пару чисел `a` и `b` с минимальным значением `f(a,b)`, где `f` передается в качестве параметра.
28. Дан массив целых. Составить функцию `accumulate`, применяющую функцию `f(s,a)`, передаваемую в качестве параметра, к каждому элементу `a` массива и записывающую результат в переменную `s`. С ее помощью найти минимальный элемент в массиве, сумму и произведение элементов массива.
29. Дан массив целых. Сформировать по нему массив, содержащий длины всех серий (подряд идущих одинаковых элементов). Одиночные элементы считать сериями длины 1.
30. Дан массив целых. Из каждой серии удалить один элемент.

31. Дан массив целых. Удалить все серии, длина которых меньше k.
32. Слить `n` массивов целых, упорядоченных по возрастанию, в один, упорядоченный по возрастанию.
    Указание 1. Для упрощения алгоритма следует записать в конец каждого массива *барьер* – самое большое число соответствующего типа. Барьер, в частности, будет определять, где заканчиваются данные в массиве.
    Указание 2. Составить функцию, в которую передать динамический двумерный массив (массив одномерных массивов чисел) и число `n` одномерных массивов. Рекомендуется во внешнем цикле завести счетчик одномерных массивов, в которых достигнут барьер. В тот момент, когда он становится равным `n`, слияние окончено. Другой вариант: при записи в результирующий массив проверять значение элемента; если оно равно значению барьера, то слияние окончено.
33. Дан массив чисел. Оформить функцию `partition`, перетаскивающую его элементы, удовлетворяющие данному условию, в начало, меняя их местами с предыдущими. Порядок элементов, удовлетворяющих условию, не менять. Условие передавать как параметр-функцию. Например, числовой массив 6 2 9 4 7 3 1 8 5 после применения к нему функции `partition` с условием «элемент > 5» будет выглядеть так: 6 9 7 8 2 4 3 1 5.
34. Дан массив чисел. По нему сконструировать массив сумм соседних элементов, по нему – еще один массив сумм и т.д. – до массива из одного элемента. Результат должен храниться в массиве указателей на одномерные массивы.

Литература

1. Б. Страуструп. Язык программирования C++. Третье издание. М. Бином. 1999.
2. С. Липпман, Ж. Лажойе. Язык программирования C++. Вводный курс. Третье издание. М. ДМК. 2001.