Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Владимирский государственный университет
имени Александра Григорьевича и Николая Григорьевича Столетовых»

М. В. ШИШКИНА

ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ ВЫСОКОГО УРОВНЯ С++

Практикум

Владимир 2022

УДК 004.42
ББК 32.973-018
Ш55

Рецензенты:
Кандидат технических наук
генеральный директор ООО «ФС Сервис»
Д. С. Квасов

Кандидат физико-математических наук, доцент
зав. кафедрой функционального анализа и его приложений
Владимирского государственного университета
имени Александра Григорьевича и Николая Григорьевича Столетовых
В. Д. Бурков

Издаётся по решению редакционно-издательского совета ВлГУ

Шишкина, М. В.

Ш55 Программирование на языке высокого уровня C++ : практикум / М. В. Шишкина ; Владим. гос. ун-т им. А. Г. и Н. Г. Столетовых. – Владимир : Изд-во ВлГУ, 2022. – 104 с.

ISBN 978-5-9984-1462-6

Изложены теоретические основы языка программирования C++, представлено достаточное количество примеров, приведены задания для лабораторных и самостоятельных работ, а также вопросы для самоконтроля и закрепления материала.

Предназначен для студентов бакалавриата направлений подготовки 01.03.02 «Прикладная математика и информатика», 02.03.02 «Математическое обеспечение и администрирование информационных систем», 02.03.01 «Математика и компьютерные науки».

Рекомендовано для формирования профессиональных компетенций в соответствии с ФГОС ВО.

Ил. 28. Табл. 2. Библиогр.: 5 назв.

УДК 004.42
ББК 32.973-018

ISBN 978-5-9984-1462-6

ПРЕДИСЛОВИЕ

Изучение программирования на современных языках высокого уровня вызывает большой интерес среди молодёжи, выбирающей свой профессиональный путь. Эта востребованность объясняется стремительным развитием информационных технологий.

Глубокое погружение в профессию, связанную с информационными технологиями, как и её полномерное освоение невозможны без изучения основ программирования, в том числе на востребованном современном языке программирования высокого уровня *C++*. Знание основ языка программирования *C++* открывает перед студентами возможности его дальнейшего освоения, постижения его нюансов. Углублённое изучение программирования позволяет понять суть языка, выбрать наиболее подходящие средства решения задачи в дальнейшей самостоятельной работе. При изучении языка программирования важен практический подход. Для его успешного освоения необходимо решение большого количества задач по каждой теме. Не менее значимы навыки понимания уже написанного другими разработчиками кода, поиска и исправления синтаксических и семантических ошибок в программном коде.

Практикум содержит задания для лабораторных и самостоятельных работ, вопросы для самоконтроля, входящие в состав учебно-методического комплекса дисциплины «Основы программирования», разработанного на базе современных федеральных государственных образовательных стандартов подготовки бакалавров по направлениям 01.03.02 «Прикладная математика и информатика», 02.03.02 «Математическое обеспечение и администрирование информационных систем», 02.03.01 «Математика и компьютерные науки».

Практикум разработан в соответствии с рабочими программами дисциплины «Основы программирования» и может быть рекомендован в качестве основного пособия для освоения дисциплины «Основы программирования».

ВВЕДЕНИЕ

Практикум позволяет повторить и углубить знания, полученные в ходе освоения школьного курса информатики и ИКТ и начала изучения дисциплины «Основы программирования».

Практикум содержит девять разделов, позволяющих закрепить ранее изученный материал, подготовиться к выполнению и защите лабораторных работ по курсу «Основы программирования».

Первый раздел предназначен для повторения ранее пройденного материала, закрепления навыков, полученных в начале изучения курса. Эта часть пособия содержит достаточное количество примеров и заданий для самостоятельной работы, а также контрольных вопросов, позволяющих закрепить полученные ранее знания и навыки, необходимые для дальнейшего успешного освоения материала.

Каждый следующий раздел содержит достаточно подробное изложение теоретического материала, сопровождаемое примерами, разъяснением приведённого кода и графической интерпретацией примеров по изучаемой теме; а также цель и постановку задачи лабораторной работы, контрольные вопросы и задания для самостоятельной работы.

Контрольные вопросы, приведённые в конце каждого раздела, составлены таким образом, что позволяют студенту повторить и закрепить материал изучаемой темы, самостоятельно оценить уровень своих знаний и подготовиться к защите лабораторной работы.

Задания для самостоятельных работ направлены на закрепление знаний и навыков, полученных студентами во время аудиторных занятий.

Структура практикума организована таким образом, что каждый следующий раздел основан на знаниях и навыках, полученных при выполнении предыдущей работы.

В приложениях содержатся требования к отчёту по лабораторной работе, рекомендации по форматированию отчёта и некоторые справочные материалы.

1. ОСНОВЫ ЯЗЫКА ПРОГРАММИРОВАНИЯ *C++*

Цель изучения любого языка программирования – использование его для дальнейшего решения профессиональных задач. Поэтому необходимо вспомнить, что любой алгоритм может быть представлен с использованием трёх алгоритмических конструкций: *следование*, *ветвление*, *цикл*. Структурные схемы основных алгоритмических конструкций приведены в прил. 1. Перед тем как приступить к реализации алгоритма в виде кода на языке программирования, необходимо представить алгоритм в виде блок-схемы. Грамотно составленная блок-схема – залог успешного написания кода. Основные блоки в соответствии с ГОСТом представлены в прил. 2.

Перед написанием кода необходимо определить назначение и, следовательно, тип переменных и констант, так как от типа данных зависят диапазон возможных значений и правила работы с ними. Базовые типы данных языка программирования *C++* представлены в прил. 3.

Для реализации алгоритмической конструкции «ветвление» в языке программирования *C++* используют операторы `if`, `switch` и тернарную операцию. Далее приведён синтаксис и примеры каждой из упомянутых конструкций языка.

Оператор ветвления `if`

`if` (*выражение*) *оператор1*; [`else` *оператор2*;]

Полная форма оператора `if`

```
if (a > b) max = a; else max = b;
```

Неполная форма оператора `if`

```
if (a > b) max = a;/* */
```

Использование составного выражения в условии

```
if (a<b &&(a>d || a==0)) b++; else {b* = a; a++;}
```

Оператор множественного ветвления `switch`

`switch` (*<целочисленное выражение>*)
{
`case` *константное выражение1*: *оператор1*;
`case` *константное выражение2*: *оператор2*;

`case` *константное выражение3*`:` *оператор3*`;`

…

`case` *константное выражение N – 1*`:` *оператор N – 1*`;`

`[default:` <*операторN*>`]`

`}`

Пример использования оператора множественного ветвления:

```
int i = 2;
switch (i)
{
    case 1:printf("1\n");
    case 2:printf("2\n");
    case 3:printf("3\n");
    case 4:printf("4\n");
    default:printf("No!\n");
    ;
    }
```

Тернарная операция

Тернарная операция позволяет реализовать ветвление и вернуть значение выполненного оператора в точку вызова

`(`<*операнд1*>`)?`<*операнд2*>`:`<*операнд3*>`;`

Пример использования тернарной операции:

```
int a = 7;
int b = 8;
int c = (a < b) ? a++: b++;
```

Для организации цикла в языке программирования *C++* используют операторы `for`, `while`, `do while`.

Цикл с предусловием `while`

Синтаксис оператора `while` определён следующим образом:

`while (`*выражение*`)` *оператор*`;`

Пример использования оператора `while`:

```
int i = 1;
```

```
while (i<=n)
{   printf("a=");
    scanf_s("%i",&a);
    s = s + a; i++;
}
```

Цикл с постусловием `do while`

Синтаксис оператора `while` определён следующим образом:

`do` <*оператор*> `while` (*выражение*)`;`

Пример использования оператора `do while`:

```
do
    {
    printf("a=");
    scanf_s("%i", &a);
    s = s + a; i++;
    } while (i < n);
    printf("s=%i\n",s);
```

Цикл с параметром `for`

Этот оператор позволяет реализовать цикл с предусловием и условием продолжения.

Правило записи оператора следующее:

`for` (*выражение1*`;` *выражение2*`;` *выражение3*) *оператор*`;`

Пример использования оператора `for:`

```
for (int i = 0; i < n; i++)
{
    printf("a=");
    scanf_s("%i", &a);
    s = s + a;
}
```

Важный момент при изучении языка программирования *C++* – работа с указателями. Особое внимание стоит уделить работе с указуемым через указатель.

Для объявления указателя необходимо указать тип данных указуемого и символ * (звёздочка) перед именем указателя

<тип данных><имя указателя>*;

```
int a = 7;
int* p = &a;
int** pp = &p;
**pp = 9;
```

Для работы с указуемым, т. е. для перехода по адресу, необходима операция *разадресации* (разыменовывания). Для этого нужно использовать символ * перед уже объявленным и проинициализированным указателем

```
*p=5;
```

Употребление символа * перед идентификатором при объявлении означает, что объявлен указатель.

Употребление символа * перед идентификатором объявленного ранее указателя означает разыменовывание указателя, т. е. обращение к указуемому.

Строка `*p=5;` означает запись в переменную `a` значения целочисленной неименованной константы `5`.

Под ***ссылкой*** понимают альтернативное имя объекта. Таким образом, обратиться к переменной можно по имени или через ссылку на эту переменную. Ссылку можно рассматривать как константный разыменованный указатель, т. е. указатель, который нельзя перенастроить, но через него можно работать с указуемым

<тип данных> `&` *<имя ссылки>* `=` *<переменная>*`;`

```
int a=10;
int b=5;
int &ref=a; //объявление ссылки ref и её связывание с переменной a.
```

Для работы с однотипными данными, объединёнными под одним именем, используют массивы.

Под ***массивом*** в программировании понимают последовательность однотипных элементов, расположенных в памяти подряд и объединённых под одним именем.

Правило объявления массива в языке программирования *C++* следующее:

<тип элементов массива> <имя массива>[*<количество элементов>*];

Индексация элементов массива начинается с нуля, поэтому номер последнего элемента массива на единицу меньше общего количества элементов массива.

Например, массив можно объявить так:

```
int mas [5];
```

Или так, задав предварительно количество элементов константой, что предпочтительнее, так как теперь при необходимости изменить количество элементов массива достаточно изменить значение константы в одной строчке кода:

```
int const N=5;
int mas [N]={1,-2,3,-4,5};
```

В приведённом примере объявление совмещено с инициализацией.

Примеры обращения к элементам массива:

```
mas[3]=95;
*(mas+3)=8;
*(mas+i)=i+i;
```

Для объявления многомерного массива необходимо указать каждое его измерение в квадратных скобках после имени массива

<тип элементов имя массива> [*<целочисленная константа*1>][*<целочисленная константа2>*];

Например, так объявлен двумерный массив:

```
int const M = 5;
int const N = 3;
int mas2[N][M];
```

А так объявлен трёхмерный массив:

```
int mas3[N][M][N];
```

Выделить динамически память под переменную или массив данных можно при помощи операции new

new <*тип данных*> [<*количество элементов*>];

Оператор new возвращает в точку вызова типизированный указатель на начало захваченной области памяти, дальнейшая работа с захваченной памятью ведётся через это значение

```
int* p = new int(15);
```

Динамическая память, захваченная при помощи new, должна быть освобождена при помощи delete

```
delete[] <указатель>;
delete p;
delete []pM;
```

Для инициализации элементов многомерного массива при объявлении необходимо указать элементы в фигурных скобках списком через запятую, если размерность указана в квадратных скобках

```
int mas2[3][2] = {1,2,3,4,5,6};
```

Либо так:

```
int mas2[3][] = { {5,8},{3,4},{1,2} };
```

Следующий пример показывает реализацию форматированного вывода элементов двумерного массива

```
for (int i = 0; i < 3; i++)
{
for (int j = 0; j < 2; j++)printf("%i ", mas2[i][j]);
printf("\n");
}
```

Рассмотрим вариант создания двумерного массива в динамической памяти

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

Пользовательский тип данных – это тип данных, который программист описывает по определённым правилам, исходя из условий решаемой задачи.

Структура – это пользовательский тип данных, позволяющий объединить разнотипные данные под одним именем.

Эти данные в рамках структуры называются *полями*.

Описание структуры начинается с ключевого слова `struct`, далее следует имя описываемого типа. Описание полей структуры заключается в фигурные скобки; заканчивается описание точкой с запятой.

Важно не путать имя типа структуры и имена переменных описанного типа

```
struct имя типа {описание полей};
struct ST_Name
{
    int pole1;
    float pole2;
    char pole3[10];
};
```

Объявить переменную описанного типа «структура» можно по тем же правилам, что и переменную любого другого типа, т. е. указав имя типа и имя переменной

<имя типа> <имя переменной>;
```
ST_Name primer1, primer2;
```

Переменные типа «структура» объявить можно сразу после описания типа, до точки с запятой

```
struct Struct_Name
{
    int pole_1;
    float pole_2;
    char pole_3[10];
} ST1;
```

Инициализация полей структуры возможна при объявлении переменной; значения полей необходимо передать списком в фигурных скобках, разделив значения запятой

```
struct Struct_Name
{
    int pole_1;
```

```
        float pole_2;
        char pole_3[10];
    }St1{-1, 2.3, "Name"};
    Struct_Name primer1{ 10,2.4,"name1" }, primer2 =
{10,2.4,"name1"};
```

Для обращения к полям структуры необходимо указать имя переменной типа «структура» и через точку – имя соответствующего поля

<имя переменной>.*<имя поля>*

```
primer1.pole1 = 9;
```

Строку можно рассматривать как массив символов, заканчивающийся признаком конца строки “\0”. Поэтому под элементы массива нужно выделить на один символ больше общего количества символов строки.

Проинициализировать объявленную строку можно при объявлении одним из двух продемонстрированных в примере способов

```
char str[5]="abcd";
char str1[5] = { 'a','b','c','d','\0' };
```

Так как строка – это массив, то возможна динамическая инициализация и работа со строкой через указатель

```
char* p_str = new char[5];
```

В библиотеке *C++* описан класс `string`, обеспечивающий работу со строками. В этом классе реализованы такие операции, как сравнение, присвоение, конкатенация, индексация, поиск подстроки и др. Ниже приведены примеры работы со строками

```
string s;
string s1 = "stroka1";
s = s1 + " + stroka2";
```

Значение переменной `s` равно `"stroka1 + stroka2"`.

```
string s2 = "adcb";
s1 = "bbbbbbb";
string ss;
(s2 > s1) ? ss=s2 : ss=s1;
```

Значение `ss` равно `"bbbbbbb"`

```
char c1[] = "abc";
char c2[] = "aaaaa";
int k = strcmp(c2,c1);
```

Функция `strcmp` возвращает целочисленное значение, это значение будет больше нуля, если первая строка больше второй; ноль, если строки равны; меньше нуля, если первая строка меньше второй.

При этом производится сравнение не длины строк, а самих строк в лексикографическом порядке.

При таком подходе строка “*aaaaaa*” меньше строки “*bc*”

`cin.getline(c1,3); //`ввод с клавиатуры значения строки *c1*

`strcpy_s(c2, c1); /*`в строку *c2* будет скопировано значение из строки *c1*`*/`

Далее приведены вопросы для самоконтроля. Без знания ответов на эти вопросы дальнейшее изучение языка не будет эффективным.

Вопросы для самоконтроля и подготовки к защите лабораторной работы

1. Что такое алгоритм?
2. Назовите основные алгоритмические структуры.
3. Назовите виды алгоритмов.
4. Перечислите базовые типы языка программирования *C++*.
5. Напишите синтаксис объявления переменной в языке программирования *C++*.
6. Назовите операторы, с помощью которых можно организовать ветвление в языке программирования *C++*.
7. Напишите синтаксис и пример использования оператора `if` в языке программирования *C++*.
8. Напишите синтаксис и пример использования оператора `switch` в языке программирования *C++*.
9. Напишите синтаксис и пример использования тернарной операции в языке программирования *C++*.
10. Назовите операторы, с помощью которых можно организовать цикл в языке программирования *C++*.
11. Напишите синтаксис и пример использования оператора `for` в языке программирования *C++*.

12. Напишите синтаксис и пример использования оператора `while` в языке программирования *C++*.

13. Напишите синтаксис и пример использования оператора `do while` в языке программирования *C++*.

14. Поясните разницу в работе постфиксного и префиксного инкрементов в языке программирования *C++*.

15. Что такое указатель в программировании?

16. Напишите синтаксис объявления и инициализации указателя в языке программирования *C++*.

17. Назовите операции, допустимые при работе с указателями; поясните их смысл.

18. Что такое ссылка? Приведите пример работы со ссылками в языке программирования *C++*; поясните смысл примера.

19. Что такое массив?

20. Напишите синтаксис и приведите пример объявления одномерного статического массива данных.

21. Назовите способы обращения к элементам массива; приведите примеры.

22. Напишите синтаксис объявления двумерного и трёхмерного статических массивов данных; приведите пример.

23. Назовите известные вам алгоритмы сортировки данных; поясните их суть.

24. Каким образом можно в языке программирования *C++* организовать работу с динамическими массивами данных?

25. Какие типы данных называют пользовательскими?

26. Что такое структура (тип данных) в языке программирования *C++*?

27. Напишите синтаксис описания структуры, объявления переменной описанного типа и обращения к полям структуры; приведите пример.

28. Что такое строка с точки зрения языка программирования *C++*?

29. Приведите примеры создания строки в языке программирования *C++*.

30. Приведите примеры обращения к элементам строки.

Для закрепления теоретических навыков на практике рекомендуется самостоятельно выполнить приведённые ниже задания.

Задания для самостоятельной работы

1. Дано натуральное число n. Найти сумму первой и последней цифры этого числа.

2. Дано натуральное число n. Переставить местами первую и последнюю цифры этого числа.

3. Даны два натуральных числа m и n ($m \le 9999$, $n \le 9999$). Проверить, есть ли в записи числа m цифры, совпадающие с цифрами в записи числа n.

4. Дано натуральное число n, $n \le 9999$. Проверить, есть ли в записи числа три одинаковые цифры.

5. Дано натуральное число n, $n \le 99$. Дописать к нему цифру k в конец и начало.

6. Даны натуральные числа n, k. Проверить, есть ли в записи чисел n и k цифра m.

7. Вычислить $y = a\sin(x)$. Для всех a и x, заданных значениями a_0 и x_0 и шагом по a и x соответственно.

8. Вычислить $y = \cos 2(x) + ac$. Для всех a и x, заданных значениями a_0 и x_0 и шагом по a и x соответственно.

9. Вычислить $z = 2\prod_{i=1}^{k}\left(a_i - \frac{1}{a_i}\right)$, где k задано, a_i задаёт пользователь вводом с клавиатуры.

10. Объявить вещественные переменные x, y, z и S. Записать фрагмент кода, присваивающий переменным x, y и z значения, вводимые с клавиатуры. Вычислить квадрат суммы трёх введённых вещественных чисел x, y, z. Вывести результат на экран.

11. Ввести значение угла в градусах, предварительно объявив соответствующую переменную. Вычислить и вывести на экран значения *cos*, *sin* и *tg* этого угла. Вывести эти значения на экран.

12. Объявить вещественную переменную x. Ввести значение переменной с клавиатуры такое, что $x > 0$. Вычислить десятичный и натуральный логарифмы введённого значения. Вывести результат на экран.

13. Объявить константу x, задать ей значение 25.5, ввести с клавиатуры значения переменной y, вычислить значение переменной f по формуле $f = 2x + x^y$. Вывести результат на экран.

14. Объявить вещественные переменные *x, y, z* и *S*. Написать программный код, реализующий следующие действия: присвоение переменным *x, y* и *z* значений, вводимых с клавиатуры; вычисление квадрата разности трёх введённых вещественных чисел $S = (x - y - z)^2$. Вывести результат на экран.

15. Даны два целых числа, определить меньшее из них.

16. Определить, является ли данный год високосным. (Год является високосным, если его номер кратен 4, но не кратен 100, а также если он кратен 400.)

17. Ввести целое число *x*, вывести значение sign*(x)*.

Функция sign*(x)* (знак числа) определена так:

sign*(x)* = 1, если $x > 0$,

sign*(x)* = –1, если $x < 0$,

sign*(x)* = 0, если $x = 0$.

18. Даны два целых числа, каждое записано в отдельной строке. Вывести число 1, если первое число больше второго; число 2, если второе больше первого; число 0, если они равны.

19. Даны три целых числа. Вывести наибольшее из данных чисел.

20. Определить, бьёт ли ладья, стоящая на клетке с указанными координатами (номер строки и номер столбца), фигуру, стоящую на другой указанной клетке. Дано четыре целых числа в интервале от 1 до 8: координаты ладьи (два числа) и координаты другой фигуры (два числа). Вывести слово *YES*, если ладья сможет побить фигуру за один ход и *NO* – в противном случае. (Шахматная ладья может ходить на любое количество клеток по горизонтали или вертикали.)

21. Определить, бьёт ли слон, стоящий на клетке с указанными координатами (номер строки и номер столбца), фигуру, стоящую на другой указанной клетке. Дано четыре целых числа в интервале от 1 до 8: координаты слона (два числа) и координаты другой фигуры (два числа). Вывести слово *YES*, если ладья сможет побить фигуру за один ход, и *NO* – в противном случае. (Шахматная фигура слон может ходить на любое количество клеток по горизонтали.)

22. Определить, бьёт ли ферзь, стоящий на клетке с указанными координатами (номер строки и номер столбца), фигуру, стоящую на

другой указанной клетке. Вводятся четыре числа: координаты ферзя и координаты другой фигуры. Координаты – целые числа в интервале от 1 до 8. Вывести слово *YES*, если ферзь может побить фигуру за один ход, и *NO* – в противном случае. (Шахматная фигура ферзь может ходить в любом направлении на любое количество клеток.)

23. Поле шахматной доски определяется парой чисел (a, b), каждое от 1 до 8, первое число задаёт номер столбца, второе – номер строки. Заданы две клетки. Определить, может ли шахматный король попасть с первой клетки на вторую за один ход. (Шахматный король может перемещаться за один ход на одну клетку в любом направлении.)

24. Определить, бьёт ли конь, стоящий на клетке с указанными координатами (номер строки и номер столбца), фигуру, стоящую на другой указанной клетке. Вводятся четыре числа: координаты коня и координаты другой фигуры. Координаты – целые числа в интервале от 1 до 8. Вывести слово *YES*, если конь может побить фигуру за один ход, и *NO* – в противном случае. (Шахматная фигура конь может ходить в форме буквы Г в любом направлении.)

25. Составить программу, выводящую на экран квадраты чисел от 10 до 20 включительно.

26. Даны натуральные числа от 35 до 87. Вывести на консоль те из них, которые при делении на 7 дают остаток 1, 2 или 5.

27. Найти сумму 1 + 2 + 3 +…+ n, где число n вводится пользователем с клавиатуры.

28. Найти произведение цифр трёхзначного числа.

29. Найти количество чётных цифр данного натурального числа.

30. Найти наибольшую цифру данного натурального числа.

31. Найти все четырёхзначные числа, сумма цифр каждого из которых равна 15.

32. Составить таблицу значений функции $y = 5 – x^2/2$ на отрезке [–5; 5] с шагом $h = 0.5$ изменения x.

33. Вводится натуральное число. Найти сумму чётных цифр, входящих в его состав.

34. Вводится целое число *A*. Преобразовать его в число *B*, цифры которого будут следовать в обратном порядке по сравнению с введённым числом.

35. Объявить указатель на вещественное число, настроить его на переменную соответствующего типа и увеличить значение этой переменной вдвое, обращаясь к ней через указатель.

36. Объявить константный указатель на вещественную константу. Настроить его на соответствующую константу, предварительно объявив и задав её значение равным значению ускорения свободного падения. Объявить и проинициализировать вещественные переменные *m* и *h*, объявить два вещественных указателя и настроить их на переменные *m* и *h*. Вычислить значение $E_p = mgh$, обращаясь к переменным *m*, *h* и константе *g* через соответствующие указатели.

37. Объявить вещественную переменную *f*, указатель на вещественную переменную *pf*, константный указатель на указатель на вещественную переменную *cp*, проинициализировать объявленные переменные и константу *cp*, связав их в единую цепочку. Изменить значение переменной *f*, обращаясь к ней через указатель *cp*.

38. Объявить и проинициализировать следующую цепочку данных: указатель на указатель на переменную типа `char`. Вывести на экран значение символа, предварительно изменив его значение, обе операции нужно выполнить, обращаясь к переменной типа `char` через указатель на указатель.

39. Объявить и проинициализировать следующую цепочку данных: константный указатель на константный указатель на вещественную константу. Вывести значение вещественной константы на экран, обращаясь к ней через указатель на указатель.

40. Объявить и проинициализировать две целочисленные переменные. Объявить указатель и ссылку на целое, настроить на одну из объявленных переменных, ссылку связать со второй переменной. Объявить третью целочисленную переменную, сохранить в неё сумму первых двух объявленных переменных. К слагаемым обращаться через указатель и ссылку соответственно.

41. Объявить три целочисленные переменные. Объявить ссылки на эти переменные. Проинициализировать две переменные значениями, введёнными с клавиатуры. В третью переменную записать удвоенную разность квадратов первых двух переменных. Все операции выполнять, обращаясь к переменным через ссылки.

42. Объявить и проинициализировать две целочисленные переменные. Объявить указатель и ссылку на целое, настроить на одну из объявленных переменных, ссылку связать со второй переменной. Увеличить значение переменных вдвое, обращаясь к ним через указатель и ссылку соответственно.

43. Объявить и проинициализировать две целочисленные переменные. Объявить два указателя на целочисленные переменные и настроить их на объявленные переменные. Поменять значения переменных местами, обращаясь к ним через указатели.

44. Объявить и проинициализировать массив из десяти целых чисел. Объявить указатель на целочисленную переменную и настроить его на минимальный элемент массива.

45. Объявить одномерный целочисленный массив из десяти элементов, проинициализировать его в цикле вводом с клавиатуры. Объявить указатель на целое и настроить его на минимальный элемент массива.

46. Объявить одномерный целочисленный массив из десяти элементов, проинициализировать его в цикле вводом с клавиатуры. Объявить указатель на целое и настроить его на максимальный элемент массива.

47. Объявить одномерный целочисленный массив из десяти элементов, проинициализировать его в цикле вводом с клавиатуры. Вычислить квадрат разности максимального и минимального значений элементов массива.

48. Объявить одномерный целочисленный массив из N элементов, проинициализировать его любым способом. Заменить на ноль все элементы массива, значение которых повторяется более одного раза.

49. Объявить одномерный целочисленный массив из N элементов, проинициализировать его любым способом. Найти минимальный по модулю элемент этого массива.

50. Объявить одномерный целочисленный массив из N элементов, проинициализировать его любым способом. Найти и вывести на экран сумму индексов максимального и минимального элементов.

51. Объявить одномерный целочисленный массив из N элементов, проинициализировать его любым способом. Найти и вывести на экран наименьший чётный элемент массива. Если такого нет, вывести первый элемент.

52. Объявить одномерный целочисленный массив из N элементов, проинициализировать его любым способом. Найти сумму элементов массива между двумя первыми нулями. Если двух нулей в массиве нет, то вывести ноль.

53. Объявить одномерный целочисленный массив из N элементов, проинициализировать его любым способом. Вычислить произведение элементов массива с нечётными номерами.

54. Объявить одномерный целочисленный массив из N элементов, проинициализировать его любым способом. Вычислить сумму отрицательных элементов массива.

55. Вычислить сумму элементов двумерного массива.

56. Найти максимальный элемент двумерного массива и его номер.

57. Найти минимальный элемент двумерного массива и его номер.

58. Заменить в двумерном целочисленном массиве все отрицательные элементы на ноль.

59. Заменить в двумерном целочисленном массиве все элементы, превосходящие значение некоторого параметра k, на вводимое с клавиатуры значение A.

60. В целочисленной прямоугольной матрице определить количество строк, не содержащих ни одного нулевого элемента.

61. В целочисленной прямоугольной матрице определить максимальное из чисел, встречающихся в заданной матрице более одного раза.

62. В двумерном целочисленном массиве найти количество столбцов, не содержащих ни одного отрицательного элемента.

63. Дан двумерный числовой массив. Переставляя строки этого массива местами, расположить их в соответствии с ростом суммы элементов в строке.

64. Дан двумерный числовой массив. Переставляя строки этого массива местами, расположить их в соответствии с ростом суммы положительных элементов в строке.

65. Реализовать алгоритм сортировки пузырьком с использованием флага.

66. Реализовать алгоритм сортировки пузырьком с использованием запоминания индекса последнего обмена.

67. Реализовать шейкер-сортировку.

68. Реализовать алгоритм сортировки пузырьком с использованием трёх улучшений, приведённых в теоретической части раздела.

69. Реализовать алгоритм сортировки Шелла.

70. Описать структуру с именем STUDENT, содержащую следующие поля:

– фамилия и инициалы;

– номер группы;

– успеваемость (массив из пяти элементов).

Написать программу, выполняющую следующие действия:

– ввод с клавиатуры данных в массив, состоящий из десяти структур типа STUDENT;

– упорядочение элементов массива по возрастанию среднего балла;

– вывод на дисплей фамилий и номеров групп для всех студентов, имеющих оценки «4» и «5»; если таких студентов нет, вывести соответствующее сообщение.

71. Описать структуру с именем TRAIN, содержащую следующие поля:

– название пункта назначения;

– номер поезда;

– время отправления.

Написать программу, выполняющую следующие действия:

– ввод с клавиатуры данных в массив, состоящий из шести элементов типа TRAIN;

– упорядочение элементов массива по времени отправления поезда;

– вывод на экран информации о поездах, направляющихся в пункт, название которого введено с клавиатуры; если таких поездов нет, выдать на экран соответствующее сообщение.

72. Описать структуру с именем NOTE, содержащую следующие поля:

– фамилия, имя;

– номер телефона;

– дата рождения (массив из трёх чисел).

Написать программу, выполняющую следующие действия:

– ввод с клавиатуры данных в массив, состоящий из восьми элементов типа NOTE;

– упорядочение элементов массива по алфавиту;

– вывод на экран информации о людях, чьи дни рождения приходятся на месяц, значение которого введено с клавиатуры; если таковых нет, выдать на экран соответствующее сообщение.

73. Объявить строку как массив символов. Выполнить сортировку элементов объявленного массива в алфавитном порядке; вывести на экран элементы отсортированного массива в обратном порядке; если элемент встречается более одного раза, вывести его только один раз.

74. Удалить из строки повторяющиеся символы.

75. Заменить в строке все заглавные буквы на строчные.

76. Дана строка. Вывести на экран все цифры, содержащиеся в этой строке.

77. Дана строка. Подсчитать и вывести на экран количество слов в ней.

78. Дана строка. Подсчитать и вывести на экран количество гласных букв в ней.

79. Дана строка. Среди символов строки содержатся цифры. Вычислить и вывести на экран сумму этих цифр и удалить их из строки.

80. Объединить три строки в одну, разделив их пробелом.

2. ФУНКЦИИ

Функцией в программировании называют именованную последовательность описаний и операторов, решающую отдельную подзадачу. Работа с функцией состоит из трёх этапов: *объявление*, *определение* и *вызов*.

Объявление может быть совмещено с определением.

Программа, написанная на языке программирования *C++*, состоит из функций; одна из этих функций должна носить имя `main().`

Выполнение программы начинается с выполнения функции `main()`.

Объявление функции (иначе – заголовок, или сигнатура) должно обязательно содержать имя функции, тип возвращаемого значения, круглые скобки, в которых может присутствовать, а может и отсутствовать список формальных параметров.

Объявление функции заканчивается точкой с запятой.

Если объявление функции совмещено с её определением, то точка с запятой не ставится

[*класс памяти*] <*тип возвращаемого значения*> <*имя функции*> ([*список формальных параметров*]);

Класс памяти позволяет задать область видимости функции.

Функции, объявленные с ключевым словом `extern`, видны во всех модулях программы.

Функции, объявленные с ключевым словом `static`, видны только в том модуле, в котором они определены (т. е. только из этого модуля можно вызвать такую функцию).

Тип возвращаемого значения может быть любым, в том числе и типом `void`, кроме массива и функции, но может быть указателем на массив или функцию.

Список формальных параметров содержит информацию о данных, которые необходимо передать в функцию при её вызове.

Элементы списка задаются типом и именем и разделяются запятой.

Например:

```
void f1();
```

В этом примере в заголовке функции класс памяти не указан, тип возвращаемого значения – `void`, список формальных параметров пуст.

```
int f2 (int, int);
```

В этом примере класс памяти также не указан, тип возвращаемого значения – `void`, список формальных параметров содержит два целочисленных параметра.

При объявлении функции имена параметров могут быть опущены, как показано в предыдущем примере.

При вызове функции на место формальных параметров будут подставлены значения конкретных данных – фактических параметров. Эти значения могут быть именованными (ранее объявленные переменные и константы) или неименованными.

Определение функции содержит заголовок функции (без точки с запятой в конце) и тело функции – последовательность определений и операторов, направленных на решение задачи.

Тело функции должно быть заключено в фигурные скобки.

Например:

```
void f1 () {cout << "это функция f1";}
int f2 (int a, int b){int c = a+b; return c;};
```

Для вызова функции необходимо указать её имя и, если это необходимо, список фактических параметров в фигурных скобках

```
f1 ();
f2 (3,4);
```

Для возвращения значения из функции необходимо использовать оператор `return`, указав после него возвращаемое значение, если функция имеет тип возвращаемого значения, отличный от `void`.

Значение, возвращаемое из функции с помощью оператора `return`, обязательно должно иметь тип, указанный в заголовке функции перед её именем.

После того как в теле функции встретился оператор `return`, работа функции будет завершена и управление будет передано в точку вызова функции.

Операторов `return` в теле функции может быть несколько, но выполнится только один.

Если после ключевого слова `return` указано возвращаемое значение, то это значение вернётся в точку вызова, где может быть использовано по усмотрению программиста.

Например:

```
f2 (3,4); //возвращаемое значение не использовано
int S = f2 (3,4); //возвращаемое значение сохранено в переменной S
cout << f2 (3,4); //возвращаемое значение выведено на экран.
```

Отчёты по всем лабораторным следует оформлять в соответствии с требованиями, приведёнными в прил. 4, 5, 6.

Лабораторная работа № 1

НАПИСАНИЕ ФУНКЦИЙ НА ЯЗЫКЕ ПРОГРАММИРОВАНИЯ *C++*

Цель работы. Закрепление навыков разработки программного кода на языке программирования *C++*. Получение навыков написания функций на языке программирования *C++*.

Постановка задачи. Написать функцию сложения двух вещественных чисел. Функция получает на вход два целочисленных параметра и возвращает целое число – сумму параметров. Написать функцию сложения элементов целочисленного одномерного массива. Функция получает на вход адрес первого элемента и количество элементов в массиве и возвращает сумму элементов в массиве. Написать функцию сортировки одномерного числового массива. Функция получает на вход данные о массиве, в качестве возвращаемого значения указать `void`. Написать функцию вывода элементов одномерного целочисленного массива на экран. В `main`-функции продемонстрировать работу всех описанных выше функций.

Вопросы для самоконтроля и подготовки к защите лабораторной работы

1. Что называют функцией в программировании?
2. Какие параметры называют формальными?
3. Какие параметры называют фактическими?
4. Что такое определение функции?
5. Где и для чего указывают тип возвращаемого значения функции?
6. Каким образом можно вернуть значение из функции?
7. Что такое тело функции?
8. Напишите синтаксис объявления функции.
9. Какие действия необходимо выполнить для вызова функции?
10. Сколько раз можно вызвать функцию?

Задания для самостоятельной работы

Написать функции для решения следующих задач. Если в задаче требуется что-то найти, найденное значение должно быть возвращено из функции.

1. Заполнение двумерного массива с клавиатуры.
2. Вывод двумерного массива в табличном виде.
3. Вычисление среднего арифметического элементов одномерного числового массива.
4. Поиск максимального значения в одномерном числовом массиве.
5. Поиск минимального значения в двумерном числовом массиве.
6. Сортировка одномерного числового массива.
7. Сортировка двумерного числового массива.
8. Поиск максимального из трёх передаваемых параметров.
9. Поиск суммы $1 + 2 + 3 + \ldots + N$, где число N вводится пользователем с клавиатуры.
10. Вывод всех квадратов натуральных чисел, не превосходящих данного числа N.

3. РЕКУРСИЯ

Под словом ***рекурсия*** (от лат. *recursio* – возвращение) понимают процесс описания какого-либо объекта или процесса, содержащего самоподобие (т. е. объект или процесс является частью описания самого себя). Понятие «рекурсия» встречается в различных областях знаний. Например, в лингвистике под рекурсией понимают способность языка порождать вложенные предложения и конструкции языка. Два поставленных друг напротив друга зеркала, образующих бесконечный рекурсивный коридор, – классический пример рекурсии из области физики. Всем известные с детства докучные сказки есть не что иное, как пример рекурсии в литературе. При организации данных также может быть использован рекурсивный подход. Рекурсия возникает, если определённый тип данных содержит внутри себя ссылку на этот же тип данных. Как, например, при описании структуры на языке программирования *C++* с целью дальнейшей организации потенциально бесконечных динамических списков

```
struct element
 {
   struct element* next; /*указатель на следующий элемент того же типа*/
   int data; /*хранимые данные*/
 };
```

Использование рекурсии в программировании обусловлено рекурсивной природой многих математических задач. При решении таких задач происходит разбиение задачи на ряд более простых подзадач. При этом обнаруживается, что для вычисления *F(N)* необходимо вычислить *F(N – 1)*, т. е. частью алгоритма вычисления функции будет вычисление этой же функции.

Функция называется ***рекурсивной,*** если в её теле встречается вызов самой себя.

Рекурсия называется ***прямой***, если в теле функции указано её имя, т. е. вызов этой же функции.

Рекурсия называется ***косвенной***, если в её теле осуществлён вызов другой функции, в теле которой, в свою очередь, происходит вызов первой функции.

Ниже приведён фрагмент кода на языке программирования *C++*, демонстрирующий пример прямой рекурсии

```
int a()
{
a(); //вызов функции a()
}
```

Пример косвенной рекурсии:

```
int a(){.....b().....};
int b(){.....c().....};
int c(){.....a().....};
```

Любой рекурсивный алгоритм может быть заменён на итерационный (циклический) алгоритм.

При выборе между рекурсивным и циклическим подходом необходимо учитывать достоинства и недостатки каждого из них. Так, например, рекурсивная функция будет более компактной по сравнению с циклической, но при этом нужно помнить, что каждый вызов функции приводит к созданию в стеке копии значений её параметров. Таким образом, достоинство рекурсии – компактная запись, а недостатки – расход времени и памяти на повторные вызовы функции и передачу ей копий параметров, а также опасность переполнения стека.

Для корректного завершения работы рекурсивная функция должна содержать хотя бы одну нерекурсивную ветвь, заканчивающуюся оператором возврата. Организацию такого ветвления следует реализовывать через оператор условия. Если этого не сделать, то после вызова такой функции выйти из неё будет невозможно.

При завершении работы рекурсивной функции (как и любой другой функции) соответствующая часть стека освобождается, управление передаётся в точку вызова функции (т. е. в случае рекурсивной функции – в тело самой функции) оператору, следующему за рекурсивным вызовом функции.

Пример ветвления в теле рекурсивной функции для случая прямой рекурсии:

```
int a(int a1)
{
    if (a1 == 0) return 0; else return a1+a(a1-1);
};
```

Если вызвать описанную функцию следующим образом:

```
printf("a=%i\n", a(10));
```

результат работы будет таким:

```
a=55
```

Пример ветвления в рекурсивной функции для случая косвенной рекурсии:

```
void  b(int a1); int  c(int a1); /*объявление функций, описание которых будет позже*/
int a(int a1) { if (a1 == 0) return 0; else b(a1); }; /*описание функции a(int  a1), в теле которой происходит вызов функции b(int  a1), в теле которой, в свою очередь, будет вызвана функция c(int a1)*/.
void b(int a1) { printf("a=%i\n", a1); c(a1); };
int  c(int a1) { return a(a1-1); }
```

Ниже представлен результат вызова функции `int a(int a1)` со значением параметра `a1=10`

```
printf("a=%i\n", a(10));
a=10
a=9
a=8
a=7
a=6
a=5
a=4
a=3
a=2
a=1
a=0
```

Приведённые примеры наглядно демонстрируют реализацию рекурсивных функций, но не имеют эффективного практического применения. Задачу сложения и вывода на экран последовательности чисел можно решить гораздо проще, используя цикл.

Рассмотрим два примера, в которых использование рекурсии обосновано лаконичностью кода.

Напишем рекурсивную функцию для вычисления факториала целого неотрицательного числа

```
int factorial1(int n) {
if (n < 2) {return 1;}
else {return n * factorial1(n - 1);}
}
```

Рассмотрим реализацию функции вычисления суммы элементов одномерного числового массива

```
int SumMas1(int p, int* mas)
{
    if (p == 0) {return mas[p];}
    else
    {
    return mas[p] + SumMas1(p - 1, mas);
    }
}
```

Теперь представим реализацию тех же задач через итерационный подход

```
int factorial2(int n)
{
    int S = 1;
    while (n > 1)
    {
        S *= n;
        n--;
    }
    return S;
}

int SumMas2(int p, int* mas)
{
    int S = 0;
    for (int i = 0; i < p; i++) {
        S += mas[i];
    }
    return S;
}
```

Алгоритм быстрой сортировки

Разберём алгоритм быстрой сортировки, или сортировки Хоара, названной именем её разработчика. Это один из самых известных и широко используемых на сегодняшний день алгоритмов сортировки. В ряде случаев алгоритм даёт наилучшую скорость сортировки.

На первом шаге алгоритма необходимо выбрать опорный элемент в массиве. Это может быть любой элемент, никакие ограничения на него не накладываются. Обычно выбирают центральный элемент следующим образом:

```
int op = a[(l + r) / 2].
```

Здесь `op` – опорный элемент, выбранный в массиве целых чисел; `a` – имя массива; `l` – номер крайнего левого элемента в массиве (на первом шаге равен нулю); `r` – номер крайнего правого элемента в массиве (на первом шаге равен номеру последнего элемента в массиве).

Далее массив разделяется на две части следующим образом: все элементы из левой части, бóльшие или равные опорному элементу, переносим в правую часть; все элементы из правой части, которые меньше или равны опорному, переносим в левую часть. Таким образом, массив получается частично упорядоченным. Все элементы левой части гарантированно меньше элементов правой части. Но между собой элементы левой и правой частей не упорядочены. Наглядно это представлено на рис. 1.

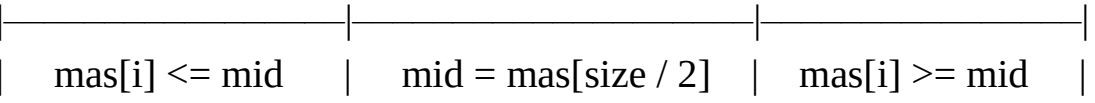


Рис. 1. Схематичное представление первой итерации сортировки Хоара

Для реализации описанного шага необходимо двигаться от границы (переданной в функцию в качестве параметра) до опорного элемента, пока не будет обнаружен первый элемент, находящийся не в своей части (больше опорного в левой части и меньше опорного в правой), а затем поменять найденные элементы местами.

Например, таким образом можно найти номер первого элемента, который необходимо переместить из левой части в правую

```
while (mas[f] < mid) f++;
```

После завершения цикла в переменной `f` будет сохранён номер соответствующего элемента.

Остаётся рекурсивно повторить описанные действия для левой и правой частей массива. Повторение производится до тех пор, пока размер соответствующей части больше единицы.

Рекуррентные соотношения

Последовательность чисел или функций, в которой каждый следующий член выражается через предыдущий, называется ***рекуррентной***, а формулы, устанавливающие эту связь, – ***рекуррентными соотношениями***. Рекуррентные соотношения используют при составлении алгоритмов вычисления величин, значения которых основаны на предыдущих значениях, при этом нет необходимости хранить все промежуточные значения. Текущее значение величины выражается через её предыдущие значения $f(x) = f(x - 1)$, при этом необходимо задать начальное значение x. Такой подход используется при решении задач на вычисление сумм (произведений) членов конечных или бесконечных последовательностей, вычисления пределов последовательностей и т. п. Простым примером служит вычисление членов арифметической $x_{n+1} = x_n + a$ и геометрической $x_{n+1} = x_n a$ последовательностей.

Количество предыдущих членов последовательности, необходимое для вычисления текущего члена, называют ***глубиной рекурсии***. Так, например, в приведённых выше формулах глубина рекурсии равна единице. А в формуле вычисления очередного члена последовательности Фибоначчи глубина рекурсии равна двум.

Последовательность Фибоначчи: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34…

$$a_i = \begin{cases} 0, & i = 0 \\ 1, & i = 1 \\ a_{i-1} + a_{i-2}, & i \geq 2 \end{cases}$$

Лабораторная работа № 2

РЕАЛИЗАЦИЯ РЕКУРСИВНЫХ АЛГОРИТМОВ НА *C ++*

Цель работы. Получение навыка разработки рекурсивных алгоритмов, написания рекурсивных функций на *C++*, закрепление навыков создания функций на языке *C++*.

Постановка задачи

1. Реализовать на языке программирования *C++* вычисление факториала целого неотрицательного числа, суммы элементов одномерного целочисленного массива. Для каждой из задач (вычисление факториала, вычисление суммы) написать две функции, первая из которых решает задачу через рекурсивный алгоритм, вторая – через циклический.

2. Написать рекурсивную функцию, реализующую алгоритм быстрой сортировки.

3. Выполнить одно задание из списка заданий для самостоятельной работы; номер задания взять в соответствии с номером вашего варианта (номером в журнале или по иному указанию преподавателя).

Вопросы для самоконтроля и подготовки к защите лабораторной работы

1. Что такое рекурсия?
2. Что такое рекурсия в программировании?
3. Назовите виды рекурсии; поясните алгоритм их реализации.
4. Что такое рекуррентное соотношение?
5. Поясните, что такое глубина рекурсии?
6. Что необходимо предусмотреть в теле рекурсивной функции для организации корректного завершения её работы?
7. На какой алгоритм можно заменить рекурсивный алгоритм?
8. Назовите преимущества и недостатки рекурсивных алгоритмов.
9. Кратко изложите суть алгоритма быстрой сортировки.
10. Назовите преимущества быстрой сортировки перед алгоритмами сортировки выбором и пузырьком.

Задания для самостоятельной работы

В следующих заданиях требуется вывести и записать рекуррентное соотношение, написать рекурсивную функцию на основе выведенного соотношения. Результатом работы функции должно быть отображение на экране указанной в задании последовательности; количество членов последовательности задать входным параметром.

1. 1, 2, 3, 4…
2. 0, 5, 10, 15…
3. 1, 1, 1, 1…
4. 1, –1, 1, –1…
5. 1, –2, 3, –4, 5, –6…
6. 2, 4, 8, 16…
7. 2, 4, 16, 256…
8. 0, 1, 2, 3, 0, 1, 2, 3, 0…
9. 1! 3! 5! 7!..
10. 1, a, a^2, a^3, a^4, a^5…
11. 1, $1 + a$, $1 + a + a^2$, $1 + a + a^2 + a^3$…
12. $1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots$

В следующих заданиях необходимо вычислить искомое значение, используя рекуррентное соотношение.

13. $a^m + 1$
14. $2a^m$
15. $(a - 1)$
16. $\frac{a^m}{(a-1)^{m+1}}$
17. $(2m - 1)!$
18. $1 + a + a^2 + \ldots + a^k$
19. $1 + a^2 + a^4 + \ldots + a^{2k}$
20. $1 - a + a^2 - a^3 + \ldots - a^{2m-1}$

4. ПЕРЕГРУЗКА ФУНКЦИЙ

Перегруженными называют функции, объявленные с одним именем, но с разным списком формальных параметров.

По типу передаваемых фактических параметров будет определено, какая именно функция должна быть вызвана. Ограничений на тип возвращаемого значения при этом не накладывается.

К механизму перегрузки прибегают при решении близких по смыслу задач с использованием различных алгоритмов.

Например, задачи сортировки

```
void sort (int*, int) //
```
объявление функции сортировки одномерного массива

```
{тело функции}
```

```
void sort (int*, int, int) //
```
объявление функции сортировки двумерного массива

```
{тело функции}
```

Если при вызове функции не найдено точного соответствия параметров, происходит так называемое *продвижение типов*. Например, `bool` и `char` будут интерпретированы как `int`; `float` как `double`. Если и на этом этапе соответствия не найдено, будет произведено стандартное преобразование типов. При этом важно учитывать, что, если типы фактических параметров соответствуют более чем одному из вариантов перегрузки, будет получено сообщение об ошибке. Так, например, если функцию, объявленную следующим образом:

```
int f(int a, int b),
```

переопределить так:

```
int f(int a, int &b),
```

то при попытке вызова будет невозможно определить, какой из функций должно быть передано управление, так как синтаксически на этапе вызова способы передачи параметров по значению и по ссылке неотличимы.

Перегруженные функции могут иметь параметры по умолчанию; разные варианты перегруженных функций могут иметь разное количество параметров по умолчанию.

Указатель на функцию

Вызов функции в языке программирования *C++* может быть организован через указатель на функцию. Объявление и инициализация такого указателя происходят по тем же правилам, что и объявление и инициализация обычных указателей, т. е. необходимо полностью описать тип указуемого и поставить символ * перед именем указателя. Под типом указуемого в данном случае понимают совокупность типа возвращаемого значения и типов передаваемых параметров, указываемых в круглых скобках после имени функции.

Можно объявить указатель на функцию, при этом важно указать тип возвращаемого значения и тип параметров.

Например:

```
void(*pf) (int, int);
```

`pf` – указатель на функцию, не возвращающую значения и принимающую два целочисленных параметра.

```
void f(int a, int b);
pf=&f; //pf настроен на функцию f
int a=3;
int b=4;
(*pf)(a,b); // вызов функции f через указатель pf/
```

Как видно из приведённого выше примера, работа с указателем на функцию ведётся по тем же правилам, что и работа с обычным указателем.

Для настройки указателя на функцию необходимо записать в него адрес этой функции; для этого используют операцию взятия адреса, обозначаемую символом &. Для работы с указуемым, т. е. для вызова функции, используют операцию разадресации, для этого перед именем указателя ставится символ *. Передача фактических параметров в функцию осуществляется таким же образом, как и при обычном вызове функции, т. е. указанием их в круглых скобках через запятую. Единственное отличие в вызове функции через указатель: списку параметров предшествует не имя функции, а разыменованный указатель.

Лабораторная работа № 3
ПЕРЕГРУЗКА ФУНКЦИЙ

Цель работы. Изучение механизма перегрузки функций на языке *C++*. Закрепление навыков разработки функций на языке *C++*, способов передачи параметров в функции, возвращения значения из функции; использование этого значения в точке вызова.

Постановка задачи

1. Написать функцию вычисления суммы элементов двумерного целочисленного массива и функцию сортировки двумерного целочисленного массива.

2. Перегрузить описанные выше функции для работы с массивами символов. При этом функция сложения должна возвращать предложение (т. е. строку, образованную сложением строк массива, слиянием их через пробел с добавлением точки после последнего символа). Функция сортировки строк должна упорядочить строки (не символы) исходного массива по алфавиту.

3. Объявить указатель на функцию, настроить его на описанную выше функцию вычисления суммы элементов двумерного массива. Вызвать соответствующую функцию через этот указатель.

Вопросы для самоконтроля и подготовки к защите лабораторной работы

1. В каком случае используют механизм перегрузки функций?
2. Каким должен быть тип возвращаемого значения перегруженных функций?
3. Могут ли перегруженные функции возвращать значения одного типа?
4. Может ли перегруженная функция иметь параметры по умолчанию?
5. В каком случае возможна ошибка при вызове перегруженной функции?
6. Что такое указатель на функцию?
7. Каким образом можно объявить указатель на функцию?
8. Каким образом происходит настройка указателя на функцию? Можно ли перенастроить указатель на функцию?

9. Каким образом организовать вызов функции через указатель на функцию?

10. Можно ли перегрузить функцию следующим образом: две функции имеют одинаковые имена, одинаковое количество и типы параметров, но возвращают значения разных типов?

Задания для самостоятельной работы

1. Описать функцию, принимающую два целочисленных параметра и возвращающую их сумму. Объявить указатель, настроить его на описанную функцию и вызвать с его помощью функцию.

2. Перегрузить функцию из первого задания для работы с вещественными числами.

3. Описать функцию, принимающую два целочисленных параметра и возвращающую квадрат разности параметров. Вызвать эту функцию через указатель.

4. Перегрузить функцию из предыдущего задания для вещественных чисел.

5. Перегрузить функцию из предыдущего задания, добавив третий параметр. Функция должна вернуть квадрат разности трёх значений.

6. Написать функцию, принимающую два целочисленных параметра и возвращающую максимальное из них. Перегрузить эту функцию для трёх целочисленных значений.

7. Перегрузить функцию из предыдущего задания для вещественных чисел.

8. Перегрузить функцию из предыдущего задания, добавив параметр по умолчанию; в зависимости от значения этого параметра будет возвращено максимальное или минимальное значение.

9. Написать функцию, которая возвращает среднее арифметическое двух целочисленных параметров. Перегрузить эту функцию для трёх параметров, для четырёх, для пяти.

10. Написать функцию, которая принимает один вещественный параметр `float a` – сторону квадрата – и возвращает вещественное число – площадь квадрата со стороной `a`. Перегрузить функцию для двух параметров: функция должна возвращать площадь прямоугольника со сторонами, равными значениям параметров; для трёх параметров: функция должна возвращать площадь треугольника со сторонами, равными значениям параметров.

5. ШАБЛОНЫ ФУНКЦИЙ

Шаблон функции позволяет описать единый алгоритм для разнотипных данных. Конкретный тип данных будет определён при вызове функции. Компилятор автоматически генерирует код функции, соответствующий переданному типу параметров. Использование шаблона позволяет сократить код и время на написание перегруженных функций, соответствующий код будет сгенерирован автоматически при первом вызове функции.

Формат описания такой функции задаётся следующим образом:

```
template <class Type> заголовок функции
{/*тело функции*/}
```

Слово Type может быть заменено любым другим словом.

Пример шаблона функции сложения двух чисел:

```
template <class Type> Type SUM (Type a, Type b)
{
    return a + b;
}
```

Далее приведён пример вызова описанного шаблона функции для целых и для вещественных чисел

```
int main()
{
    int a = 1; int b = 2;
    cout << "S=" << SUM(a,b) << "\n";
//вызов функции SUM для целых чисел

    float fa = 1.2; float fb = 2.4;
    cout << "S=" << SUM(fa, fb) << "\n";
//вызов функции SUM для вещественных чисел
}
```

Лабораторная работа № 4
ШАБЛОНЫ ФУНКЦИЙ

Цель работы. Расширение навыков работы с функциями на языке программирования *C++*, получение опыта создания шаблонов функций. Закрепление навыков написания кода на языке *C++* с использованием функций, способов передачи параметров в функции, возвращения значения из функции.

Постановка задачи. Написать шаблон функции сортировки двумерного массива. Вызвать функцию для целочисленного, вещественного и символьного двумерных массивов. Результаты сортировки массивов отобразить на экране (вывести массивы в виде таблицы).

Вопросы для самоконтроля и подготовки к защите лабораторной работы

1. Что такое шаблон функции?
2. Для чего используют шаблоны функций?
3. Напишите синтаксис объявления шаблона функции.
4. В каком случае используют шаблон функции, а в каком – перегрузку функции?
5. Сколько параметров можно передать в шаблон функции?
6. Обязательно ли функция, заданная через шаблон, должна возвращать значение?
7. Может ли тип возвращаемого значения шаблона функции быть задан как тип `T`?
8. Напишите шаблон для функции сложения двух чисел.
9. Каким образом происходит вызов функции для определённого типа входных параметров?
10. Можно ли перегрузить функцию, описанную через шаблон?

Задания для самостоятельной работы

1. Описать шаблон функции, возвращающей максимальное значение из двух переданных параметров. Вызвать эту функцию для целых и вещественных чисел.

2. Перегрузить описанный выше шаблон функции для трёх параметров. Вызвать описанный шаблон три раза с параметрами типов `int, float, double.`

3. Описать шаблон функции сортировки одномерного массива методом выбора. Вызвать описанную функцию для массивов, хранящих данные типов `int, float, char.`

4. Написать шаблон функции, выводящей на экран одномерный массив. Вызвать описанную функцию для массивов, хранящих данные типов `int, float, char.`

5. Написать шаблон функции, выводящей на экран двумерный массив в табличном виде. Вызвать описанную функцию для массивов, хранящих данные типов `int, float, char.`

6. Написать шаблон функции, возвращающей максимальное значение в одномерном массиве. Вызвать описанную функцию для массивов, хранящих данные типов `int, float, double, char.`

7. Перегрузить функцию из задания № 1 для поиска максимального значения в двумерном массиве. Вызвать описанную функцию для массивов, хранящих данные типов `int, float, double, char.`

8. Написать шаблон функции быстрой сортировки массива методом выбора. Вызвать описанную функцию для массивов, хранящих данные типов `int, float, char.`

9. Написать шаблон функции, возвращающей среднее арифметическое значение элементов одномерного числового массива. Вызвать описанную функцию для массивов, хранящих данные типов `int, float, double.`

10. Перегрузить описанный шаблон для двумерного и трёхмерного массивов. Вызвать описанные функции для массивов данных типов `int` и `float`.

6. РАБОТА С ФАЙЛАМИ

Под ***файлом*** понимают поименованную последовательность данных, расположенных на внешнем носителе, завершающуюся маркером конца файла `EOF`.

По способу доступа файлы можно разделить на *последовательные*, в этом случае чтение и запись производятся последовательно байт за байтом, и файлы с *производным доступом*, допускающие чтение и запись в указанную позицию.

При работе с файлами используют термин ***поток***. Под потоком понимают логическую абстракцию, представляющую собой последовательность байтов. Такой подход позволяет привести к единообразию работу по чтению и записи данных. При этом стандартные устройства ввода/вывода, такие как дисплей и клавиатура, рассматриваются как потоки.

Работа с потоком может вестись в двух режимах: текстовом и бинарном. В случае *бинарного* потока данные рассматриваются так, как они хранятся в памяти, т. е. как последовательность битов. В случае *текстовых* файлов данные рассматриваются как строки, заканчивающиеся символом `'\n'`. В текстовом режиме прочитанная из потока комбинация символов *CR* (возврат каретки) и *LF* (перевод строки) преобразуется в один символ конца строки `'\n'`. При записи в поток в текстовом режиме осуществляется обратное преобразование, т. е. символ `'\n'` преобразуется в комбинацию *CR* и *LF*. Если файл хранит не текстовую, а произвольную двоичную информацию, то такие преобразования не нужны.

Режим потока определяется при установлении связи с файлом.

В языке программирования *C++* работа с файлами организована с помощью *методов стандартных классов*. Работа с файлами через стандартные классы языка программирования *C++* в данном пособии не рассматривается, этот вопрос будет изучаться позднее в курсе объектно-ориентированного программирования. Поэтому далее будут рассмотрены функции для работы с файлами, унаследованные от языка программирования *C*. Именно эти функции следует использовать при выполнении лабораторной работы № 5.

Информация о файле, необходимая для работы с ним, содержится в стандартной структуре типа FILE, описанной в stdio.h, там же определены функции для работы с файлами.

Поэтому если в программе планируется работа с файлами, необходимо подключить соответствующий заголовочный файл, используя директиву include

```
#include <stdio.h>;
```

Для организации работы с файлами необходимо объявить указатель на файл следующим образом:

```
FILE * идентификатор;
```

Например, так: `FILE* F;`

Функция открытия потока получает на вход два параметра: имя файла и режим доступа к данным и имеет следующий формат:

```
FILE * fopen (const char* filename, const char*mode);
```

При успешном открытии функция возвращает указатель на структуру типа FILE. В противном случае функция возвращает NULL. Более безопасной считается функция fopen_s, имеющая следующую сигнатуру:

```
errno_t fopen_s(FILE** pFile, const char *filename, const char *mode);
```

Функция fopen_s принимает три параметра: указатель на файл, имя файла и режим работы с файлом. Возвращает нуль в случае удачного завершения работы или код ошибки – в противном случае.

Функции с постфиксом '_s', считаются более безопасными в вопросе защиты данных; эти функции выполняют дополнительные проверки параметров и возвращают коды ошибок. Однако ошибки не могут быть исправлены автоматически, поэтому необходимо анализировать возвращаемое функцией значение.

Функции, использование которых может привести к ошибкам безопасности, так как их работа не предотвращает операции, которые могут перезаписывать значения в памяти, считаются устаревшими. Их использовать не рекомендуется. По умолчанию компилятор выдаёт

предупреждение об устаревании функции, если в коде присутствует такая функция, и предлагает заменить её перегруженными шаблонами *C++*, которые помогают упростить переход к более безопасным вариантам. Так, вместо функции `fopen` предпочтительнее использовать функцию `fopen_s`. Ниже приведён пример работы с этой функцией

```
FILE* pf;
errno_t errFile3;
if ((errFile3=fopen_s(&pf, "c:\\fRezult.txt", " r
")) != 0) {printf("Не удалось открыть файл"); return 0;
}
```

В приведённом примере объявлен указатель `pf`, далее осуществлена попытка его связи с файлом `fRezult.txt`, расположенным на диске `C`, и открытие потока для чтения. Если файл открыть не удалось, на экране будет отображено сообщение `"Не удалось открыть файл"`.

Режим открытия файла:

`"r"` – открытие файла для чтения;

`"w"` – открытие пустого файла для записи;

`"a"` – открытие файла для добавления элементов в его конец;

`"r+"` – открытие файла для чтения и записи, при этом файл должен существовать;

`"w+"` – открытие пустого файла для чтения и записи; если файл с указанным именем уже существует, его информация стирается;

`"a+"` – открытие файла для чтения и добавления информации в его конец.

Режим открытия файла может содержать ещё один символ: `t` – текстовый файл или `b` – двоичный файл.

По умолчанию файл открывается в текстовом режиме. При вводе в поток информация накапливается в буфере до его заполнения или до закрытия потока.

После завершения работы с файлом поток должен быть закрыт. Для этого используют функцию `int fclose(FILE* pf)`.

Для корректного завершения предыдущего примера поток необходимо закрыть следующим образом:

```
fclose(pf);
```

Рассмотрим основные функции для работы с потоками ввода/вывода, унаследованные от языка *C*.

Чтение/запись потока байтов `fread()/fwrite()`

```
size_t fread(void* буфер, size_t число_байту size_t объем, FILE *fp);
size_t fwrite(const void *буфер, size_t число_байт, size_t объем, FILE *fp);
```

Пример использования функции `fread`:

```
FILE *pf;
char content[100];
pf = fopen("the_file.txt", "rb");
fread(&content, sizeof(char), 100,pf);
```

Функция `fread_s` считывает данные из потока. Это версия функции `fread` с усовершенствованиями, касающимися безопасности работы с данными

```
size_t fread_s(void *buffer, size_t bufferSize, size_t elementSize, size_t count, FILE *pf);
```

Функция `fread_s` возвращает то количество элементов, которое было считано в буфер. Возвращаемое значение может быть меньше количества, указанного для чтения элементов. Это может произойти в случае, если возникла ошибка чтения или конец файла был обнаружен раньше достижения заданного для чтения количества элементов. Чтобы отличить ошибку, возникшую при чтении из файла от условия конца файла, следует использовать функцию `feof` или `ferror.`

Пример использования функции `fread_s`:

```
FILE *pf;
int numread = fread_s( list, BUFFERSIZE, ELEMENTSIZE, ELEMENTCOUNT, stream );
```

Функция `int ferror(FILE *pf)` проверяет, имеются ли файловые ошибки в потоке `pf.` Если возвращаемое значение равно нулю – ошибки отсутствуют; отличная от нуля величина указывает на наличие ошибки.

Пример использования функции `fwrite`:

```
FILE *stream;
char list[30];
. . .
int numwritten = fwrite( list, sizeof( char ),
DATASIZE, stream );
fclose( stream );
```

Для более наглядной иллюстрации работы рассмотренных функций приведём пример их работы в связке

```
char ch[10];
for (int i = 0; i < 10; i++) ch[i] = i + 33;
FILE* pf;
errno_t errF1_1;
if(errF1_1=open_s(&pf,"c:\\file_new.txt","w")  !=0)
{
printf("Not File"); return 0;
}
else
{
    printf("open 'W'\n");
    fwrite(ch, sizeof(char), 10, pf);
    fclose(pf);
}

if(errF1_1=fopen_s(&pf,"c:\\file_new.txt","r")!=0)
{printf("Not File"); return 0;
}
else {
    printf("open 'r'\n");
    char *c=new char[10];
    if (!ferror(pf)) { fread_s(c, 10, 1, 10, pf); }
}
    fclose(pf);
    }
```

В приведённом выше примере объявлен указатель на массив ch, состоящий из десяти символов. Массив проинициализирован последовательностью из символов, начиная с символа под номером 33. Под номером 33 в таблице *ASCII* (прил. 7) находится символ '!'. Далее объявлен указатель на файл pf. Осуществляется попытка создания файла file_new.txt, связывание его с указателем pf и открытие потока на запись. Если создание и открытие произвести не удалось, на экране будет отображена надпись "Not File" и произведён выход из функции. В случае успеха на экране будет отображена надпись "open 'W'", говорящая пользователю о том, что файл успешно открыт для записи. Таким образом, подготовительный этап для записи данных в файл успешно завершён, и следующее действие – запись данных в файл при помощи функции fwrite. В файл, связанный с указателем pf, будут записаны десять элементов из массива ch размером sizeof(char), после чего поток будет закрыт и вновь открыт на чтение. Если открытие прошло успешно, в массив символов c из потока pf будет считана последовательность из десяти элементов размером один байт, после чего поток будет вновь закрыт.

Реализуем рассмотренный выше пример при помощи функций чтения и записи символа из потока.

Чтение символа из потока реализуется с помощью функции fgetc. Запись символа в поток реализуется с помощью функций fputc.

Функция int (FILE* stream) возвращает следующий за текущей позицией символ во входном потоке stream и даёт приращение указателю положения в файле. Символ считывается как данные типа unsigned char, преобразованные к переменной целого типа.

При достижении конца файла fgetc() возвращает константу – EOF, но поскольку EOF имеет значение целого типа, при работе с двоичными файлами для контроля достижения конца файла необходимо использовать feof(). Если fgetc() обнаруживает ошибку, то также возвращается значение EOF

```
FILE* pf;
errno_t errF;
if (errF = fopen_s(&pf, "c:\\file_new.txt","w")!=0)
```

```
{
    printf("Not F1"); return 0;
}
else
{
    printf("open 'W'\n");
    for (int i=0; i<10; i++) { fputc(i + 33, pf); }
    fclose(pf);
}

if (errF=fopen_s(&pf,"c:\\file_new.txt","r") != 0)
{
    printf("NoT F1"); return 0;
}
else {
    printf("open 'r'\n");
    while (!feof(pf)) {
    int c = fgetc(pf);
if (c!=EOF ) printf_s("%c\t", c);
    }
        fclose(pf);
    }
```

В приведённом примере стоит обратить внимание на дополнительную проверку на конец файла перед выводом считанного значения. Проверка обусловлена тем, что функция `fgetc(pf)` возвращает значение, следующее за значением текущей позиции в файле, поэтому условия продолжения цикла недостаточно. На рис. 2 приведён результат работы рассмотренного фрагмента программного кода. К такому же результату при выводе данных приводит предыдущий пример и два разобранных ниже.

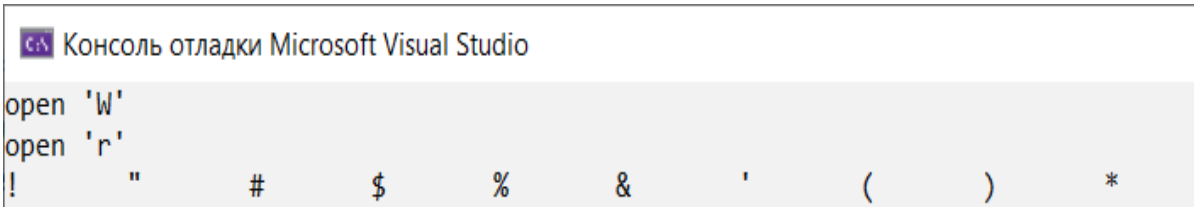


Рис. 2. Результат примера использования функций

Рассмотрим функции чтения/записи строки в файл. Функция `fgets()` реализует чтение строки из указанного потока. Функция `fputs()` реализует запись строки в указанный поток

```
char *fgets(char *str, int длина, FILE *fp);
int fputs(const char *str, FILE *fp);
```

Далее приведён пример использования этих функций для рассмотренной выше задачи. Для корректной работы функции чтения строки из файла в исходный массив добавлен символ завершения строки.

Результат работы приведённого ниже фрагмента кода такой же, как на рис. 3.

```
char ch[11]; /*объявление массива символов,
заполнение объявленного строкой выше массива*/
for (int i = 0; i < 10; i++) ch[i] = i + 33;
ch[10] = '\0';
FILE* pf; //объявление указателя на файл
errno_t errF;
//открытие файла для записи
if (errF=fopen_s(&pf,"c:\\file_new.txt", "w") != 0)
{//вывод информативного сообщения в случае неудачи открытия
    printf("Not F1"); return 0;
}
else
{//вывод информативного сообщения в случае удачного открытия
    printf("open 'W'\n");
    fputs(ch, pf); //запись строки
    fclose(pf); //закрытие потока
}
//открытие файла для чтения
if (errF=fopen_s(&pf, "c:\\file_new.txt", "r")!= 0)
{//вывод информативного сообщения в случае неудачи открытия
    printf("NoT F1"); return 0;
}
else
{//создание массива, в который будет записана строка
char ch[126];
```

```
/*вывод информативного сообщения в случае удачного открытия*/
        printf("open 'r'\n");
    //запись строки в массив и вывод строки из массива на экран
        if (fgets(ch, 126, pf)) printf_s("%s\t", ch);
    }
    fclose(pf);//закрытие потока
```

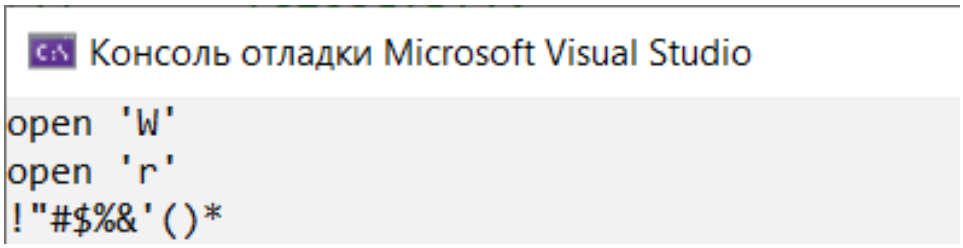


Рис. 3. Результат примера использования функций fgets() и fputs

Из рис. 3 видно, что на экране отображены те же символы, что и на рис. 2, но без пробелов. Это объясняется тем, что данные выведены как строка, а в предыдущем примере данные выводились посимвольно и были разделены управляющим символом табуляции `\t`.

В ряде случаев удобнее использовать функции форматированного ввода/вывода из потока.

Форматированный ввод `fscanf()`,

Форматированный вывод `fprintf()`.

```
int fscanf(FILE *fp, char *format, ...);
int fprintf(FILE *fp, char *format, ...);
```

Реализуем задачу, рассмотренную выше, с помощью этих функций

```
char ch[11];
for (int i = 0; i < 10; i++) ch[i] = i + 33;
ch[10] = '\0';
FILE* pf;
errno_t errF;
if (errF = fopen_s(&pf, "c:\\file_new.txt", "w") !=
0) {
        printf("Not F1"); return 0;
    }
```

```
else {
    printf("open 'W'\n");
     {if( fprintf(pf, ch)); }
    fclose(pf);
}

if (errF=fopen_s(&pf,"c:\\file_new.txt","r") != 0)
{
    printf("NoT F1"); return 0;
}
else
{char  ch[126];
    printf("open 'r'\n");
    if (fscanf_s(pf, "%s", ch, _countof(ch)))
        printf_s("%s", ch);
}
fclose(pf);
```

`_countof` возвращает количество элементов массива.

Из разобранных выше примеров очевидно, что одну и ту же задачу можно решить, используя различные функции языка программирования *C++*. При выборе функции для решения конкретной практической задачи следует руководствоваться спецификой каждой из функций и личными предпочтениями разработчика.

Значения, возвращаемые функциями работы с потоком, необходимо анализировать в программе на корректность работы, для того чтобы вовремя отследить возможные ошибки, например, при открытии файла.

При чтении данных из файла работа часто ведётся до его завершения. Для анализа файла на завершение используют функцию `feof`.

`int  feof(FILE*)` возвращает нуль, если достигнут конец файла, и отличное от нуля значение – в противном случае.

Алгоритм сортировки слиянием

Сортировка слиянием предполагает объединение двух последовательностей в одну с сохранением упорядоченности.

Таким образом, входными данными алгоритма являются две упорядоченные последовательности с возможностью поэлементного доступа. На выходе получаем одну упорядоченную последовательность, состоящую из двух объединённых входных последовательностей.

При этом элементы входных последовательностей сразу располагаются в правильном порядке, без дальнейшей перестановки.

Сортировка слиянием относится к внешним сортировкам, т. е. предполагается, что данные находятся на внешнем носителе с возможностью поэлементного последовательного доступа к ним. Пример: записанная в файл числовая последовательность.

Для изучения алгоритма и освоения его реализации на языке программирования возможно применение этого метода к предварительно упорядоченным массивам данных.

Суть алгоритма сводится к следующему.

Из входных последовательностей считываются первые элементы.

Считанные элементы сравниваются между собой, минимальный или максимальный из сравниваемых элементов (в зависимости от направления сортировки) записывается в результирующую последовательность.

Далее считывается следующий элемент из той последовательности, элемент которой был записан в итоговую последовательность на предыдущем шаге.

Процесс повторяется до тех пор, пока одна из входных последовательностей не закончится. Так как элементы входных последовательностей упорядочены, оставшуюся нерассмотренной часть второй последовательности следует просто дописать к результирующей последовательности.

Рассмотрим пошагово пример сортировки слиянием двух последовательностей. Для наглядности каждый шаг сортировки проиллюстрирован. На рис. 4 представлены входные последовательности.

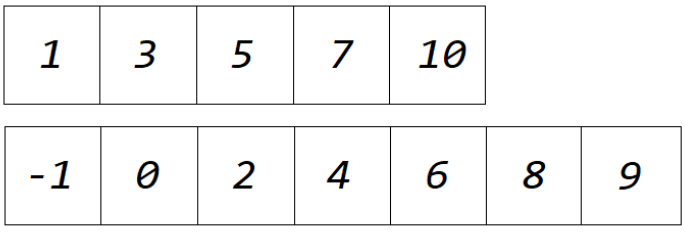


Рис. 4. Входные последовательности

На первом шаге считываем в дополнительные переменные начальные элементы последовательностей (рис. 5).

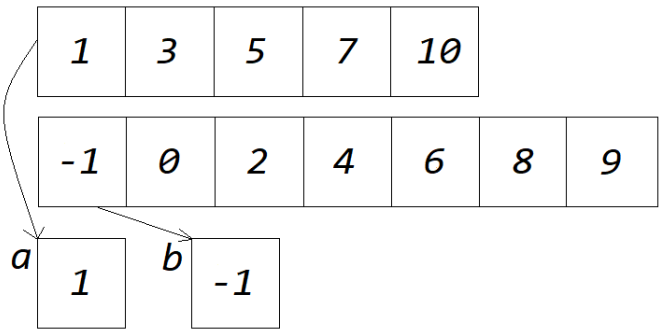


Рис. 5. Считаны первые элементы

Сравниваем считанные элементы и записываем меньший в результирующую последовательность (рис. 6).

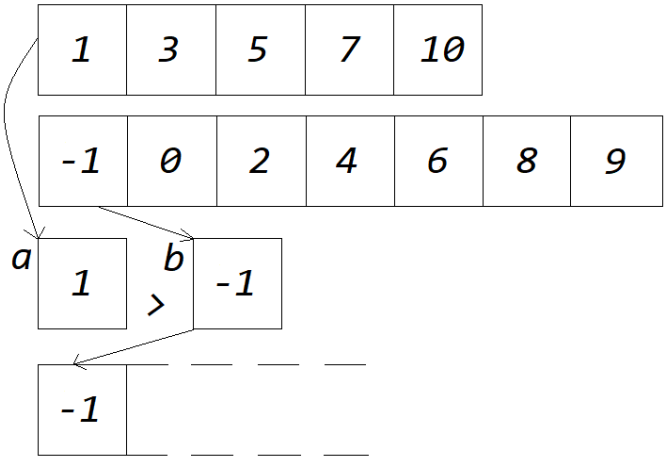


Рис. 6. Первый элемент записан
в выходную последовательность

Далее согласно алгоритму считываем следующий элемент из последовательности, элемент которой был только что записан в результирующую последовательность (рис. 7).

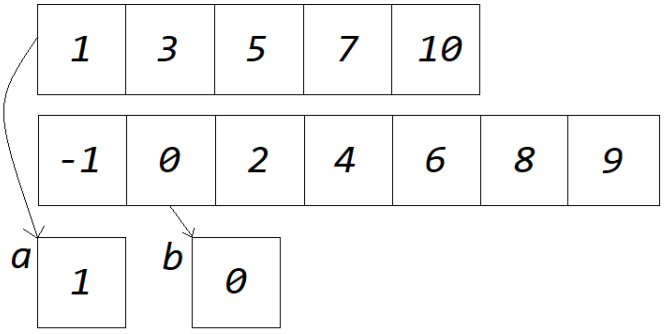


Рис. 7. Считан второй элемент

Вновь сравниваем считанные элементы и записываем меньший в результирующую последовательность (рис. 8).

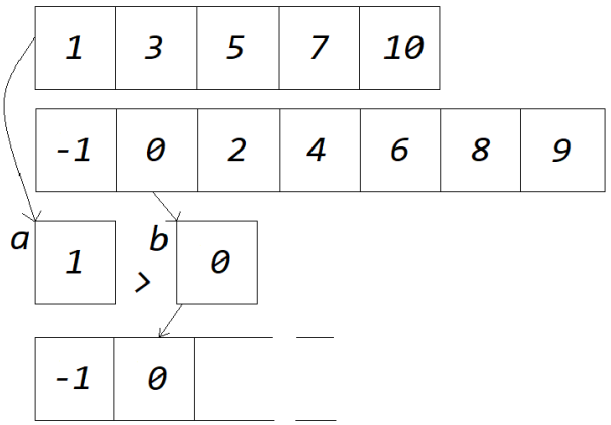


Рис. 8. Второй элемент записан
в выходную последовательность

Повторяем процесс: в переменную ***b*** будет считано значение 2, далее после сравнения значений переменных ***a*** и ***b*** меньшее значение из переменной ***a*** будет записано в результирующую последовательность (рис. 9).

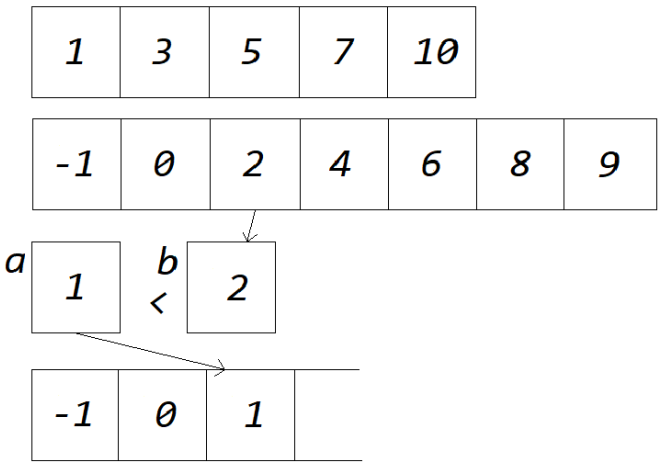


Рис. 9. Записан элемент из переменной ***a***

Очевидно, что на следующем шаге будет считано значение 3 в переменную ***a***. После сравнения значений переменных ***a*** и ***b*** в итоговую последовательность будет записано значение 2 из переменной ***b*** (рис. 10).

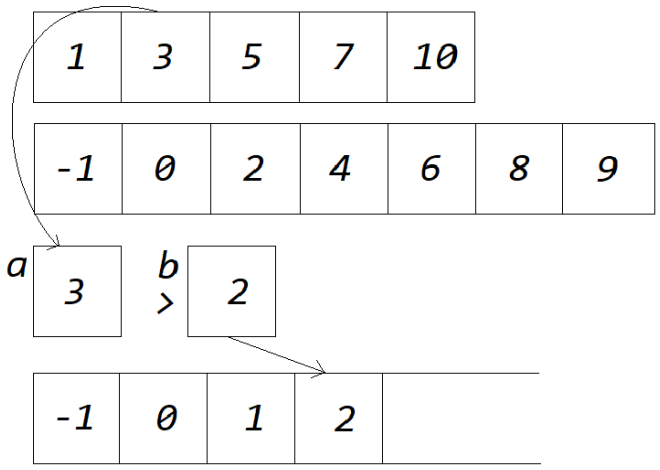


Рис. 10. Записан очередной элемент

Повторяем процесс до тех пор, пока одна из последовательностей не будет исчерпана. В данном случае это последовательность, расположенная на рисунке второй. Несмотря на то что нижняя последовательность содержит большее количество элементов, значение наибольшего элемента этой последовательности меньше последнего значения первой последовательности. Поэтому второй набор данных будет перебран и записан в результирующую последовательность быстрее. После этого останется переписать остаток первой входной последовательности в выходную. Дозапись производится поэлементно, без сравнения, так как элементы уже упорядочены, а элементы, с которыми производилось сравнение, уже исчерпаны. В данном случае нужно записать только значение 10 из первой входной последовательности. Завершающий этап слияния представлен на рис. 11.

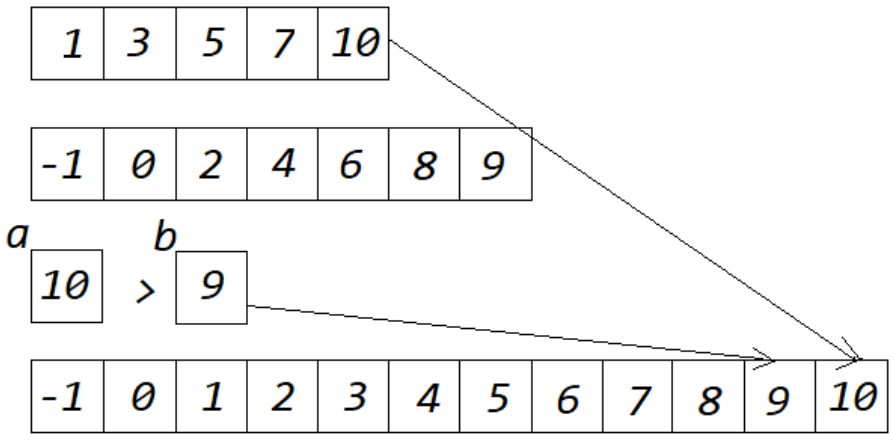


Рис. 11. Завершение слияния двух последовательностей

Лабораторная работа № 5
РАБОТА С ФАЙЛАМИ

Цель работы. Изучение способов работы с файлами, чтение и запись данных из файлов с помощью функции языка *C++*.

Постановка задачи. Реализовать сортировку слиянием двух одномерных упорядоченных целочисленных массивов. Упорядоченные массивы даны в файлах, необходимо записать их в третий файл с сохранением упорядоченности.

Методические указания

Создать два одномерных целочисленных массива; количество элементов в массивах должно быть разным. Упорядочить массивы по возрастанию, используя для этого функции, описанные в предыдущих лабораторных работах.

Записать массивы в файлы. Написать для реализации этой задачи соответствующую функцию.

Реализовать алгоритм сортировки слиянием, использовав в качестве входных данных созданные файлы. Результат сортировки сохранить в третьем файле.

Вывести данные из результирующего файла на экран.

При реализации алгоритма сортировки слиянием информация о количестве элементов в исходных массивах не должна быть использована. Для проверки достижения конца файла следует использовать функцию `feof`.

В отчёте по лабораторной работе представить несколько различных тестовых примеров; учесть, что файлы могут быть разной длины, могут содержать повторяющиеся значения. Результирующий файл должен содержать все элементы из исходных файлов.

Вопросы для самоконтроля и подготовки к защите лабораторной работы

1. Что такое файл?
2. Что такое указатель на файл?
3. Как связать указатель на файл с файлом в памяти?

4. Что такое поток?

5. На какие две группы можно разделить файлы?

6. Каким образом в коде можно определить завершение файла?

7. Какие режимы открытия файла вы знаете?

8. Каким образом можно в программе на языке *C++* записать данные в файл?

9. Каким образом можно в программе на языке *C++* считывать данные из файла?

10. Какую функцию и для какой цели нужно использовать при завершении работы с потоком?

Задания для самостоятельной работы

Пояснение: во всех задачах, где указано, что дан файл, его необходимо создать и записать в него соответствующую информацию.

1. Дан текстовый файл. Слова в файле могут быть разделены одним или более пробелом. Прочитать и вывести на экран слова из файла, игнорируя лишние пробелы, разделяя слова одним пробелом.

2. В файле записано предложение, содержащее числа. Вывести эти числа на экран.

3. Дан текстовый файл, в котором встречаются цифры. Вывести на экран текст, заменив все цифры на символ *.

4. Дан текстовый файл. Подсчитать и вывести на экран количество слов в файле; вывести на экран содержащийся в файле текст.

5. Дан текстовый файл, содержащий цифры. Вывести на экран текст; подсчитать и вывести на экран количество цифр в тексте.

6. Дан текстовый файл. Подсчитать и вывести на экран количество символов в файле до первого пробела.

7. Дан текстовый файл, содержащий некоторое количество слов, по одному слову в строке. В файле не более 30 слов длиной не более 20 символов. Создать новый файл, переписав в него все слова из первого в алфавитном порядке.

8. Дан текстовый файл. Определить, сколько раз в этом файле встречается определённый символ. Символ вводится с клавиатуры.

9. В данном текстовом файле хранятся вещественные числа. Вывести их на экран и вычислить их количество.

10. Дан текстовый файл. Подсчитать количество слов в файле; вывести на экран текст из файла и подсчитанное количество слов.

11. В текстовом файле могут содержаться числа, записанные в десятичной системе счисления. Вычислить сумму всех цифр, содержащихся в файле; если цифр нет, выдать соответствующее сообщение.

12. Дан текстовый файл. Определить, сколько раз в этом файле встречается заданное слово.

13. Дан текстовый файл. Вывести на экран слова из этого файла, состоящие не менее чем из трёх и не более чем из пяти букв; подсчитать количество таких слов.

14. Дан текстовый файл. Вывести на экран слова из этого файла, начинающиеся с гласной латинской буквы; подсчитать количество таких слов.

15. Дан текстовый файл. Подсчитать и вывести на экран количество предложений в нём.

7. МНОГОМОДУЛЬНОЕ ПРОГРАММИРОВАНИЕ

Под ***модулем*** понимают файл с исходным кодом и расширением `*.cpp`. Многомодульное программирование позволяет разбить задачу на подзадачи. При этом открывается возможность совместной работы над проектом, в ходе которой каждый разработчик решает свою задачу. Применение такого подхода открывает возможности использования ранее написанного кода, для этого достаточно подключить соответствующий модуль к проекту. В одном из модулей проекта обязательно должна содержаться `main`-функция, являющаяся точкой входа в программу. При работе над проектом рекомендуется в модуле с `main`-функцией реализовывать только сценарий работы – вызов соответствующих функций. Описание функций и пользовательских типов группировать в отдельных модулях с учётом их предназначения. Заголовки описаний должны быть вынесены в заголовочные файлы. ***Заголовочным файлом*** называют файл с кодом на *C++* и расширением `*.h`. В заголовочные файлы выносят объявления функций, пользовательских типов данных и некоторые директивы.

Получение из исходного кода исполняемого файла происходит в несколько этапов: препроцессинг, компиляция (трансляция), сборка (линковка).

Препроцессинг включает в себя следующие действия: ;

– текстовое включение файлов `#include`;

– макроподстановки `#define`;

– обработка директив условной компиляции `#if, #ifdef, #elif, #else, #endif`.

Компиляция. Назначение компиляции – перевод программы, написанной на языке высокого уровня, в набор команд, близких к машинным. Таким образом, после компиляции файл с исходным кодом программы преобразуется в файл, имеющий расширение `*.obj`, который называется *объектным модулем*.

Процесс компиляции состоит из следующих этапов:

1) лексический анализ;
2) синтаксический анализ;
3) семантический анализ;
4) оптимизация;
5) генерация кода.

Лексический анализ. Последовательность символов исходного файла преобразуется в последовательность лексем. *Лексема* – это элементарная составляющая языка, несущая смысловую нагрузку, например: имя, ключевое слово, символ операции, разделитель. Во время лексического анализа происходит разделение программы на предложения, далее предложения делятся на лексемы. После распознавания лексем происходит их перевод в двоичное представление.

Синтаксический анализ. В процессе синтаксического анализа из лексем собирают выражения, а из выражений – операторы. Последовательность терминальных символов преобразуется в нетерминалы. Невозможность достижения очередного нетерминала – признак синтаксической ошибки. Под *терминалом* понимают конечный символ грамматики. Например: `if, else, while, +, -, (, ), ++, --, '\n', 3.14159` и т. д. *Нетерминал* – это символ грамматики, для которого найдётся правило перевода его в цепочку из комбинаций терминалов и/или нетерминалов. Например: оператор присваивания, объявление переменной, цикл и т. д.

Семантический анализ – проверка смысловой правильности синтаксических конструкций. Например, если в выражении используется переменная, то она должна быть предварительно объявлена и проинициализирована. В семантический анализ также входят проверка совместимости и определение типов выражений.

Оптимизация – упрощение кода с сохранением его смысла и удаление лишних конструкций.

Генерация кода. Из промежуточного представления порождается объектный код – результат компиляции. На этом этапе происходит замена операторов языка высокого уровня инструкциями ассемблера, а затем – последовательностью машинных команд. Результат преобразования исходного текста программы записывается в виде двоичного файла, имеющего расширение `*.obj`, который называют *объектным модулем*.

Сборка (линковка, или связывание) Это последний этап процесса получения исполняемого файла, который заключается в связывании воедино всех объектных файлов проекта. На этом этапе могут быть выявлены ошибки связывания, например, если функция была объявлена, но не определена, ошибка обнаружится только на этом этапе.

В ходе трансляции устанавливается связь между операндами и адресами памяти, в которых хранятся, к примеру, результаты вычислений. Компоновщик отвечает за то, чтобы конкретному операнду соответствовала определённая область памяти. К компонуемой программе добавляются коды библиотечных функций (обеспечивающих выполнение конкретных действий – вычисления, вывода на экран), а также код, обеспечивающий размещение программы в памяти, её корректное начало и завершение. Преобразованная компоновщиком программа называется *загрузочным,* или *исполняемым, модулем* и имеет расширение `*.exe.`

В языке программирования *C++* не существует специальных конструкций, описывающих программную структуру многомодульного проекта. При этом возможна реализация одного из двух режимов компиляции проекта: раздельного и совместного.

Совместная компиляция. При совместной компиляции на вход компилятору подаётся один исходный файл, содержащий в себе все исходные модули, подключённые через директиву `#include`. Далее полученный `obj`-файл обрабатывается компоновщиком, который, в свою очередь, собирает `exe`-файл. Модули с исходным кодом, подключённые с помощью директивы `#include` в модуль, содержащий `main`-функцию, из проекта необходимо исключить (*remove*). Иначе возникнет ошибка, связанная с переопределением.

Раздельная компиляция. Все модули компилируются независимо друг от друга, генерируется столько `obj`-файлов, сколько было модулей с исходным кодом. Во время компоновки все `obj`-файлы собираются в один `exe`-файл. Для успешной компоновки необходимо, чтобы во всех компонуемых файлах содержались объявление всех используемых функций (как пользовательских, так и стандартных) и определение внешних переменных (объявленных с модификатором `extern`). В противном случае возникнет ошибка компоновки. Для реализации вышесказанного необходимо подключение через директиву `#include` заголовочных файлов во все модули, где планируется вызов объявленных в них функций.

При работе с переменными и функциями важное значение имеют понятия *область видимости* и *время жизни*.

Область видимости – это часть кода программы, из которой возможно обращение к переменной.

Время жизни – это время, в течение которого переменная физически существует в памяти и к ней можно обратиться через соответствующий идентификатор.

Время жизни и область видимости определяют класс памяти, указываемый при объявлении.

В языке *C++* выделяют четыре класса памяти, которые и определяют время жизни и область видимости переменных, констант и функций. При этом время жизни любой функции равно времени работы программы, а вот область видимости может изменяться в зависимости от класса памяти.

Модификатор класса памяти указывают при объявлении переменной или функции перед именем переменной и в начале заголовочной строки функции

класс памяти тип имя переменной;

Например:

```
static int a=7;
static void sort (int* a, int N);
```

`auto` – класс памяти по умолчанию для локальных переменных. Применяется только к переменным. Время жизни и область видимости локальны.

`registr` – класс памяти, применяемый только к локальным переменным типа `char, int`. Переменные, объявленные с таким классом памяти, будут размещены в одном из доступных регистров; если такой возможности нет, то работа с такими переменными ведётся так же, как и с переменными, имеющими класс памяти `auto.`

`static` – класс памяти, применяемый к переменным, константам и функциям. Время жизни `static`-элементов – с момента объявления до конца работы программы, так как объект расположен по фиксированному адресу. Область видимости локальна, т. е. для переменных это блок, для функций – модуль, в котором она объявлена и определена. Следовательно, такая функция недоступна из других модулей.

`extern` – внешний статический класс памяти. Время жизни – с момента объявления до конца работы программы. Область видимости глобальна, т. е. объект, объявленный с таким модификатором, доступен во всех модулях программы. Чтобы такую переменную использо-

вать во всех модулях программы, её необходимо объявить во всех модулях, где предполагается её использование, а проинициализировать только во одном.

При задании класса памяти переменной или функции руководствуются необходимым временем жизни и областью видимости определяемых объектов.

Лабораторная работа № 6

МНОГОМОДУЛЬНОЕ ПРОГРАММИРОВАНИЕ

Цель работы. Совершенствование навыков написания кода на языке программирования *C++* с использованием функций; закрепление знаний о способах передачи параметров в функции, возвращении значений из функции. Получение навыков разработки многомодульных проектов и способов компиляции в среде разработки *Visual Studio*.

Постановка задачи. Создать многомодульный проект в среде разработки *Visual Studio*, реализующий сортировку слиянием двух упорядоченных целочисленных последовательностей, хранимых в отдельных файлах. Реализовать два режима компиляции проекта: раздельную и совместную.

Указания к работе

Для выполнения работы необходимо создать модуль с расширением `*.cpp`, например `functions.cpp`, в котором будут описаны следующие функции:

– функция заполнения целочисленного массива;

– функция сортировки одномерного целочисленного массива (любым способом);

– функция вывода элементов массива на экран;

– функция записи элементов массива в файл.

Создать заголовочный файл, например `functions.h`, в который будут вынесены заголовки всех функций, описанных в модуле `functions.cpp`.

В главном модуле проекта, содержащем функцию `main()`, необходимо подключить модули `functions.cpp` и `functions.h` соответствующим для реализации раздельной компиляции образом и организовать описанный ниже сценарий работы.

Объявить два одномерных целочисленных массива; количество элементов массива должно быть различным.

Для каждого массива вызвать следующие функции в указанном порядке: заполнение массива, сортировка массива, вывод на экран, запись элементов в файл.

Реализовать сортировку слиянием элементов массива. Алгоритм сортировки реализовать в виде функции, получающей в качестве входных данных два указателя на файл.

Вывести на экран данные из третьего файла. Вывод данных из файла также рекомендуется оформить в виде функции.

После отладки проекта и добавления текста модулей в отчёт по лабораторной работе необходимо реализовать режим совместной компиляции. При этом в отчёте нужно продемонстрировать только разницу в подключении модулей, так как код, содержащийся в модулях `functions.cpp`, `functions.h` и `main.cpp`, уже добавлен в отчёт при реализации раздельной компиляции.

Вопросы для самоконтроля и подготовки к защите лабораторной работы

1. Что такое модуль?
2. Какой файл называют заголовочным?
3. Какое расширение имеет заголовочный файл?
4. Что такое компиляция?
5. Какие языки программирования называют языками высокого уровня, какие – низкого?
6. Назовите этапы сборки проекта.
7. Что такое `obj`-файл?
8. Каким образом можно программно реализовать режим раздельной компиляции? Опишите его этапы.
9. Каким образом можно программно реализовать режим совместной компиляции? Опишите его этапы.
10. В каких случаях следует использовать режим раздельной компиляции?
11. В каких случаях следует использовать режим совместной компиляции?
12. Возможен ли вызов в `main`-функции `static`-функции, описанной в отдельном модуле `f.cpp`, если в главном модуле (модуле с `main`-функцией), подключён с помощью директивы заголовочный файл `#include f.h`?

Задания для самостоятельной работы

Во всех заданиях необходимо создать многомодульный проект, содержащий как минимум три файла: `functions.cpp, functions.h, main.cpp.` Файл `functions.cpp` должен содержать описания функций, необходимых для решения задачи; файл `functions.h` должен содержать заголовки функций, описанных в `functions.cpp`; файл `main.cpp` должен содержать функцию `main()`, реализующую поставленную задачу через последовательный вызов функций, описанных в модуле `functions.cpp.` Имена файлов могут отличаться от предложенных, однако они обязательно должны быть информативными, т. е. из названия файла должно быть понятно его назначение. Реализовать два режима компиляции: раздельный и совместный.

1. Реализовать работу с целыми числами. Написать и последовательно вызвать функции, возвращающие максимальное из двух целых чисел, минимальное, среднее арифметическое, удвоенную сумму, квадрат разности.

2. Обобщить предыдущую задачу, написав шаблоны функций для трёх параметров. В `main`-функции реализовать сценарий работы с целыми и вещественными числами.

3. Реализовать многомодульный проект по работе с массивами данных (проект должен содержать функции ввода данных в массив с клавиатуры), сортировки массива различными способами, вычисления среднего арифметического значений массива, вывода данных на экран.

4. Реализовать многомодульный проект «двумерная целочисленная матрица» с функционалом заполнения матрицы, вывода матрицы на экран в табличном виде, сложения матриц, умножения матрицы на матрицу, умножения матрицы на число.

5. Обобщить предыдущую задачу, написав шаблоны функций. Реализовать в `main`-функции сценарий работы с целыми и вещественными числами.

6. Создать многомодульный проект по работе с текстовыми файлами. В файле `file.cpp` реализовать функции создания файла и записи в него данных, чтения данных из файла и вывода их на экран,

подсчёта количества символов в файле, подсчёта количества предложений в файле.

7. Создать многомодульный проект по работе с текстовыми файлами. Для этого расширить функционал предыдущего проекта функциями вывода на экран всех содержащихся в файле цифр, функцией подсчёта суммы содержащихся в файле чисел. Все функции по работе с файлами объявить с классом памяти `static`. Сценарий работы реализовать в отдельной функции того же модуля, имеющей класс памяти `extern`. Вызвать эту функцию в главном модуле программы.

8. Создать многомодульный проект «сортировка». Создать модуль, содержащий описания как минимум трёх шаблонов функций сортировки одномерного массива различными способами.

9. Расширить предыдущий проект шаблонами функций сортировки двумерного массива. Реализовать сценарий вызова описанных функций в сценарной `extern`-функции. Остальные функции объявить с модификатором `static`.

10. Разработать многомодульный проект «калькулятор», реализовав в модуле `calc.cpp` следующие функции: ввод строки – выражения; анализ строки на корректность, включающий в себя выделение операндов и операторов; вычисление выражения; вывод результата вычисления.

8. ЛИНЕЙНЫЕ ДИНАМИЧЕСКИЕ СТРУКТУРЫ

В общем случае под *структурой данных* понимают множество данных и множество связей между ними.

Мы будем рассматривать динамические структуры данных, т. е. структуры, в которых количество элементов и связи между этими элементами могут изменяться во время исполнения программы. Такие структуры можно разделить на две группы по типу организации связи между элементами: линейные и нелинейные.

Линейные	Нелинейные
– массивы;	– двоичное дерево;
– структуры;	– АВЛ-деревья;
– списки;	– мультисписки;
– очередь;	– ориентированные графы и др.
– стек и др.	

Массив – совокупность однотипных элементов, расположенных в памяти подряд, обращение к которым производится по имени массива и номеру элемента.

Под ***структурой***, или ***записью***, понимают совокупность разнотипных данных, объединённых под одним именем.

Список – это динамическая структура данных, элементы которой связаны между собой определённым образом и не обязательно расположены в памяти подряд. Размер структуры и порядок следования элементов могут изменяться во время выполнения программы. К линейным спискам относят ***очередь*** и ***стек***.

К использованию динамических структур данных прибегают в ситуациях, когда заранее неизвестен объём данных, с которыми предстоит работать, и предполагается увеличение или уменьшение их количества во время исполнения программы.

Для связи между элементами динамической структуры используют указатели.

Линейные списки могут быть *однонаправленными* и *двунаправленными*; и те и другие могут быть кольцевыми (закольцованными).

Список называют ***кольцевым***, если его последний элемент указывает на первый. Если кольцевой (или закольцованный) список двунаправленный, то каждый элемент указывает на следующий и предыдущий; соответственно, последний элемент настроен на первый и предпоследний, а первый элемент указывает на второй и последний.

Каждый элемент списка должен содержать два вида полей: *информационное* (их может быть несколько, типы и количество информационных полей зависят от решаемой задачи) и *поля-указатели*, служащие для связи между элементами.

Исходя из того что элемент списка должен объединять в себе разнотипные поля, в языке программирования *C++* его описывают как структуру.

В данном случае под словом *структура* понимают тип данных (в языке программирования Паскаль используют тип «запись»). Под упомянутым выше словосочетанием «структура данных» подразумевают подход к организации данных.

Таким образом, чтобы двигаться дальше, нужно чётко понимать разницу между понятиями *структура* как тип данных и *структура данных*.

В качестве примера рассмотрим описание элемента динамического списка. Это будет структура с двумя полями: информационным и полем-указателем на следующий элемент. Информационное поле будет целочисленным.

Как вы думаете, на какой тип данных нужно объявить поле-указатель? Попробуйте самостоятельно описать такую структуру, а потом свериться с кодом, приведённым в качестве примера

```
struct elem
{int inf;
 elem * p;};
```

Поле `p` имеет тип *указатель на элемент* (где элемент – это описанная выше структура).

Нельзя объявить переменную типа «структура» до завершения её описания, но можно объявить указатель на описываемую структуру в качестве поля этой структуры.

Схематично элемент списка показан на рис. 12.

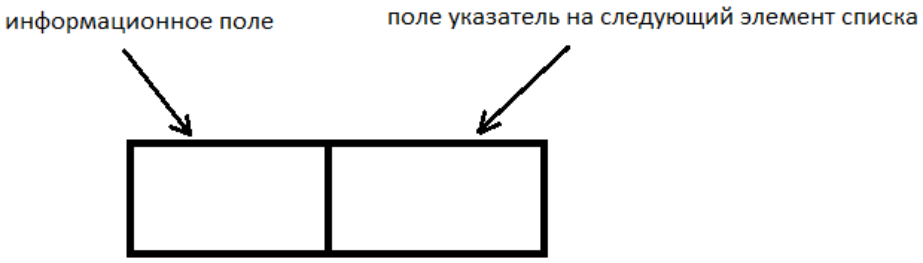


Рис. 12. Схематичное отображение элемента динамического списка

Создадим и проинициализируем элемент описанного выше типа

```
elem El = { 2, NULL };
```

На рис. 13 схематично показана структура данных *линейный список*, которую можно организовать, используя в качестве элементов описанный тип `El`.

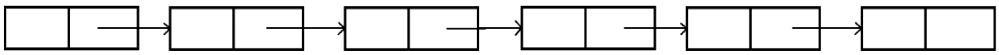


Рис. 13. Схематичное представление линейного списка

Из схемы, представленной на рис. 13, понятно, что для работы с линейным списком необходим как минимум указатель на его начало для обращения к элементам.

Первый элемент не может быть именованным, как в нашем примере (`elem El = { 2, NULL };`), так как в процессе работы он может быть удалён или стать не первым, а при работе со списком необходима возможность обращения в любой момент к его началу. Поэтому работа по созданию динамического списка начинается с объявления указателя на список. Этот указатель должен иметь тип указателя на эту структуру. Для дальнейшей корректной работы может понадобиться указатель на текущей элемент. Результат объявления настройки таких указателей показан на рис. 14.

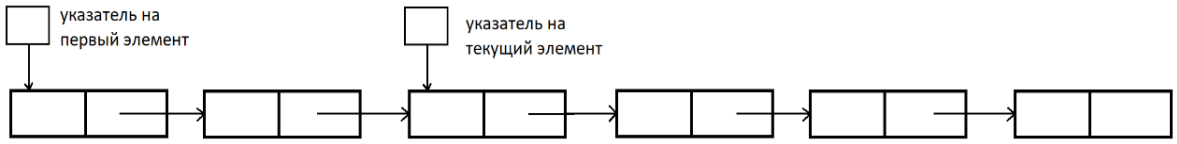


Рис. 14. Схематичное представление линейного списка и указателей, настроенных на первый и текущий элементы

Если список кольцевой, то последний элемент необходимо настроить на первый (рис. 15).

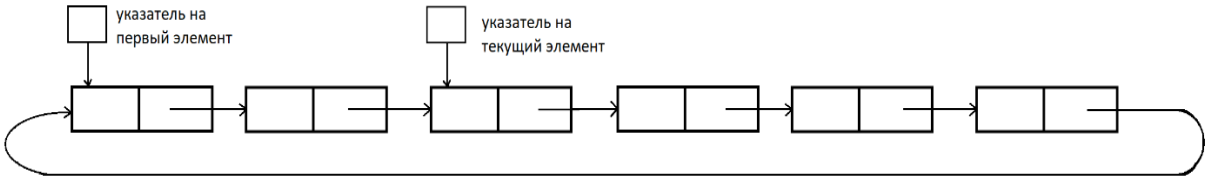


Рис. 15. Однонаправленный кольцевой список

Если список двунаправленный, каждый его элемент должен содержать два поля-указателя, настроенные на следующий и на предыдущий элементы

```
struct elem
{
    int data;
    elem* prev;
    elem* next;
};
```

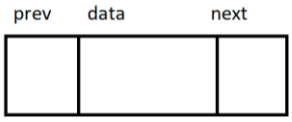


Рис. 16. Элемент двунаправленного списка

Схематично элемент типа `struct elem` представлен на рис. 16.

Если двунаправленный список – кольцевой, предыдущим элементом для первого является последний, а следующим для последнего – первый (рис. 17).

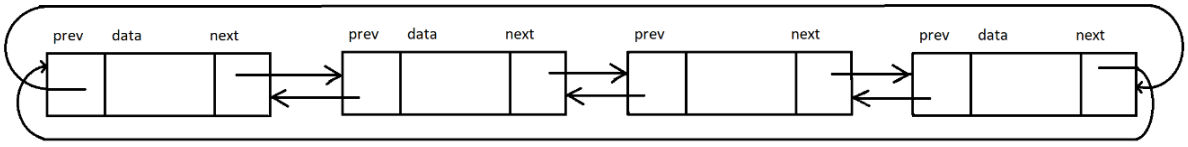


Рис. 17. Двунаправленный кольцевой список

Для работы с разобранными выше динамическими структурами данных необходимо определить следующие операции:

– создание первого элемента;
– добавление элемента в начало или конец списка;
– поиск элемента по ключу;
– вставка по ключу;

– упорядочивание по ключу;
– удаление элемента из начала или конца списка;
– удаление по ключу.

При добавлении и исключении элементов из списка необходимо осуществлять проверку списка на пустоту следующим образом:

```
head != NULL;
```

где `head` – указатель на первый элемент списка (иначе называемый головой, или вершиной, списка).

Понятно, что в зависимости от результатов проверки алгоритм добавления будет различным для первого и последующего элементов. То же касается и удаления. Предварительно нужно убедиться, есть ли элементы в списке.

Добавление элемента в однонаправленный линейный список

Добавление первого элемента:

– захватить память под элемент;

– заполнить соответствующим образом информационные поля, поле-указатель настроить на `NULL` (рис. 18).

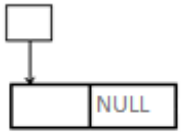


Рис. 18. Создан первый элемент списка

Последующие элементы:

– захватить память под элемент (рис. 19);

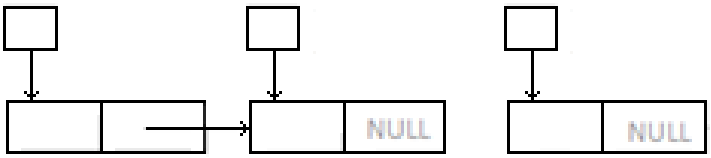


Рис. 19. Захвачена память под новый элемент, который будет добавлен к существующему списку

– настроить соответствующим образом поля-указатели данного элемента и предыдущего элемента списка в зависимости от того, куда будет добавлен новый элемент: в голову или хвост списка (рис. 20).

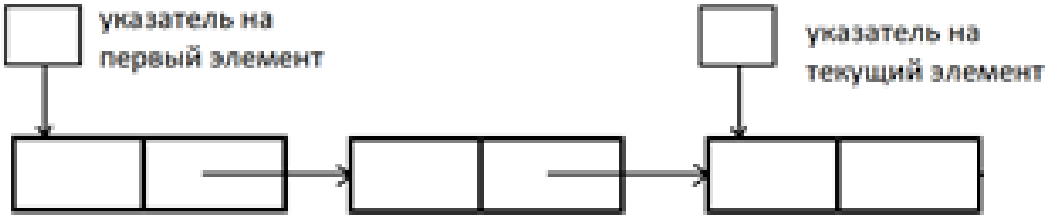


Рис. 20. Элемент добавлен в хвост линейного динамического списка

Вставка по ключу

Предварительно необходимо осуществить поиск по ключу нужного элемента.

Допустим, нам необходимо вставить новый элемент за элементом с искомым значением.

Для этого нужно настроить вспомогательный указатель на голову списка и перемещать его в цикле на следующий элемент до тех пор, пока не будет достигнуто искомое значение либо конец списка.

Для того чтобы найти искомое значение, нужно сравнивать ключ с информационным полем элемента списка, на который настроен вспомогательный указатель. Результат поиска схематично представлен на рис. 21.

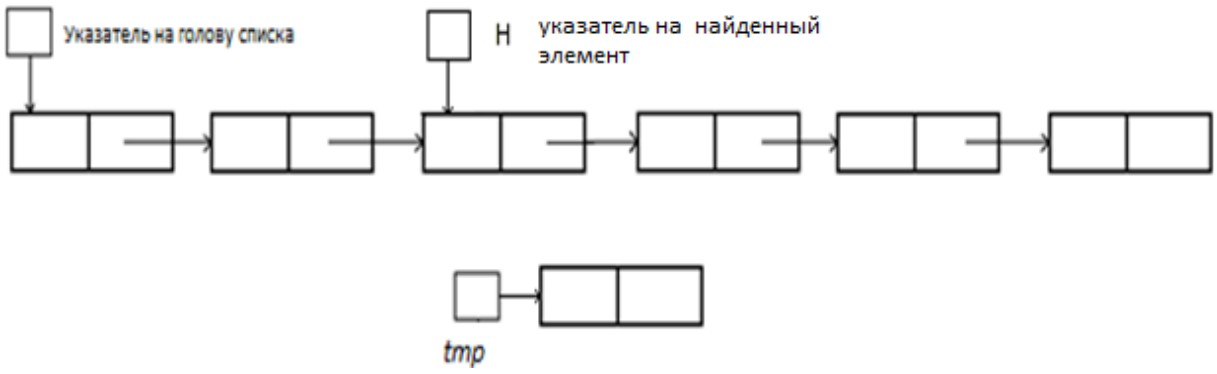


Рис. 21. Поиск элемента по ключу

```
int k;//ключ
. . .
if (H->inf==k) H=H->p;
```

где `H` – указатель на найденный элемент.

Далее необходимо настроить поле-указатель нового элемента списка на тот элемент, на который указывал элемент с совпавшим ключом

```
tmp->p=H->p;
```

А поле-указатель найденного элемента настроить на новый элемент

```
H->p=tmp;
```

Результат такой перенастройки представлен на рис. 22.

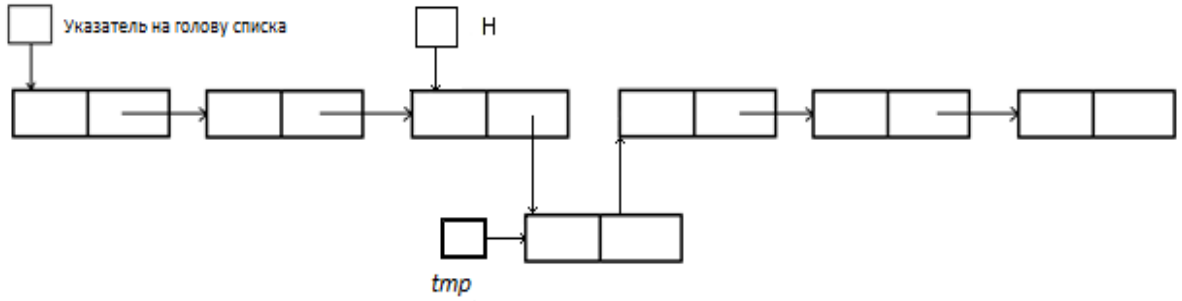


Рис. 22. Элемент вставлен в список

Если необходимо вставить новый элемент перед элементом с заданным ключом, то проверяют на совпадение с ключом элемент, следующий за элементом, на который настроен вспомогательный указатель.

Отдельно стоит рассмотреть ситуацию, когда необходимо вставить новый элемент перед первым элементом. На рис. 23 представлен следующий момент: новый элемент уже создан, но ещё не вставлен в список.

Рис. 23. Подготовка к добавлению элемента

В этом случае поле-указатель нового элемента настраивают на первый элемент, после чего перенастраивают указатель на голову списка на вновь добавленный элемент; либо поле-указатель первого элемента настраивают на новый в зависимости от организации направления списка. Таким образом, если на схеме стрелки будут направлены в другую сторону, это означает, что поля-указатели настроены на предыдущий элемент, а не на следующий. На рис. 24 схематично представлен результат добавления элемента в голову списка.

Рис. 24. Элемент добавлен в голову списка

Исключение элемента из списка

Прежде чем освободить память, занимаемую элементом, необходимо выполнить перенастройку указателей.

Алгоритм удаления первого элемента:

– настроить вспомогательный указатель на голову (рис. 25);

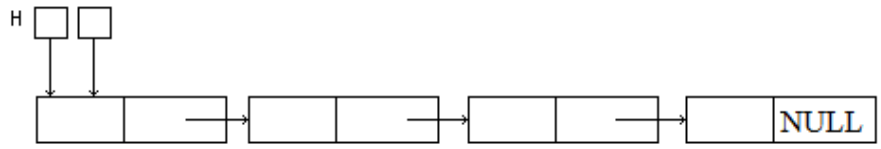


Рис. 25. Дополнительный указатель настроен на первый элемент динамического списка

– перенастроить указатель на голову на следующий элемент (рис. 26);

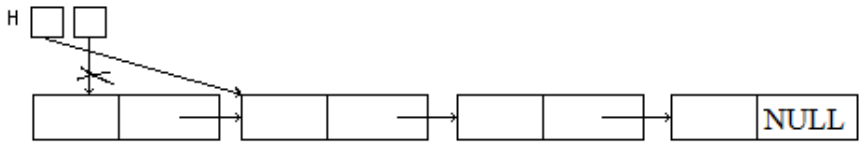


Рис. 26. Элемент удалён из головы динамического списка

– освободить память, занимаемую первым элементом.

Удаление элемента по ключу

Для реализации этой задачи необходимо два вспомогательных указателя: указатель на удаляемый элемент и указатель на предыдущий элемент.

Поле-указатель элемента, предшествующего удаляемому, настраиваем на элемент, следующий за удаляемым, после чего освобождаем память, занимаемую удаляемым из списка элементом.

Линейный список, организованный по принципу «первый вошёл – первый вышел» (*FIFO – first in, first out*), называют ***очередью***.

Для наглядности целесообразно представить очередь в бытовом понимании, например очередь в магазин: кто первый в очередь встал, тот первым и получит товар (так должно быть). Для динамического списка это означает, что элемент, который был добавлен в список первым, будет и извлечён из очереди первым.

Для реализации такого подхода необходимо объявить указатель на голову (вершину) списка и указатель на хвост очереди для возможности добавления элементов в конец очереди. Эти два указателя можно объединить в одной переменной – структуре. Эта структура будет не того же типа, что и элементы списка, это другой пользовательский тип данных. Этот тип должен быть описан отдельно после описания типа элемента списка. Например, так:

```
struct Q
{
    elem* head;
    elem* tail;
};
```

Если планируется работа только с одним списком, поля `head` и `tail` можно объявить отдельными переменными вне структуры.

Для добавления элемента в очередь потребуется дополнительный указатель

```
elem* tmp;
tmp = new elem;
```

Алгоритм добавления элемента в очередь:

– захватить память под новый элемент списка

```
elem* tmp;
tmp = new elem;
```

– заполнить его поля

```
tmp->inf = a;
```

– если это первый элемент списка (т. е. указатель на голову ни на что не указывает `head == NULL`), настраиваем его на новый элемент

```
head = tmp;
```

Этот же элемент будет последним в силу единственности, т. е. указатель на хвост нужно также настроить на него;

– если элемент не первый, то указатель на голову не трогаем. Перенастраиваем указатель на хвост на вновь захваченный элемент

```
tail = tmp;
```

– предварительно настраиваем на него же поле-указатель предыдущего элемента очереди

```
tail->p = tmp;
```

Линейный список, организованный по принципу «последний вошёл – первый вышел» (*LIFO – last in, first out*), называют *стеком.*

Рис. 27. Наглядное представление стека

Для понимания структуры такой организации данных и способов работы с ними стек представляют в виде узкой трубки, закрытой с одного конца и заполненной теннисными мячиками, как показано на рис. 27.

Из рис. 27 видно, что в противоположность очереди элемент, добавленный в стек первым, будет извлечён из него последним.

Добавление элемента в стек:

– на первом шаге захватываем память под элемент и заполняем его информационное поле

```
elem* tmp;
tmp = new elem;
tmp->inf = a;
```

– дальнейшая настройка зависит от того, первый это элемент или в стеке уже есть значения

```
(head == NULL);
```

– если элемент первый, указатель на вершину стека настраиваем на него

```
head = tmp;
tmp->p = NULL;
```

– если в стеке уже есть значения, производим перенастройку указателей таким образом, чтобы не потерять связь между элементами и при этом новый элемент стал вершиной стека

```
tmp->p = head;
head = tmp;
```

При написании функций работы с динамическими списками параметры-указатели передают через ссылки; если указатель передать по значению, изменённое значение не сохранится после выхода из функции.

Лабораторная работа № 7
ЛИНЕЙНЫЕ ДИНАМИЧЕСКИЕ СТРУКТУРЫ

Цель работы. Закрепление навыков разработки функций на языке программирования *C++*, в том числе способов передачи параметров в функции, возвращения значения из функции, использования этого значения в точке вызова.

Получение навыков организации динамических структур данных и дальнейшей работы с ними на языке программирования *C++*.

Постановка задачи. Реализовать создание линейных динамических структур (по принципу стека и очереди) для хранения целочисленных данных.

Для этого написать функции добавления элементов в стек и удаления их из стека; функции добавления и удаления элементов из очереди; в функциях предусмотреть проверку на пустоту списка.

При этом в функциях добавления элемента, если список пуст, необходимо создать первый элемент. Если список пуст и при этом вызвана функция извлечения элемента, нужно выдать соответствующее сообщение.

Создать и проинициализировать целочисленный массив.

В цикле перебора элементов добавить элементы в стек и очередь. Каждый элемент должен быть добавлен в обе структуры.

После того как все элементы добавлены, вывести элементы на экран из стека и из очереди.

Вопросы для самоконтроля и подготовки к защите лабораторной работы

1. Что такое динамическая структура данных?
2. Какие динамические структуры называют линейными?
3. Приведите примеры линейных динамических структур.
4. Каким образом должен быть описан элемент динамической структуры данных на языке программирования *C++?*
5. Назовите правило для организации линейной динамической структуры данных *очередь*.
6. Назовите правило для организации линейной динамической структуры данных *стек*.
7. Перечислите этапы добавления элемента в линейную динамическую структуру.

8. Перечислите этапы удаления элемента из линейной динамической структуры.

9. Каким образом нужно передать указатель на голову и хвост очереди в функцию добавления элемента?

10. Почему именно этим способом?

Задания для самостоятельной работы

1. Написать функцию добавления элемента в очередь.

2. Написать функцию добавления элемента в стек.

3. Написать функцию проверки линейного динамического списка на пустоту.

4. Написать функцию извлечения элементов из линейного динамического списка.

5. Написать функцию поиска элемента в списке по ключу.

6. Написать функцию удаления по ключу элемента из линейного динамического списка.

7. Написать функцию вставки по ключу элемента в линейный динамический список.

8. Написать функцию добавления элемента в однонаправленный кольцевой список.

9. Написать функцию добавления элемента в двунаправленный кольцевой список.

10. Написать функцию удаления элемента по ключу из двунаправленного кольцевого динамического списка.

11. Элементы целочисленного массива записать в очередь. Написать функцию извлечения элементов из очереди до тех пор, пока первый элемент очереди не станет чётным.

12. Даны две очереди, содержащие целые числа. Объединить очереди в одну таким образом, чтобы первой в результирующей очереди оказалась та, сумма элементов которой больше.

13. Даны две непустые очереди одинаковой длины. Объединить очереди в одну, в которой элементы исходных очередей чередуются.

14. Даны две непустые очереди. Элементы каждой из очередей упорядочены по возрастанию. Объединить очереди в одну с сохранением упорядоченности элементов.

15. Пусть имеются файл действительных чисел и некоторое число *A*. Используя очередь, вывести на экран сначала все элементы, меньшие числа *A*, а затем все остальные элементы.

9. ОБРАТНАЯ ПОЛЬСКАЯ ЗАПИСЬ

Рассмотрим классическую задачу преобразования формы записи арифметического выражения в обратную польскую запись (ОПЗ) с использованием стека.

ОПЗ – форма записи математических и логических выражений, в которой операнды расположены перед знаками операций. Такая форма записи не содержит скобок. В литературе можно встретить другие названия: обратная польская нотация, бесскобочная символика Лукашевича (названа именем польского математика Яна Лукашевича, предложившего такой подход в 1920 году).

Алгоритм преобразования выражения в ОПЗ с использованием стека

Входным значением алгоритма является строка, содержащая арифметическое выражение в привычной нам форме записи. На выходе получаем строку, содержащую запись этого выражения в обратной польской нотации, при этом выражение не содержит скобок и знаки операций расположены за операндами.

Для реализации алгоритма используется стек для переменных типа `char`. В стек будут помещены знаки операций и открывающая скобка.

На первом шаге рассматриваем поочерёдно каждый символ входной строки:

– если этот символ – *число* (или переменная), то просто помещаем его в выходную строку;

– если символ – *знак операции* (+, –, ×, /), то проверяем приоритет данной операции.

Операции умножения и деления имеют наивысший приоритет (положим его равным трём). Операции сложения и вычитания имеют меньший приоритет – два. Наименьший приоритет – единицу – имеет открывающая скобка.

Далее необходимо проверить стек.

Если стек все ещё пуст или находящиеся в нём символы (знаки операций и открывающая скобка) имеют меньший приоритет, чем приоритет текущего символа, то помещаем текущий символ в стек.

Если символ, находящийся на вершине стека, имеет приоритет, больший или равный приоритету текущего символа, то извлекаем символы из стека в выходную строку до тех пор, пока выполняется это условие; затем переходим к предыдущему пункту.

Если текущий символ – открывающая скобка, то помещаем её в стек. Если текущий символ – закрывающая скобка, то извлекаем символы из стека в выходную строку до тех пор, пока не встретим в стеке открывающую скобку (т. е. символ с приоритетом, равным единице), которую следует просто уничтожить. Закрывающая скобка также уничтожается.

Если вся входная строка разобрана, а в стеке ещё остаются знаки операций, извлекаем их из стека в выходную строку.

Рассмотрим алгоритм на примере выражения ***a + (b – c) d.***

Согласно изложенному выше алгоритму поочерёдно перебираем все символы входной строки.

Результат пошагового применения алгоритма приведён в табл. 1.

Таблица 1

Преобразование выражения в ОПЗ

Символ	Действие	Выходная строка после совершённого действия	Стек после совершённого действия
a	‘***a***’ – переменная. Помещаем её в выходную строку	***a***	пуст
+	‘+’ – знак операции. Помещаем его в стек (поскольку стек пуст, приоритеты можно не проверять)	***a***	+
(	‘(’ – открывающая скобка. Помещаем в стек	***a***	**+ (**
b	‘***b***’ – переменная. Помещаем её в выходную строку	***a b***	**+ (**
–	‘–’ – знак операции, который имеет приоритет, равный двум. Проверяем стек: на вершине находится символ ‘(’, приоритет которого равен единице. Следовательно, мы должны просто поместить текущий символ ‘–’ в стек	***a b***	**+ (–**

Окончание табл. 1

Символ	Действие	Выходная строка после совершённого действия	Стек после совершённого действия
c	‘***c***’ – переменная. Помещаем её в выходную строку	***a b c***	**+ (–**
)	‘)’ – закрывающая скобка. Извлекаем из стека в выходную строку все символы, пока не встретим открывающую скобку. Затем уничтожаем обе скобки	***a b c –***	**+**
×	‘×’ – знак операции, который имеет приоритет, равный трём. Проверяем стек: на вершине находится символ ‘+’, приоритет которого равен двум, т. е. он меньше, чем приоритет текущего символа ‘×’. Следовательно, мы должны просто поместить текущий символ ‘×’ в стек	***a b c –***	**+ ×**
d	‘***d***’ – переменная. Помещаем её в выходную строку	***a b c – d***	**+ ×**

Входная строка разобрана, но в стеке ещё остаются знаки операций, которые необходимо поочерёдно извлечь в выходную строку.

Поскольку стек – это структура, организованная по принципу *LIFO*, сначала извлекается символ ‘×’, затем символ ‘+’.

Получаем выражение в обратной польской нотации: ***a b c – d*** **× +**

Алгоритм вычисления выражения, записанного в ОПЗ

Для реализации этого алгоритма используют стек чисел (или переменных, если они встречаются в исходном выражении). Алгоритм достаточно прост. В качестве входной строки теперь рассматриваем выражение, записанное в ОПЗ.

1. Если очередной символ входной строки – число, то помещаем его в стек.

2. Если очередной символ – знак операции, то извлекаем из стека два верхних числа, используем их в качестве операндов для этой операции, затем помещаем результат обратно в стек.

Когда вся входная строка будет разобрана, в стеке останется одно число, которое и будет результатом данного выражения.

Рассмотрим этот алгоритм на примере выражения: 7 5 2 – 4 × +
Результат пошагового применения алгоритма приведён в табл. 2.

Таблица 2

Преобразование выражения в ОПЗ

Символ	Действие	Состояние стека после совершённого действия
7	‘7’ – число. Помещаем его в стек	7
5	‘5’ – число. Помещаем его в стек	7 5
2	‘2’ – число. Помещаем его в стек	7 5 2
–	‘–’ – знак операции. Извлекаем из стека два верхних числа (5 и 2) и совершаем операцию 5 – 2 = 3, результат которой помещаем в стек	7 3
4	‘4’ – число. Помещаем его в стек	7 3 4
×	‘×’ – знак операции. Извлекаем из стека два верхних числа (3 и 4) и совершаем операцию 3 × 4 = 12, результат которой помещаем в стек	7 12
+	‘+’ – знак операции. Извлекаем из стека два верхних числа (7 и 12) и совершаем операцию 7 + 12 = 19, результат которой помещаем в стек	19

Теперь строка разобрана и в стеке находится одно число – 19, которое является результатом исходного выражения.

Лабораторная работа № 8

ОБРАТНАЯ ПОЛЬСКАЯ ЗАПИСЬ

Цель работы. Закрепление навыков разработки функций на языке программирования *C++*, в том числе способов передачи параметров в функции, возвращения значения из функции, использования этого значения в точке вызова.

Закрепление навыков работы с линейными динамическими структурами, навыков разработки комбинированных алгоритмов и реализации их на языке программирования *C++*.

Постановка задачи. Преобразовать арифметическое выражение в ОПЗ. Реализовать вычисление полученного выражения. Алгоритмы преобразования выражения в ОПЗ и алгоритм вычисления выражения реализовать, используя функции организации данных в виде стека.

Вопросы для самоконтроля и подготовки к защите лабораторной работы

1. Что такое ОПЗ?
2. Перечислите этапы преобразования выражения в ОПЗ.
3. Назовите этапы вычисления выражения, записанного в ОПЗ.
4. Где может быть применена работа с выражениями, записанными в обратной польской нотации?
5. Назовите другие названия ОПЗ, встречающиеся в литературе.
6. Для чего в алгоритме перевода выражения в обратную польскую нотацию служит приоритет символа?
7. Какие действия необходимо выполнить, если при переводе выражения в ОПЗ встречается открывающая скобка?
8. Какие действия необходимо выполнить, если при переводе выражения в ОПЗ встречается закрывающая скобка?
9. При переводе выражения в ОПЗ происходит удаление из выражения скобок; за счёт чего при этом достигается корректность вычисления выражения?
10. Какое действие необходимо выполнить после совершения одной из математических операций при вычислении с использованием стека выражения, представленного в ОПЗ?

Задания для самостоятельной работы

Решить следующие задачи с использованием линейных динамических списков.

1. Написать функцию, получающую на вход арифметическое выражение в виде строки и преобразующую его в выражение в обратной польской нотации.

2. Создать многомодульный проект ОПЗ – калькулятор.

3. Организовать циклическую работу ОПЗ – калькулятора через запрос пользователю на повторный ввод выражения.

4. Реализовать динамическую структуру *дек* (список с возможностью добавления элементов с обоих концов).

5. Написать функцию извлечения элементов из дека.

6. Дано натуральное число N. Вывести на экран количество различных цифр, встречающихся в его десятичной записи; вывести на экран количество цифр и сами цифры через запятую.

7. Дана скобочная последовательность. Узнать, является ли она верной, т. е. проверить соблюдение скобочного баланса с учётом соответствия формы скобок.

8. По заданному текстовому файлу получить новый текстовый файл, в котором слова из первого файла расположены в обратном порядке.

9. Реализовать дек и методы работы с ним при помощи статического массива.

10. Расположить элементы одномерного целочисленного массива длиной N в обратном порядке с использованием стека.

10. НЕЛИНЕЙНЫЕ ДИНАМИЧЕСКИЕ СТРУКТУРЫ

Под ***нелинейными динамическими структурами данных*** понимают такой способ организации хранения данных, при котором количество данных в структуре может изменяться во время работы программы и порядок расположения элементов в памяти может не совпадать с порядком их добавления в структуру. При решении ряда задач, например поиска, такой подход к хранению данных эффективнее. К нелинейным динамическим структурам данных относят: графы, различные виды деревьев, мультисписки. В этом разделе в качестве примера нелинейной динамической структуры будут рассмотрены бинарные деревья.

Бинарным деревом называют определённым образом организованную динамическую структуру данных, позволяющую хранить большие объёмы данных с возможностью их быстрого поиска.

Далее рассмотрены двоичные деревья поиска. Элементы такой структуры называют *узлами (вершинами)*. На каждый узел имеется только *одна ссылка*. Узел содержит как минимум одно информационное поле и два поля-указателя на левое и правое поддеревья.

Наглядно двоичное дерево представлено на рис. 28.

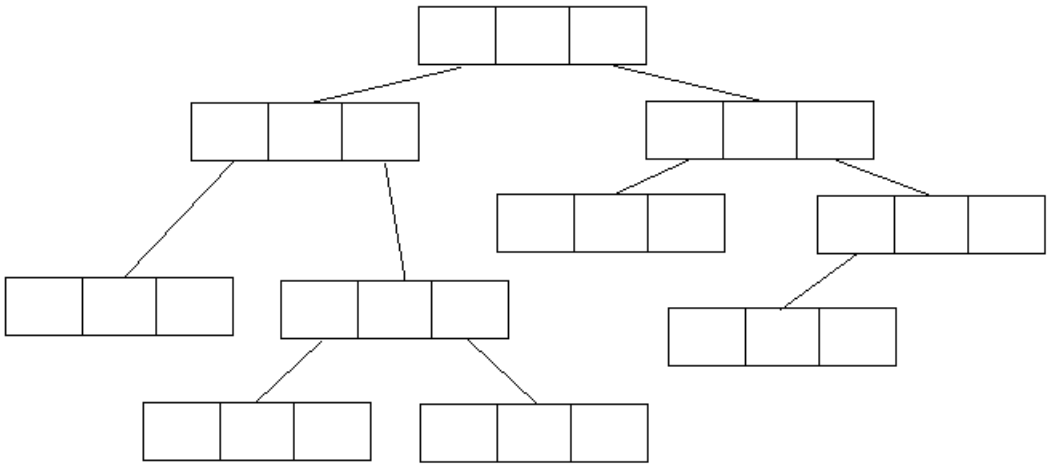


Рис. 28. Наглядное схематичное представление нелинейной динамической структуры «дерево»

При этом важно помнить, что такая иллюстрация отражает метод логической организации данных, а сами ячейки памяти располагаются по возрастанию адресов.

Добавление элементов в такую динамическую структуру подчинено следующему правилу: для каждого узла все ключи его левого поддерева меньше ключа рассматриваемого узла, а ключи правого поддерева больше ключа рассматриваемого узла. Одинаковые ключи не допускаются.

Поиск элемента в такой структуре осуществляется гораздо быстрее, чем в линейном списке.

При работе с двоичными деревьями применяют следующую терминологию.

Начальный узел принято называть *корнем дерева*.

Узел, не имеющий поддерева, называют *листом*.

Если существует путь от узла ***a*** до узла ***b***, то говорят, что узел ***a*** – *предок* узла ***b***, а узел ***b*** – *потомок* узла ***a.***

Самый длинный путь от корня до листа называют *высотой дерева*.

Так как каждое *поддерево* представляет собой дерево, очевидно, что эта структура рекурсивная, поэтому и действия с ней удобнее реализовывать при помощи рекурсивных функций, передавая в качестве параметра указатель на корень дерева.

Для бинарных деревьев определены следующие операции:

– включение узла в дерево;

– удаление узла из дерева;

– поиск значения по ключу;

– обход дерева, при обходе дерева узлы не удаляются.

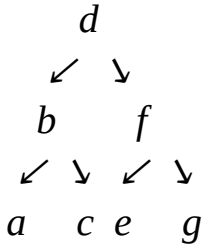
Рассмотрим три порядка обхода дерева: обход в симметричном порядке (симметричный обход (*inorder – LCR*)), обход в прямом порядке (обход сверху, или обход в ширину (*preorder – CLR*)), обход в обратном порядке (обход в глубину, обратный обход, обход снизу (*postorder – LRC*)).

При *обходе в симметричном порядке* (*LCR*) обрабатывается сначала левое поддерево, потом корень, а затем правое поддерево.

При *обходе в прямом порядке* (*CLR*) обрабатывается сначала корень, потом левое поддерево, а затем правое.

При *обходе в обратном порядке* (*LRC*) сначала обрабатывается левое поддерево, потом правое и, наконец, корень.

Пусть нам дано дерево



Результаты его обхода описанными выше способами представлены ниже.

Симметричный обход ***a b c d e f g***
Прямой обход ***d b a c f e g***
Обход снизу ***a c b e g f d***

Алгоритмы обхода дерева

Обход в прямом порядке. Каждый узел посещается до того, как были посещены его потомки. Для корня дерева рекурсивно выполняются следующие действия:

– посетить узел;
– обойти левое поддерево;
– обойти правое поддерево.

Симметричный обход. Посещаем сначала левое поддерево, потом узел, затем правое поддерево. Для корня дерева рекурсивно выполняются следующие действия:

– обойти левое поддерево;
– посетить узел;
– обойти правое поддерево.

Обход в обратном порядке. Узлы посещаются снизу вверх.

Для корня дерева рекурсивно выполняются следующие действия:

– обойти левое поддерево;
– обойти правое поддерево;
– посетить узел.

Из схемы, представленной на рис. 28, и всего вышесказанного следует, что элемент нелинейной динамической структуры – бинарное дерево динамической структуры – должен объединять в себе разнотипные поля. Это информационные поля, содержащие данные, для хране-

ния которых создаётся динамическая структура, и поля-указатели, служащие для организации этой структуры. Количество информационных полей определяется условиями задачи; полей типа *указатель на элемент* необходимо два.

Пример описания элемента динамической структуры *двоичное дерево*:

```
struct elem
{int data;
  elem* left;
  elem* right;
};
```

Для организации обращения к дереву необходим указатель на его начало – корень.

```
elem* root=NULL;
```

В приведённом примере `root` – имя указателя, `elem*` – его тип, указатель на структуру `elem`.

Операции над деревом реализуют в виде рекурсивных функций, передавая в них в качестве параметра ссылку на корень дерева. Передача этого параметра по значению не позволит изменить структуру за пределами функции. В функцию добавления элемента в дерево также необходимо передать значение информационного поля. Функции работы с деревом не должны возвращать значения, т. е. перед именем функции нужно указать `void.`

Пример вывода значения информационного поля элемента, на который настроен указатель `root:`

```
printf("%i",root->data);
```

Пример вызова функции обхода дерева:

```
viewTree(root->left);
```

Фрагмент кода добавления нового элемента в дерево:

```
elem* tmp=new elem; //захвачена память под новый элемент
tmp->data=a; //заполняем информационное поле
tmp->left= NULL; //записываем в указатели NULL
tmp->right= NULL;
*root=tmp; //настраиваем корень на новый элемент
```

При написании функции добавления элемента необходимо организовать правильное перемещение по узлам дерева в соответствии с правилом: *для каждого узла все ключи его левого поддерева меньше ключа рассматриваемого узла, а ключи правого поддерева больше ключа рассматриваемого узла. Одинаковые ключи не допускаются.*

Таким образом, если входной элемент меньше текущего, перемещаемся на один уровень влево, для этого рекурсивно вызываем функцию, передавая ей соответствующее поле-указатель текущего элемента; иначе – перемещаемся вправо; в ситуации равенства входного и текущего элементов переходим к следующему входному элементу.

Перемещаемся до тех пор, пока соответствующий указатель текущего элемента не будет пуст, настраиваем этот указатель на новый элемент, предварительно выделив для него память и заполнив поля.

Для добавления в дерево элементов, хранимых в массиве, необходимо вызывать функцию добавления элементов дерева в цикле перебора элементов массива. Например:

```
for (int i=0; i< n; i++){printf("%i ",mas[i]);
addTree(&root,mas[i]);}
```

Вывод на экран элементов массива в примере используется для демонстрации порядка добавленных элементов.

При организации обхода дерева необходимо правильно реализовать перемещение по поддеревьям.

Если в алгоритме указано обойти сначала левое поддерево, то перемещаемся по левой ветви до тех пор, пока указатель на левую ветвь не будет равен `NULL`

```
if(root) {
    viewTree(root->left);
.   .   .
}
```

Отображаем значение текущего элемента

```
printf("%i ",root->data);
```

Переходим к корню текущего элемента или правому поддереву в зависимости от вида обхода

```
viewTree(root->right);
```

Лабораторная работа № 9
НЕЛИНЕЙНЫЕ ДИНАМИЧЕСКИЕ СТРУКТУРЫ

Цель работы. Совершенствование навыков написания кода на языке программирования *C++* с использованием функций. Закрепление знаний о способах передачи параметров в функции, возвращении значений из функции. Закрепление навыков реализации рекурсивных алгоритмов. Получение навыков создания нелинейных динамических структур. Изучение и реализация алгоритмов обхода бинарного дерева.

Постановка задачи. Написать функцию добавления элементов в бинарное дерево (предусмотрев проверку на пустоту).

Написать рекурсивные функции симметричного, прямого обхода и обхода в обратном порядке.

Методические указания к работе

При описании элемента дерева информационное поле задать целочисленным.

При обходе дерева соответствующий элемент выводить на экран.

Создать нелинейную динамическую структуру «двоичное дерево», добавив в неё элементы целочисленного массива. Вызвать поочерёдно функции обхода дерева.

Вопросы для самоконтроля и подготовки к защите лабораторной работы

1. Какие структуры данных называют динамическими нелинейными?
2. Перечислите известные вам нелинейные структуры данных.
3. Для чего используют нелинейные структуры данных?
4. Какую структуру называют двоичным деревом?
5. Что такое корень дерева?
6. Что такое узел дерева?
7. Приведите пример описания на языке программирования *C++* элемента структуры данных «двоичное дерево».
8. Назовите правило добавления нового элемента в дерево.
9. Почему говорят, что дерево имеет рекурсивную структуру?
10. Назовите известные вам алгоритмы обхода деревьев.

Задания для самостоятельной работы

Реализовать следующие алгоритмы двумя способами: используя рекурсивный подход и циклический.

1. Прямой алгоритм обхода дерева.
2. Обратный алгоритм обхода дерева.
3. Симметричный алгоритм обхода дерева.

Реализовать следующие задачи, описав соответствующие функции.

4. Поиск значения по ключу в дереве.
5. Удаление элемента из дерева по ключу.
6. Подсчёт количества узлов дерева.

Используя написанные ранее функции и полученные знания, решить следующие задачи.

7. Дан указатель *P1* на корень непустого дерева. Вывести максимальное из значений его узлов.
8. Дан указатель *P1* на корень. Вывести количество листьев в этом дереве. Предусмотреть проверку корня на пустоту.
9. Дан указатель *P1* на корень. Вывести значение всех листьев в этом дереве. Предусмотреть проверку корня на пустоту.
10. Дан указатель *P1* на корень. Вывести значение и адрес максимального узла, не являющегося листом.
11. Дан указатель *P1* на корень. Вывести количество и значения нечётных узлов дерева, не являющихся листьями.
12. Дан указатель *P1* на корень. Вывести максимальное из значений узлов, не являющихся листьями.
13. Дан указатель *P1* на корень. Вывести сумму всех его чётных листьев.
14. Дан указатель *P1* на корень. Вывести сумму всех его нечётных элементов, не являющихся листьями.

ЗАКЛЮЧЕНИЕ

Работа по подготовке специалистов в области информационных технологий, прикладной математики, компьютерных наук предполагает освоение ими дисциплин программирования и получение навыков в области разработки, отладки, тестирования и модификации программного обеспечения. Работа специалиста в этой области немыслима без перечисленных выше навыков.

Овладение базовыми навыками программирования на языке высокого уровня *C++* даёт студенту возможность получения опыта в области разработки алгоритмических решений, представления решения в виде блок-схемы, самостоятельной разработки кода на языке программирования высокого уровня, использования и отладки для дальнейшей работы ранее написанного кода, в том числе и кода, написанного другими разработчиками.

Полученные знания и навыки будут применены студентами при решении широкого спектра учебных и профессиональных задач, а также для дальнейшего совершенствования профессионального мастерства.

РЕКОМЕНДАТЕЛЬНЫЙ БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Павловская, Т. А. C/C++. Программирование на языке высокого уровня : учеб. для вузов / Т. А. Павловская. – СПб. : Питер, 2009. – 432 с. – ISBN 978-5-91180-174-8.

2. Страуструп, Б. Программирование. Принципы и практика с использованием C++ / Б. Страуструп. – М. : Вильямс, 2016. – 1328 с. – ISBN 978-5-8459-1949-6.

3. Шилдт, Г. Полный справочник по C++ : пер. с англ. / Г. Шилдт. – 4-е изд., стер. – М. : Вильямс, 2015. – 800 с. – ISBN 978-5-8459-2047-8.

4. Воронова, Л. М. Типовые алгоритмические структуры для вычислений : учеб. пособие / Л. М. Воронова ; Владим. гос. ун-т. – 2-е изд., перераб. и доп. – Владимир : Ред.-издат. комплекс ВлГУ, 2004. – 96 с. – ISBN 5-89368-503-2.

5. Прата, С. Язык программирования C++. Лекции и упражнения : пер. с англ. / С. Прата. – М. : Вильямс, 2017. – 1248 с. – ISBN 978-5-8459-2048-5.

ПРИЛОЖЕНИЯ

Приложение 1

Структурные схемы основных алгоритмических конструкций

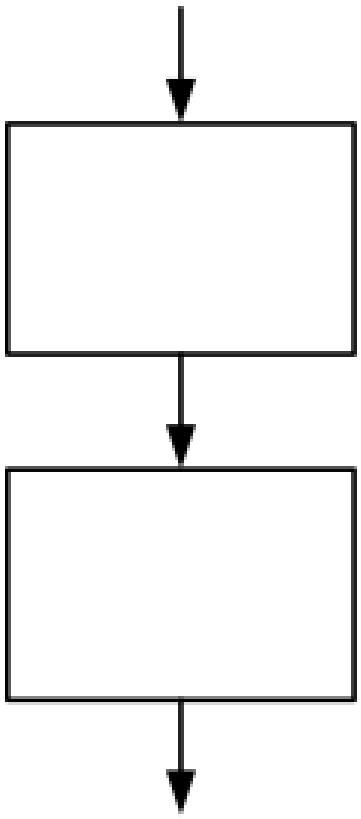
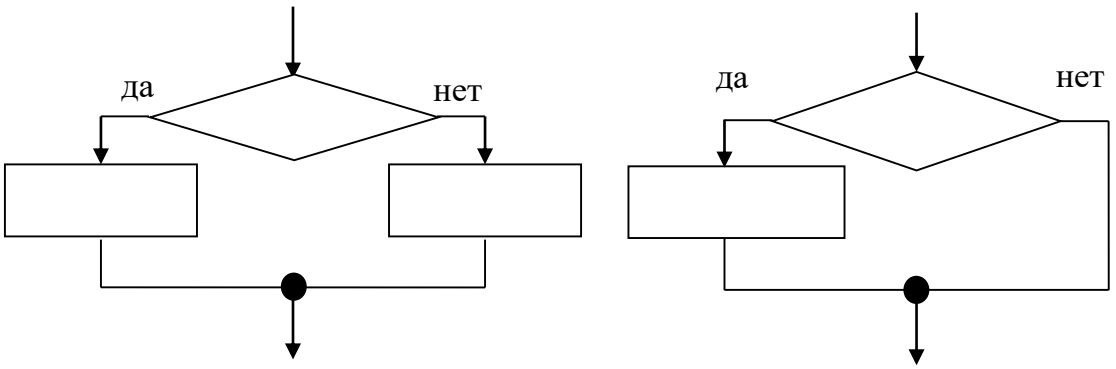
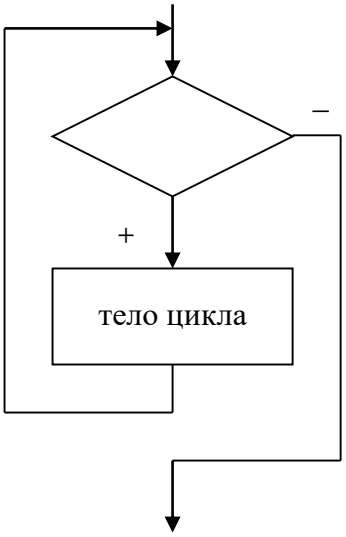


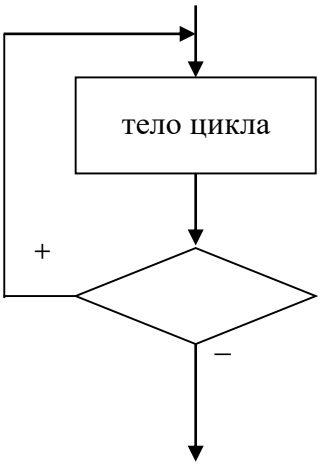
Рис. П.1. Схематичное представление алгоритмической структуры «следование»



а) *б)*

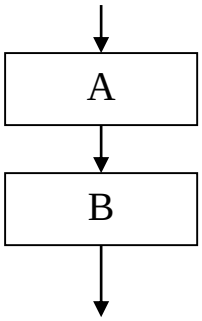
Рис. П2. Алгоритмическая структура «ветвление»: а – полное ветвление; б – неполное ветвление

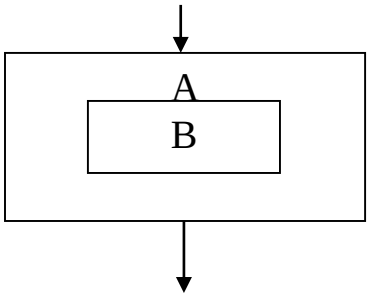




а) *б)*

Рис. П3. Алгоритмическая конструкция «повторение»:
а – цикл с предусловием; б – цикл с постусловием



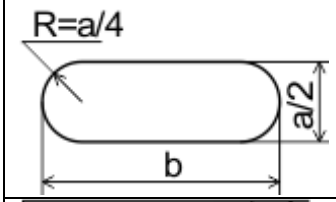
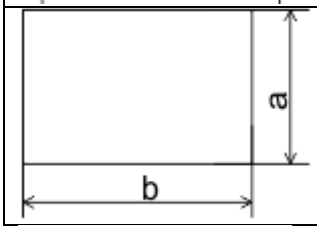
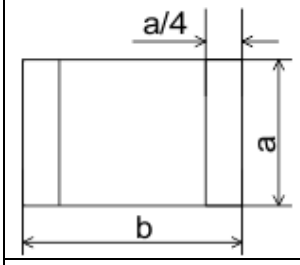
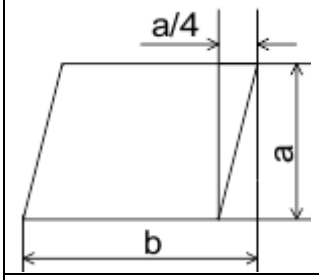
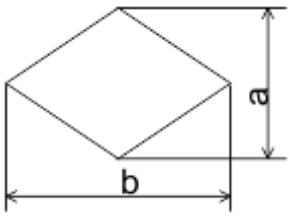
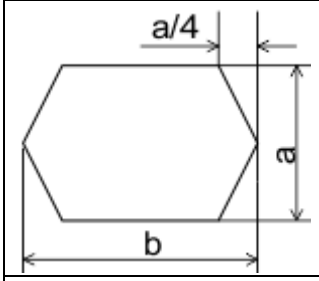


а) *б)*

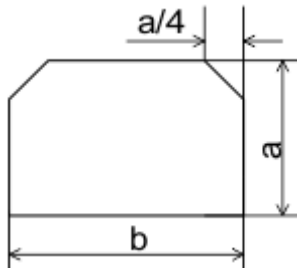
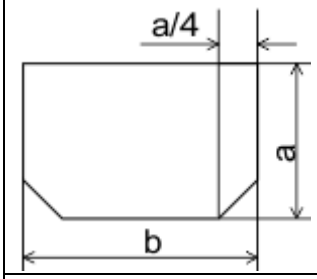
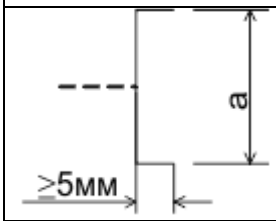
Рис. П4. Алгоритмическая конструкция «повторение»:
а – последовательные циклы; б – вложенный цикл

Приложение 2

Основные блоки

Форма блока	Назначение блока
	**Терминатор** Обозначает вход и выход во внешнюю среду. «Начало», «Конец» либо прерывание обработки данных
	**Процесс** Обозначает одно или несколько действий обработки данных. Выполнение этих действий ведёт к изменению данных, их значения или формы хранения
	**Предопределённый процесс** (функция) Использование ранее созданных и отдельно описанных алгоритмов или программ
	**Данные** (ввод – вывод) Используют при вводе или выводе данных. Преобразование данных в форму, подходящую для обработки (ввод) или отображения результатов обработки (вывод)
	**Решение** (условие), или переключатель Выбор направления выполнения алгоритма. Блок содержит некоторое условие, имеет один вход и несколько альтернативных выходов. По какой ветви будет произведена дальнейшая работа – определяется значением этого условия. Значения могут быть подписаны рядом с соответствующими линиями
	**Подготовка** (цикл с параметром) Может содержать установку переключателя, модификацию индекса

Окончание

Форма блока	Назначение блока
Начало цикла  Конец цикла  	**Граница цикла** Состоит из двух частей, отображающих начало и конец цикла, имеющих один и тот же идентификатор; условия инициализации, приращения, завершения цикла помещают в начале или конце цикла в зависимости от того, какой это цикл: с предусловием или с постусловием
 	**Соединитель** Используют при необходимости обрыва линии и продолжении её в другом месте. Соответствующие символы-соединители должны содержать одно и то же уникальное обозначение
 	**Комментарий** Используют для описания и/или пояснения каких-то действий
 	Пропуск Три точки используют для отображения пропущенного символа или группы символов

Приложение 3

Базовые типы данных *C++*

В таблице представлены основные типы данных языка *C++*.

В первом столбце указано зарезервированное слово, определяющее тип данных.

Во втором столбце указано количество байт, отводимое под переменную с соответствующим типом данных.

В третьем столбце приведён диапазон допустимых значений.

Тип	Байт	Диапазон допустимых значений
Целочисленный (логический) тип данных		
bool	1	0 / 255
Целочисленный (символьный) тип данных		
char	1	0 / 255
Целочисленные типы данных		
short int	2	–32 768 / 32 767
unsigned short int	2	0 / 65 535
int	4	–2 147 483 648 / 2 147 483 647
unsigned int	4	0 / 4 294 967 295
long int	4	–2 147 483 648 / 2 147 483 647
unsigned long int	4	0 / 4 294 967 295
Типы данных с плавающей точкой		
float	4	–2 147 483 648.0 / 2 147 483 647.0
long float	8	–9 223 372 036 854 775 808.0 / 9 223 372 036 854 775 807.0
double	8	–9 223 372 036 854 775 808.0 / 9 223 372 036 854 775 807.0

Приложение 4

Образец титульного листа отчёта по лабораторной работе

Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования
«Владимирский государственный университет
имени Александра Григорьевича и Николая Григорьевича Столетовых»
(ВлГУ)

Кафедра ФиПМ

ЛАБОРАТОРНАЯ РАБОТА № 8

по дисциплине «Основы программирования»
на тему: «Нелинейные динамические структуры»

Выполнил:
ст. группы ПМИ-122
Иванов И. И.
Принял:
ст. преподаватель каф. ФиПМ
Шишкина М. В.

Владимир 2022

Приложение 5

Требования к форматированию отчёта по лабораторной работе

Отчёт по лабораторной работе должен быть создан в текстовом процессоре *Microsoft Word*. Отчёт должен быть распечатан на одной стороне чистого белого листа формата А4, не допускается использование оборотной стороны использованных листов.

Титульный лист должен быть оформлен в соответствии с примером, приведённым в прил. 4.

При наборе текста используется шрифт *Times New Roman*, кегль 14 пт.

Поля задать следующим образом: левое поле – 30 мм; правое – 10 мм; верхнее и нижнее – 20 мм. Междустрочный интервал полуторный; режим выравнивания по ширине; отступ в начале абзаца (красная строка) 15 мм.

Все листы должны быть пронумерованы. Титульному листу номер присваивают, но не проставляют его. Следовательно, номер страницы, следующей после титульного листа, – 2. Номер проставляют по центру листа, кегль 12 пт.

Для допуска к защите лабораторной работы необходимо продемонстрировать работоспособность написанной программы, а также представить отчёт, написанный и оформленный в соответствии с требованиями.

Для защиты работы необходимо ответить на вопросы по её теоретической части, подробно пояснить строки кода программы, указанные преподавателем. Самостоятельно выполнить выданное преподавателем задание, по теме и уровню сложности соответствующее заданию лабораторной работы.

Приложение 6

Требования к содержанию отчёта

Отчёт по лабораторной работе должен содержать следующие части.

1. Титульный лист, оформленный по образцу, приведённому в прил. 4.

2. Цель работы.

3. Постановку задачи.

Формулировка цели работы и постановка задачи должны точно соответствовать формулировкам, указанным преподавателем.

4. Пять – семь ключевых слов на русском и английском языках. В качестве ключевых слов стоит выбирать слова, опираясь на которые можно составить исчерпывающий ответ по теме работы.

5. Краткая теоретическая часть должна содержать от двух до четырёх страниц связного самостоятельно написанного текста с примерами. Теоретическая часть не должна быть копией лекций, но лекционный материал должен быть взят за основу. При этом должны быть рассмотрены все вопросы, затронутые в работе, с теоретической точки зрения и на примерах, реализованных автором.

6. Ход работы должен содержать программный код, написанный на изучаемом языке программирования, и несколько тестов, результатов работы программы с объяснением полученных результатов.

7. Выводы по работе. В этой части необходимо указать, к какому выводу пришёл автор во время выполнения работы. Например: достоинства и недостатки изученного метода, разработанного алгоритма; для решения какого класса задач используют этот подход.

Приложение 7

Таблицы кодов *ASCII*

0 -	16 - ▶	32 -	48 - 0	64 - @	80 - P	96 - '	112 - p
1 - ☺	17 - ◀	33 - !	49 - 1	65 - A	81 - Q	97 - a	113 - q
2 - ☻	18 - ↕	34 - "	50 - 2	66 - B	82 - R	98 - b	114 - r
3 - ♥	19 - !!	35 - #	51- 3	67 - C	83 - S	99 - c	115 - s
4 - ♦	20 - ¶	36 - $	52 - 4	68 - D	84 - T	100 - d	116 - t
5 - ♣	21 - §	37 - %	53 - 5	69 - E	85 - U	101 - e	117 - u
6 - ♠	22 - ▬	38 -&	54 - 6	70 - F	86 - V	102 - f	118 - v
7 -	23 - ↕	39 - '	55 - 7	71 - G	87 - W	103 - g	119 - w
8 -	24 - ↑	40 - (	56 - 8	72 - H	88 - X	104 - h	120 - x
9 -	25 - ↓	41 -)	57 - 9	73 - I	89 - Y	105 - i	121 - y
10 -	26 - →	42 - *	58 - :	74 - J	90 - Z	106 - j	122 - z
11 -	27 - ←	43 - +	59 - ;	75 - K	91 - [	107 - k	123 - {
12 -	28 - ∟	44 - ,	60 - <	76 - L	92 - \	108 - l	124 - |
13 -	29 - ↔	45 - -	61 - =	77 - M	93 - j	109 - m	125 - }
14 - ♫	30 - ▲	46 - .	62 ->	78 - N	94 - ^	110 - n	126 - ~
15 - ☼	31 - ▼	47 - /	63 - ?	79 - O	95 - ÷	111 - o	127 - ⌂
16 - ▶	32 -	48 -0	64 - @	80 - P	96 -	112 - p	

128 -А	144 - Р	160 - а	176 - ▒	192 - └	208 - ╨	224 -р	240 - Ё
129 - Б	145 - С	161 - б	177 - ▒	193 - ┴	209 - ╤	225 -с	241- ё
130 -В	146 - Т	162 - в	178 - ▓	194 - ┬	210 - ╥	226 -т	242 - Є
131 - Г	147 - У	163 - г	179 - |	195 - ├	211 - ╙	227 -у	243 - є
132 -Д	148 - Ф	164 - д	180 - ┤	196 - —	212 - ╘	228 -ф	244 - Ї
133 -Е	149 - Х	165 - е	181 - ╡	197 - ┼	213 - ╒	229 - х	245 - ї
134 - Ж	150 - Ц	166 - ж	182 - ╢	198 - ╞	214 - ╓	230 - ц	246 - Ў
135 -З	151 - Ч	167 - з	183 - ╖	199 - ╟	215 - ╫	231 - ч	247 - ў
136 -И	152 -Ш	168 - и	184 - ╕	200 - ╚	216 - ╪	232 - ш	248 - °
137 - Й	153 -Щ	169 - й	185 - ╣	201 - ╔	217 - ┘	233 - щ	249 - ●
138- К	154 - Ъ	170 - к	186 - ║	202 - ╩	218 - ┌	234 - ъ	250 - .
139 -Л	155 - Ы	171 - л	187 - ╗	203 - ╦	219 - ▉	235 - ы	251 - √
140 -М	156 - Ь	172 - м	188 - ╝	204 - ╠	220 - ▄	236 - ь	252- №
141 -Н	157 - Э	173 - н	189 - ╜	205 - ═	221 - ▌	237 - э	253 - ¤
142 -О	158 - Ю	174 - о	190 - ╛	206 - ╬	222 - ▐	238 - ю	254 - ▮
143- П	159 - Я	175 - п	191 - ┐	207 - ╧	223 - ▀	239 - я	255 -
144 - Р	160 - а	176 - ▒	192 - └	208 - ╨	224 - р	240 - Ё	

ОГЛАВЛЕНИЕ

Учебное издание

ШИШКИНА Мария Викторовна

ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ ВЫСОКОГО УРОВНЯ С++

Практикум

Редактор Е. А. Платонова

Технические редакторы Ш. В. Абдуллаев, Н. В. Пустовойтова

Компьютерная верстка Л. В. Макаровой

Выпускающий редактор А. А. Амирсейидова

Подписано в печать 28.06.22.
Формат 60×84/16. Усл. печ. л. 6,05. Тираж 77 экз.
Заказ
Издательство
Владимирского государственного университета
имени Александра Григорьевича и Николая Григорьевича Столетовых.
600000, Владимир, ул. Горького, 87.