Министерство образования Республики Беларусь
Учреждение образования
«Белорусский государственный университет
информатики и радиоэлектроники»

Факультет информационных технологий и управления

Кафедра информационных технологий автоматизированных систем

А. А. Навроцкий

ОСНОВЫ АЛГОРИТМИЗАЦИИ И ПРОГРАММИРОВАНИЯ В СРЕДЕ VISUAL C++

Рекомендовано УМО по образованию в области информатики и радиоэлектроники в качестве учебно-методического пособия для специальностей общего высшего, специального высшего образования, закрепленных за УМО

2-е издание, дополненное и пересмотренное

Минск БГУИР 2026

УДК 004.4’6(076)
ББК 32.973.26-018.2я73
Н15

Р е ц е н з е н т ы:

кафедра естественно-научных дисциплин
ГУО «Университет Национальной академии наук Беларуси»
(протокол № 10 от 25.03.2025);

директор ООО «Софтарекс Технолоджиес» А. А. Пыхтин

Навроцкий, А. А.

Н15 Основы алгоритмизации и программирования в среде Visual C++ : учеб.-метод. пособие / А. А. Навроцкий. – 2-е изд., доп. и пересм. – Минск : БГУИР, 2026. – 156 с. : ил.

ISBN 978-985-543-842-8.

Содержит теоретические сведения о языке C++. Рассмотрены примеры написания программ в среде Microsoft Visual Studio C++. Представлены задания для лабораторных работ.

Адресовано студентам 1–2 курсов университета, изучающим дисциплину «Основы алгоритмизации и программирования».

Первое издание выпущено в 2014 году.

УДК 004.4’6(076)
ББК 32.973.26-018.2я73

ISBN 978-985-543-842-8

СОДЕРЖАНИЕ

ВВЕДЕНИЕ **7**
ТЕОРЕТИЧЕСКИЙ РАЗДЕЛ **8**
1. Базовые элементы языка C++ **8**
1.1. Идентификаторы 8
1.2. Ключевые слова 8
1.3. Знаки операций 9
1.4. Константы 9
1.5. Разделители 9
1.6. Комментарии 9
1.7. Структура программы C++ 9
1.8. Директивы препроцессора 10
1.9. Стандартные библиотеки C++ 10
1.10. Функции библиотеки cmath 11
1.11. Потоковый ввод/вывод данных 12
2. Базовые типы данных **14**
2.1. Типы данных 14
2.2. Объявление переменных и констант 14
2.3. Целый тип данных 14
2.4. Символьный тип данных 15
2.5. Вещественный тип данных 17
2.6. Логический тип данных 18
2.7. Тип void 18
2.8. Объявление auto 18
2.9. Математические константы 19
2.10. Неявное преобразование типов 19
2.11. Явное преобразование типов 21
3. Операции в языке C++ **22**
3.1. Арифметические операции 22
3.2. Операция присваивания 22
3.3. Операции сравнения 23
3.4. Логические операции 23
3.5. Поразрядные логические операции 24
3.6. Приоритет операций в C++ 24
3.7. Использование блоков 26
4. Организация разветвляющихся алгоритмов **27**
4.1. Оператор условного ветвления (if-else) 27
4.2. Условный оператор 28
4.3. Оператор множественного выбора 28

5. Организация циклических алгоритмов 31
5.1. Оператор цикла for 31
5.2. Оператор цикла while 33
5.3. Оператор цикла do-while 33
5.4. Операторы и функции передачи управления 33
5.5. Организация циклических алгоритмов 35
6. Использование массивов 38
6.1. Одномерные массивы 38
6.2. Алгоритмы работы с одномерными массивами 39
6.3. Многомерные массивы 40
6.4. Алгоритмы работы с двумерными массивами 41
7. Использование указателей 44
7.1. Объявление указателя (необработанного указателя) 44
7.2. Операции над указателями 44
7.3. Инициализация указателей 46
7.4. Работа с динамической памятью 46
7.5. Создание одномерного динамического массива 47
7.6. Создание двумерного динамического массива 47
8. Использование строковых переменных 49
8.1. Объявление строк 49
8.2. Функции для работы со строками 49
8.3. Алгоритмы работы со строками 54
9. Типы данных, определяемых пользователем 56
9.1. Объявление и использование структур 56
9.2. Объявление и использование объединений 59
9.3. Объявление и использование перечислений 60
10. Использование файлов 61
10.1. Понятие файла 61
10.2. Функции для работы с файлами 61
11. Функции 69
11.1. Понятие функции 69
11.2. Параметры функции 71
11.3. Перегрузка функций 79
11.4. Встраиваемые функции 80
11.5. Указатель на функцию 81
11.6. Ссылка на функцию 83
12. Область видимости и классы памяти 84
13. Рекурсивные алгоритмы 85
13.1. Понятие рекурсии 85

13.2. Условие окончания рекурсивного алгоритма 86
13.3. Типы рекурсивных алгоритмов 86
13.4. Примеры рекурсивных алгоритмов 86
13.5. Целесообразность использования рекурсии 89
14. Алгоритмы сортировки 91
14.1. Простые методы сортировки 91
14.2. Улучшенные методы сортировки 93
15. Алгоритмы поиска 99
15.1. Линейный поиск 99
15.2. Двоичный (бинарный) поиск 99
15.3. Интерполяционный поиск 100
16. Хеширование 101
16.1. Понятие хеширования 101
16.2. Схемы хеширования 102
16.3. Хеш-таблица с линейной адресацией 102
16.4. Хеш-таблицы с квадратичной и произвольной адресацией 104
16.5. Хеш-таблица с двойным хешированием 105
16.6. Хеш-таблица на основе связанных списков 105
16.7. Метод блоков 107
17. Динамические структуры данных 108
17.1. Понятие списка, стека и очереди 108
17.2. Работа со стеком 109
17.3. Работа со однонаправленной очередью 112
17.4. Работа с двусвязанными списками 113
17.5. Работа с двусвязанными циклическими списками 116
18. Нелинейные списки 117
18.1. Древовидные структуры данных 117
18.2. Использование древовидных структур 118
18.3. Двоичное дерево поиска 119
19. Синтаксический анализ арифметических выражений 128
19.1. Алгоритм преобразования выражения в форму ОПЗ 128
ЛАБОРАТОРНЫЙ ПРАКТИКУМ 134
Лабораторная работа 1. Программирование линейных алгоритмов 134
Лабораторная работа 2. Программирование разветвляющихся алгоритмов 136
Лабораторная работа 3. Программирование циклических алгоритмов 138
Лабораторная работа 4. Использование одномерных массивов 140
Лабораторная работа 5. Использование двумерных массивов 141
Лабораторная работа 6. Программирование с использованием строк 142

Лабораторная работа 7. Программирование с использованием структур 142
Лабораторная работа 8. Программирование с использованием файлов 144
Лабораторная работа 9. Использование функций 144
Лабораторная работа 10. Программирование рекурсивных алгоритмов ... 146
Лабораторная работа 11. Алгоритмы сортировки 147
Лабораторная работа 12. Алгоритмы поиска 148
Лабораторная работа 13. Хеширование 149
Лабораторная работа 14. Использование стеков 149
Лабораторная работа 15. Использование двусвязанных списков 150
Лабораторная работа 16. Работа с бинарным деревом поиска 151
Лабораторная работа 17. Вычисление алгебраических выражений 152
ПРИЛОЖЕНИЕ. РАБОТА В СРЕДЕ MICROSOFT VISUAL C++ 153
1. Консольный режим работы 153
2. Выполнение программы 153
3. Отладка программы 154
СПИСОК РЕКОМЕНДОВАННОЙ ЛИТЕРАТУРЫ 155

ВВЕДЕНИЕ

Учебно-методическое пособие предназначено для изучения дисциплины «Основы алгоритмизации и программирования» (ОАиП). В нем содержатся сведения об элементарных конструкциях языка *C++*, а также программировании алгоритмов на структурах данных.

Учебно-методическое пособие состоит из двух частей – теоретической части и лабораторного практикума.

Теоретическая часть охватывает основы программирования на языке *C++*, а также программирование динамических структур данных, таких как стеки, очереди, деревья, алгоритмов сортировки и поиска, хеширования, рекурсивных и других алгоритмов.

Лабораторный практикум состоит из 17 работ, которые позволят получить практические знания на основе изученного материала.

Материал учебно-методического пособия рассчитан на последовательное освоение материала – от простых понятий к более сложным. Пропуск любой главы при изучении может не позволить качественно освоить материал последующих глав.

Учебно-методическое пособие содержит большое количество примеров, в которых рассматриваются особенности использования изучаемых концепций и механизмов программирования на языке *C++*.

Все примеры программ были выполнены и проверены в *Microsoft Visual Studio C++* в 64-разрядных системах.

ТЕОРЕТИЧЕСКИЙ РАЗДЕЛ

1. Базовые элементы языка *C++*

Алфавит языка *C++* включает прописные и строчные буквы латинского алфавита, арабские числа, специальные, пробельные и разделительные символы.

Из символов алфавита формируются ***токены*** – наименьшие конструкции языка, имеющие значение для компилятора.

К токенам относятся: идентификаторы, ключевые слова, знаки операций, константы, разделители.

1.1. Идентификаторы

Идентификатор – последовательность цифр и букв латинского алфавита, а также специальных символов при условии, что первой стоит буква или знак подчеркивания. Идентификатор чуствителен к регистру (идентификаторы, для образования которых используются совпадающие строчные и прописные буквы, считаются различными).

	aa, AA, Aa, aA – различные идентификаторы.

В идентификаторе допустимо использовать любое количество символов, однако значимыми считаются только первые 2048.

При выборе идентификатора:

– необходимо следить, чтобы идентификатор не совпадал с ключевыми зарезервированными словами и именами библиотечных функций;

– нежелательно использование идентификаторов, начинающихся со знака подчеркивания или с двойным знаком подчеркивания в любом месте, так как такие идентификаторы используются в стандартной библиотеке (их использование может привести к некорректной работе программы).

	Общепринятые правила именования идентификаторов: – имена переменных и функций пишутся строчными буквами; – имена типов начинаются с прописной буквы; – имена констант пишутся прописными буквами; – имя идентификатора отражает внутреннюю сущность объекта.

1.2. Ключевые слова

Ключевое слово – зарезервированный токен, имеющий специальное значение. Некоторые ключевые слова, определенные стандартом *C++*: auto, bool, break, case, catch, char, class, const, const_cast, constexpr, continue, default, delete, do, double, dynamic_cast, else, enum, explicit, extern, false, float, for, friend, goto, if, inline, int, long, namespace, new, noexcept, nullptr, operator,

private, protected, public, register, reinterpret_cast, return, short, signed, sizeof, static, static_cast, struct, switch, template, this, throw, true, try, typedef, typeid, typename, unsigned, using, virtual, void, volatile.

1.3. Знаки операций

Знак операции – один или несколько символов, определяющих действие над операндами. Использование пробелов внутри знака операции не допускается.

1.4. Константы

Константа – токен, значение которого не может быть изменено во время выполнения программы.

Для объявления констант используются ключевые слова const и constexpr:

```
const double pi = 3.14159265359;
constexpr int y = 5;
```

Отличие const от constexpr состоит в том, что constexpr может быть применен и к функциям.

	Задание констант с помощью директивы препроцессора #define в *C++* не рекомендуется.

1.5. Разделители

Элементы языка *C++* отделяются друг от друга символами-разделителями. В качестве символов-разделителей используются пустые значения (пробелы), а также знаки препинания (например, точка с запятой, фигурные скобки, кавычки и др.).

1.6. Комментарии

Комментарий – текстовая или символьная информация, используемая для пояснения участков программы. Комментарии не влияют на ход выполнения программы, так как не являются токенами и не включаются в содержимое исполняемого файла (компилятор обрабатывает их как пробел).

В *C++* комментарии:

– начинаются последовательностью «//» и заканчиваются концом строки;

– начинаются последовательностью «/*» и заканчиваются последовательностью «*/» (вложение комментариев не разрешено).

1.7. Структура программы *C++*

Программа *C++* состоит из одной или нескольких функций. Обязательным является присутствие функции main, которой передается управление при запуске программы.

Вид функции:

```
int main()
{
// Тело функции
}
```

После завершения выполнения функции main завершается выполнение программы. Функция должна возвращать значение целого типа (если возвращаемое значение не указано, то компилятор возвращает нуль).

Упрощенная структура программы имеет вид:

```
<Директивы препроцессора>
<Описание типов пользователя и глобальных переменных>
<Прототипы функций>
<Описание глобальных переменных>
<Функция main()>
<Определения пользовательских функций>
```

1.8. Директивы препроцессора

Препроцессор – специальная часть компилятора, обрабатывающая директивы до начала процесса компиляции программы. Директива препроцессора начинается с символа «#», который должен быть первым символом строки, затем следует название директивы. В конце директивы точка с запятой не ставится. В случае необходимости переноса директивы на следующую строку применяется символ «\».

Для подключения к программе заголовочных файлов используется директива include. Если идентификатор файла заключен в угловые скобки, то поиск файла будет вестись в стандартном каталоге, если в двойные кавычки, то поиск проводится в следующем порядке:

- каталог, в котором содержится файл, включивший директиву;
- каталоги файлов, которые были уже включены директивой;
- текущий каталог программы;
- каталоги, указанные опцией компилятора «\I»;
- каталоги, заданные переменной окружения include.

Обработка препроцессором директивы include сводится к тому, что на место директивы помещается копия указанного в директиве файла.

Директива define используется для написания макросов и определения символических констант (не рекомендуется использование в *C++*).

1.9. Стандартные библиотеки *C++*

При создании исполняемого файла к исходной программе подключаются библиотечные файлы (как правило, имеют расширение *lib*), содержащие уже откомпилированный набор функций. Для осуществления связи с библиотечным

файлом к программе подключается (с помощью директив препроцессора) заголовочный файл, который содержит информацию об именах и типах функций, расположенных в библиотеке. На этапе компиляции компоновщик извлекает из библиотечных файлов используемые в программе функции.

	В *C++* заголовочные файлы не имеют расширения, например, #include <iostream>. Для файлов, унаследованных от *C*, следует указывать расширение: #include <math.h>.

1.10. Функции библиотеки *cmath*

Все аргументы в тригонометрических функциях задаются в радианах. Параметры остальных функций имеют тип double. Некоторые математические функции перечислены в табл. 1.1.

Таблица 1.1

Математическая функция	Функция библиотеки *math.h*	Содержание вычислений
$\|x\|$	abs(x) fabs(x)	Вычисление абсолютного значения числа. Например: s = abs(-3.9) → Результат: *s* = 3.9 s = fabs(-4.5) → Результат: *s* = 4.5
arccos(*x*)	acos(x)	Вычисление значения арккосинуса числа *x*. Значение *x* может быть задано только из диапазона −1...1. Например: s = acos (0.4) → Результат: *s* = 1.15928 s = acos (1.5) → Результат: *s* = −1.#IND
arcsin(*x*)	asin(x)	Вычисление значения арксинуса числа *x*. Значение *x* может быть задано только из диапазона –1...1. Например: s = asin (−1) → Результат: *s* = −1.5708 s = asin (0.9) → Результат: *s* = 1.11977
arctg(*x*)	atan(x)	Вычисление значения арктангенса. В результате выполнения функции возвращается число из диапазона −π/2…π/2. Например: x = atan (3.5) → Результат: *s* = 1.2925
$\sqrt[3]{x}$	cbrt(x)	Возвращает значение кубического корня от *x*
cos(*x*)	cos(x)	Вычисление cos(*x*)
ch(*x*)	cosh(x)	Вычисление косинуса гиперболического

Окончание табл. 1.1

Математическая функция	Функция библиотеки *math.h*	Содержание вычислений
e^x	exp(x)	Вычисление экспоненты числа *x*
Минимум	fmin(x, y)	Вычисление минимального значения из *x* и *y*
Максимум	fmax(x, y)	Вычисление максимального значения из *x* и *y*
Остаток от деления *x* на *y*	fmod(x,y)	Функция возвращает действительное значение, соответствующее остатку от деления *x* на *y*. Например: fmod (3, 4) → Результат: *s* = 3 fmod (6.4, 3.1) → Результат: *s* = 0.2
ln(*x*)	log(x)	Вычисление натурального логарифма *x*
$\lg_{10}(x)$	log10(x)	Вычисление десятичного логарифма *x*
x^y	pow(x, y)	Возведение *x* в степень *y*
Округление	round(x)	Функция возвращает ближайшее к аргументу целое значение
sin(*x*)	sin(x)	Вычисление sin(*x*)
sh(*x*)	sinh(x)	Вычисление синуса гиперболического *x*
$\sqrt{x}$	sqrt(x)	Вычисление квадратного корня *x*
tg(*x*)	tan(x)	Вычисление тангенса *x*
tgh(*x*)	tanh(x)	Вычисление тангенса гиперболического *x*
Выделение целой части	trunc(x)	Выделение целой части числа trunc (3.4) → Результат: *s* = 3

1.11. Потоковый ввод/вывод данных

Поток – это логическое устройство, которое осуществляет передачу данных от источника к приемнику. В библиотеке *iostream* определены четыре страндартных потока:

cout – стандартный поток ввода, направлен из оперативной памяти на внешнее устройство (по умолчанию – на экран компьютера);

cin – стандартный поток вывода, направлен от внешнего устройства (по умолчанию – с клавиатуры) в оперативую память;

cerr – поток, предназначенный для вывода на экран сообщений об ошибках и программной диагностики. Объект немедленно выводится на экран, минуя буферизацию. Поток нельзя перенаправить;

clog – поток, предназначенный для ведения журналов. Не может быть перенаправлен, но проходит буферизацию.

Для работы с потоками применяются операции вставки в поток (<<) и извлечения из потока (>>).

Введем, например, с клавиатуры переменную *x* и выведем ее на экран:

```
cout << "Enter value x: " << endl;
 cin >> x;
cout << "x =  " << x << endl;
```

Здесь манипулятор endl переводит курсор в начало следующей строки.

> В языке *C* для перехода на новую строку использовали управляющий символ «\n». Манипулятор endl, появившийся в *C++*, кроме переноса строки производит сброс буферов потока вывода, что повышает надежность программы, но несколько снижает скорость ее выполнения.

Для управления вводом/выводом используются флаги или манипуляторы форматирования. *Флаги* устанавливают параметры ввода/вывода, которые будут действовать на все последующие операторы до тех пор, пока не будут отменены. *Манипуляторы* (библиотека *iomanip*) помещаются в операторы ввода/вывода непосредственно перед форматируемым значением. В табл. 1.2 приведены некоторые манипуляторы форматирования.

Таблица 1.2

Манипулятор	Описание
setw(n)	Задает ширину поля вывода в *n* символов
setprecision(m)	Задает количество цифр (*m*-1) в дробной части числа
boolalpha	Вывод логических величин в текстовом виде
scientific	Экспоненциальная форма вывода вещественных чисел
fixed	Фиксированная форма вывода вещественных чисел (по умолчанию)

2. Базовые типы данных

2.1. Типы данных

Тип данных позволяет определить, какие значения могут принимать переменные, какая их структура, какое количество ячеек используется для их размещения и какие операции допустимо над ними выполнять

Данные можно разбить на две группы: скалярные (простые) и структурированные (составные).

К ***скалярному (простому) типу*** относятся данные, представляемые одним значением из определенного диапазона.

Структурированные (составные) типы определяются как комбинация скалярных и описанных ранее структурированных типов.

Базовыми типами данных являются: целый, действительный, логический и символьный.

Данные могут быть ***константами*** и ***переменными***. В отличие от переменных константы не могут изменять свое значение во время выполнения программы.

2.2. Объявление переменных и констант

Объявление переменных можно сделать в любом месте программы до первого их использования.

	Для повышения читабельности объявление значимых переменных и констант желательно делать в начале программы.

При объявлении сначала указывается тип данных, а затем через запятую список переменных данного типа, например:

```
int  x, y, z;
double  a, b;
```

Используются две формы объявления переменных:

– объявление, не приводящее к выделению памяти;

– объявление, при котором выделяется память в соответствии с указанным типом.

При объявлении с выделением памяти можно сразу инициализировать переменную (присвоить ее начальное значение), например:

```
int sum = 0, k = 10;
```

2.3. Целый тип данных

Целый тип данных предназначен для хранения чисел, не имеющих дробной части. В языке *C++* определены следующие целые типы данных:

int (__int32, signed**)** – занимает 4 байта памяти, может хранить значения из диапазона от –2 147 483 648 до 2 147 483 647;

long (long int) – занимает 4 байта памяти, может хранить значения из диапазона от –2 147 483 648 до 2 147 483 647; в 64-разрядных системах совпадает с типом int;

short (short int) – занимает 2 байта памяти, может хранить значения из диапазона от –32 768 до 32 767. Данный тип данных использовать нежелательно, так как имея меньшую длину, он обрабатывается медленнее типа int.

long long – занимает 8 байт памяти, может хранить значения из диапазона от –9 223 372 036 854 775 808 до 9 223 372 036 854 775 807.

Для смещения границ диапазона только в положительную область используется атрибут **unsigned**. Например, unsigned int имеет длину от 0 до 4 294 967 295.

Константы целого типа – последовательность цифр, начинающаяся со знака «минус» для отрицательных констант и со знака «плюс» (или без него) для положительных констант. Для обозначения констант типа long после числа ставят букву L или l.

Константы могут быть представлены в различных системах счисления.

Десятичные константы: последовательность чисел от 0 до 9, начинающаяся не с нуля, например 334.

Восьмеричные константы: последовательность чисел от 0 до 7, начинающаяся с нуля, например 045.

Шестнадцатеричные константы: последовательность чисел от 0 до 9 и букв от *A* до *F*, начинающаяся с символов 0x, например 0xF5C3.

2.4. Символьный тип данных

Символьный тип предназначен для хранения одного символа, для чего достаточно выделить 1 байт памяти. Данные такого типа рассматриваются компилятором как целые, поэтому в переменных типа signed char можно хранить целые числа из диапазона –128…127. Для хранения символов используется unsigned char, который позволяет хранить 256 символов кодовой таблицы *ASCII* (*American Standard Code for Information Interchange* – американский стандартный код для обмена информацией). Стандартный набор символов *ASCII* использует только 7 битов для каждого символа (диапазон 0…127). Добавление восьмого разряда позволило увеличить количество кодов таблицы *ASCII* до 255. Коды от 128 до 255 представляют собой расширение таблицы *ASCII* для хранения символов национальных алфавитов, а также символов псевдографики.

Значения кодовой таблицы *ASCII* с номерами 0…32 и 127 содержат непечатаемые символы, которые не имеют графического представления, но влияют на отображение текста. Символы с кодами 32…127 представлены в табл. 2.1. Символы с кодами 128…255 (кодовая таблица 866 – *MS-DOS*) представлены в табл. 2.2.

Таблица 2.1

Код	Символ	Код	Символ	Код	Символ	Код	Символ
32	пробел	56	8	80	P	104	h
33	!	57	9	81	Q	105	i
34	“	58	:	82	R	106	j
35	#	59	;	83	S	107	k
36	$	60	<	84	T	108	l
37	%	61	=	85	U	109	m
38	&	62	>	86	V	110	n
39	'	63	?	87	W	111	o
40	(	64	@	88	X	112	p
41	)	65	A	89	Y	113	q
42	*	66	B	90	Z	114	r
43	+	67	C	91	[	115	s
44	,	68	D	92	\	116	t
45	-	69	E	93	]	117	u
46	.	70	F	94	^	118	v
47	/	71	G	95	_	119	w
48	0	72	H	96	`	120	x
49	1	73	I	97	a	121	y
50	2	74	J	98	b	122	z
51	3	75	K	99	c	123	{
52	4	76	L	100	d	124	|
53	5	77	M	101	e	125	}
54	6	78	N	102	f	126	~
55	7	79	O	103	g	127	del

Таблица 2.2

Код	Символ	Код	Символ	Код	Символ	Код	Символ
128	А	160	а	192	└	224	р
129	Б	161	б	193	┴	225	с
130	В	162	в	194	┬	226	т
131	Г	163	г	195	├	227	у
132	Д	164	д	196	─	228	ф
133	Е	165	е	197	┼	229	х
134	Ж	166	ж	198	╞	230	ц
135	З	167	з	199	╟	231	ч
136	И	168	и	200	╚	232	ш
137	Й	169	й	201	╔	233	щ
138	К	170	к	202	╩	234	ъ

Окончание табл. 2.2

Код	Символ	Код	Символ	Код	Символ	Код	Символ
139	Л	171	л	203	╦	235	ы
140	М	172	м	204	╠	236	ь
141	Н	173	н	205	═	237	э
142	О	174	о	206	╬	238	ю
143	П	175	п	207	╧	239	я
144	Р	176	▒	208	╨	240	Е
145	С	177	▒	209	╤	241	е
146	Т	178	▓	210	╥	242	Є
147	У	179	│	211	╙	243	є
148	Ф	180	┤	212	╘	244	Ї
149	Х	181	╡	213	╒	245	ї
150	Ц	182	╢	214	╓	246	Ў
151	Ч	183	╖	215	╫	247	ў
152	Ш	184	╕	216	╪	248	°
153	Щ	185	╣	217	┘	249	·
154	Ъ	186	║	218	┌	250	·
155	Ы	187	╗	219	█	251	√
156	Ь	188	╝	220	▄	252	№
157	Э	189	╜	221	▌	253	¤
158	Ю	190	╛	222	▐	254	■
159	Я	191	┐	223	▀	255	

Значение переменной символьного типа записывается в одиночных кавычках.

2.5. Вещественный тип данных

Вещественный тип данных характеризуется присутствием в числе дробной части. Число представляется в экспоненциальной форме: $\pm n.mE\pm p$, где $n.m$ – мантисса (n – целая часть, m – дробная часть), p – порядок.

В языке *C++* используются следующие типы вещественных данных:

float – занимает 4 байта памяти, может хранить значения из диапазона от $3.4 \cdot 10^{-38}$ до $3.4 \cdot 10^{+38}$. Данный тип позволяет хранить числа с точностью до семи знаков после запятой. Не рекомендуется к использованию в *C++*;

double (long double) – занимает 8 байт памяти, может хранить значения из в диапазона от $1.7 \cdot 10^{-308}$ до $1.7 \cdot 10^{+308}$. Данный тип позволяет хранить числа с точностью до 15 знаков после запятой.

Вещественное число хранится в памяти компьютера в нормализованной форме ($1 \le M < 2$). При нарушении нормализации мантиссу сдвигают влево до тех пор, пока старшей цифрой мантиссы не станет единица, которая называется

неявной единицей и в памяти не хранится. Порядок числа сдвигается таким образом, чтобы весь интервал значений находился в положительной области (знак «плюс» в памяти не хранится). Сэкономленные 2 бита позволяют повысить точность хранения числа.

При использовании вещественных констант после числа добавляется буква F для типа float, D – для типа double (по умолчанию) и L для типа long double.

2.6. Логический тип данных

Логический тип bool занимает 1 байт памяти, может принимать два значения: *true* (1) или *false* (0). Так как тип bool занимает 1 байт памяти, то он может получать значения от 0 до 255. Значения от 1 до 255 трактуются как *true* (1), а значение 0 – как *false* (0).

```
        bool b;
cout.setf(ios::boolalpha); // Вывод логического 0 или 1,
                           // соответственно, как «false» или «true»
    b = true;     cout << b << endl;     // Выводит: true
    b = 1;        cout << b << endl;     // Выводит: true
    b = 225;      cout << b << endl;     // Выводит: true
    b = false;    cout << b << endl;     // Выводит: false
    b = 0;        cout << b << endl;     // Выводит: false
```

2.7. Тип void

Тип описывает пустой набор значений. Тип, как правило, используется для описания функций, не возвращающих значение, или для объявления нетипизированных указателей (для использования требуется их приведение к определенному типу). Запрещено объявление переменных типа void, ссылок и константных указателей типа const void*.

2.8. Объявление auto

Ключевое слово auto используется для объявления переменной, тип которой определяется исходя из типа инициализацирующего выражения.

Формат:

auto инициализатор = инициализирующее выражение;

Например:

```
auto x = 5;              // x – переменная типа int
auto y = 7.8;            // y – переменная типа double
auto m1 = { 1, 2, 3 };   // m1 – массив типа int
auto m2 = { 1.5, 2.4 };  // m2 – массив типа double
```

2.9. Математические константы

Математические константы находятся в библиотеке *numbers* (пространство имен std::numbers):

```
#include <numbers>
using namespace std::numbers;
```

Математические константы представлены в табл. 2.3.

Таблица 2.3

Константа	Математическая формула	Значение
e	Число e	2.718281828459045
log2e	$\log_2(e)$	1.4426950408889634
log10e	$\log_{10}(e)$	0.4342944819032518
ln2	$\ln(2)$	0.6931471805599453
ln10	$\ln(10)$	2.302585092994046
pi	π	3.141592653589793
inv_pi	$1/\pi$	0.3183098861837907
inv_sqrtpi	$1/\sqrt{\pi}$	0.5641895835477563
sqrt2	$\sqrt{2}$	1.4142135623730951
sqrt3	$\sqrt{3}$	1.7320508075688772
inv_sqrt3	$1/\sqrt{3}$	0.5773502691896257

2.10. Неявное преобразование типов

Неявное преобразование типов – автоматическое преобразование компилятором данных одного типа в данные другого типа в случае, если требуется согласование этих типов данных.

Неявное преобразование типов происходит, если:

- операнды имеют различный тип данных;
- типы аргументов функций отличаются от типов формальных параметров.

Неявное преобразование при выполнении арифметических операций. Данные преобразуются к типам, имеющим больший приоритет (увеличивается слева направо):

int → unsigned int → long → unsigned long → long long → unsigned long long → float → double → long double.

	Преобразование данных типа long long (unsigned long long) в тип float может привести к потере точности.

Данные типов bool, char, signed char, unsigned char, short и unsigned short всегда преобразуются к типу int (или к типу с еще большим приоритетом).

Логические преобразования. Если операция требует использования логического значения (например, в операторе if), то любой скалярный тип неявно преобразуется в тип bool.

Неявное преобразование типов является безопасным по определению. Это достигается повышением уровня приоритета (увеличением размера типа данных).

При использовании операции присваивания различают расширяющее и сужающее преобразование.

При ***расширяющем преобразовании*** значение переменной с меньшим приоритетом присваивается переменной с большим приоритетом. Операция является безопасной (потеря данных невозможна).

Например, необходимо вычислить: s = a + b, где *s* − переменная типа double, *a* − символ, *b* − переменная типа int. Пусть a = 'd', b = 45. Выражение будет рассчитываться следующим образом. Так как в арифметическом выражении присутствуют две переменные различных типов, то переменная с меньшим приоритетом (*a*) будет приведена к типу int. Для этого в памяти компьютера создается временная переменная типа int, которая будет хранить номер символа 'd', равный 100 (см. табл. 2.1). После этого выполняется операция суммирования, результат которой будет равен 145 (100 + 45). Полученное значение (145) присваивается переменной *s*. При выполнении операции присваивания преобразование типа не происходит, но размер переменной типа double больше размера переменной типа int, поэтому потеря информации в этом случае не происходит.

При ***сужающем преобразовании*** значение переменной с большим приоритетом присваивается переменной с меньшим приоритетом. При таком преобразовании возможна потеря данных.

Преобразование из типа с плавающей запятой в целочисленный тип всегда является узким преобразованием, так как дробная часть отбрасывается и теряется.

При нахождении узкого преобразования компилятор выполняет его неявным образом, но выдает предупреждение. Предупреждение не останавливает компиляцию, однако результат выполнения программы может быть неверным:

```
int m = 3.2; // warning C4244: инициализация:
        // преобразование "double" в "int", возможна потеря данных
    cout << m; // Выводит: 3
    cout << endl;
    short k = 500000; // warning C4305: инициализация:
                        // усечение из "int" в "short"
// warning C4309 : инициализация: усечение константного значения
    cout << k; // Выводит: -24288
```

```
cout << endl;
int n = INT_MAX + 1; // warning C4307: "+":
                     // переполнение целой константы со знаком
cout << n; // Выводит: -2147483648
cout << endl;
bool b = m;
cout << b; // Выводит: 1
```

2.11. Явное преобразование типов

Если неявное преобразование типов не приводит к требуемому результату, можно сделать преобразование типов явным образом:

```
static_cast <тип> (переменная)
```

Например,

```
int a = INT_MAX, b = INT_MAX;
int s = (a * b) / a;
    cout << s << endl;  // Выводит: 0 - Ошибка!
int f = (static_cast <double> (a) * b) / a;
    cout << f << endl;   // Выводит: 2147483647
```

При расчете переменной *s* вычисленное значение произведения *a* на *b* выходит за границы диапазона значений, которые могут храниться в переменной типа int. Создаваемая для хранения промежуточного результата временная переменная типа int получает ошибочную информацию, поэтому результат вычисления будет неверным.

В следующей строке переменная *a* явным образом приводится к типу double. Следовательно, результат вычисления произведения будет храниться во временной переменной наибольшего по длине типа (из int и double) − типа double. Полученный результат не выходит за границы диапазона значений типа double, поэтому ошибка не возникает.

Оператор static_cast возвращает ошибку, если типы, используемые для приведения, полностью несовместимы.

В языке *C* использовалась в настоящее время устаревшая (не рекомендуемая разработчиком) форма приведения типов:

```
(тип) переменная
```

или

```
тип (переменная)
```

3. Операции в языке *C++*

3.1. Арифметические операции

Язык *C++* содержит пять простых бинарных операций: «+», «–», «*», «/» и «%». Операции сложения, вычитания, умножения и деления применимы ко всем базовым типам. Операция получения остатка от деления «%» используется только для работы с целочисленными данными.

Например: 10 % 6 = 4, 7 % 10 = 7, 10 % 5 = 0.

3.2. Операция присваивания

Формат операции:

операнд1 **=** *операнд2*;

В *операнд1* заносится значение *операнда2*.

В качестве *операнда1* можно только *lvalue*. В качестве *операнда2* можно использовать как *lvalue,* так и *rvalue*.

	lvalue (изменяемое выражение слева от оператора присваивания) – объект, занимающий идентифицируемое место в памяти (имеет адрес). *lvalue* не может быть константой, функцией, массивом. *rvalue* (выражение справа от оператора присваивания), в качестве которого может использоваться константа, переменная, выражение, массив или функция. Все *lvalue* могут использоваться как *rvalue*, но не все *rvalue* могут быть *lvalue*.

Допустимо использовать следующее написание:

a = b = c = d; что равнозначно a = d; b = d; c =d ;

Для сокращения записи конструкции

операнд1 **=** *операнд1* ***знак_операции*** *операнд2*;

можно использовать конструкцию

операнд1 ***знак_операции=*** *операнд2*;

Например, оператор

x = x + 2;

можно заменить оператором

x += 2;

Если *операнд2* для операций суммирования и вычитания равен единице (или минус единице), то используются операции инкремента:

операнд1++;

или декремента

операнд1--;

Например, вместо оператора i = i + 1 используется оператор i++, а вместо оператора i = i - 1 оператор i--.

Операторы инкремент или декремент могут иметь две формы: *префиксную* (например, ++i) или *постфиксную* (например, i++). Форма оператора определяет его приоритет при выполнении операций в выражении.

При использовании префиксной формы оператора сначала выполняется инкремент или декремент, а затем арифметические операции. Если используется постфиксная форма, то сначала выполняются арифметические операции, а затем инкремент или декремент.

3.3. Операции сравнения

Операции сравнения применяются при работе с двумя операндами и возвращают *true* (1), если результат сравнения – истина, и *false* (0), если результат сравнения – ложь. В языке *C* определены следующие операции сравнения:

< (меньше), **<=** (меньше или равно), **>** (больше),

>= (больше или равно), **!=** (не равно), **= =** (равно).

Операнды должны иметь одинаковый тип (допустимо сравнивать целый и действительный типы).

	В *C*++20 добавлен оператор трехстороннего сравнения (<=>). Результатом выполнения операции является объект, содержащий результат сравнения на больше, меньше и равно.

3.4. Логические операции

Логические операции возвращают результат (ложно или истинно) сравнения операндов скалярных типов. В *C++* определены три логические операции: «!», «&&» и «||» (табл. 3.1).

Таблица 3.1

Логическая операция	Обозначение	Результат выполнения	Пример

НЕ (**!**) (унарная)	**!**a
	или
	not a
		Возвращает *true*, если *a=false*, и *false*, если *a=true*	a = 3;
			x = **!**(a > 0);
			Результат: *x= false*

И (**&&**) (бинарная)	a **&&** b
	или
	a **and** b
		Возвращает *true*,
		если *a=true* **и** *b=true*,
		иначе возвращает *false*
			a = 7;
			x = (a > 0 **&&** a < 5);
			Результат: *x= false*

ИЛИ (**||**) (бинарная)	a **||** b
	или
	a **or** b
		Возвращает *true*,
		если *a=true* **или** *b=true*,
		иначе возвращает *false*
			a = 7;
			x = (a > 0 **||** a < 5);
			Результат: *x=true*

Допускается использование различных логических операций в одном выражении:

```
a = 7;
x = !(a >= 0 && a < 10 || a != 7);
```

Результат: *x= false*.

3.5. Поразрядные логические операции

Поразрядные логические операции работают с двоичным представлением целых чисел.

Определены следующие операции:

«~» – поразрядное отрицание (унарная операция). Операция инвертирует каждый бит операнда (0 заменяется на 1, а 1 на 0);

«<<» – поразрядный сдвиг влево. Операция сдвигает биты левого операнда на число разрядов, указанное правым операндом. Сдвиг на *n* разрядов влево аналогичен умножению числа на 2^n;

«>>» – поразрядный сдвиг вправо. Операция сдвигает биты левого операнда на число разрядов, указанное правым операндом. Сдвиг на *n* разрядов вправо аналогичен делению числа на 2^n;

«&» – поразрядное И;

«|» – поразрядное ИЛИ;

«^» – поразрядное исключающее ИЛИ.

Таблица истинности для операций «&», «|», «^» представлена ниже (табл. 3.2).

Таблица 3.2

Значение битов	b_1 & b_2	b_1 | b_2	b_1 ^ b_2
$b_1 = 0, b_2 = 0$	0	0	0
$b_1 = 0, b_2 = 1$	0	1	1
$b_1 = 1, b_2 = 0$	0	1	1
$b_1 = 1, b_2 = 1$	1	1	0

3.6. Приоритет операций в *C++*

Приоритет операций – порядок выполнения операций в арифметическом выражении. Операторы с более высоким приоритетом будут выполняться раньше операторов с более низким приоритетом. Если операторы имеют одинаковый уровень приоритета, то порядок их выполнения определяется ассоциативностью (слева направо или справа налево).

Приоритет операций в языке *C++* представлен в табл. 3.3. Приоритет уменьшается сверху вниз. Ассоциативность указана с помощью стрелок: → (слева направо) и ← (справа налево).

Таблица 3.3

Уровень приоритета и ассоциатив-ность	Тип операции	Операторы
1	Разрешение области действия	::
2 (←)	Префиксные инкремент и декремент	++, --
3 (→)	Выбор элемента для указателя (объект или указатель)	. (точка), -> (стрелка)
	Индекс массива	[ ]
	Вызов функции, инициализация, скобки	( )
	Имя типа объекта	typeid
	Явное приведение типа	const_cast, dynamic_cast, reinterpret_cast, static_cast
4 (←)	Размер объекта или типа	sizeof
	Поразрядное отрицание	~ (compl)
	Логическое отрицание	! (not)
	Унарные плюс и минус	+, –
	Взятие адреса и разадресация	&, *
	Создание и уничтожение объекта	new, delete
	Явное приведение типа	()
5 (→)	Указатель на элемент	.*, ->*
6 (→)	Арифметические операции	*, /, %
7 (→)	Арифметические операции	+, –
8 (→)	Сдвиг	<<, >>
9 (→)	Операции сравнения	<, >, >=, <=
10 (→)	Операции сравнения	==, != (not_eq)
11 (→)	Побитовое И	& (bitand)
12 (→)	Побитовое исключающее ИЛИ	^ (xor)
13 (→)	Побитовое ИЛИ	| (bitor)
14 (→)	Логическое И	&& (and)
15 (→)	Логическое ИЛИ	|| (or)
16 (→)	Условная операция	? :
17 (→)	Постфиксные инкремент и декремент	++, – –
18 (←)	Присваивание	=, *=, /=, %=, +=, –=, <<=, >>=, &= (and_eq), |= (or_eq), ^= (xor_eq), throw
19 (→)	Последовательность	, (запятая)

3.7. Использование блоков

Группа операторов, заключенная в фигурные скобки, называется ***блоком***. Компилятор рассматривает такую группу операторов как один составной оператор. В любой конструкции языка *C++* простой оператор можно заменить блоком. Например, вместо

оператор;

можно поставить

```
{
оператор1;
…
операторn;
 }
```

4. Организация разветвляющихся алгоритмов

Разветвляющийся алгоритм – алгоритм, содержащий несколько ветвей (последовательностей команд), отличающихся друг от друга содержанием вычислений. Выход вычислительного процесса на ту или иную ветвь алгоритма определяется текущими данными.

4.1. Оператор условного ветвления (if-else)

Формат оператора выбора:

if (*логическое_выражение***)** *оператор1*;
else *оператор2*;

Если *логическое_выражение* истинно, то выполняется *оператор1*, иначе – *оператор2*.

Например:

```
if (f > 10) x = 3;
    else x = 42;
```

Истинным *логическое_выражение* считается, если оно равно:

– true;

– ненулевому арифметическому значению;

– значению указателя, отличному от nullptr;

– ненулевому значению типа класса, определяющего однозначное преобразование к арифметическому, логическому типу или типу указателя.

Оператор имеет сокращенную форму:

if (*логическое_выражение***)** *оператор1***;**

Например:

```
if (f == 0) x = 3;
```

Логическое_выражение всегда располагается в круглых скобках. Если *оператор1* или *оператор2* содержит более одного оператора, то используется блок.

В качестве *оператора1* и *оператора2* могут быть использованы операторы if. Такие операторы называют *вложенными*. Во вложенных операторах if ключевое слово else принадлежит ближайшему предшествующему ему if.

Например:

```
if (логическое_выражение1) оператор1;
if (логическое_выражение2) оператор2;
   else  оператор3;
```

Оператор3 будет выполняться, если *логическое_выражение2* ложно. Значение *логического_выражения1* не оказывает влияния на работу *оператора3.*

Изменить порядок проверки можно, используя фигурные скобки:

```
if (логическое_выражение1) {
        оператор1;
if (логическое_выражение2) оператор2;
                                }
   else оператор3;
```

Оператор3 будет выполняться, если *логическое_выражение1* ложно. Значение *логического_выражения2* не оказывает влияния на выбор *оператора3*.

4.2. Условный оператор

Формат условного тренарного оператора:

```
условие ? операнд1 : операнд2;
```

Если значение *условия* истинно, то результатом выполнения оператора является *операнд1*, иначе – *операнд2*.

Например, найти наибольшее из двух чисел:

```
max = a > b ? a : b;
```

Условие может быть любым скалярным выражением. Операнды должны быть одного типа, или их разрешено привести к одному типу неявным преобразованием.

Применение условного оператора сокращает код, однако не увеличивает скорость выполнения программы.

4.3. Оператор множественного выбора

При наличии в программе большого числа ветвлений, зависящих от значения одного операнда, используется оператор множественного выбора switch.

Формат оператора:

```
switch(переменная_выбора) {
        case константа1: оператор1 ;   break;
                                …
        case константаN: операторыn;   break;
        default : операторыn+1;
                                }
```

Переменная_выбора должна быть целочисленного типа (или типа неявно преобразуемого в целочисленный тип). Тип *констант* должен соответствовать типу *переменной_выбора*, а значение должно быть уникальным (наличие констант с одинаковым значением не допускается).

При совпадении значения *переменной_выбора* со значением одной из констант управление передается первому оператору, стоящему после этой константы. Если совпадений не было, то управление передается первому оператору блока default (при наличии) или первому оператору за конструкцией switsh.

Пример 4.1. Написать программу, реализующую простейший калькулятор.

```
int a, b, res;   char znak;
cout << " Введите a :";            cin >> a;
cout << " Введите b :";           cin >> b;
cout << " Введите знак операции :";cin >> znak;
switch(znak)      {
      case '+': res = a + b; break;
      case '-': res = a - b; break;
      case '*': res = a * b; break;
      case '/': res = a / b; break;
            default: cout << "Ошибка!"; exit(1);
}
cout << res;
```

В конце каждого набора операторов ставится оператор break, который завершает выполнение оператора switch. Если не поставить break, то после выполнения соответствующей секции управление будет передано операторам, относящимся к другим ветвям switch. Отсутствие break, как правило, приводит к ошибке в вычислениях. Однако в некоторых случаях использование секций case без break оправдано, например, если необходимо для различных значений констант сравнения выполнять одинаковую последовательность операторов.

Пример 4.2. Определить пору года по номеру месяца.

```
int mes;
cout << "Введите номер месяца : ";          cin >> mes;
      switch (mes) {
      case 12:   case 1:   case 2:   cout << "Зима"; break;
      case 3:    case 4:   case 5:   cout << "Весна"; break;
      case 6:    case 7:   case 8:   cout << "Лето"; break;
      case 9:    case 10:  case 11: cout << "Осень"; break;
            default: cout << "Ошибка!";
      }
```

	При выводе текста в окно консоли происходит преобразование кириллических букв в стандарт ср866, который отображает эти буквы неправильно. Для корректного вывода русских букв (в потоке вывода) можно использовать оператор:
	system("chcp 1251");
	или
	setlocale(LC_ALL, "Russian");

Пример 4.3. Вычислить значение выражения

$$s = \begin{cases} 2 \cdot y + f(x), & \text{если} \quad x \cdot y > 10, \\ \sin(f(x)), & \text{если} \quad x \cdot y \le 3, \\ \sqrt[3]{f(x)+3} & \text{иначе.} \end{cases}$$

При выполнении задания предусмотреть выбор вида функции $f(x)$: $\ln(x)$ или $\cos(x)$.

```
double x, y, f, p, res;
   int k;
   cout << "Введите x ";  cin >> x;
   cout << "Введите y ";  cin >> y;
   cout << "Введите 1 (если f(x) = ln(x)) или 2 (если f(x) = cos(x)) :" ;
   cin >> k;
   switch (k) {
      case 1: f = log(x); break;
      case 2: f = cos(x); break;
      default: cout << "Функция не выбрана";  return 1;
   }
   p = x * y;
   if (p > 10) res =2*y+ f;
      else
            if (p <= 3) res =sin(f);
                  else res = cbrt(f + 3);
   cout << "Результат = " << res << endl;
```

5. Организация циклических алгоритмов

Циклический алгоритм – многократное выполнение одной и той же последовательности операторов при различных значениях исходных данных.

5.1. Оператор цикла for

Общий вид оператора:

```
for (инициализирующее_выражение; логическое_выражение;
                                  инкрементирующее_выражение)
        {
  // Тело цикла
        }
```

Обычно все три выражения содержат одну переменную, которую называют ***счетчиком цикла***.

Инициализирующее выражение выполняется один раз в начале выполнения циклического алгоритма. Как правило, используется для инициализации счетчика цикла. Может содержать объявления и операторы.

Логическое выражение проверяется перед выполнением тела цикла. Если результат имеет отличное от нуля целочисленное значение (*true*), то тело цикла выполняется, иначе выполняется следующий за телом цикла оператор. Если логическое выражение отсутствует, то считается, что оно имеет значение *true*.

Инкрементирующее выражение предназначено для изменения значения счетчика цикла. Модификация счетчика происходит после каждого выполнения тела цикла.

Тело цикла – последовательность операторов, которая выполняется до тех пор, пока не будет выполнено условие выхода из цикла. Тело цикла может содержать внутри себя любые конструкции языка *C++*, в том числе любое количество вложенных циклов.

Схема работы цикла for представлена на рис. 5.1.

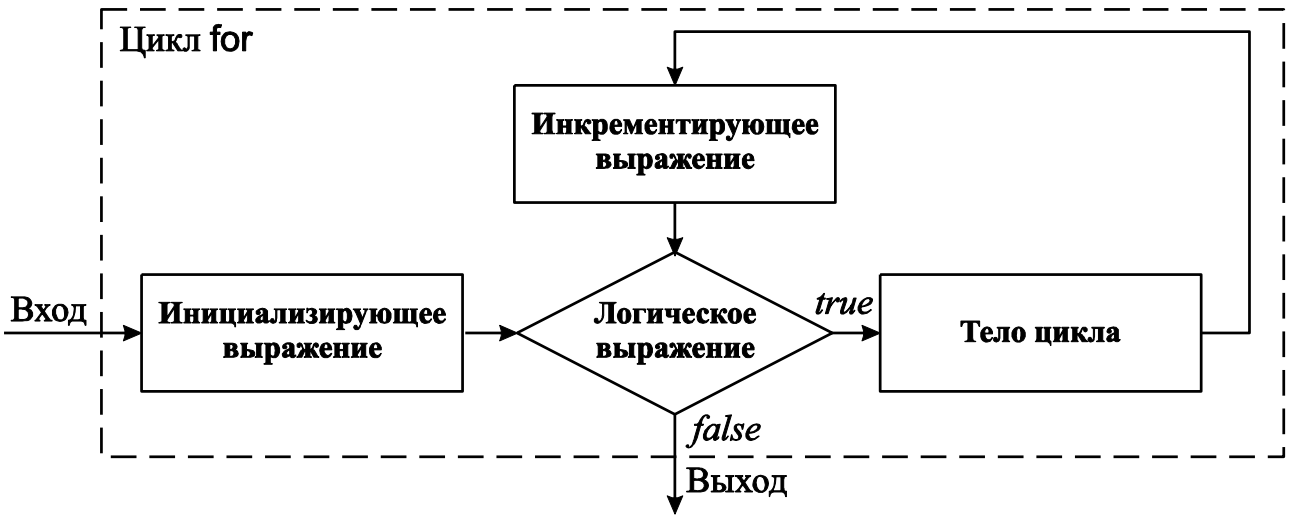

Рис. 5.1

При выполнении оператора

```
for (i = 1;   i < 10;  i++) cout << i << endl;
```

сначала (*в инициализирующем выражении*) в переменную i будет занесено число 1. После этого будет проверено значение логического выражения, и так как оно имеет значение *true* (1 < 10), то будет выполнено тело цикла – текущее значение i (1) будет выведено на экран. Затем выполняется инкрементирующее выражение (i++) и снова проверяется значение логического выражения.

Тело цикла будет выполняться до тех пор, пока значение логического выражения не примет значения *false* (10 < 10).

В результате работы циклического алгоритма на экран будут выведены числа от 1 до 9.

Если необходимо вывести числа от 1 до 10, то можно использовать конструкцию

```
for (i = 1;   i <= 10;  i++) cout << i << endl;
```

однако в *C++* чаще используют конструкции со строгим неравенством:

```
for (i = 1;   i < 11;  i++) cout << i << endl;
```

Разрешено совмещать выполнение инкрементирующего выражения с описанием счетчика цикла:

```
for (int i = 1;   i < 11;  i++)
```

Такое объявление удобно тем, что переменная i согласно стандарту языка *C++* будет существовать только внутри цикла. После выполнения цикла память, выделенная для хранения i, освобождается.

Любая из секций в операторе for не является обязательной, поэтому может отсутствовать любое количество секций. Допустимо такое написание бесконечного цикла:

```
for ( ; ; )
```

Для размещения в одной секции оператора for нескольких операторов используется операция «запятая», которая позволяет в тех местах, где допустимо использование только одного оператора, размещать несколько операторов. Формат операции:

Оператор1, Оператор2, …, ОператорN

Программа для вычисления факториала числа *n* может выглядеть следующим образом:

```
for (f=1, i=1;  i<=n;  f*=i, i++);
```

Точка с запятой в конце оператора for означает, что тело цикла отсутствует.

Имеется форма оператора for, основанная на диапазоне (*range-based for*), которая позволяет обращаться последовательно к каждому элементу коллекции (множеству упорядоченных элементов).

```
for (элемент : имя_коллекции)
{
    // Тело цикла
}
```

5.2. Оператор цикла while

Оператор цикла с предусловием

```
while (логическое_выражение)
{
    // Тело цикла
}
```

выполняет операторы тела цикла до тех пор, пока значение логического выражения истинно. Если значение логического выражения становится равным 0 (*false*), циклический процесс прекращается и выполняется первый после цикла оператор. Если условие сразу равно 0 (*false*), то тело цикла не выполняется ни разу.

5.3. Оператор цикла do-while

Оператор цикла с постусловием

```
do {
    // Тело цикла
} while (логическое_выражение);
```

организует выполнение операторов тела цикла, пока значение логического выражения истинно. Если значение логического выражения становится равным 0 (*false*), циклический процесс прекращается и выполняется первый после тела цикла оператор.

	Оператор цикла do-while опасен тем, что тело цикла выполняется хотя бы один раз (вне зависимости от значения условия). Поэтому, если это возможно, следует избегать использования этого оператора.

5.4. Операторы и функции передачи управления

Операторы и функции передачи управления позволяют изменить стандартный порядок выполнения операторов.

5.4.1. Оператор **continue**

Позволяет в циклическом алгоритме пропускать операторы тела цикла, находящиеся после оператора continue, и передать управление следующему циклу. Оператор continue обычно используется вместе с оператором if.

После выполнения оператора continue выполняется логическое выражение оператора цикла.

	С осторожностью использовать continue в операторах do-while и while, так как инкрементирующее выражение может попасть в пропускаемую часть оператора (что приведет к бесконечному циклу).

5.4.2. Оператор **break**

Позволяет перейти к следующему за блоком оператору. Например, в циклах он обеспечивает досрочный выход из цикла, а в операторе switch – выход из блока выбора.

	Оператор break позволяет выйти только из текущего блока, т. е. в случае использования вложенных циклов выход происходит только из одного циклического алгоритма.

5.4.3. Оператор **return**

Завершает выполнение функции и передает управление в точку ее вызова (или в ОС, если это функция main()). Вызывающая функция помещает результат вычисления выражения (если функция его возвращает) в точку вызова и возобновляет свою работу.

Формат оператора:

return *выражение*;

В функции можно использовать любое количество операторов return.

5.4.4. Функция **exit**

Находится в библиотеке *stdlib.lib*. Корректно прерывает выполнение программы, записывая все буферы, закрывая все потоки. Формат функции:

void exit(**int** *возвращаемое_значение*)

Параметр является служебным сообщением системе. Как правило, 0 говорит об успешном завершении программы, ненулевые значения – об ошибке.

Функция

void quick_exit(**int** *возвращаемое_значение*)

быстро завершает программу без полного освобождения ресурсов.

5.4.5. Функция ***abort***

Находится в библиотеке *stdlib.lib*. Генерирует «молчаливое» исключение и прерывает выполнение программы. Функция abort не закрывает открытые и временные файлы, не очищает буферы потоков. Формат функции:

```
void abort()
```

5.4.6. Оператор безусловного перехода ***goto***

Передает управление оператору, отмеченному меткой.

	Использование оператора goto существенно снижает читабельность программы и увеличивает вероятность ошибки. Поэтому использование goto в программах **нежелательно**.

Примером обоснованного применения оператора безусловного перехода может служить необходимость организации выхода сразу из нескольких вложенных циклов, например:

```
for ( i = 0; i < n; i++)
  for (j = 0; j < m; j++)   {
            if (логическое_выражение) goto met;
                              }
met:   …
```

5.5. Организация циклических алгоритмов

Пример 5.1. Вывести таблицу значений функции $y(x) = \sin(x)$ на интервале от a до b с шагом h.

Вариант 1 (с использованием оператора цикла for):

```
for (double x=a; x<b+h/2; x+=h)
cout << "x = " << x << " y = " << sin(x) << endl;
```

Так как значение h является действительным числом, то при суммировании могут накапливаться ошибки округления. Например, если значение b равно 3.0, а h после выполнения некоторого количества итераций приняло значение 3.00000000000001, то логическое выражение x <= b будет равно *false*, и, следовательно, последнее значение таблицы не будет выведено на экран. Поэтому для гарантированного выполнения последней итерации значение правой границы интервала увеличивается на некоторую величину, не превышающую h (например, на $h/2$).

Вариант 2 (с использованием оператора цикла while):

```
x = a;
while(x<b+h/2) {
   cout << "x = " << x << " y = " << sin(x) << endl;
```

```
  x += h;
}
```

Пример 5.2. Вычислить интеграл $s = \int_a^b \sin x \, dx$ методом средних.

```
h = (b - a) / 100;
  s = 0;  // Начальное значение для расчета площади
x = a + h / 2; // Центр прямоугольника
for (; x < b; x += h) s += sin(x) * h;
```

Пример 5.3. Вычислить сумму $s(x) = \sum_{k=1}^{100} (-1)^k \frac{x^k}{k!}$.

Для расчета такой последовательности удобно использовать рекуррентную формулу. ***Рекуррентная формула*** – формула, которая выражает каждый член последовательности через *n* предыдущих.

Для получения формулы вычисляются значения слагаемых при различных значениях *k*:

при $k = 1;\ a_1 = -1\frac{x}{1};$

при $k = 2;\ a_2 = 1\frac{x \cdot x}{1 \cdot 2};$

при $k = 3;\ a_3 = -1\frac{x \cdot x \cdot x}{1 \cdot 2 \cdot 3}$ и т. д.

Видно, что на каждом шаге слагаемое домножается на $-1\frac{x}{k}$. Исходя из этого формула рекуррентной последовательности будет иметь вид $a_k = -a_{k-1}\frac{x}{k}$.

Полученная формула позволяет избавиться от многократного вычисления факториала и возведения в степень.

```
s =0;        // Начальное значение суммы
a=1;         // Начальное значение для вычисления очередного
             //  члена рекуррентной последовательности
  for ( int k=1; k<=100; k++)
{
  a*=-x/k;   // Вычисление очередного члена
             // рекуррентной последовательности
  s+=a;      // Суммирование слагаемых
}
```

Пример 5.4. Вычислить сумму $s(x)=\sum_{k=0}^{100}(-1)^k\frac{x^{2k}}{(2k)!}\sin(x)$.

В данной формуле получить рекуррентную зависимость для sin(x) сложно, поэтому функция sin(x) будет рассчитываться отдельно (считается нерекуррентной частью). Для оставшейся части формулы $\sum_{k=0}^{100}(-1)^k\frac{x^{2k}}{(2k)!}$ рассчитываются значения слагаемых при различных значениях k:

при $k=0;\ a_1=1\frac{1}{1};$

при $k=1;\ a_1=-1\frac{\boldsymbol{x^2}}{1\cdot\boldsymbol{2}};$

при $k=2;\ a_2=1\frac{x^2\cdot\boldsymbol{x^2}}{1\cdot2\cdot\boldsymbol{3}\cdot\boldsymbol{4}};$

при $k=3;\ a_3=-1\frac{x^2\cdot x^2\cdot\boldsymbol{x^2}}{1\cdot2\cdot3\cdot4\cdot\boldsymbol{5}\cdot\boldsymbol{6}}$ и т. д.

Формула рекуррентной последовательности будет иметь следующий вид: $a_k=-a_{k-1}\frac{x^2}{(2k-1)\cdot(2k)}$. Расчет удобно начинать не с нулевого элемента, а с первого. Поэтому значение нулевого элемента рассчитывается вручную и подставляется в начальное значение суммы.

```
        s = sin(x);          // Значение суммы для нулевого элемента
        a = 1;
for (int k=1; k<=100; k++)
 {
   a *= -sqr(x)/(2*k*(2*k-1));
   s += a*sin(x);            // Учитывается нерекуррентная часть
 }
```

6. Использование массивов

Массив – структура однотипных данных, каждый элемент которой хранится в отдельной ячейке, доступ к которой осуществляется по ее номеру. Массив характеризуется: имением массива, типом хранимых данных, размером (количеством элементов) и размерностью (формой представления элементов массива). Номер ячейки массива называется ***индексом***. Индексы массивов должны иметь целый тип, а элементы массивов могут иметь любой тип.

6.1. Одномерные массивы

Объявление одномерного массива:

тип имя_массива [**размер**];

Пример объявления массива:

```
int c[4];
```

Размер массива задается константой или константным выражением целого типа (размер не может быть изменен во время выполнения программы).

Индексы массивов в языке *C++* начинаются с 0. Например, вышеобъявленный массив состоит из четырех элементов: c[0], c[1], c[2] и c[3]. Расположение элементов массива в памяти указано на рис. 6.1.

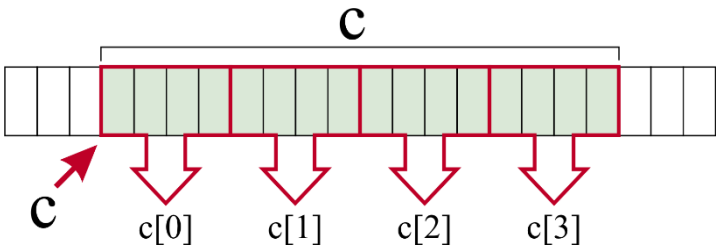

Рис. 6.1

Одновременно с объявлением можно инициализировать (задавать начальные значения) элементы массива:

```
double mas1[5] = {1.5, 3.3, 4.5, 2.7, 3.0};
int mas2[4] = {2, 5};
```

Если в группе инициализации не хватает начальных значений, то оставшиеся элементы заполняются нулями, например массив mas2: mas2[0] = 2, mas2[1] = 5, mas2[2] = 0 и mas2[3] = 0.

При объявлении со списком инициализации количество элементов можно не указывать. В этом случае размер массива будет равен количеству начальных значений. Объявление

```
char mc[ ] = {'e', 'k', 'q'}
```

создаст массив из трех элементов.

Обращение к элементу массива происходит через указание имени массива и в квадратных скобках номера элемента массива. Например:

```
x = a[3];    a[4] = b[0] + a[2];
```

Правила работы с элементами массива соответствуют правилам работы с переменными соответствующего типа.

6.2. Алгоритмы работы с одномерными массивами

Пример 6.1. Ввести с клавиатуры и вывести на экран одномерный массив.

```
int mas[10], n;
// Ввод одномерного массива
cout << "Введите размер массива : ";
cin >> n;
for (int i = 0; i < n; i++)
{
        cout << "Введите элемент [" << i << "]= ";
        cin >> mas[i];
}
// Вывод одномерного массива
for (int i = 0; i < n; i++)

        cout << mas[i] << "   ";
```

Пример 6.2. Найти сумму и произведение элементов одномерного массива.

```
   s = 0; p = 1;
 for (i=0; i<n; i++)
{
    s += a[i];   // Расчет суммы элементов
    p *= a[i];   // Расчет произведения элементов
}
```

Пример 6.3. Найти минимальный и максимальный элементы одномерного массива.

Вариант 1:

```
min = max = a[0];
  for (int i = 1; i < n; i++)
        if (mas[i] < min) min = a[i];
                    else
                          if (mas[i] > max) max = a[i];
```

Вариант 2:

```
min = max = a[0];
for (int x : mas)
{
        min = fmin(min, x);
        max = fmax(max, x);
}
```

Пример 6.4. Удалить из одномерного массива все отрицательные элементы.

```
for (int i=0; i<n; i++)
  if (mas[i] < 0)
  {
    for (j=i+1; j<n; j++) mas[j-1] = mas[j];
 n--;    i--;
  }
```

Пример 6.5. Отсортировать массив по неубыванию значений элементов.

```
for (int i=0; i<n-1; i++)
  for (int j=i+1; j<n; j++)
        if (mas[i] > mas[j])
              {
               tmp = mas[i];
               mas[i] = mas[j];
               mas[j] = tmp;
              }
```

6.3. Многомерные массивы

Объявление одномерного массива:

тип имя_массива [*размер*_1] [*размер*_2] … [*размер_N*];

Пример объявления двумерного массива:

```
int m[4][5];  // Двумерный массив из 4 х 5 = 20 элементов
```

Одновременно с объявлением можно инициализировать элементы массива:

```
int s[2][3] = { {3, 4, 2}, {6, 3, 4} };
```

В одномерном массиве первый индекс является номером строки, а второй – номером столбца. Поэтому, например, значение элемента s[1][0] равно 6. Математически массив s представляет собой матрицу вида

3 4 2
6 8 5

В памяти компьютера такой массив располагается последовательно по строкам (рис. 6.2).

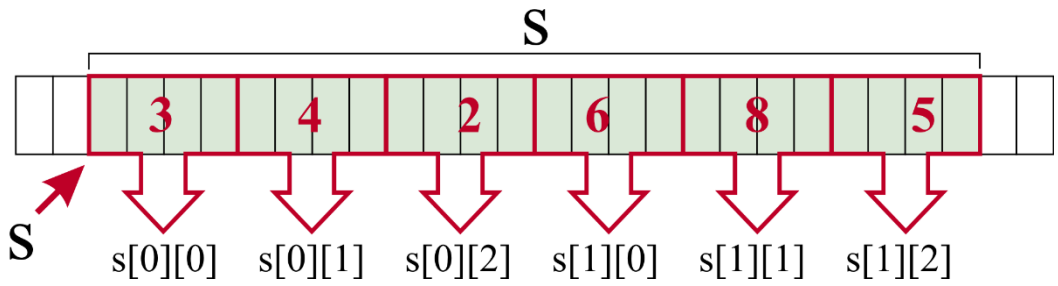

Рис. 6.2

Обращение к элементу двумерного массива происходит через указание имени массива и в квадратных скобках номера строки и номера столбца. Например:

```
x = s[0][2];
s[1][2] = m[3][2] + s[0][1];
```

6.4. Алгоритмы работы с двумерными массивами

Пример 6.6. Ввести с клавиатуры и вывести на экран двумерный массив целых чисел.

```
int n, m;
double mas[10][10];
// Ввод
cout << "Введите число строк и столбцов:" << endl;
cin >> n >> m;
for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
        {
        cout << "Введите элемент [" << i << "] [" << j << "]: ";
        cin >> mas[i][j];
        }
// Вывод
for (int i = 0; i < n; i++)
{
        for (int j = 0; j < m; j++)
                cout << setw(8) << mas[i][j] << "  ";
  cout << endl;
}
```

Для выравнивания столбцов используется модификатор setw (библиотека *iomanip*), который устанавливает ширину поля вывода.

Пример 6.7. Вывести на экран двумерный массив действительных чисел.

```
// Вывод
for (int i = 0; i < n; i++)
{
        for (int j = 0; j < m; j++)
        cout << fixed << setw(10) << setprecision(3) << mas[i][j] << " ";
        cout << endl;
}
```

Используется вывод действительного числа с фиксированной точкой (модификатор fixed) с тремя знаками после запятой (модификатор setprecision(3)).

Пример 6.8. Заполнить двумерный массив случайными действительными числами из диапазона от 30 до 70.

```
mt19937 mt(time(nullptr));  // Генератор случайных чисел
// Задание диапазона генерируемых чисел
uniform_real_distribution<> rnd(30, 70);
 for (int i = 0; i < n; i++)
        for (int j = 0; j < m; j++)
                mas[i][j] = rnd(mt);
```

Используется генератор случайных чисел mt19937 (библиотека *random*).

Пример 6.9. Найти сумму элементов, лежащих на побочной диагонали.

```
s = 0;
for (int i=0; i<n; i++)
        s += mas[i][n-i-1];
```

Пример 6.10. Найти координаты элементов, содержащих максимальное и минимальное значения.

```
imin = jmin = imax = jmax = 0;
for (int i = 0; i < n; i++)
   for (int j = 0; j < m; j++) {
      if (mas[i][j] < mas[imin][jmin])
      {
         imin = i;   jmin = j;
      }
     else
```

```
        if (mas[i][j] > mas[imax][jmax])
        {
            imax = i;  jmax = j;
        }
    }
```

Пример 6.11. Найти сумму элементов, лежащих выше главной диагонали.

```
s = 0;
        for (int i=0; i<n-1; i++)
          for (int j=i+1;  j<m;  j++)
                        s += mas[i][j];
```

Пример 6.12. Упорядочить строки матрицы по неубыванию элементов, содержащих максимальные значения.

```
for (int i = 0; i < n; i++)
{
        b[i] = mas[i][0];
        for (int j = 1; j < m; j++)
                if (mas[i][j] > b[i]) b[i] = mas[i][j];
}

for (int i = 0; i < n - 1; i++)
        for (int j = i + 1; j < m; j++)
                if (b[i] > b[j])
                {
                        tmp = b[i];
                        b[i] = b[j];
                        b[j] = tmp;
                        for (int k = 0; k < m; k++)
                        {
                                tmp = mas[i][k];
                                mas[i][k] = mas[j][k];
                                mas[j][k] = tmp;
                        }
                }
```

7. Использование указателей

7.1. Объявление указателя (необработанного указателя)

Память компьютера представляет собой массив последовательно пронумерованных ячеек. При объявлении данных в памяти выделяется непрерывная область для их хранения. Адрес первого байта памяти, выделенной под переменную, называется ***адресом*** этой переменной.

Указатель – это переменная, предназначенная для хранения адреса участка памяти. Для хранения указателя выделяется участок памяти размером **8 байт** (в 32-разрядных системах – 4 байта).

Указатели используются:

- для динамического выделения памяти;
- передачи параметров в функциях;
- обращения к элементам структур данных.

Формат объявления указателя:

*Тип_переменной *имя_указателя*;

Например:

```
int *a;
double *b, *d;
char *c;
```

На один и тот же участок памяти может ссылаться любое количество указателей (в том числе различных типов). Допустимо описывать переменные типа «указатель на указатель» (указатель на ячейку памяти, которая в свою очередь содержит адрес другой ячейки памяти). Например:

```
int *um1, **um2, ***um3;
```

В языке *C* определены три вида указателей:

1. Указатель на объект известного типа.

2. Указатель типа void. Применяется в случаях, когда тип объекта заранее не определен.

3. Указатель на функцию. Позволяет обращаться с функциями, как с переменными.

7.2. Операции над указателями

7.2.1. Унарные операции

Определены две унарные операции:

1. «**&**» («**взять адрес**»). Операция позволяет получить адрес переменной.

2. «*****» («**разадресация**»). Позволяет получить доступ к величине, расположенной по указанному адресу.

7.2.2. Арифметические операции и операции сравнения

При выполнении арифметических операций с указателями автоматически учитывается размер типа данных указателя.

Инкремент и декремент. Перемещает указатель к следующему или предыдущему элементу массива.

Например:

```
int* um, mas[5] = { 1,2,3,4,5 };
um = mas;
        cout << *um << endl;   // Выводит: 1
um++; um++;
        cout << *um << endl;   // Выводит: 3
um--;
        cout << *um << endl;   // Выводит: 2
```

Добавление или вычитание. Перемещение указателя на число байт, равное произведению размера типа данного, на которое ссылается указатель, на величину добавляемой или вычитаемой константы. Например:

```
int* um, mas[5] = { 1,2,3,4,5 };
um = mas;
        cout << *um << endl;   // Выводит: 1
um += 3;
        cout << *um << endl;   // Выводит: 4
um -= 2;
        cout << *um << endl;   // Выводит: 2
```

Разность указателей. Разность двух указателей равна числу объектов соответствующего типа, размещенных в данном диапазоне адресов. Например:

```
int mas[5];
int* um = &mas[0];
int* un = &mas[4];
        int k = un - um;
                cout << k << endl;   //  Выводит: 4
```

Операции сравнения. Сравнивают адреса объектов. Результат определяется исходя из взаимного расположения объектов в адресном пространстве программы.

	Операции сравнения для указателей имеют смысл при определении принадлежности указателей к одному объекту (== и !=) или для работы с последовательно расположенными элементами (например, с массивом).

7.3. Инициализация указателей

Инициализация пустым значением. Например:

```
// Стиль C
        int* a = NULL;
        int* b = 0;
// Стиль C++ (начиная с v.11)
        int* c = nullptr;
```

Присваивание указателю адреса уже существующего объекта. Например:

```
int k = 23;
        int* uk = &k;   // или   int *uk(&k);
        int* us = uk;
```

Присваивание указателю адреса выделенного участка динамической памяти:

```
int * s = new int;
        int* k = (int*)malloc(sizeof(int));
```

	Операция sizeof() определяет размер указанного параметра в байтах.

7.4. Работа с динамической памятью

Динамическая память (heap) – специальная область памяти, позволяющая во время выполнения программы выделять и освобождать место в соответствии с текущими потребностями. Доступ к выделенным участкам памяти осуществляется через указатели. Для работы с динамической памятью в языке *C* (библиотека *malloc.lib*) определены следующие функции:

void ***malloc**(size_t size) – выделяет область памяти размером size байт. Возвращает указатель void* на выделенный блок памяти. Для получения указателя заданного типа используется явное приведение типов. Если для выделения заданного блока памяти недостаточно свободного места, то функция возвращает NULL.

void ***calloc**(size_t n, size_t size) – выделяет область памяти размером n блоков по size байт. Возвращает адрес выделенного блока памяти. Если недостаточно свободного места для выделения заданного блока памяти, то возвращает NULL. Вся выделенная память заполняется нулями.

void ***realloc**(void *u, size_t size) – изменяет размер ранее выделенной памяти, связанной с указателем u, на новое число (size) байт. Если память под указатель не выделялась, то функция ведет себя как malloc. Если недостаточно свободного места для выделения заданного блока памяти, то функция возвращает значение NULL.

void **free**(*u) – освобождает участок памяти, связанный с указателем u.

	size_t – беззнаковый целочисленный тип данных, позволяющий хранить максимальный размер любого теоретически возможного объекта.

В языке *C++* для выделения и освобождения памяти определены операторы new и delete.

Имеются две формы операторов:

тип ***указатель** = **new** *тип* (*инициализатор*) – выделяет область памяти в соответствии с указанным типом и заносит туда значение инициализатора (не обязательно).

delete *указатель* – освобождение выделенной памяти.

тип ***указатель** = **new** *тип*[n] – выделение участка памяти размером n блоков указанного типа.

delete []*указатель* – освобождение выделенной памяти.

Оператор delete не уничтожает значения, связанные с указателем, а разрешает компилятору использовать данный участок памяти.

	Каждому оператору, выделяющему динамическую память, соответствует свой оператор освобождения памяти.

7.5. Создание одномерного динамического массива

Для создания одномерного динамического массива необходимо знать тип элементов массива и их количество. Например, для создания одномерного динамического массива, состоящего из *n* действительных чисел, можно использовать следующие функции:

```
umas1 = static_cast <double*> (malloc(n*sizeof(double)));
```

(освобождение памяти – free(umas1))

или

```
umas1 = static_cast <double*> (calloc (n,sizeof(double)));
```

(освобождение памяти – free(umas1))

или

```
umas1 = new double[n];
```

(освобождение памяти – **delete** []umas1)

7.6. Создание двумерного динамического массива

При создании многомерного динамического массива значения всех размерностей, кроме первой, должны быть указаны целочисленными константами. Например:

```
double (*mas1)[5] = new double[n][5];
```

В качестве первого параметра может быть использована целочисленная переменная.

Такой подход не очень удобен, так как не позволяет задавать нужное количество столбцов во время выполнения программы.

Для работы с двумерными массивами используется конструкция, являющаяся массивом указателей на одномерные массивы (рис. 7.1).

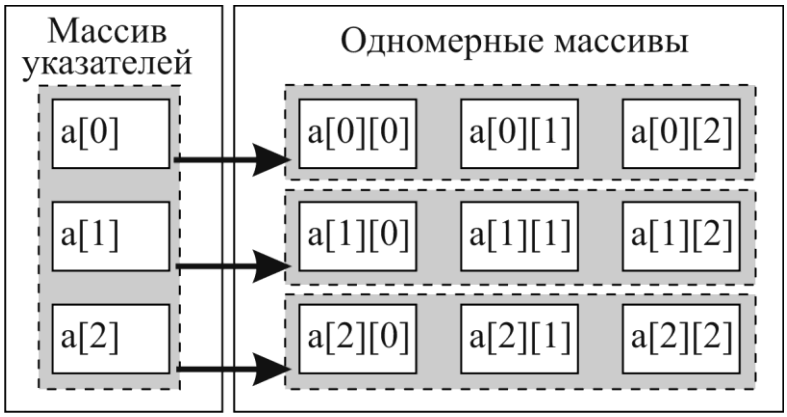

Рис. 7.1

При использовании такого способа выделения памяти имеется возможность обращения к элементам двумерного динамического массива таким же образом, как и к элементам двумерного нединамического массива.

При создании двумерного динамического массива вначале выделяется память под одномерный массив указателей, затем каждый указатель получает адрес созданного одномерного динамического массива (освобождение памяти осуществляется в обратном порядке).

```
double** umas2;                    // Объявление указателя на массив
// Выделение памяти для размещения массива указателей
   umas2 = new double* [n];
// Выделение памяти для размещения одномерных массивов
   for (i = 0; i < n; i++)  umas2[i] = new double[m];

   …          // Работа с массивом

// Освобождение памяти, выделенной для одномерных массивов
      for (i = 0; i < n; i++)  delete[]umas2[i];
// Освобождение памяти, выделенной для массива указателей
      delete[]umas2;
   umas2 = nullptr;        // Очистка указателя
```

8. Использование строковых переменных

В языке *C++* имеется два основных способа работы со строковыми данными: использование *массива символов типа char* (*нуль-терминальные строки*) и использование класса string. В данном учебно-методическом пособии рассматривается только первый способ организации работы со строками.

8.1. Объявление строк

Объявление строки аналогично объявлению массива:

```
char имя строки [размер]
```

В отличие от обычного массива строка должна заканчиваться нулевым символом '\0' – ***нуль-терминатором***. Длина строки равна количеству символов плюс нулевой символ. При вводе данных нулевой символ помещается в конец строки автоматически. Например, в строке

```
char str1[10] = "123456789";
```

символы располагаются следующим образом:

'1'	'2'	'3'	'4'	'5'	'6'	'7'	'8'	'9'	'\0'

При объявлении строки со списком инициализации количество элементов можно не указывать. В этом случае размер строки будет равен количеству начальных значений плюс один (для нулевого символа). Объявление:

```
char str2[ ] = "absd";
```

Доступ к отдельным символам строки осуществляется по их индексам. Например:

```
str2[2] = 'e';
```

	В языке *C++* одиночные кавычки используются для обозначения символов, а двойные – для обозначения строк.

Массив строк объявляется следующим образом:

```
char имя[количество строк][количество символов в строке];
```

Например:

```
char str[10][5].
```

Обращение к третьей строке массива строк:

```
str[2].
```

8.2. Функции для работы со строками

Для ввода/вывода строк и символов используются функции библиотеки *stdio.lib*, для работы со строками – функции библиотеки *string.lib*, для преобразования типов – функции библиотеки *stdlib.lib*, а для распознавания символов – функции библиотеки *ctype.lib*.

В новых стандартах языка многие функции стандартной библиотеки объявлены устаревшими и заменены их версиями с более высоким уровнем безопасности. По умолчанию компилятор запрещает выполнение устаревших функций и предлагает использовать безопасные перегруженные функции.
Если программист уверен в своем коде и хочет использовать устаревшие функции, то перед include можно добавить отключение проверки на безопасность:

```
#define _CRT_SECURE_NO_WARNINGS
```

Для использования автоматической перегрузки небезопасных шаблонов можно использовать

```
#define _CRT_SECURE_CPP_OVERLOAD_STANDARD_NAMES 1
```

Для перегрузки функций, использующих в качестве аргументов числа, необходимо указать

```
#define _CRT_SECURE_CPP_OVERLOAD_STANDARD_NAMES_COUNT 1
```

Наиболее часто применяются следующие функции:

int puts(**const char** *str) – выводит на экран строку str. Переводит указатель на следующую строку.

char ***gets_s**(**char** *str, **int** n) – помещает n - 1 символов, введенных с клавиатуры, в строку str. Возвращает NULL в случае ошибки.

errno_t **strcpy_s**(char * str1, int d, const char *str2) – копирует содержимое строки str2 в строку str1. Параметр d задает размер буфера, который используется для переноса строки. Для получения необходимого размера буфера можно использовать макрос _countof. Функция возвращает нуль в случае ошибки и код ошибки в случае неудачи.

Например:

```
strcpy_s(str, _countof(str), "xyz");
```

Результат: *str* = "*xyz*".

errno_t – целочисленный тип данных, предназначенных для хранения кодов ошибок.

errno_t **strcat_s**(**char** * str1, int d, const char *str2) – добавляет в конец строки str1 содержимое строки str2.

Например:

```
char str[10]="xyz";
      strcat_s(str, _countof(str), "abc");
```

Результат: *str* = "*xyzabc*".

int strcmp(**const char** *str1, **const char** *str2) – сравнивает содержимое строк str1 и str2. Если str1 < str2, то результат равен –1, если str1 = -2 − результат равен нулю, если str1 > str2 – результат равен 1.

```
char st1[40] = "ABCD",   st2[40] = "xyz";
k = strcmp(st1, st2)
```

Результат: $k = -1$.

char ***strchr**(**char** * str, **int** ch) – возвращает указатель на первое появление символа ch в строке str.

Например, требуется определить позицию первого появления символа d в строке.

```
char str[40] = "AbCdFGh";
char* s = strchr(str, 'd');
int k = static_cast <int> (s - str);
```

Результат: $k = 3$.

char *strstr(**char** *str1, **const char** *str2) – возвращает указатель на первое появление строки str2 в строке str1.

Например, требуется определить позицию первого вхождения строки *BC* в строку *ABCD*.

```
char str1[40] = "AbCdFGh";
char str2[40] = "FG";
char* s = strstr(str1, str2);
int k = static_cast <int> (s - str1);
```

Результат: $k = 4$.

char* **strtok_s**(char* str, const char* dlm, char** context) – возвращает указатель на токен, находящийся в строке str (токеном считается набор символов, отделенный от других токенов символом-разделителем, находящимся в строке dlm). Параметр context используется для хранения сведений о непроверенной части строки.

При первом вызове функции strtok_s функция пропускает ведущие разделители и возвращает указатель на первый токен в строке str (следующий после токена символ заменяется нулевым символом). При последующих вызовах функции со значением NULL в качестве первого аргумента указатель аналогичным образом переходит к следующим токенам. После нахождения всех токенов указатель получает значение NULL.

Пример 8.1. Вывести на экран лексемы, разделенные символами пробела, тире и запятой.

```
char str[50] = {"Простота - залог надежности программы."};
char *ctn = NULL;
char sp[] = " -.";
```

```
char* wrd = strtok_s(str, sp, &ctn);
while (wrd != NULL)
{
        puts(wrd);
        wrd = strtok_s(NULL, sp, &ctn);
}
```

Выводит:

Простота
залог
надежности
программы

size_t **strlen**(const char* str) – возвращает длину строки str (нуль-терминатор '\0' не учитывается).

```
char str[40] = "ABCD";
        int k = strlen(str);
```

Результат: *k* = 4.

char ***_strrev**(char *str) – изменяет порядок следования символов в строке str на противоположный.

```
char str[40] = "ABCD";
        _strrev(str);
```

Результат: *str* = "*DCBA*".

char *_**strdup**(const char *str) – возвращает копию строки str. Для выделения памяти под новою строку функция вызывает malloc, следовательно, необходимо использовать free() для очистки памяти в конце работы.

```
char str1[40] = "ABCD";
char* str2;
        str2 = _strdup(str1);
…
        free(str2);
```

Результат: *st2* = "*ABCD*".

errno_t _**strlwr_s**(char *str, size_t n) – преобразует прописные символы строки str в строчные. Параметр n задает размер буфера.

```
char str[40] = "aBcD";
        _strlwr_s(str, strlen(str) + 1);
```

Результат: *st* = "*abcd*".

errno_t _**strupr**_**s**(char *str, size_t n) – преобразует строчные символы строки str в прописные. Параметр n задает размер буфера.

```
char str[40] = "aBcD";
    _strupr_s(str, strlen(str) + 1);
```

Результат: *st* = "*ABCD*".

int **atoi**(const char *str) – преобразует символьное представление целого числа str (до первого символа, не являющегося цифрой) в число целого типа. Пробелы в начале строки пропускаются.

```
char st1[40] = "  354dg4f";
    int k = atoi(st1);
```

Результат: *k* = 354.

double **atof**(const char *str) – преобразует символьное представление действительного числа str (до первого символа не соответствующего числу) в число действительного типа. Пробелы в начале строки пропускаются.

```
char str[40] = "354.55dd3";
double b = atof(str);
```

Результат: *b* = 354.55.

errno_t _**itoa**_**s**(int k, char *str, size_t n, int d) – преобразует n символов десятичного целого числа k в строку str (в заданной d системе счисления (от 2 до 36)).

```
_itoa_s(25, str, _countof(str), 10);
```

Результат: *str* = 25 в десятичной системе счисления.

```
_itoa_s(25, str, _countof(str), 2);
```

Результат: *str* = 11001 в двоичной системе счисления.

errno_t _**gcvt**_**s**(char *str, size_t n, double val, int dgt) – преобразует число действительного типа val в строку str. Размер буфера n рекомендуется устанавливать равным значению константы _CVTBUFSIZE (309+40). Количество десятичных разрядов dgt должно быть не более 18.

```
double a = -254.2965;
char str[_CVTBUFSIZE];
    _gcvt_s(str, _CVTBUFSIZE, a, 7);  // st = "-254.2965"
    _gcvt_s(str, _CVTBUFSIZE, a, 5);  // st = "-254.3"
    _gcvt_s(str, _CVTBUFSIZE, a, 3);  // st = "-254"
    _gcvt_s(str, _CVTBUFSIZE, a, 1);  // st = "-3e+003"
```

Функции распознавания символов (библиотека *cctype.lib*):

int isalnum(***символ***) возвращает ненулевое значение (*true*), если *символ* – буква или цифра;

int isalpha(*символ*) возвращает ненулевое значение (*true*), если *символ* – буква;

int isdigit(*символ*) возвращает ненулевое значение (*true*), если *символ* – цифра;

int ispunct(*символ*) возвращает ненулевое значение (*true*), если *символ* – знак пунктуации;

int islower(*символ*) возвращает ненулевое значение (*true*), если *символ* – буква нижнего регистра;

int isupper(*символ*) возвращает ненулевое значение (*true*), если *символ* – буква верхнего регистра;

int isspace(*символ*) возвращает ненулевое значение (*true*), если *символ* – пробел, знак табуляции, возврат каретки, символ перевода строки, вертикальной табуляции, перевода страницы.

8.3. Алгоритмы работы со строками

	При одновременном использовании форматированного и неформатированного ввода возможен некорректный ввод данных, поэтому необходимо делать очистку буфера с помощью манипулятора ws (например, cin >> n >> ws;) или методом cin.ignore().

Пример 8.2. Проверить, присутствует ли слово *visual* в заданной строке.

```
char str[30];
        puts("Введите строку ");
gets_s(str, 20);
char* ch = strstr(str, "visual");
        if (ch != nullptr) puts("Присутствует");
                        else puts("Не присутствует");
```

Пример 8.3. В строке *str* удалить все символы ‘*w*’.

```
char* ch;
while (ch = strstr(str, "w"))
        while (*ch != '\0')
        {
                ch[0] = ch[1];
                ch++;
        }
```

Пример 8.4. Выделить и вывести на печать все слова произвольной строки. Слова отделяются друг от друга одним или несколькими пробелами.

```
char str[100], sl[100];     int k = 0;
 gets_s(str, 100);
```

```
  strcat_s(str, _countof(str), " ");
    int n = strlen(str);
for (int i = 0; i < n; i++)
   if (str[i] != ' ') sl[k++] = str[i];
    else
      if (k > 0) {
                sl[k] = '\0';
                  puts(sl);
         sl[0] = '\0';
         k = 0;
               }
```

Пример 8.5. Определить, является ли строка палиндромом, т. е. читается ли она слева направо так же, как и справа налево (например, «А роза упала на лапу Азора»).

```
        setlocale(LC_ALL, "ru-RU");
char str[80] = "А Роза упала на лапу Азора";
_strlwr_s(str, strlen(str) + 1);
int i = 0, j = strlen(str) - 1;
bool bl = true;
while (i <= j) {
        while (str[i] == ' ') i++;
        while (str[j] == ' ') j--;
                if (str[i++] != str[j--])
        {
                bl = false;
                break;
        }
}
        if (bl)  cout << "Палиндром" << endl;
                else cout << "Не палиндром" << endl;
```

	Работа некоторых функций зависит от установленного языкового стандарта. Например, для корректной работы с кириллическими символами надо установить `system("chcp 1251");`

9. Типы данных, определяемых пользователем

9.1. Объявление и использование структур

Структура – составной тип данных, в котором под одним именем объединены функции и данные различных типов. Объявление структуры:

```
struct имя_структуры
{
   Список элементов структуры
} ;
```

Элементами структуры могут быть данные, которые называются ***полями***, и функции, которые называются ***методами***.

Правила описания полей и методов аналогичны правилам описания данных и функций.

Пример описания структуры с несколькими полями:

```
struct Tstr
{
      int m1;
      double m2, m3;
};
```

Поля структуры могут иметь любой тип, в том числе «массив» и «структура».

Объявление структурной переменной:

```
Tstr x;
```

Разрешено совмещать описание структуры и объявление переменных соответствующего типа (между закрывающей фигурной скобкой и точкой с запятой):

```
struct Tstr
{
    int m1;
    double m2, m3;
} a, *b;
```

К полям и методам структуры можно обращаться через составное имя. Формат обращения:

имя_структуры.имя_поля_или_метода

или

указатель_на_структуру **->** *имя_поля_или_метода*

Например, обратиться к полям структуры Tstr можно следующим образом (после выделения памяти для структуры, связанной с указателем b):

```
a.m1 = 35;          b->m1 = 35;
```

или

```
(&a)->m1 = 35;   (*b).m1 = 35;
```

Правила работы с полями структуры идентичны правилам работы с переменными соответствующих типов.

Разрешено при объявлении структуры инициализировать поля:

```
struct Tstr
{
    int m1 = 4;
    double m2 = 2.0, m3 = 3.53;
};
```

Инициализировать переменные-структуры можно путем помещения за объявлением списка инициализации:

```
struct Tstr
{
    int m1;
    double m2, m3;
} a = {5, 2.6, 34.2};
```

В качестве полей могут быть использованы другие структуры:

```
struct Tstr
{
    int m1;
    double m2, m3;
      struct
      {
            int mm1;
      } m4;
} s;
```

Обращение к полю mm1 в этом случае будет следующим:

```
s.m4.mm1 = 3;
```

Если имя структуры не указывается, то такое определение называется ***анонимным***.

Разрешено использовать операцию присваивания (только для структур одного типа). Например:

```
Tstr x, y;
  x = y;
```

В этом случае все значения полей структуры y копируются в соответствующие поля структуры x.

Из структур, как правило, формируют массивы:

```
Tstr ms[100];  // Объявление массива структур
…
ms[99].m1 = 56;  // Обращение к полю массива структур
```

Пример 9.1. Имеется список жильцов многоквартирного дома. Каждый элемент списка содержит следующую информацию: фамилия владельца, номер квартиры, количество комнат в квартире. Вывести в алфавитном порядке фамилии владельцев двухкомнатных квартир. Память для хранения списка выделять динамически.

```
struct Tlist
{
    char fio[50];
    int nomer;
    int nrooms;
} *spisok;

int n, i, j;
cout << "Введите число квартир: " << endl;
cin >> n;
spisok = new Tlist[n];
for (i = 0; i < n; i++)
{
    cout << "Введите фамилию: ";
     cin >> spisok[i].fio;
    cout << "Введите номер квартиры: ";
     cin >> spisok[i].nomer;
    cout << "Введите число комнат: ";
      cin >> spisok[i].nrooms;
    cout << endl;
}

Tlist tmp;
for (i = 0; i < n - 1; i++)
    for (j = i + 1; j < n; j++)
        if (spisok[i].nrooms == 2 && spisok[j].nrooms == 2
            && strcmp(spisok[i].fio, spisok[j].fio) == 1)
```

```
            {
               tmp = spisok[i];
               spisok[i] = spisok[j];
               spisok[j] = tmp;
            }

      for (i = 0; i < n; i++)
         if (spisok[i].nrooms == 2)
            cout << spisok[i].fio << ", квартира номер - "
            << spisok[i].nomer << endl;
      delete[]spisok;
```

9.2. Объявление и использование объединений

Объединение (union) – размещение под одним именем некоторой совокупности данных таким образом, чтобы размер выделяемой памяти был достаточен для размещения любого данного. Переменная типа union в любой момент времени может хранить не более одного объекта из списка элементов. Такие структуры используются в случаях, когда отдельные поля существуют в различные моменты времени.

Объявление объединения:

```
union имя_объединения
{
    // Набор полей
};
```

Например,

```
union per {
     int a;
     double b;
     char  c;
  } un;

  un.a = 567;
  cout << un.a << endl; // Значение mun.a равно 567
  un.b = 8.2;
  cout << un.a << endl;  // 1717986918 - Ошибка!
  cout << un.b << endl;  // Значение mun.b равно 8.2
```

9.3. Объявление и использование перечислений

Перечисление (enum) задает множество значений для заданной пользователем переменной.

Объявление перечисления:

```
enum имя {набор_значений};
```

Элементы в перечислении представлены в виде именованных констант и называются ***перечислителями***.

```
enum week { Mon, Tue, Wed, Thu, Fri, Sat, Sun };
```

Каждому перечислителю присваивается номер. По умолчанию первый перечислитель имеет номер 0, второй – 1 и т. д.

Можно устанавливать нумерацию, отличную от заданной, по умолчанию:

```
enum week { Mon=1, Tue, Wed, Thu, Fri, Sat, Sun };
```

Каждый перечислитель должен быть уникальным. Номера могут повторяться.

```
enum week { Mon=3, Tue, Wed, Thu=2, Fri, Sat, Sun };
```

В этом случае перечеслители Mon, Tue, Wed, Thu, Fri, Sat, Sun имеют номера 3, 4, 5, 2, 3, 4 и 5 соответственно.

Перечисления могут неявно преобразовываться в целочисленные типы, обратное преобразование запрещено. Для преобразования значения перечеслителя можно использовать явное приведение типа:

```
enum week { Mon = 1, Tue, Wed, Thu, Fri, Sat, Sun } w;
   w = Sat;
   cout << w;        // Выводит: 6
   // w = 7;         // Ошибка!
   w= static_cast<week>(7);
   int n = w;
   cout << n;        // Выводит: 7
   if (w == Sun) cout << "Day off!";    // Выводит: Day off!
   if (w == 7) cout << "Day off!";      // Выводит: Day off!
```

10. Использование файлов

10.1. Понятие файла

Файл − именованная совокупность данных, расположенных на внешнем носителе. Для получения доступа к данным файл необходимо открыть. После выполнения любой операции над данными указатель сдвигается на одну позицию вперед. В конце работы файл требуется закрыть (доступ к данным, размещенным в файле, будет запрещен). Информация о файле хранится в управляющей структуре, имеющей тип FILE.

Различают два вида файлов: текстовые и двоичные.

Текстовые файлы хранят информацию в виде последовательности символов. Вывод осуществляется аналогично выводу на экран. Текстовые файлы могут быть отредактированы в любом текстовом редакторе.

Бинарные (или двоичные) файлы предназначены для хранения последовательности байтов. Структура такого файла определяется программно.

Файлы, размещаемые на носителях информации, имеют структуру, представленную на рис. 10.1.

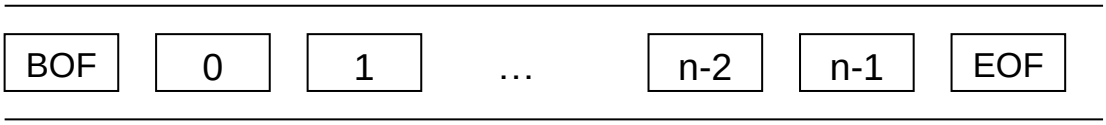

Рис. 10.1

В начале файла записана информация о файле BOF (*Begin of File*), его имя, тип, длина и т. д., в конце файла помещается признак конца файла EOF (*End of File*). Если файл пуст, то BOF и EOF совмещены.

При работе с файлами используются следующие макросы:

– NULL − определяет пустой указатель;

– EOF − значение, возвращаемое при попытке чтения после конца файла;

– FOPEN_MAX − возвращает максимальное число одновременно открытых файлов.

	В последних стандартах языка *C++* рекомендуется вместо макроса NULL использовать ключевое слово nullptr, которое менее уязвимо и в большинстве случаев работает лучше.

10.2. Функции для работы с файлами

Функции для работы с файлами размещены в библиотеках *stdio.lib* и *io.lib*. При работе с файлами используются указатели типа FILE. Формат объявления указателя на файл следующий:

```
FILE *указатель_на_файл;
```

Например:

```
FILE *fl1, *fl2;
```

Указатель содержит адрес структуры, включающей в себя различные сведения о файле, например, его имя, статус, указатель на начало файла.

Функция

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

открывает файл и связывает его с потоком. Возвращает нуль в случае успешного открытия или код ошибки в случае неудачи;

pFile – файловый указатель, который получает указатель на открытый файл;

filename – указатель на строку символов, в которой хранится имя файла и путь к нему. Например: *d:\\work\\lab2.dat*;

mode – указатель на строку символов, в которой указывается режим открытия файла. По умолчанию файл открывается в текстовом режиме. Допустимые режимы приведены в табл. 10.1.

Таблица 10.1

Режим открытия	Действие
r (или rt)	Открывает текстовый файл для чтения. В случае отсутствия файла с указанным именем возникает ошибка
r+ (или rt+)	Открывает текстовый файл для чтения и записи данных
rb	Открывает двоичный файл для чтения. В случае отсутствия файла с указанным именем возникает ошибка
rb+ (или r+b)	Открывает двоичный файл для чтения и записи данных
w (или wt)	Создает текстовый файл для записи. Если файл с указанным именем существует, то прежняя информация уничтожается
w+ (или wt+)	Создает текстовый файл для чтения и записи данных
wb	Создает двоичный файл для записи. Если файл с указанным именем существует, то прежняя информация уничтожается
wb+ (или w+b)	Создает двоичный файл для чтения и записи данных
a (или at)	Открывает текстовый файл для записи. Указатель устанавливается в конец файла
a+ (или at+)	Открывает текстовый файл для чтения и записи данных. Указатель устанавливается в конец файла. Если файл с указанным именем отсутствует, то он будет создан
ab	Открывает двоичный файл для записи. Указатель устанавливается в конец файла
ab+ (или a+b)	Открывает двоичный файл для чтения и записи данных. Указатель устанавливается в конец файла. Если файл с указанным именем отсутствует, то он будет создан

В режим открытия могут добавляться другие параметры (табл. 10.2).

Таблица 10.2

Режим открытия	Действие
x	При использовании совместно с "w" или "w+" вызывает ошибку, если указанный в параметрах функции файл уже существует
T	Определяет файл как временный. По возможности он не сбрасывается на диск
D	Определяет файл как временный. Он удаляется с диска, после того как закрывается последний указатель файла

Для создания файла можно записать:

```
FILE* fl;
 errno_t  err = fopen_s(&fl, "lab.dat", "w+b");
  if (err == 0)  cout << "The file was opened";
   else {
     cout << "The file was not opened";
     return 1;
         }
```

Для исключения ошибки, возникающей при открытии несуществующего файла, можно использовать конструкцию

```
err = fopen_s(&fl, "lab.dat", "r");
if (err) err = fopen_s(&fl, "lab.dat", "w");
```

При записи данных обмен происходит не непосредственно с файлом, а с некоторым буфером. Информация из буфера переписывается в файл только при переполнении буфера или при закрытии файла.

Для закрытия файла используется функция

int **fclose**(FILE **указатель_на _файл*);

Функция закрывает поток, открытый с помощью вызова fopen(), и записывает в файл все данные, которые находятся в дисковом буфере. Доступ к файлу после выполнения функции будет запрещен. Если файл был закрыт без ошибок, то функция возвращает нуль, иначе − EOF.

Для закрытия всех открытых файлов используется функция

int **_fcloseall**(**void**);

Функция

int **fputc**(**int** *символ*, FILE **указатель_на _файл*);

записывает один ***символ*** в текущую позицию указанного открытого файла. Если функция выполнилась успешно, то она возвращает записанный символ, иначе – EOF.

Функция

int **fgetc**(FILE *_указатель_на_файл_);

читает один символ из текущей позиции указанного открытого файла. После чтения указатель сдвигается на одну позицию вперед. Если достигнут конец файла, то функция возвращает значение EOF.

Функция

int **feof**(FILE *_указатель_на_файл_);

возвращает отличное от нуля значение (*true*) при попытке чтения данных после конца файла и нуль (*false*), если конец файла не достигнут. Функция работает с файлами всех типов.

Функция

int **fputs** (const **char** * _строка_, FILE *_указатель_на_файл_);

записывает строку символов в текущую позицию указанного открытого файла. В случае ошибки эта функция возвращает EOF. Нулевой символ в файл не записывается.

Функция

char ***fgets**(**char** *_строка_, **int** _длина_, FILE *_указатель_на_файл_);

читает строку символов из текущей позиции указанного открытого файла до тех пор, пока не будет прочитан символ перехода на новую строку или количество прочитанных символов не станет равным длина – 1. В случае ошибки функция возвращает NULL.

Функция

int ***fprintf**(FILE *_указатель_на_файл_,
const char *_строка форматирования_ [, _аргументы_]);

записывает форматированные данные в файл.

Строка форматирования состоит из обычных символов и спецификаторов формата.

Общий вид этого спецификатора формата:

% [_флаг_] [_ширина_] [._точность_] <_символ_формата_>

Параметр флаг определяет выравнивание числа при выводе. Некоторые значения флага приведены в табл. 10.3.

Таблица 10.3

Флаг	Назначение
–	Выравнивает выводимое число по левому краю поля
+	Всегда будет выводиться знак числа
Пробел	Устанавливает пробел перед положительным числом и минус перед отрицательным

Параметр ширина определяет минимальное количество выводимых символов.

Параметр точность имеет различное назначение для различных типов выводимых данных. Для действительных чисел, выводимых с использованием спецификаторов %f или %e, точность определяет количество десятичных разрядов, а с использованием спецификатора %g – количество значащих цифр. При выводе строк точность определяет максимальную длину поля вывода, а при выводе целых чисел − максимальное количество цифр.

Некоторые *символы формата* приведены в табл. 10.4.

Таблица 10.4

Символ формата	Значение
c	Вывод одного символа
d	Вывод целого десятичного числа со знаком
e	Вывод числа в экспоненциальном формате ($\pm x.xx\, e \pm xx$)
f	Вывод числа с плавающей точкой ($\pm xx.xxx$)
s	Вывод строки символов
p	Вывод значения указателя

Функция

```
int fscanf_s(FILE *указатель_на_файл,
        const char *строка_форматирования [, аргументы]);
```

читает форматированные данные из файла. Строка форматирования аналогична строке форматирования функции fprintf.

Функция

```
void rewind(FILE *указатель_на_файл);
```

устанавливает указатель текущей позиции в начало файла.

Функция

```
int ferror(FILE *указатель_на_файл);
```

возвращает ненулевое значение, если при последней операции с файлом произошла ошибка, иначе − возвращает 0 (*false)*.

Функция

```
size_t  fwrite(const void *записываемое_данное,
            size_t размер_элемента, size_t число_элементов,
                        FILE *указатель_на_файл);
```

записывает в файл заданное число данных указанного размера. Размер данных задается в байтах. Функция возвращает число записанных элементов.

Функция

```
size_t  fread(void *переменная,
              size_t размер_элемента, size_t число_элементов,
                                      FILE *указатель_на_файл);
```

считывает в указанную переменную заданное число данных указанного размера. Размер данных задается в байтах. Функция возвращает число прочитанных элементов.

Функция

```
int _fileno(FILE *указатель_на_файл);
```

возвращает значение дескриптора указанного файла (дескриптор – логический номер файла для заданного потока).

Функция

```
long _filelength(int дескриптор);
```

возвращает длину файла с соответствующим дескриптором в байтах.

Функция

```
int _chsize(int дескриптор, long  размер);
```

устанавливает новый размер файла с соответствующим дескриптором. Если размер файла увеличивается, то в конец добавляются нулевые символы, если размер файла уменьшается, то все лишние данные удаляются. В случае успешного изменения функция возвращает 0 (иначе –1).

Функция

```
int fseek(FILE * указатель_на _файл,
                  long int число_байтов, int точка_отсчета);
```

устанавливает указатель в заданную позицию. Заданное количество байтов отсчитывается от начала отсчета, которое задается следующими макросами: начало файла – SEEK_SET, текущая позиция – SEEK_CUR, конец файла – SEEK_END. При успешном завершении работы функция возвращает нуль, а в случае ошибки – ненулевое значение.

Пример 10.1. Написать программу сортировки данных в файле по невозрастанию.

```
fopen_s(&fl, "f:\\lab10.dat", "rb+");
int nb = sizeof(int), a, b, nwrt;
int n = _filelength(_fileno(fl)) / nb;
for (int i = 0; i<n - 1; i++)
  for (int j = i + 1; j<n; j++) {
      fseek(fl, i*nb, SEEK_SET);
            nwrt = fread(&a, nb, 1, fl);
      fseek(fl, j*nb, SEEK_SET);
```

```
                nwrt = fread(&b, nb, 1, fl);
if (a>b)
{
        fseek(fl, i*nb, SEEK_SET);
                nwrt = fwrite(&b, nb, 1, fl);
        fseek(fl, j*nb, SEEK_SET);
                nwrt = fwrite(&a, nb, 1, fl);
}  }
fclose(fl);
```

Пример 10.2. Написать программу для работы с бинарным файлом, содержащим список студентов. Каждый элемент списка содержит следующую информацию: фамилия студента, номер группы, средний балл за последнюю сессию. Занести данные в файл. Прочитать данные из файла и вывести на экран и в текстовый файл фамилии студентов, сдавших сессию со средним баллом выше 7.

```
struct TStud
{
   char fio[50];
   int no;
   double score;
} list;

int main()
{
   system("chcp 1251");
   FILE *fl, *ft;
   char fname1[20], fname2[20];
   int n;
   cout << "Введите имя файла : " << endl;
   cin >> fname1;

   if (fopen_s(&fl, fname1, "wb")) {  // Создание файла
      cout << "Error" << endl;
      return 1;
   }

   cout << "Введите число студентов: ";
   cin >> n;
```

```
    for (int i = 0; i < n; i++) {
        cout << "Введите фамилию: ";       cin >> list.fio;  cin.ignore();
        cout << "Введите номер группы: ";  cin >> list.no;
        cout << "Введите средний балл: ";  cin >> list.score;
        fwrite(&list, sizeof(TStud), 1, fl);
    }
    fclose(fl);  // Закрытие файла

    cout << "Введите имя текстового файла: " << endl;
    cin >> fname2;

    if (fopen_s(&ft, fname2, "w")) {  // Создание файла
        cout << "Error" << endl;
        return 1;
    }

    if (fopen_s(&fl, fname1, "rb")) {  // Открытие файла
        cout << "Error" << endl;
        return 1;
    }

    n = _filelength(_fileno(fl)) / sizeof(TStud);
    for (int i = 0; i < n; i++)
    {
        fread(&list, sizeof(TStud), 1, fl);
        if (list.score >= 7) {
            cout << list.fio << " - средний балл : " << list.score << endl;
            fprintf(ft, "%s - средний балл  %f\n", list.fio, list.score);
        }
    }

    fclose(fl);
    fclose(ft);
}
```

11. Функции

11.1. Понятие функции

Функция – последовательность операторов, оформленная таким образом, что ее можно вызвать по имени из любого места программы.

Объявление функции:

```
тип_возвращаемого_значения имя_функции (список_параметров)
        {
    // Тело функции
  return возвращаемое_значение;
        }
```

Первая строка данного описания называется ***заголовком функции***.

Результат вычисления функции, помещаемый в точку вызова, называется ***возвращаемым значением***. Тип возвращаемого значения может быть любым, кроме массива или функции. Если функция не возвращает значение, то указывается тип void.

Список параметров функции (формальных параметров) представляет собой набор конструкций следующей формы:

```
тип_параметра имя_параметра
```

Пример вызова функции:

```
double f(int a, double b)
{
        return a + b;
}
void main() {
        cout << f(5, 9.3);
}
```

При вызове функции программа приостанавливает свою работу и передает управление функции. После завершения работы функции результат ее работы передается в точку вызова и программа продолжает выполняться.

Правила оформления тела функции такие же, как и для любого другого участка программы. Все объявления носят локальный характер, т. е. объявленные переменные доступны только внутри функции.

Не допускается вложение функций друг в друга.

Выход из функции происходит при достижении закрывающей функцию скобки или после выполнения оператора return.

Функция должна быть объявлена до ее вызова, иначе компиляция завершится ошибкой. Для инициализации указателя на функцию до ее определения используется *прототип функции*. Прототип функции аналогичен заголовку

функции, за исключением того, что имена формальных параметров не указываются (остаются только типы) и в конце ставится точка с запятой:

```
double f(int, double);
```

Широкое использование прототипов вызвано следующим:

– функции, имеющие прототипы, могут быть размещены в других модулях;

– использование прототипов позволяет размещать функции в произвольном порядке (а не до первого их использования);

– размещение прототипов в одном месте делает программу более читабельной.

Функция не может возвращать массив, однако может вернуть указатель на него:

```
// int [10] f1();  // Ошибка!
int* f2();
```

Для возврата нескольких значений в качестве возвращаемого значения можно использовать структуру

```
struct St
{
   int x;
   double y;
   int mas[5];
};
St f();
```

Разрешено использовать ключевое слово auto для обозначения типа возвращаемого результата. В этом случае компилятор выводит тип автоматически, исходя из типа данных, передаваемых оператору return:

```
auto f() {
  return 0.3;
}
```

В примере функция будет возвращать результат действительного типа.

В новых стандартах языка *C++* для возврата значений разрешено использовать структурированные привязки:

```
struct St {
   int x;
   double y;
   char c;
};
St f() {
   int a=4;
   double b=7.3;
```

```
    char c = 'w';
    return { a, b, c };
}
void main() {
    auto [p1, p2, p3] = f();
    cout << p1 << " " << p2 << " " << p3 << endl;
}
```

В операторе auto [p1, p2, p3] = f() создаются переменные, тип которых соответствует типу возвращаемых данных.

Спецификатор constexpr используется для обозначения константной переменной или функции, значение которой вычисляется на этапе компиляции (если это возможно).

```
constexpr int f(int a, int b)
{
  return a + b;
}
void main() {
      int x=3, y=7, z;
      cin >> z;
int s1 = f(x, y);  // Знач. будет вычислено на этапе компиляции
int s2 = f(x, z); // Знач. будет вычислено во время выполн. прогр.
}
```

11.2. Параметры функции

Описание функции в языке *C* обязательно должно содержать раздел параметров (открытая и закрытая скобки). Если функция не имеет параметров, то скобки остаются пустыми:

```
int fun()  или  int fun(void)
```

Для каждого параметра должен указываться его тип. Сокращенная запись (один тип для нескольких параметров) запрещена:

```
// double f(int x, y)        // Ошибка!
double f(int x, int y)       // Правильно
```

Область видимости параметров – до конца функции, в которой они объявлены.

Количество *аргументов (фактических параметров),* их типы и порядок следования должны соответствовать количеству, типам и порядку следования *параметров (формальных параметров)*, указанных при объявлении функции. Имена параметров и аргументов могут не совпадать.

Существует три основных способа передачи параметров: передача по значению, по указателю и по ссылке.

11.2.1. Передача параметров по значению

При использовании передачи параметров по значению при вызове функции выделяется память в соответствии с типом параметров и инициализируется значениями соответствующих аргументов. Доступ к аргументам из функции отсутствует, т. е. функция использует локальные копии аргументов. Память, выделенная под такие параметры, освобождается при выходе из функции.

Пример передачи параметров по значению:

```
double f(double, int, char);   // Прототип функции
int main() {
    double a1 = 2.3;
    int a2 = 10;
    char a3 = 'c';
    cout << a1 << " " << a2 << " " << a3 << endl; // Выводит: 2.3 10 c
    double s = f(a1, a2, a3);                // Вызов функции
     cout << a1 << " " << a2 << " " << a3 << endl; // Выводит: 2.3 10 c
       cout << "s= " << s << endl;          // Выводит: s = 116.5
    return 0;
}
double f(double a, int b, char c)  // Заголовок функции
{
    a += 5.2; b--; c++;
    cout << a << " " << b << " " << c << endl;  // Выводит: 7.5 9 d
    return a + b + c;
}
```

Достоинства передачи параметров по значению:

1. В качестве аргументов можно использовать переменные, константы, выражения, структуры, классы, перечисления. Так, в примере выше вызов функции можно описать следующим образом:

```
double s = f(1+1.3,10,'c');
```

2. Защищены данные в вызывающей функции.

Недостатки передачи параметров по значению:

1. Затраты времени и памяти на копирование значений. Использование в качестве параметров объектов структур и классов может привести к значительному снижению производительности.

2. Функция не может изменять значения аргументов.

11.2.2. Передача параметров по указателю

При использовании передачи параметров по указателю в качестве параметров используются не аргументы, а указатели на них. Поэтому любые изменения параметров функции (с помощью операции разадресации) приводят к изменению аргументов в вызывающей функции:

```
double f(double*, int*, char*);   // Прототип функции
int main() {
   double a1 = 2.3;
   int a2 = 10;
   char a3 = 'c';
   cout << a1 << " " << a2 << " " << a3 << endl; // Выводит: 2.3 10 c
    double s = f(&a1, &a2, &a3);                  // Вызов функции
   cout << a1 << " " << a2 << " " << a3 << endl; // Выводит: 7.5 9 d
     cout << "s= " << s << endl;                   // Выводит: s = 116.5
   return 0;
}
double f(double *a, int *b, char *c)  // Заголовок функции
{
   *a += 5.2; (*b)--; (*c)++;
   cout << *a << " " << *b << " " << *c << endl;  // Выводит: 7.5 9 d
   return *a + *b + *c;
}
```

Достоинства передачи параметров по указателю:

1. Экономия ресурсов, связанная с тем, что при передаче не происходит копирование аргументов.

2. Возможность передачи в вызывающую функцию любого количества значений.

3. Возможность изменения значения аргументов.

Недостатки передачи параметров по указателю:

1. В качестве аргументов нельзя использовать объекты, которые не имеют идентифицированного места в памяти (rvalue).

2. Функция может изменять значение аргумента, что может привести к ошибкам в программе. Для решения этой проблемы можно использовать ключевое слово const перед соответствующим аргументом:

```
double f(const double*, const int*, const char*); // Прототип функции
// Заголовок функции
double f(const double *a, const int *b, const char *c)
```

11.2.3. Передача параметров по ссылке

Использование указателей не всегда удобно из-за необходимости использования операции разадресации, поэтому в языке *C++* введена передача параметров по ссылке. Ссылочный параметр («алиас») является псевдонимом соответствующего аргумента.

```
int x = 3, y = 9;
   int* p = &x;       // Указатель
   int& l = y;        // Ссылка
     cout << *p << " " << l << endl;     // Выводит: 3 9
```

Разница между ссылками и указателями состоит в том, что ссылка инициализируется один раз и после этого изменить ее нельзя, а указатель может изменять свое значение (в том числе на nullptr):

```
int x = 3, y = 9;
   int* p = &x;
       p = &y;
    cout << *p << " " << x << " " << y << " " << endl;  // Выводит: 9 3 9
       int& l = x;
        l = y;
     cout << l << " " << x << " " << y << " " << endl;  // Выводит: 9 9 9
```

При обращении к ссылочным параметрам функции происходит обращение к соответствующим аргументам в вызывающей функции. Например:

```
double f(double &, int &, char &);       // Прототип функции
int main() {
   double a1 = 2.3;
   int a2 = 10;
   char a3 = 'c';
   cout << a1 << " " << a2 << " " << a3 << endl; // Выводит: 2.3 10 c
    double s = f(a1, a2, a3);               // Вызов функции
   cout << a1 << " " << a2 << " " << a3 << endl; // Выводит: 7.5 9 d
     cout << "s= " << s << endl;          // Выводит: s = 116.5
   return 0;
}
double f(double &a,  int &b, char &c) {         // Заголовок функции
   a += 5.2; b--; c++;
   cout << a << " " << b << " " << c << endl;  // Выводит: 7.5 9 d
   return a + b + c;
}
```

Достоинства и передачи параметров по адресу аналогичны достоинствам и недостатками передачи параметров по указателю.

По ссылке могут передаваться любые параметры, в том числе указатели:

```
void f(int *& a, int *&b)
{
   int* t;
   t = a; a = b; b = t;
}
void main() {
   int x = 3, y = 9;
   int* p1 = &x, * p2 = &y;
   cout << *p1 << " " << *p2 << endl; // Выводит: 3 9
   f(p1, p2);
   cout << *p1 << " " << *p2 << endl; // Выводит: 9 3
}
```

11.2.4. Передача параметров через глобальные переменные

Глобальные переменные доступны во всех функциях программы (описанных ниже объявления глобальной переменной). Функции могут использовать эти переменные для обмена данными:

```
int x = 10, y = 5, s;        // Глобальные переменные
void sum() { s = x + y; }
void raz() { s = x - y; }
void main() {
 sum();
  cout << s << endl;       // Выводит: 15
 raz();
  cout << s << endl;       // Выводит: 5
}
```

Достоинство передачи параметров через глобальные переменные: доступность данных из любого места программы.

Недостатки передачи параметров через глобальные переменные:

1. Высокая вероятность ошибок, так как изменение глобальной переменной оказывает влияние на все функции, ее использующие.

2. Сложность отладки программ и поиска ошибок (область поиска – вся программа).

11.2.5. Параметры со значениями по умолчанию

При объявлении функции для некоторых аргументов можно задавать значение по умолчанию, которое передается в функцию в случае, если при вызове

соответствующий аргумент не задан. Так как компилятор присваивает имеющиеся значения последовательно слева направо, то аргументы, имеющие заданное по умолчанию значение, должны располагаться правее аргументов, не имеющих такого значения. Значения по умолчанию могут задаваться только в одном месте: в прототипе или в заголовке функции. По соглашению значения по умолчанию задаются в прототипе функции. Пропуск аргументов при вызове функции запрещен.

Пример:

```
void f(double a = 5.5, int b = 10, char c = 'a');
int main() {
    f();              // Выводит: 5.5 10 a
    f(9.9);           // Выводит: 9.9 10 a
    f(9.9, 6);        // Выводит: 9.9 6 a
    f(9.9, 6, 'b');   // Выводит: 9.9 6 b
   // f(9.9, , 'b');  // Ошибка!
    f('b'); // 98 10 a
     return 0;
}
void f(double a, int b, char c)
{
    cout << a << " " << b << " " << c << endl;
}
```

11.2.6. Передача массивов в функции

Массив в *C++* запрещено передавать по значению. Так как имя массива является указателем на первый элемент массива, то передача в функцию всегда осуществляется по указателю:

```
void f(int []);
```

или

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

Одновременно с указателем на массив принято передавать и его размер:

```
void f1(int[], int);          // Прототип функции
void f2(int[20], int);        // Прототип функции
                              // (компилятор игнорирует размерность)
void f3(int*, int);           // Прототип функции
void main() {
    const int n = 10;
    int a[n];
    f1(a, n);                 // Вызов функции
```

```
    f2(a, n);          // Вызов функции
    f3(a, n);          // Вызов функции
}
void f1(int b[], int k) {
    //  Тело функции
}
void f2(int b[35], int k) {
    //  Тело функции
}
void f3(int* b, int k) {
    //  Тело функции
}
```

При передаче массива по указателю возможно изменение его элементов в функции, что может привести к ошибкам. Для запрета изменения элементов массива в функции указывается ключевое слово const:

```
void f(const int []);
```

или

```
void f(const int *);
```

Имя массива всегда передается как указатель. При необходимости использования ссылки следует указывать размер массива (ссылка на массив указанного размера):

```
const int n = 10;
void f(int(&)[n]);  // Прототип функции
void main() {
    int a[10];
    f(a);       // Вызов функции
   // f(&a);   // Ошибка!
    int b[5];
   // f(b);     // Ошибка!
}
void f(int(&b)[n]) {
    //  Тело функции
}
```

При передаче многомерного массива скобки для первой размерности остаются пустыми, а для других размерностей должен указываться размер:

```
void f(int[][3]);
void f2( int (*)[3]);
void main() {
```

```
    const int n = 3;
    int a[n][n];
f(a);          // Вызов функции
f2(a);         // Вызов функции
}
void f(int b[][3]) {
    //  Тело функции
}
void f2(int (*b)[3]) {
    //  Тело функции
}
```

11.2.7. Передача переменного числа параметров

Формат объявления функции с переменным числом:

тип_возвращаемого_значения имя_функции (*список_параметров*, **…**)

Список параметров содержит хотя бы один обязательный параметр. Многоточие (эллипсис, англ. *ellipsis*) указывает на возможность добавления любого числа параметров.

Для работы с параметрами определен тип списка va_list и три макроса:

void va_start(va_list *указатель*, *имя_послед._обязат._аргумента*)

начинает работу со списком. Устанавливает указатель на первый необязательный аргумент.

void va_arg(va_list *указатель*, *тип_аргумента*)

возвращает значение очередного аргумента из списка. Каждый запуск макроса переводит указатель на следующий аргумент. Достижение последнего аргумента списка не контролируется.

void va_end(va_list *указатель*)

завершает работу со списком и освобождает память.

Пример 11.1. Подсчитать сумму введенных аргументов. Первый параметр функции должен передавать количество элементов.

```
#include <iostream>
#include <cstdarg>
using namespace std;
int f(int, ...);          // Прототип функции
void main() {
    cout << f(5, 1, 2, 3, 4, 5) << endl;  // Выводит: 15
    cout << f(3, 1, 2, 3) << endl;        // Выводит: 6
}
```

```
int f(int n, ...) {    // Заголовок функции
   int ar, s = 0;
   va_list argm;
   va_start(argm, n);
   for (int i = 0; i < n; i++)
      s += va_arg(argm, int);
   va_end(argm);
   return s;
}
```

11.3. Перегрузка функций

Под перегрузкой функций понимается использование различных функций с одинаковым именем, объявленных в одной области видимости. Перегрузка применяется для ситуаций, когда требуется выполнить эквивалентные по смыслу, но разные по алгоритму действия. Например, вычисление площади для различных фигур или нахождение минимума для различных наборов данных.

Перегруженные функции различаются компилятором по типам и числу параметров. Например, необходимо вычислить площадь для круга и прямоугольника:

```
double S(double a, double b) { return a * b; }
double S(double r) { return 3.14 * pow(r, 2); }
void main() {
      cout << "Circle area " << S(4) << endl;
      cout << "Rectangle area " << S(3,5) << endl;
}
```

При выборе перегруженной функции имена параметров, тип возвращаемого результата, спецификатор const (если не относится ко всей функции) и способ передачи массива в функцию не учитываются:

```
int f(int x) {}
//double f(int x) {}          // Ошибка
void f(double x) {}
int f(int x[]) {}
// int f(int* x) {}           // Ошибка
 int f(char x) {  }
// int f(const char x) { }    //Ошибка
```

Если типы аргументов при вызове функции не совпадают с типами параметров, то подходящая функция ищется исходя из возможных преобразований типов.

Пример 11.2. Вывести на экран таблицу значений функции $y(x) = \ln(x+1)$ и ее разложения в ряд $s(x) = \frac{x}{1} - \frac{x^2}{2} + ... + (-1)^{n+1}\frac{x^n}{n}$ с точностью ε = 0.001. Вывести число итераций, необходимое для достижения заданной точности.

```
double f(double);
double f(double, int &, double eps=1e-6);

int main() {
        double a = 0.1, b = 0.9, h = 0.1;
        int k;
cout << setw(5) << "x" << setw(12) << "y(x)" << setw(12) << "s(x)"
                                                << setw(5) << "k" << endl;

        for (double x = a; x < b + h / 2; x += h)
cout << setw(5) << x << setw(12) << f(x) << setw(12) << f(x, k)
                                                << setw(5) << k <<endl;
}

double f(double x) {
        return log(x+1);
}

double f(double x, int &k, double eps) {
        double a, c, sum;
        sum = c = a = x;
         k = 2;
        while (fabs(a) > eps)
        {
                c *= -x;
                a = c/k++;
                sum += a;
        }
        return sum;
}
```

11.4. Встраиваемые функции

Встраивание функции – способ оптимизации программы, при котором в места вызова функции вставляется ее тело.

Использование функций удобно, с точки зрения программиста, однако для исполняемого модуля это не всегда хорошо, так как тратится время на организацию вызовов функций. При использовании маленьких функций накладные расходы на организацию работы с ними превышают затраты, возникающие при непосредственном внесении тела функции в код программы.

Для описания встраиваемой функции используется ключевое слово inline.

Например, если функцию объявить

```
inline double f(int a, double b) {
    return a*b;
}
```

то при компиляции во всех местах, где встречается обращение к этой функции, будет вставлено тело функции.

Наличие ключевого слова inline не обязывает компилятор встраивать тело функции в программу. Компилятор игнорирует спецификацию inline в следующих случаях:

– при наличии операторов организации циклов (for, do, while), переключателей (switch) и безусловного перехода (goto) в функциях, возвращающих значения;

– при наличии оператора return в функциях, не возвращающих значения;

– при обнаружении рекурсивного вызова функции;

– при использовании статических переменных (с атрибутом static).

Достоинство использования встраиваемых функций: увеличение скорости выполнения программы.

Недостатки использования встраиваемых функций:

– вставленные тела функций увеличивают размер программы;

– изменение функций inline требует перекомпиляции всех программ, их использующих.

11.5. Указатель на функцию

Имя функции (как и имя массива) является константным указателем на начало функции в оперативной памяти. Разрешается использовать указатели на функции в программе.

Например, имеется функция

```
double y(double x, int n)
{
// Тело функции
}
```

Указатель на такую функцию имеет вид

```
double (*fun)(double, int);
```

Если присвоить указателю fun адрес функции y:

```
fun = y;
```

то функцию можно вызывать

```
x = fun(t, m);
```

Указатели на функции, как правило, применяются при использовании функций в качестве аргументов других функций.

Пример 11.3. Вывести на экран таблицу значений функции $y(x) = \sin x$ и ее разложения в ряд $s(x) = x - \frac{x^3}{3!} + \ldots + (-1)^n \frac{x^{2n+1}}{(2n+1)!}$ с точностью ε = 0.001. Вывести число итераций, необходимое для достижения заданной точности.

```
typedef double (*uf)(double, double, int&);
void tabl(double, double, double, double, uf);
double y(double, double, int&);
double s(double, double, int&);
int main()
{
cout << setw(8) << "x" << setw(15) << "y(x)" << setw(10) << "k" <<
endl;
	tabl(0.1, 0.8, 0.1, 0.001, y);
cout << endl;
cout << setw(8) << "x" << setw(15) << "s(x)" << setw(10) << "k" <<
endl;
	tabl(0.1, 0.8, 0.1, 0.001, s);
	return 0;
}
void tabl(double a, double b, double h, double eps, uf fun)
{
	int k = 0;
	double sum;
	for (double x = a; x < b + h / 2; x += h)
	{
		sum = fun(x, eps, k);
cout << setw(8) << x << setw(15) << sum << setw(10) << k << endl;
	}
}
```

```
double y(double x, double eps, int& k)
{
	return sin(x);
}
double s(double x, double eps, int& k)
{
	double a, c, sum;
	sum = a = c = x;
	k = 1;
	while (fabs(c) > eps)
	{
		c = pow(x, 2) / (2 * k * (2 * k + 1));
		a *= -c;
		sum += a;
		k++;
	}
	return sum;
}
```

11.6. Ссылка на функцию

Допустимо использование ссылок на функцию. Например:

```
void f(int); // Прототип функции
void(&sdf)(int) = f;
sf(3); // Вызов функции (то же самое, что и f(3);)
```

Для использования ссылки в примере 11.3 требуется переписать первый оператор:

```
using uf = double (double, double, int&);
```

Использование ссылок на функции не имеет преимуществ по сравнению с использованием указателей (к тому же имеются ограничения), поэтому они используются редко.

12. Область видимости и классы памяти

Область видимости определяет, в каких частях программы возможно использование данной переменной, а ***класс памяти*** – время, в течение которого переменная существует в памяти компьютера. Период времени между созданием и уничтожением переменной называется ***временем жизни*** переменной.

В языке *C++* определены четыре класса памяти:

Автоматический локальный (auto) класс памяти. Область видимости локальных переменных ограничена функцией или блоком, в котором она объявлена. Время жизни автоматической локальной переменной − промежуток времени между ее объявлением и завершением работы функции или блока, в которых она объявлена. Ограничение времени жизни переменной позволяет экономить оперативную память. Этот класс памяти используется по умолчанию.

Статический локальный (static) класс памяти. Переменная имеет такую же область видимости, как и автоматическая. Время жизни статической локальной переменной − промежуток времени между ее объявлением и окончанием работы программы. Инициализация переменной происходит только при первом обращении к ней. Компилятор хранит значение переменной от одного вызова функции до другого. Если статическая переменная не инициализирована явно, она по умолчанию имеет значение 0.

Внешний глобальный (extern) класс памяти. Глобальные переменные объявляются вне функций и доступны во всех функциях, находящихся ниже описания глобальной переменной. Время жизни глобальной переменной совпадает с временем работы программы. В момент создания глобальная переменная инициализируется нулем. Включение ключевого слова extern позволяет функции использовать внешнюю переменную, даже если она определяется позже в этом или другом файле. Память для глобальных переменных выделяется в начале программы и освобождается при завершении ее работы.

Регистровый локальный (register) класс памяти. Является «пожеланием» компилятору помещать часто используемую переменную в регистры процессора для ускорения скорости выполнения программы. Если компилятор отказался помещать переменную в регистры процессора, то переменная становится автоматической.

Если при объявлении переменной класс памяти не указан явно, то он задается автоматически в зависимости от местоположения переменной в тексте программы. Переменные, объявленные внутри функции, по умолчанию имеют класс памяти auto, а остальные – extern.

13. Рекурсивные алгоритмы

13.1. Понятие рекурсии

Рекурсивным называется способ построения объекта, при котором определение объекта включает в себя аналогичный объект в виде некоторой его части. Решать задачу рекурсивно – это значит разложить ее на подзадачи, которые затем аналогичным образом (т. е. рекурсивно) разбиваются на еще меньшие подзадачи, и так до тех пор, пока на определенном уровне подзадачи не становятся настолько простыми, что могут быть решены тривиально. Путем последовательного решения всех элементарных подзадач получается решение всей задачи. Функция называется рекурсивной, если в ее теле содержится вызов аналогичной функции.

Например, необходимо вычислить факториал числа n ($n!$). Известно, что $n! = n \cdot (n-1)!$. Следовательно, для вычисления $n!$ необходимо вычислить $n \cdot (n-1)!$, в свою очередь для вычисления $(n-1)!$ вычисляем $(n-1) \cdot (n-2)!$, для вычисления $(n-2)!$ вычисляем $(n-2) \cdot (n-3)!$ и т. д. На каждом шаге значение вычисляемого факториала уменьшается на единицу. Задача разбивается до тех пор, пока значение n не станет равным 0, т. е. не будет получено тривиальное решение $0! = 1$. Текст программы вычисления факториала:

```
int fact(int n)
{
        if (n <= 0) return 1;
              else return n*fact(n-1);
}
```

Рассмотрим работу функции для расчета 4!. Процесс рекурсивных вызовов и возврата значений показан на рис. 13.1.

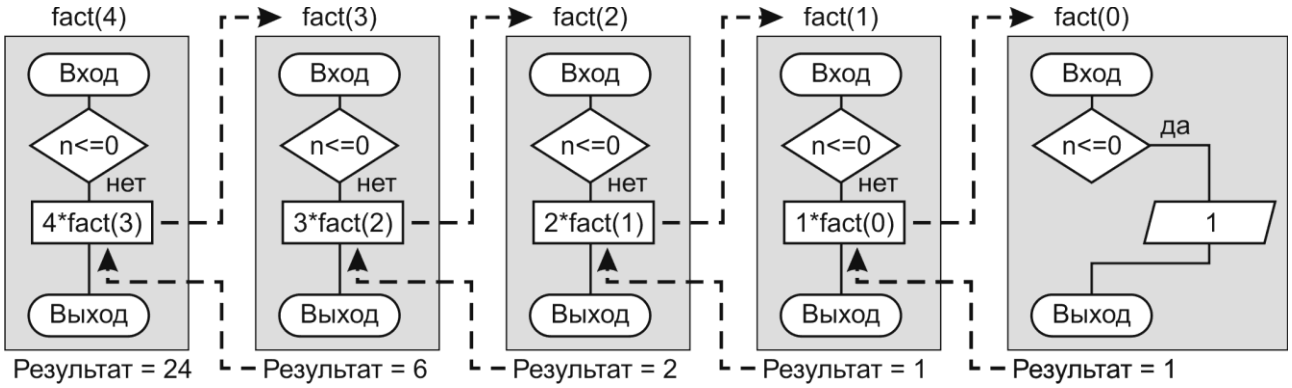

Рис. 13.1

При каждом рекурсивном вызове вызывающая функция приостанавливает свою работу и все ее данные сохраняются в специальной области памяти, называемой *стеком*. Структура стека такова, что в него можно последовательно вносить данные, а затем извлекать в обратном порядке (первый вошел − последний вышел). После достижения дна рекурсии происходит последовательная выборка данных из стека.

13.2. Условие окончания рекурсивного алгоритма

Если в рекурсивном алгоритме не выполняется условие окончания рекурсивных вызовов, то такой алгоритм будет вызывать функцию бесконечно (до тех пор, пока не будет переполнен стек). В программе обязательно должен присутствовать оператор, прекращающий рекурсивный вызов функции при достижении определенных значений текущих данных. Для предотвращения переполнения программного стека требуется делать оценку максимальной глубины рекурсии.

Бесконечная рекурсия может возникать не только при отсутствии условия прекращения рекурсивного вызова функции, но и при неполном учете всех возможных путей движения рекурсии. Например, если факториал рассчитывается следующим образом:

```
int fact(int n)
{
      if (n == 0) return 1;
      else return n*fact(n-1);
}
```

то при вводе числа, меньшего нуля, функция будет вызываться бесконечно.

13.3. Типы рекурсивных алгоритмов

Можно выделить пять основных типов рекурсивных алгоритмов:

– линейная рекурсия. Рекурсивная функция вызывается один раз. Результат вызова используется в финальной операции;

– хвостовая рекурсия. Рекурсивная функция вызывается один раз, но результат вызова является последним действием;

– множественная рекурсия. Рекурсия в одном операторе вызывается несколько раз;

– взаимная рекурсия. Несколько рекурсивных функций взаимно (циклически) вызывающих друг друга;

– вложенная рекурсия. Одним из параметров функции является вызов аналогичной функции.

13.4. Примеры рекурсивных алгоритмов

Пример 13.1. Найти сумму $S_n = \sum_{i=1}^{n} a_i$

```
int sumr(int i) {
  if (i < 0) return 0;
    else return a[i] + sumr(i-1);
}
```

Пример 13.2. Найти наибольший общий делитель двух чисел, используя следующее соотношение: если B делится на A нацело, то НОД(A, B) = A; иначе НОД(A, B) = НОД(B %A, A).

```
int nodr(int a, int b)  {
      if( b%a == 0) return a;
                  else return nodr(b%a,a);      }
```

Пример 13.3. Найти max($a_0 ... a_{n-1}$).

Данную задачу можно разбить на следующие **элементарные подзадачи**: max(max($a_0 ... a_{n-2}$), a_{n-1}), и далее max(max(max($a_0 ... a_{n-3}$), a_{n-2}), a_{n-1}), …, каждая из которых решается выбором: если $x > y$, тогда $mx = x$, иначе $mx = y$. На последнем уровне окажется **тривиальная задача** – max(a_0) → $mx = a_0$, после чего находится max(mx, a_1) и т. д.

```
int maxr2(int i) {
      if (i == 0) return a[0];
        int mx = maxr2(i-1);
          if (a[i] > mx)  return a[i];
                    else return mx;
}
```

Пример 13.4. Найти сумму элементов одномерного массива (при рекурсивном разбиении массив следует делить на две части).

```
int sumr(int a[], int i, int j)
{
      if (i == j) return a[i];
      else
       return sumr(a,i,(j+i)/2) + sumr(a,(j+i)/2+1,j);
}
```

Пример 13.5. Вычислить числа Фибоначчи, которые определяются следующим рекурсивным соотношением: $b_0 = 0$; $b_1 = 1$; $b_n = b_{n-1} + b_{n-2}$.

```
int fibr (int n)
{   if (n <= 1) return n;
     else return fibr(n-1)+fibr(n-2);    }
```

Данная реализация алгоритма имеет изящный код, однако работает неэффективно. Каждое обращение к функции приводит к вызову еще двух функций. С увеличением n число вызовов возрастает как 2^{n-1}. Например, при $n = 5$ дерево вызовов программы будет иметь вид, представленный на рис. 13.2.

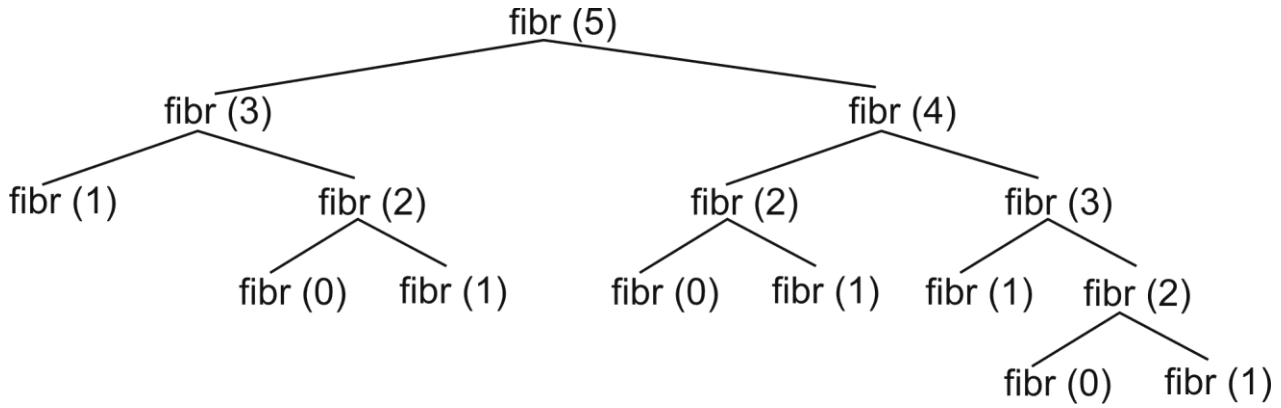

Рис. 13.2

Видно, что при выполнении программы требуется стековая память для хранения данных для 16 (2^4) функций. Большим недостатком алгоритма является многократный вызов функции с одинаковыми параметрами.

Так как функция Фибоначчи растет достаточно быстро, то для больших значений *n* рекурсивный алгоритм будет работать медленно или совсем перестанет работать из-за переполнения стека. Поэтому практического интереса такая рекурсивная программа не представляет.

Рассмотренный выше пример показывает, что компактная и красивая программа не всегда эффективна. Для вычисления чисел Фибоначчи удобно использовать обычный итерационный алгоритм или алгоритм с одним рекурсивным вызовом:

```
int fibri (int x1, int x2, int n)
{
if (n == 1) return x2;
    else   if (n == 0) return x1;
                else    {
  x2 += x1;
  x1 = x2-x1;
       return fibri(x1, x2, n-1);
       }
}
```

Обращение к функции: `fibri(0,1,n);`

Пример 13.6. Задача о Ханойской башне. Даны три стержня, на один из которых нанизаны *n* колец. Кольца отличаются размером и расположены меньшее на большем. Необходимо перенести башню на другой стержень. За один раз разрешается переносить только одно кольцо, причем нельзя класть большее кольцо на меньшее. Для промежуточного хранения дисков можно использовать третий стержень.

Решение задачи для одного диска: переложить диск с *первого* стержня на *второй* стержень.

Решение задачи для двух дисков:

− переложить диск с *первого* стержня на *третий* стержень;

− переложить диск с *первого* стержня на *второй* стержень;

− переложить диск с *третьего* стержня на *второй* стержень.

Башня, состоящая из *n* дисков, рассматривается как башня из двух дисков: первый диск − верхний диск башни, а второй диск − все диски, располагающиеся под верхним диском. После перестановки этих двух составных дисков задача решается для $n - 1$ дисков.

```
void hanoy(int n, int sterg1, int sterg2, int sterg3)
{
      if (n > 0) {
      hanoy(n-1,sterg1,sterg3,sterg2);
          cout << "perenesty disk s "<<sterg1<<" na "<<sterg2<<endl;
      hanoy(n-1,sterg3,sterg2,sterg1);
                }
}
```

Обращение к функции: hanoy(n,1,2,3);

Пример 13.7. Вычислить $y = x^n$ по следующему алгоритму: $y = (x^{n/2})^2$, если *n* четное; $y = x \cdot x^{n-1}$, если *n* нечетное.

```
double st(int n)
{   if (n == 0) return 1;
 else   {
       if (n%2 == 0)   {
              double p = st(n/2);
              return p * p;
                    }
       else return x * st(n-1);
        }
}
```

13.5. Целесообразность использования рекурсии

Рекурсивные алгоритмы хорошо подходят для задач, допускающих рекурсивное разбиение на элементарные подзадачи. Однако это не означает, что для решения таких задач бесспорно использование рекурсивных программ. В большинстве случаев использование рекурсии абсолютно неэффективно.

Недостатки рекурсии:

1. Большой расход памяти и ресурсов. При каждом рекурсивном вызове система сохраняет в стеке все локальные данные. Обработка сложной цепочки рекурсивных вызовов требует выделение больших ресурсов системы.

2. Часто ход выполнения рекурсивного алгоритма требует многократного вычисления функций, имеющих одинаковые входные данные, что существенно снижает быстродействие программы. Классический пример – вычисление чисел Фибоначчи.

3. Несмотря на кажущуюся простоту, рекурсивные программы сложны для понимания и для отладки.

В большинстве случаев нерекурсивный алгоритм выполняется быстрее рекурсивного, однако существует ряд задач, решить которые без использования рекурсии достаточно сложно.

14. Алгоритмы сортировки

Сортировка – процесс перегруппировки элементов массива, приводящий к их упорядоченному расположению относительно заданного ключа. Ключом в зависимости от решаемой задачи может считаться любое поле структуры. Целью сортировки является облегчение поиска элементов.

Сортировку данных в оперативной памяти принято называть ***внутренней***, а сортировку на внешних носителях информации – ***внешней***.

Для оценки эффективности сортировки часто используют следующие критерии:

1. *Скорость* сортировки. Определяется числом сравнений и обменов. Оценивается также скорость сортировки *в наилучшем и наихудшем случаях*, так как существуют алгоритмы, которые, имея хорошую среднюю скорость, очень медленно работают в наихудшем случае.

2. *Естественность* сортировки. Сортировка называется *естественной*, если время сортировки минимально для уже упорядоченных данных и увеличивается по мере возрастания их степени неупорядоченности.

3. *Устойчивость* сортировки. Алгоритм сортировки является устойчивым, если в отсортированном массиве элементы с одинаковыми ключами располагаются в том же порядке, в котором они располагались в исходном массиве. Лучшими считаются алгоритмы, не переставляющие элементы с одинаковыми ключами.

Часто оценивается *сложность алгоритма* − зависимость объема работы, выполняемой некоторым алгоритмом, от размера входных данных.

Существует три основных класса сортировок:

1. Обмен. При таком способе меняются местами элементы, расположенные не по порядку. Обмен продолжается до тех пор, пока все элементы не будут упорядочены.

2. Выбор. Вначале ищется наименьший элемент и ставится на первое место, затем ищется следующий по значимости элемент и устанавливается на второе место и т. д. В результате все элементы помещаются в нужные позиции.

3. Вставка. Последовательно перебираются все элементы. Каждый элемент перемещается в ту позицию, в которой он должен стоять.

14.1. Простые методы сортировки

14.1.1. Метод пузырька

Данная сортировка относится к классу обменных сортировок. Ее алгоритм содержит многократные сравнения соседних элементов и при необходимости их обмен. Элементы ведут себя подобно пузырькам воздуха в воде − каждый из них поднимается на свой уровень.

```
void s_puz(tmas a[], int n) {
   tmas t;
   for (int i = 1; i < n; i++)
```

```
for (int j = n - 1; j > = i; j - )
    if (a[j - 1].key > a[j].key)
    {
        t = a[j - 1];
        a[j - 1] = a[j];
        a[j] = t;
    }
```

Особенности пузырьковой сортировки:

– сложность сортировки – $O(N^2)$;

– с увеличением степени отсортированности массива количество обменов уменьшается, а количество сравнений всегда остается одинаковым;

– сортировка устойчива;

– проста в понимании и реализации.

14.1.2. Сортировка выбором

В массиве выбирается элемент с наименьшим значением и меняется местами с первым элементом. Затем из оставшихся элементов находится наименьший и меняется местами со вторым элементом и т. д.

```
void s_vb(tmas a[], int n)
{
   int imin, i, j;
    tmas t;
 for(i=0; i<n-1; i++)
 {
    imin = i;
  for(j=i+1; j<n; j++)
  if (a[imin].key > a[j].key) imin = j;
   if (imin != i)
   {
         t = a[imin];
         a[imin] = a[i];
         a[i] = t;
    }
} }
```

Особенности сортировки выбором:

– сложность сортировки – $O(N^2)$;

– с увеличением степени отсортированности массива количество обменов уменьшается, а количество сравнений всегда остается одинаковым;

– сортировка неустойчива;

– количество обменов намного меньше, чем в пузырьковой сортировке.

14.1.3. Сортировка вставкой

Сначала сортируются два первых элемента массива. Затем вставляется третий элемент в необходимую позицию по отношению к первым двум элементам. После этого четвертый элемент помещается в соответствующую позицию списка из трех элементов и т. д. Процесс повторяется до тех пор, пока не будут вставлены все элементы.

```
void s_vst(tmas a[], int n)
{
 int i, j;
 tmas t;
 for(i=1; i<n; i++)
 {
        t = a[i];
  for(j=i-1;  j>=0 && t.key<a[j].key;  j--) a[j+1] = a[j];
        a[j+1] = t;
} }
```

Особенности сортировки вставкой:

– сложность сортировки в лучшем случае – $O(N)$, а в худшем – $O(N^2)$;

– с увеличением степени отсортированности массива уменьшается и количество обменов, и количество сравнений;

– сортировка устойчива.

14.2. Улучшенные методы сортировки

Все алгоритмы, рассмотренные выше, имеют один фатальный недостаток – они работают очень медленно. Применяемые методы оптимизации кода не дают существенного прироста производительности алгоритма. Существует правило: если используемый в программе алгоритм слишком медленный сам по себе, никакой объем ручной оптимизации не сделает программу достаточно быстрой. Решение заключается в применении лучшего алгоритма сортировки.

14.2.1. Метод Шелла

Общая идея заимствована из сортировки вставкой. Недостатком сортировки вставкой является то, что меньшие элементы множества быстро помещаются в необходимую позицию, а большие – достаточно медленно. Для преодоления этого недостатка на начальных этапах алгоритм передвигает не все элементы, а только элементы, отстоящие на некотором расстоянии друг от друга. Интервал постепенно уменьшается, пока на станет равным единице. Например, сначала сортируются элементы, расположенные на расстоянии трех позиций друг от друга. Затем сортируются элементы, расположенные на расстоянии двух

позиций. Наконец, сортируются все соседние элементы. Последовательность шагов может быть любой, однако последний шаг обязательно должен быть равен единице.

```
void s_shell(tmas a[], int n)
{
   int i, j;
   tmas t;
for(int d=3; d>0; d--)
   for(i=d; i<n; i++)
 {
     t = a[i];
  for(j=i-d;  j>=0 && t.key<a[j].key;  j-=d)  a[j+d] = a[j];
        a[j+d] = t;
 }
}
```

Особенности сортировки вставкой:

– сложность сортировки в лучшем случае равна $O(N\log^2 N)$, а в худшем – $O(N^2)$ (зависит от выбранной последовательности шагов. Оптимальная последовательность не определена до сих пор);

– количество сдвигов элементов существенно снижено по сравнению с простыми методами сортировки;

– средняя скорость сортировки значительно выше, чем у сортировки вставкой;

– сортировка естественна;

– сортировка неустойчива.

14.2.2. Сортировка слиянием

Алгоритм сортировки слиянием следующий:

1. Сортируемый массив рекурсивно разбивается на смежные участки примерно одинакового размера до тех пор, пока в каждом участке не останется по одному элементу.

2. Смежные упорядоченные участки массива соединяются в один упорядоченный участок, для чего последовательно извлекаются наименьшие элементы из этих участков и помещаются в результирующий массив. Когда в одном из смежных участков элементы заканчиваются, все оставшиеся элементы из другого участка массива перемещаются в результирующий массив. Результирующий массив записывается на место рассмотренных смежных участков.

3. Алгорим прекращает работу в тот момент, когда будут соединены все смежные участки.

Функция слияния:

```
void slip(int left, int m, int right)
{
   int i = left, j = m + 1, k = 0;
   while ((i <= m) && (j <= right))
     if (a[i].key < a[j].key) { c[k++] = a[i++];}
                              else { c[k++] = a[j++];}
   while (i <= m)    c[k++] = a[i++];
   while (j <= right) c[k++] = a[j++];
   // Запись отсортированного участка в массив
   k = 0; i = left;
   while (i <= right) a[i++] = c[k++];
}
```

Функция сортировки:

```
void s_sl(int left, int right)
{
   if (left < right)
   {
      int m = (left + right) / 2;
      s_sl(left, m);
      s_sl(m + 1, right);
      slip(left, m, right);
   }

}
```

Вызов:

```
s_sl(0, n-1);
```

Особенности сортировки слиянием:

– сложность – $O(N^{\log N})$;

– средняя скорость сортировки значительно выше, чем у сортировки вставкой;

– сортировка не является естественной (скорость не зависит от упорядоченности исходных данных);

– сортировка устойчива;

– для работы алгоритма требуется дополнительный массив, поэтому он, как правило, используется для внешней сортировки.

14.2.3. Метод QuickSort (быстрая сортировка, сортировка Хоара)

В начале сортировки задается базовый элемент (средний или выбранный случайным образом). Затем элементы, большие или равные базовому, перемещаются на одну сторону, а меньшие − на другую. После этого аналогичные действия повторяются отдельно для каждой части массива. Процесс повторяется до тех пор, пока массив не будет отсортирован. Алгоритм по своей сути рекурсивный, поэтому его можно реализовать в виде рекурсивной функции:

```
void s_qsr(int left, int right)
{
   int i = left, j = right;
   tmas t, x;
x = a[(i+j)/2];
do {
     while (a[i].key < x.key && i < right) i++;
     while (a[j].key > x.key && j > left)   j--;
  if (I <= j) {
       t = a[i];   a[i] = a[j];   a[j] = t;
  i++;  j--;
   }
  } while( i<=j );
  if(left < j)  s_qsr(left, j);
  if(i < right) s_qsr(i, right);
}
```

Для быстрой работы алгоритма *QuiskSort* необходимо правильно выбрать базовый элемент. Если значение базового элемента при каждом делении будет равно наибольшему значению, то сортировка по скорости станет равной пузырьковой. Методика выбора базового элемента должна отталкиваться от природы сортируемого массива. Например, если данные расположены достаточно равномерно, то удобнее выбирать средний элемент. В других случаях можно использовать случайный выбор базового элемента.

Описанная выше рекурсивная реализация сортировки имеет красивый и понятный алгоритм, однако знание особенностей функционирования рекурсивных программ позволяет предположить, что нерекурсивная реализация будет работать лучше:

```
struct St {
        int l;
        int r;
} stack[10];
```

```
void push(int l, int r, int &s) {
        stack[s].l = l;
        stack[s].r = r;
        s++;
}

void pop(int &l, int &r, int& s) {
        s--;
        l = stack[s].l;
        r = stack[s].r;
}

void s_qs(tmas a[], int n) {
 int i, j, left, right, s = 0;
        tmas x;
 push(0, n-1, s);
        while (s != -1) {
          pop(left, right, s);
                while (left < right) {
                        i = left; j = right; x = a[(left + right) / 2];
                        while (i <= j) {
                                while (a[i].key < x.key) i++;
                                while (a[j].key > x.key) j--;
                                if (i <= j) { swap(a[i], a[j]); i++;  j--; }
                        }
                if ((j - left) < (right - i)) {   // Выбор более короткой части
                                if (i < right) push(i, right, s);
                          right = j;
                        }
                        else {
                                if (left < j) push(left, j, s);
                          left = i;
                        }
                }
        }
}
```

В данной функции в качестве базового выбирается средний элемент. Массив просматривается слева направо до тех пор, пока не будет найден элемент,

больший или равный базовому, и справа налево до тех пор, пока не будет найден элемент, меньший или равный базовому. Найденные элементы меняются местами. Если найденный слева элемент стоит правее элемента, найденного при поиске справа, то поиск прекращается. Массив разбивается на два новых участка. Поиск и разбиение продолжаются до тех пор, пока каждая из частей не будет состоять из одного единственного элемента.

Особенности быстрой сортировки:

– сложность сортировки – $O(N \log N)$, в худшем случае (маловероятно) – $O(N^2)$;

– сортировка естественна;

– сортировка неустойчива (при данной реализации алгоритма);

– для работы алгоритма не требуется использование дополнительного массива;

– количество сравнений намного меньше, чем у любого ранее рассмотренного метода.

15. Алгоритмы поиска

Цель ***поиска*** состоит в нахождении элемента, имеющего заданное значение ключевого поля.

15.1. Линейный поиск

Линейный поиск используется в случае, когда массив не отсортирован по заданному ключу. Поиск представляет собой последовательный перебор элементов массива до обнаружения требуемого ключа или до конца массива, если ключ не обнаружен:

```
int p_lin1(tmas a[], int  n, int x)
{
for(int i=0; i < n; i++)
        if (a[i].key == x) return i;
   return -1;
}
```

В данном алгоритме на каждом шаге делается две проверки: проверка на равенство ключевого поля и искомого ключа и проверка условия продолжения циклического алгоритма. Для исключения проверки условия продолжения циклического алгоритма вводится вспомогательный элемент – *барьер*, который предохраняет от выхода за пределы массива:

```
int p_lin2(tmas a[], int n, int x)
{
 a[n].key = x;
   int i = 0;
 while (a[i].key != x) i++;
  return i;
}
```

Если функция возвращает значение, равное *n*, то это говорит о том, что искомый элемент не обнаружен. Эффективность такого алгоритма почти в два раза выше предыдущего.

15.2. Двоичный (бинарный) поиск

Двоичный поиск используется в случае, когда данные упорядочены, например, по неубыванию ключевого поля. Алгоритм состоит в последовательном исключении той части массива, в которой искомого элемента быть не может. Для этого берется средний элемент, и если значение ключевого поля этого элемента больше, чем значение искомого ключа, то можно исключить из рассмотрения правую половину массива, иначе исключается левая половина массива. Процесс продолжается до тех пор, пока в рассматриваемой части массива не останется один элемент.

```
int p_dv(tmas a[], int n, int x)
{
 int i=0, j=n-1, m;
while(i < j)  {
m=(i + j)/2;
if (x > a[m].key) i = m+1; else j = m;
  }
if (a[i].key == x) return i;
return -1;
}
```

15.3. Интерполяционный поиск

Для массивов с равномерным распределением элементов можно использовать формулу, позволяющую определить примерное местоположение элемента:

$$m = i + \frac{(i - j)(x - a[i].key)}{a[i].key - a[j].key},$$

где i, j – начало и конец интервала; x – искомое значение ключевого поля.

```
int p_dv(tmas a[], int n, int x) {
 int i = 0, j = n-1, m;
while(i < j)
{
 if (a[i].key == a[j].key) // Предотвращение деления на нуль
    if (a[i].key == x) return i;
                  else return -1;
      m=i + (j - i) * (x - a[i]) / (a[j] - a[i]);
if (a[m].key == x) return m;
  else
   if (x > a[m].key) i = m+1; else j = m-1;
}
 return -1;
}
```

Данный поиск быстрее двоичного в 3–4 раза.

	Интерполяционный поиск вблизи ключевого поля может вести себя неустойчиво. Поэтому обычно с использованием интерполяционного поиска делают несколько первых шагов, а затем используют двоичный поиск.

16. Хеширование

16.1. Понятие хеширования

Для решения задачи быстрого поиска используется алгоритм ***хеширования*** (*hashing*), при котором ключи данных записываются в особую хеш-таблицу. Затем при помощи простой функции *i = h(key)* алгоритм хеширования определяет положение искомого элемента в таблице по значению его ключа.

Рассмотрим **пример**.

Имеется массив структур из семи элементов, значения ключей которых находятся в диапазоне 0…15.

```
mas[0].key = 5;
mas[1].key = 15;
mas[2].key = 1;
mas[3].key = 10;
mas[4].key = 8;
mas[5].key = 3;
mas[6].key = 11;
```

Допустим, что надо найти элемент с ключом 3. Для этого метод линейного поиска сделает шесть шагов, а для использования двоичного поиска потребуется предварительная сортировка. Количество шагов зависит от способа сортировки, но затраты в этом случае будут выше, чем при линейном поиске.

Для ускорения поиска создадим новый массив (хеш-таблицу), в котором номер элемента будет равен значению ключа:

```
H[ Mas[i].key ] = Mas[i];
```

Все неиспользуемые элементы массива H имеют значение –1:

```
H[0].key = -1;
H[1] = mas[2];          //  H[1].key = 1
H[2].key = -1;
H[3] = mas[5];          //  H[3].key = 3
H[4].key = -1;
H[5] = mas[0];          //  H[5].key = 5
H[6].key = -1;
H[7].key = -1;
H[8] = mas[4];          //  H[8].key = 8
H[9].key = -1;
H[10] = mas[3];         //  H[10].key = 10
H[11] = mas[6];         //  H[11].key = 11
H[12].key =-1;
H[13].key = -1;
```

```
H[14].key = -1;
H[15] = mas[1];            //  H[15].key = 15
```

При такой организации для нахождения любого элемента достаточно сделать только один шаг `x = H[key]` (сложность алгоритма $O(1)$). Для удаления элемента достаточно поставить значение –1 в соответствующее поле.

Для решения реальных задач такой подход неприемлем, так как размер массива должен быть достаточен для размещения элемента с максимальным ключом, что существенно увеличивает размер хеш-таблицы. Например, для хранения телефонной базы с семизначными номерами необходим массив из 9 999 999 элементов. Для уменьшения размера хеш-таблицы используются различные схемы хеширования.

16.2. Схемы хеширования

При сжатии таблицы несколько различных элементов могут получить одинаковый номер в хеш-таблице, поэтому схема хеширования должна содержать ***алгоритм разрешения конфликтов***, определяющий поведение программы в случае, если новый ключ попадает на уже занятую позицию.

Схема работы ***алгоритма размещения*** элементов в хеш-таблице:

1. По значению ключа вычисляется номер позиции в хеш-таблице $i = key \% m$ (m – количество элементов в хеш-таблице).
2. Если полученная позиция уже занята, то алгоритм разрешения конфликтов находит новую позицию.
3. Если новая позиция тоже занята, повторяется п. 2 до тех пор, пока не будет найдена свободная позиция.

Алгоритм поиска по значению ключа находит позицию искомого элемента в хеш-таблице. Если значение ключа элемента не совпадает с искомым ключом, то осуществляется дальнейший поиск в соответствии с выбранным алгоритмом разрешения конфликтов.

16.3. Хеш-таблица с линейной адресацией

Алгоритм разрешения конфликтов: если найденная для элемента позиция i уже занята, то ищется первая незанятая позиция (начиная с $i + 1$).

Например, имеется следующий массив:

```
Mas[0].key = 5;
Mas[1].key = 15;
Mas[2].key = 3;
Mas[3].key = 10;
Mas[4].key = 125;
Mas[5].key = 333;
Mas[6].key = 11;
Mas[7].key = 437;
```

Данные размещаются в хеш-таблице из 10 элементов. Функция размещения: *i = key %* 10.

Полученная хеш-таблица:

```
H[0] = Mas[3];      // H[0].key = 10;
H[1] = Mas[6];      // H[1].key = 11;
H[2] =  -1
H[3] = Mas[2] ];    // H[3].key = 3;
H[4] = Mas[5] ];    // H[4].key = 333;
H[5] = Mas[0] ];    // H[5].key = 5;
H[6] = Mas[1] ];    // H[6].key = 15;
H[7] = Mas[4] ];    // H[7].key = 125;
H[8] = Mas[7] ];    // H[8].key = 437;
H[9]  =  -1
```

Пример 16.1. Создать хеш-таблицу, использующую алгоритм размещения с линейной адресацией.

```
void sv_add(int key, int m, int* H)
{
   int i = abs(key % m);
     while (H[i] != -1) {
          i++;
          if (i == m) i = 0;
       }
   H[i] = key;
}

int sv_seach(int key, int m, int *H)
{
   int i = abs(key % m);
   while (H[i] != -1)
   {
      if (H[i] == key) return i;
         i++;
         if (i == m) i = 0;
   }
   return -1;
}
```

```
int main()
{
   const int n = 8;          // Число элементов в массиве
   int mas[n] = { 5, 15, 3, 10, 125, 333, 11, 437 };
   const int m = 10;         // Число элементов в хеш-таблице
   int H[m];
   int i;
   for (i = 0; i < m; i++) H[i] = -1; // Все позиции свободны
   for (i = 0; i < n; i++) sv_add(mas[i], m, H);
   // Поиск элемента с ключом 333
   int key = 333;
   int k = sv_seach(key, m, H);
     if (k == -1) cout << "Item not found" << endl;
       else cout << H[k] << endl;
}
```

Достоинство: простой алгоритм размещения и поиска элементов.

Недостатки:

1. Фиксированный размер хеш-таблицы.

2. Сложный алгоритм удаления элемента, так как удаление элемента часто приводит к необходимости перестройки всей таблицы. Для преодоления данного недостатка можно использовать несколько состояний ячейки: «занята», «не занята», «удалена». Если во время поиска алгоритм попадает на ячейку со статусом «удалена», то поиск продолжается далее. При добавлении данных ячейка со статусом «удалена» считается свободной.

3. Если данные в таблице расположены неравномерно, то скорость поиска может быть очень низкой. Для преодоления данного недостатка можно использовать следующую хеш-функцию: $i = (key + r) \% 10$, где r – простое число, сгенерированное датчиком случайных чисел. Для правильной работы алгоритмов поиска и размещения датчик должен всегда устанавливаться в одинаковое начальное положение.

16.4. Хеш-таблицы с квадратичной и произвольной адресацией

В отличии от метода с линейной адресацией, в *методе с квадратичной адресацией* поиск свободной ячейки ведется не последовательно (i++), а по формуле $i = i + p^2$ (p – номер попытки).

В *методе с произвольной адресацией* незанятая позиция ищется по формуле: $i \mathrel{+}= i + r[p]$ (r – заранее сгенерированный массив случайных чисел; p – номер попытки).

По сравнению с линейной адресацией данные методы дают более равномерное распределение данных в таблице, однако работают несколько медленнее.

16.5. Хеш-таблица с двойным хешированием

Алгоритм метода:

1. Найти позицию элемента в хеш-таблице по формуле $i = key \% m$.
2. Если ячейка с номером i свободна, то перейти к п. 6.
3. Вычислить $c = 1 + (key \% (m - 2))$.
4. Найти позицию элемента в хеш-таблице по формуле $i = i - c$ (если $i < 0$, то $i = i + m$).
5. Если ячейка с найденным номером i занята, то перейти к п. 4.
6. Вставить элемент в найденную позицию.

По сравнению с предыдущими данный метод из-за наличия независимых друг от друга цепочек поиска свободной ячейки дает более равномерное распределение данных в хеш-таблице. Усложнение алгоритма приводит к снижению скорости его работы.

16.6. Хеш-таблица на основе связанных списков

Одним из наиболее эффективных методов разрешения конфликтов состоит в том, что элементы, попадающие на одну и ту же позицию, размещаются в связанных списках (см. разд. 17). Например, имеется следующий массив:

```
Mas[0].key = 5;
Mas[1].key = 15;
Mas[2].key = 3;
Mas[3].key = 10;
Mas[4].key = 125;
Mas[5].key = 333;
Mas[6].key = 11;
Mas[7].key = 437;
```

Данные размещаются в хеш-таблице из 10 элементов. Функция размещения: $i = key \%$ 10. Каждый элемент таблицы является указателем на вершину стека.

Полученная хеш-таблица:

```
H[0] ← Mas[3]
H[1] ← Mas[6]
H[2] ← nullptr
H[3] ← Mas[2] ← Mas[5]
H[4] ← nullptr
H[5] ← Mas[0] ← Mas[1] ← Mas[4]
H[6] ← nullptr
H[7] ← Mas[7]
H[8] ← nullptr
H[9] ← nullptr
```

Пример 16.2. Создать хеш-таблицу, использующую алгоритм размещения на основе связанных списков.

```
struct TNode  // Описание элемента стека
{
   int inf;    // Информационная часть структуры
   TNode* a;   // Адресная часть структуры
};

TNode** sv_create(int m)
{
    TNode** H = new  TNode * [m];
   for (int i = 0; i < m; i++) H[i] = nullptr;
   return H;
}

void sv_add(int inf, int m,  TNode** H)
{
    TNode* spt = new TNode;
   spt->inf = inf;
   int i = abs(inf % m);
   if (H[i]) { spt->a = H[i];  H[i] = spt; }
   else  { H[i] = spt; spt->a = nullptr; }
}

TNode* sv_seach(int inf, int m,  TNode** H)
{
   int i = abs(inf % m);
    TNode* spt = H[i];
   while (spt)  {
      if (spt->inf == inf) return spt;
      spt = spt->a;
   }
   return nullptr;
}

void sv_delete(int m,  TNode** H)
{
    TNode* spt, * sp;
   for (int i = 0; i < m; i++)
```

```
    {
        sp = H[i];
       while (sp)  {
          spt = sp;
          sp = sp->a;
          delete spt;
       }
    }
    delete[]H;
}

int main()
{
    int n = 8; // Число элементов  в массиве
    int mas[] = { 5, 15, 3, 10, 125, 333, 11, 437 };
    int m = 10; // Число элементов  в хеш-таблице
      TNode** H = sv_create(m);
    for (int i = 0; i < n; i++) sv_add(mas[i], m, H);
     int key = 333;
     TNode* p = sv_seach(key, m, H);
     if (!p)  cout << "Item not found" << endl;
     else cout << p->inf << endl;
    sv_delete(m, H);
}
```

Достоинства:

1. Достаточно простой алгоритм вставки и поиска элементов.
2. Связанная таблица не может быть переполнена.

Недостаток: плохая работа с неравномерно размещенными данными. Для преодоления этого недостатка используется методика, рассмотренная в подразд. 16.3.

16.7. Метод блоков

Используется массив одномерных массивов одинакового размера (блоков).

Вначале находится номер блока, в который помещается элемент. Если блок переполнен, то элемент помещается в специальный блок переполнения. Поиск ведется в найденном блоке и в блоке переполнения. Метод хорошо зарекомендовал себя при хранении хеш-таблицы на файле, так как запись и чтение из файла можно осуществлять поблочно, что быстрее поэлементной работы.

17. Динамические структуры данных

17.1. Понятие списка, стека и очереди

Объект данных считается динамической структурой, если его размер, взаимное расположение и взаимосвязи его элементов изменяются в процессе выполнения программы.

Список (list) – последовательность однотипных данных, работа с которыми ведется в оперативной памяти. В процессе работы список может изменять свой размер. Наибольшее распространение получили две формы работы со списком – очередь и стек.

Стек (stek) – список с одной точкой входа. Данные добавляются в список и удаляются из него только с одной стороны последовательности (вершины стека). Таким образом реализуется принцип «последний пришел – первый вышел».

Очередь (turn) – список с одной или двумя точками входа. Данные добавляются в конец очереди, а извлекаются из начала очереди. Таким образом реализуется принцип «первый пришел – первый вышел».

Для работы со списками предусмотрен специальный *рекурсивный тип данных*, в описании которого содержится указатель на аналогичную этому типу структуру.

Чаще всего используется следующая конструкция рекурсивного типа данных:

```
struct TInf
{
  // Набор полей структуры
};

struct TNode
{
  TInf inf;    // Информационная часть структуры
  TNode *a;   // Адресная часть структуры
};
```

Для упрощения рассмотрения в дальнейшем будет использоваться структура следующего вида:

```
struct TNode
{
  int  inf;          // Информационная часть структуры
  TNode *a;      // Адресная часть структуры
} ;
```

Однонаправленные связанные списки организуются следующим образом: память для каждого элемента выделяется отдельно (по мере необходимости). В информационную часть помещаются нужные данные, а в адресную часть – адрес предыдущей или последующей структуры.

	Адресация, при которой каждый элемент, кроме информационной части, хранит адрес другого элемента последовательности, называется ***косвенной адресацией***. В отличие от адресации по индексу косвенная адресация менее наглядна, однако обладает большей гибкостью.

17.2. Работа со стеком

Для работы со стеком достаточно знать указатель на вершину стека. Для перемещения по стеку необходимо последовательно переходить от одной ячейки к другой:

```
spt = top;       // Установка текущего указателя в начало стека
spt = spt->a;    // Перемещение к следующему элементу
spt=spt->a->a;   // Перемещение на два элемента
```

Пример 17.1. Работа со стеком.

```
#include <iostream>
using namespace std;

struct TNode  // Описание элемента стека
{
  int inf;    // Информационная часть структуры
  TNode* a;   // Адресная часть структуры
};

struct stack // Структура для работы со стеком
{
  TNode* top=nullptr;  // Указатель на вершину стека
  int size=0;          // Количество элементов стека

    bool empty()  {     // Проверка наличия элементов в стеке
       if (top) return false;
       else return true;
    }

    void push(int inf) {   // Добавление элемента в стек
       TNode* spt = new TNode;
```

```
    spt->inf = inf;
    spt->a = top;
    top = spt;
    size++;
}

void pop() {   //  Удаление элемента из стека
     TNode *spt  = top;
     top = top->a;
     delete spt;
     size--;
}

 void print() {     // Вывод содержимого стека на экран
   TNode* spt = top;
   while (spt != nullptr)
   {
       cout << spt->inf << " ";
       spt = spt->a;
   }
   }

  TNode* search(int x) {      // Поиск элемента с заданным ключом
   if (!top) return nullptr;
      TNode* spt = top;
   while (spt->inf != x && spt->a != nullptr) spt = spt->a;
    if (spt->inf == x) return spt;
    else return nullptr;
  }

 // Поиск предыдущего элемента (исключая первый)
  TNode* searchp(int x) {
    if (!top || !top->a) return nullptr;
      TNode* spt = top;
      while (spt->a->inf != x && spt->a->a != nullptr) spt = spt->a;
      if (spt->a->inf == x) return spt;
      return nullptr;
  }
```

```
    // Удаление элемента с заданным ключем
      void del(int x) {
        if (!top) return;
        if (top->inf == x) pop();
        TNode* spt, * spp;
         spp = searchp(x);
         spt = spp->a;
        spp->a = spp->a->a;
      delete spt;
      }

      // Обмен следующих за указанным элементов
      void exchange(TNode* sp) {
         TNode* spt;
         spt = sp->a->a;
         sp->a->a = spt->a;
         spt->a = sp->a;
         sp->a = spt;
      }
} ;

int main() {
  stack s;
   s.push(4);
   s.push(2);
   s.push(1);
   s.push(6);
   s.push(9);
   s.print(); // Выводит: 9 6 1 2 4
   TNode* d1 = s.search(1);  cout << d1->inf; // Выводит: 1
   TNode* d2 = s.searchp(1); cout << d2->inf; // Выводит: 2
    s.exchange (d2);
    s.print(); // Выводит: 9 6 2 1 4
    s.del(6);
    s.print(); // Выводит: 9 2 1 4
 while (!s.empty()) s.pop();
 if (s.empty()) cout << "Stack is empty";
}
```

17.3. Работа со однонаправленной очередью

Работа с однонаправленной очередью аналогична работе со стеком за исключением того, что данные помещаются в конец списка, а извлекаются из начала.

Пример 17.2. Работа с однонаправленной очередью.

```
struct TNode {
  int inf;    // Информационная часть структуры
  TNode* a;   // Адресная часть структуры
};

struct queue {
    TNode* front = nullptr;  // Указатель на начало  очереди
    TNode* back = nullptr;  // Указатель на конец очереди

    bool empty() {    // Проверка наличия элементов в очереди
        if (front) return false;
        else return true;
    }

    void push(int inf) {     // Добавление элемента в очередь
        TNode* spt = new TNode;
        spt->inf = inf;
        spt->a = nullptr;
        // Если элементы отсутствуют
        if (!front) front = back = spt;
        else {
            back->a = spt;
            back = spt;
        }
    }

    void pop() {    //  Удаление элемента из очереди
        TNode* spt = front;
        front = front->a;
        delete spt;
        if (!front) back = nullptr;
    }
```

```
    void print() {   // Вывод элементов очереди на экран
        TNode* spt = front;
        while (spt != nullptr) {
            cout << spt->inf << " ";
            spt = spt->a;
        }
    }
};

int main() {
    queue s;
   s.push(4);
   s.push(2);
   s.push(1);
   s.push(6);
   s.push(9);
   s.print(); // Выводит: 4 2 1 6 9
 while (!s.empty()) s.pop();
 if (s.empty()) cout << "Queue is empty";
}
```

17.4. Работа с двусвязанными списками

Двусвязанный список состоит из структур, содержащих поля для хранения адресов предыдущего и последующего элементов. Такая организация позволяет осуществлять перемещение по списку в любом направлении.

Пример 17.3. Работа с двусвязанным списком.

```
struct TNode {
  int inf;    // Информационная часть структуры
  TNode* left;   // Адресная часть
  TNode* right;  // Адресная часть
};

struct list {
    TNode* front = nullptr;  // Указатель на начало  очереди
    TNode* back = nullptr;  // Указатель на конец очереди
```

```
bool empty() {   // Проверка наличия элементов в очереди
    if (front) return false;
    else return true;
}

void push(int inf) {     // Добавление элемента в очередь
  TNode* spt = new TNode;
        spt->inf = inf;
        spt->right = nullptr;
     if (!front) {
       spt->left = nullptr;
       front = back = spt;
       return;
       }
     back->right = spt;
     spt->left = back;
     back = spt;
 }

void pop() {     //  Удаление элемента из очереди
    TNode* spt = front;
    front = front->right;
    delete spt;
    if (!front) back = nullptr;
    else
    front->left = nullptr;
}

void print() {   // Вывод элементов очереди на экран
    TNode* spt = front;
    while (spt != nullptr) {
        cout << spt->inf << " ";
        spt = spt->right;
    }
}

TNode* search(int x) {     // Поиск элемента с заданным ключом
    if (!front) return nullptr;
```

```
        TNode* spt = front;
        while (spt->inf != x && spt->right != nullptr) spt = spt->right;
        if (spt->inf == x) return spt;
        else return nullptr;
    }

    void del(int x) {     //  Удаление элемента с заданным ключом
        TNode* spt = search(x);
        if (!spt) return;
        if (front == back) {
          front = nullptr;
          back = nullptr;  }
         else
            if (!spt->left) {
                front = spt->right;
                front->left = nullptr;
        }
            else
                if (!spt->right) {
                    back = spt->left;
                    back->right = nullptr;  }
                else {
                    spt->right->left = spt->left;
                    spt->left->right = spt->right;
                }
        delete spt;
    }

    // Добавление элемента после заданного
    void pushleft(TNode* spp, int inf) {
        TNode* spt = new TNode;
        spt->inf = inf;
        spt->right = spp->right;
        spt->left = spp;
        spp->right = spt;
        if (spt->right) spt->right->left = spt;
    }
};
```

```
int main() {
    list s;
   s.push(4);
   s.push(2);
   s.push(1);
   s.push(6);
   s.push(9);
   s.print();          // Выводит: 4 2 1 6 9
   s.del(6);
   s.print();          // Выводит: 4 2 1 9
   s.pushleft(s.search(2), 7);
   s.print();          // Выводит: 4 2 7 1 9
   while (!s.empty()) s.pop();
}
```

17.5. Работа с двусвязанными циклическими списками

Циклические списки – одно- или двунаправленные очереди, в которых последний элемент указывает на начало очереди (рис. 16.1). Понятия начала и конца очереди здесь не имеют смысла, достаточно знать адрес любого элемента очереди.

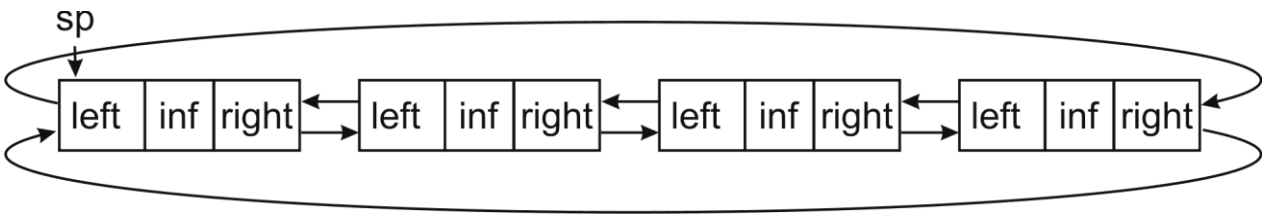

Рис. 16.1

18. Нелинейные списки

18.1. Древовидные структуры данных

Рассмотрим древовидную структуру данных (рис. 18.1).

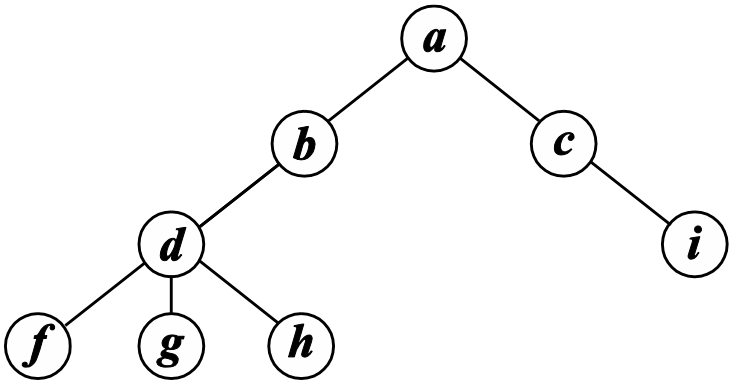

Рис. 18.1

Все данные называются ***узлами***.

Связи между узлами называется ***ветвями***.

Самый верхний узел – ***корень*** дерева (*a*).

Узлы, из которых не выходят связи, – ***листья*** дерева (*f, g, h, i*).

Узел, находящийся непосредственно над другим, называется ***родительским узлом*** (*для узла d узел b является родительским*). Узел, находящийся непосредственно ниже, называется ***дочерним*** (*для узла b узел d является дочерним*).

Все узлы, находящиеся выше рассматриваемого, являются его ***предками*** (*для узла d предки b и a*), а все узлы, находящиеся ниже, – ***потомками*** (*для узла b потомки – d, f, g, h*).

Узлы, имеющие одного и того же родителя, называются ***сестринскими*** (*f, g, h*).

Узел, не являющийся листом, называется ***внутренним*** (*b, или d, или c, или a*).

Порядок узла (или ***степень узла***) – количество дочерних узлов (*для узла b порядок 1, для узла d порядок 3*).

Степень дерева – это максимальный порядок его узлов (*рассматриваемое дерево имеет третий порядок*). Дерево второй степени называется ***бинарным*** или ***двоичным***.

Глубина узла – число предков плюс единица (*например, для узла d глубина равна 3*).

Глубина дерева – наибольшая глубина всех узлов (*для данного дерева – 4*).

18.2. Использование древовидных структур

Для работы с древовидными структурами используется следующая конструкция рекурсивного типа:

```
struct ttree
 {
   tinf inf;
   ttree *a1;
   ttree *a2;
…
   ttree *an;
 } *proot, *p;
```

Рассмотрим размещение в памяти структуры, приведенной на рис. 18.1:

```
ttree *proot, *p;
 proot = new ttree;
 proot->inf = 'a';     proot->a2 = nullptr;
 p = proot;
 p->a1 = new ttree;
        p = p->a1;
        p->inf = 'b';              p->a2 = nullptr;                p->a3 = nullptr;
p->a1 = new ttree;
        p = p->a1;
         p->inf = 'd';
p->a1 = new ttree;
         p->a1->inf = 'f';       p->a1->a1 = nullptr;          p->a1->a2 = nullptr;
        p->a1->a3 = nullptr;
p->a2 = new ttree;
         p->a2->inf = 'g';     p->a2->a1 = nullptr;          p->a2->a2 = nullptr;
        p->a2->a3 = nullptr;
p->a3 = new ttree;
         p->a3->inf = 'h';     p->a3->a1 = nullptr;          p->a3->a2 = nullptr;
        p->a3->a3 = nullptr;
         p = proot;
p->a3 = new ttree;
         p = p->a3;
         p->inf = 'c';              p->a1 = nullptr;       p->a2 = nullptr;
p->a3 = new ttree;
         p = p->a3;
         p->inf = 'i';     p->a1 = nullptr;      p->a2 = nullptr;     p->a3 = nullptr;
```

Как видно из приведенного выше фрагмента программы, непосредственное заполнение даже небольшого дерева требует довольно громоздкой последовательности команд. Поэтому для работы с деревьями используют набор специфических алгоритмов.

Обходом дерева называется последовательное обращение ко всем его узлам. Следующая рекурсивная процедура осуществляет такой обход с выводом каждого узла:

```
void obh(ttree *p)  // Обход всего дерева
{
 if (p == nullptr) return;
 // Вывод при прямом обходе
  obh (p->a1);
  obh (p->a2);

  obh tree(p->an);
// Вывод при обратном обходе
}
```

Прямой обход: ***a b d f g h c i***.
Обратный обход: ***f g h d b i c a***.

18.3. Двоичное дерево поиска

Если ключевые поля в дереве расположены таким образом, что для любого узла значение ключа у левого преемника меньше, чем у правого, то такое дерево называется ***двоичным деревом поиска*** (*Binary Search Tree*). Предположим, что имеется набор данных, упорядоченных по ключу: *key*: 1, 5, 6, 9, 14, 21, 28, 32, 41. Для таких данных двоичное дерево поиска выглядит следующим образом (рис. 18.2).

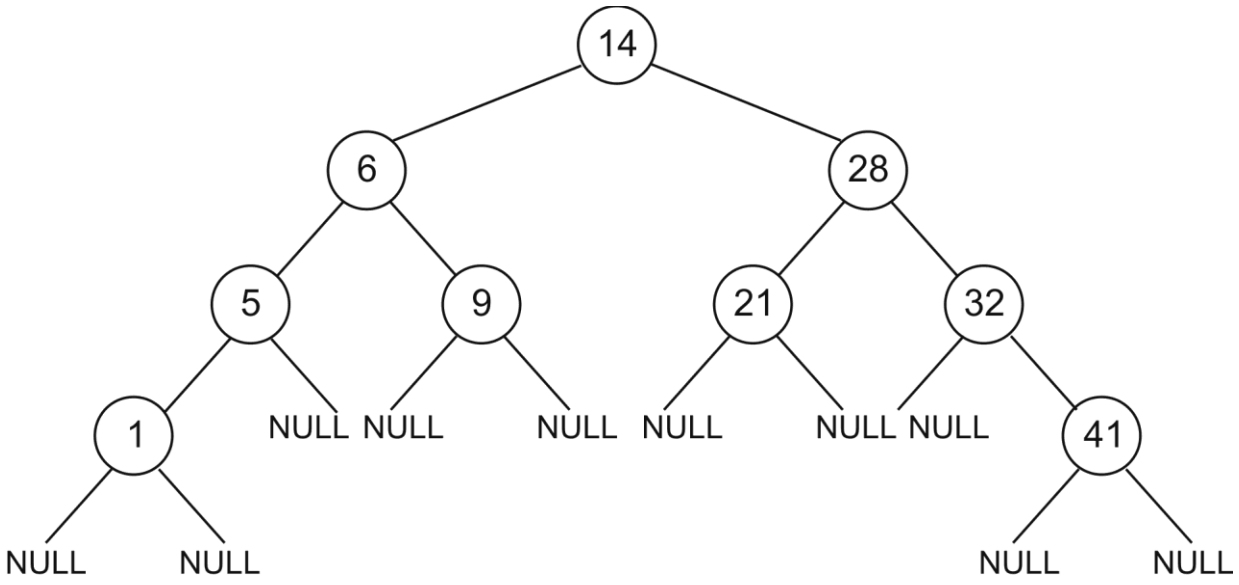

Рис. 18.2

Дерево, у которого узлы, имеющие только одну дочь, располагаются не выше двух последних уровней, называется ***сбалансированным деревом*** или ***AVL-деревом*** (Г. М. **А**дельсон-**В**ельский и Э. М. **Л**андис). Эффективность поиска информации в такой динамической структуре данных сравнима с эффективностью двоичного поиска в массиве ($O(\log^2 n)$).

AVL-дерево считается сбалансированным, если для любого узла глубина левого поддерева отличается от глубины правого поддерева не более чем на единицу. Разница глубин называется ***коэффициентом сбалансированности*** (*balance factor*). Дерево считается сбалансированным, если коэффициент равен одному из трех значений: –1, 0 и 1.

После добавления нового элемента дерево может стать разбалансированным. Существует четыре варианта разбалансировки дерева (рис. 18.3).

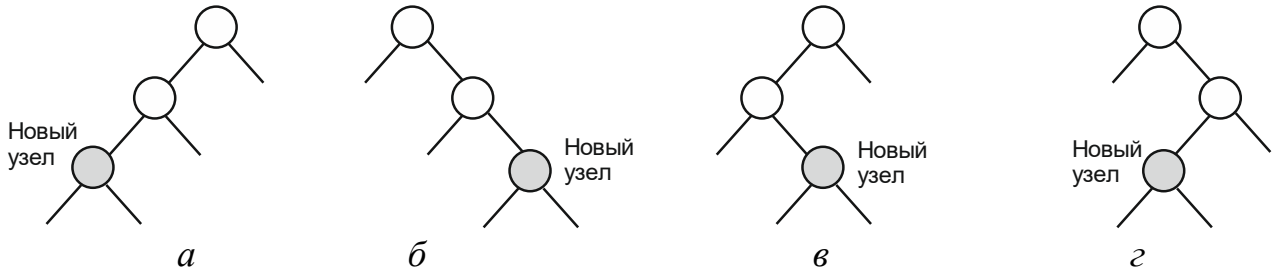

Рис. 18.3

Балансировка выполняется с помощью действий, называемых *вращениями узлов*.

В случаях *а* и *б*, с нарушенной балансировкой, достаточно одного вращения (малого левого или малого правого).

Малое правое вращение осуществляется так, как показано на рис. 18.4.

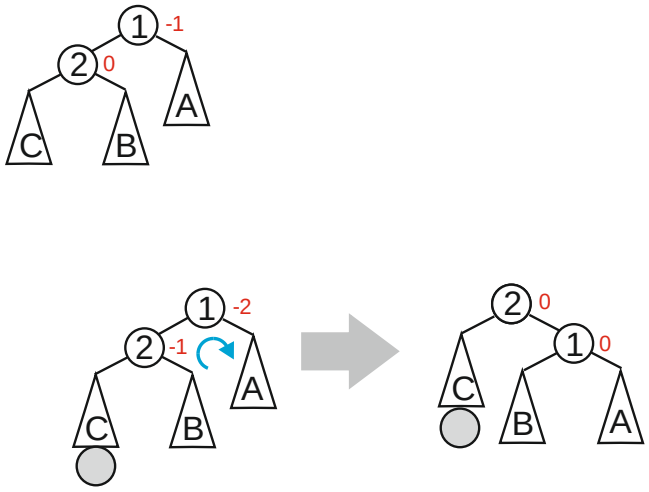

Рис. 18.4

Малое левое вращение осуществляется аналогично, но в другую сторону.

В случаях *в* и *г*, с нарушенной балансировкой, требуется двойное левое-правое или правое-левое вращение. Такие вращения называются большим правым и большим левым.

Большое правое вращение осуществляется так, как показано на рис. 18.5.

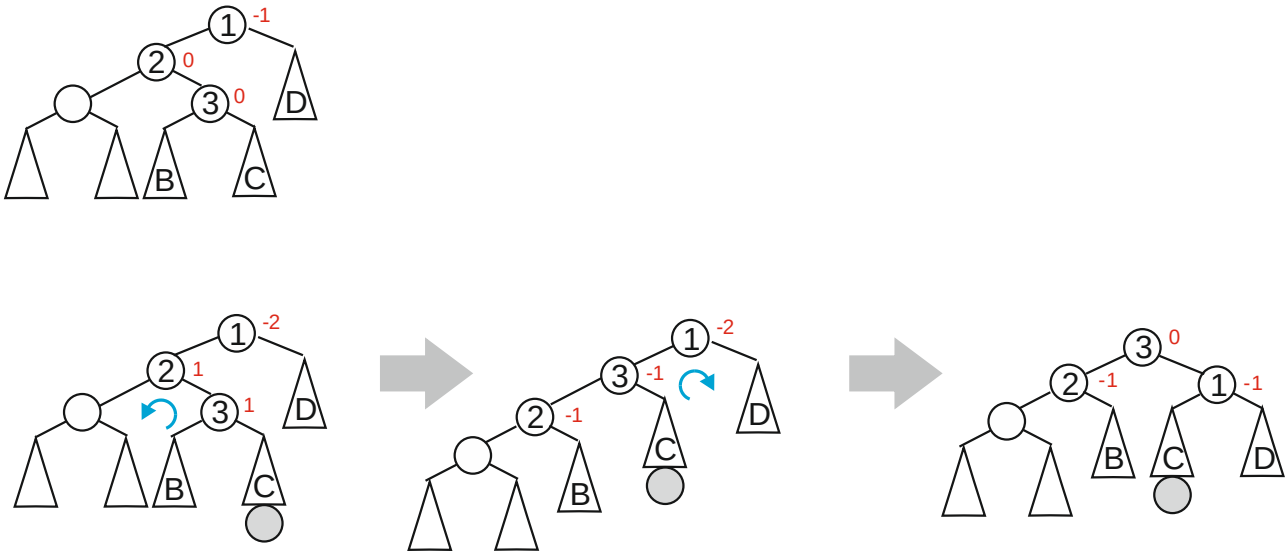

Рис. 18.5

Большое левое вращение осуществляется аналогично, но в другую сторону.

Алгоритм балансировки при удалении такой же, как и при добавлении узла.

Удаление узла дерева, осуществляется с учетом возможных вариантов его размещения:

1. Если удаляется узел, не имеющий потомков (лист), то дополнительных изменением в дерево вносить не требуется (рис. 18.6).

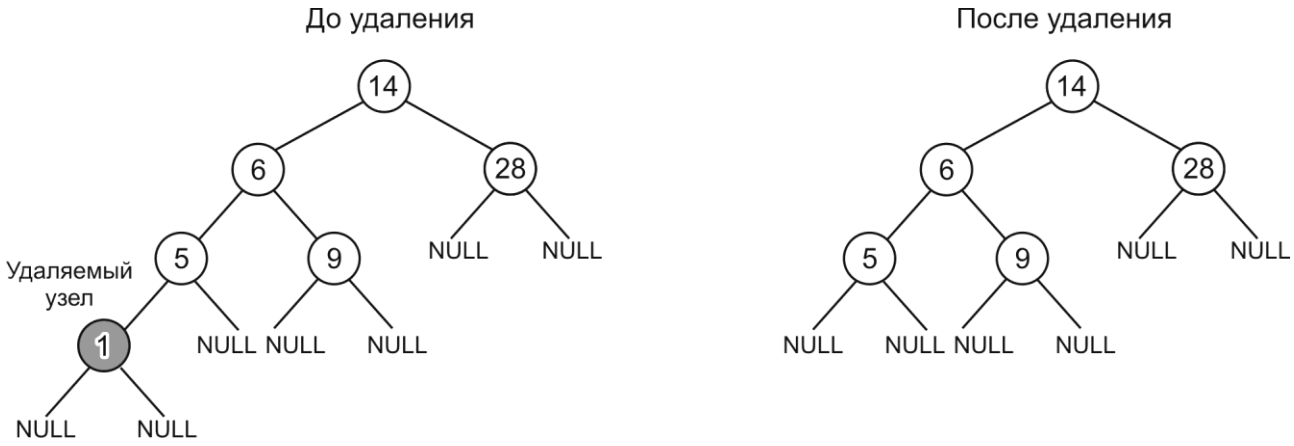

Рис. 18.6

2. Если удаляется узел, имеющий только один дочерний узел (слева или справа). Удаляемый узел заменяется этим дочерним узлом (рис. 18.7).

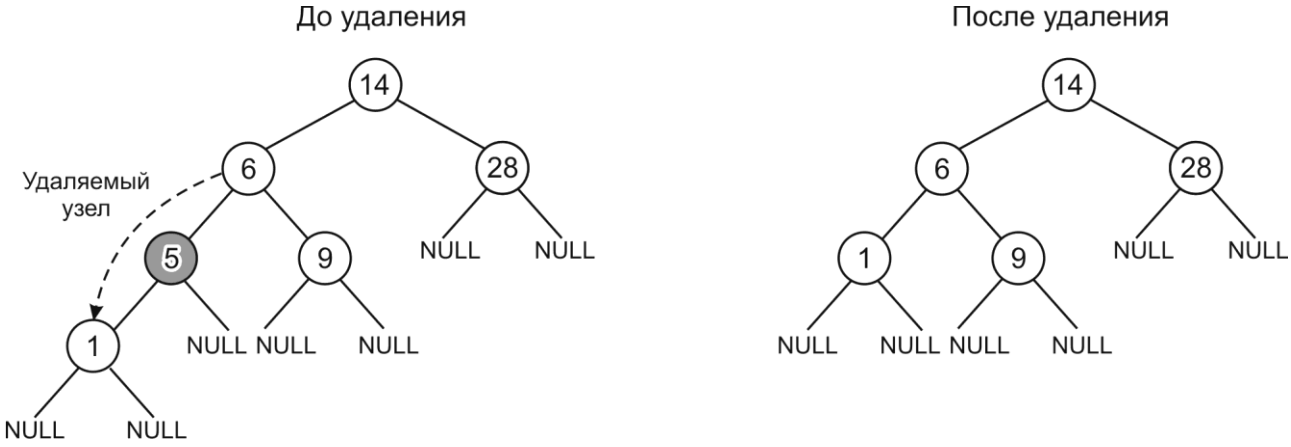

Рис. 18.7

3. Если удаляется узел, имеющий два дочерних узла, то удаляемый узел заменяется узлом, имеющим наибольший ключ в левом поддереве либо наименьший ключ в правом поддереве (рис. 18.8).

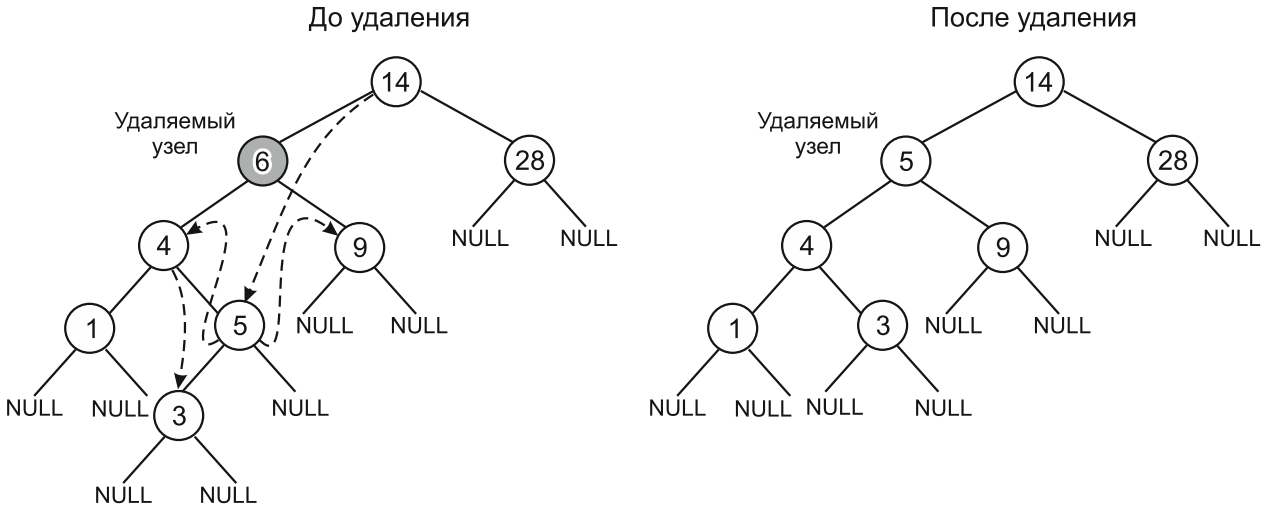

Рис. 18.8

Пример 18.1. Работа с двоичным деревом поиска.

```
struct TNode // Узел
{
        int inf;   // Информационная часть
        TNode* left = nullptr;  // Адресная часть
        TNode* right = nullptr; // Адресная часть
        int depth = 1; // Глубина поддерева
};
```

```
int dph(const TNode* p)  // Чтение глубины поддерева
{
if (!p) return 0; // Если поддерево отсутствует
 return p->depth;
}

int bfactor(const TNode* p) // Вычисление разбалансированности
{
return dph(p->right) - dph(p->left);
}

void maxdepth(TNode* p)  // Вычисление глубины поддерева
{
  int l = dph(p->left);
  int r = dph(p->right);
if (l > r) p->depth = l + 1;
      else p->depth = r + 1;
}

TNode* rotateright(TNode* root) // Правый поворот вокруг p
{
        TNode* p = root->left;
        root->left = p->right;
        p->right = root;
         maxdepth(root);
         maxdepth(p);
        return p;
}

TNode* rotateleft(TNode* root) // Левый поворот вокруг q
{
        TNode* p = root->right;
        root->right = p->left;
        p->left = root;
         maxdepth(root);
         maxdepth(p);
        return p;
}
```

```
TNode* AVL(TNode* p) // Балансировка узла p
{
	maxdepth(p); // Вычисление глубины поддерева
	if (bfactor(p) == 2) // Если требуется балансировка
	{
	 if (bfactor(p->right) < 0)   p->right = rotateright(p->right);
		return rotateleft(p);
	}
	if (bfactor(p) == -2) // Если требуется балансировка
	{
	 if (bfactor(p->left) > 0)     p->left = rotateleft(p->left);
		return rotateright(p);
	}
	return p; // Балансировка не требуется
}

TNode* push(TNode* p, int key) // Добавление нового элемента
{
	if (!p) {
		p = new TNode;
		p->inf = key;
		return p;
	}
	if (key < p->inf) p->left = push(p->left, key);
		else p->right = push(p->right, key);
	return AVL(p);
}

TNode* find_max(TNode* p)  // Поиск узла с минимальным ключом
{
	TNode* q = p;
	while (q->right != nullptr) q = q->right;
	return q;
}

TNode* find_min(TNode* p) // Поиск узла с минимальным ключом
{
	TNode* q = p;
```

```
        while (q->left != nullptr) q = q->left;
        return q;
}

// Отключение узла с максимальным ключом из поддерева
TNode* removemin(TNode* p)
{
        if (!p->right) return p->left;
        p->right = removemin(p->right);
        return AVL(p);
}

TNode* pop(TNode* p, int key) // Удаление узла с ключом k
{
        if (!p) return nullptr;    // Ключ не обнаружен
        if (key < p->inf) p->left = pop(p->left, key);
         else
        if (key > p->inf) p->right = pop(p->right, key);
        else {          // Ключ найден
                TNode* q = p->left;
                TNode* r = p->right;
                delete p;
                if (!r) return q;
                TNode* rep = find_max(q);
                rep->left = removemin(q);
                rep->right = r;
                return AVL(rep);
        }
        return AVL(p);
}

TNode* pop(TNode* p) // Удаление всего дерева
{
        if (!p) return nullptr;
        pop(p->left);
        pop(p->right);
        delete(p);
}
```

```
TNode* search(TNode* p, int key) // Поиск по ключу
{
	TNode* q = p;
	while (q)
	{
		if (q->inf == key) return q;
		if (key < q->inf) q = q->left;
		else q = q->right;
	}
	return q;
}

void prints(TNode* p) // Симметричный обход дерева
{
	if (!p) return;
	prints(p->left);
	cout << p->inf << " ";
	prints(p->right);
}

bool empty(TNode* p) {  // Проверка наличия элементов в дереве
	if (p) return false;
	else return true;
}


void print(TNode *p, int r = 0)  // Вывод дерева на экран
{
	if (!p) return;
cout << setiosflags(ios::right);
 print(p->right, r + 5);
   cout << setw(r) << p->inf << endl;
 print(p->left, r + 5);
}

int main() {
	TNode* root = nullptr; const int n = 10;
	for (int i = 1; i < n; i++) root = push(root, i*2);
```

```
    print(root);  // Выводит дерево
    root = pop(root, 4); // Удаляет узел с ключом 4
    root = pop(root, 8); // Удаляет узел с ключом 8
    print(root); // Выводит дерево
    TNode* q = search(root, 12); // Поиск элемента с ключом 12
    if (q) cout <<"Element = " << q->inf << endl;
    else cout << "Not found" << endl;
    root = pop(root); // Удаление дерева
    if (empty(root)) cout << "Tree removed";
    else  cout << "Tree exists";
}
```

19. Синтаксический анализ арифметических выражений

Выражения в математике обычно записываются в *инфиксной* форме, например (*a* + *b*) * (*k* – *d*). Однако для компьютерной обработки такая форма не удобна, так как при вычислениях необходимо учитывать приоритет операций, который к тому же может быть изменен с помощью скобок.

Имеются эквивалентные формы записи:

– префиксная (знак операции ставится перед операндами, например **+ab-kd*);

– постфиксная (знак операции ставится после операндов, например *ab+kd-**).

Наиболее удобной для программирования является использование *постфиксной* формы представления арифметических выражений, предложенной польским математиком Я. Лукашевичем. Такая форма записи арифметических выражений получила название ***обратной польской записи*** (ОПЗ). Удобство использования ОПЗ состоит в том, что при записи выражений скобки не нужны, а полученная последовательность операндов и операций удобна для расшифровки.

19.1. Алгоритм преобразования выражения в форму ОПЗ

Алгоритм *преобразования выражений из инфиксной формы в форму ОПЗ* (алгоритм Дейкстры) заключается в следующем. Строка последовательно просматривается слева направо. Имеются выходная строка и стек для временного хранения символов. Операнды добавляются в выходную строку, а остальные символы обрабатываются следующим образом:

1. Если текущий символ – знак операции, а стек пуст, то операция записывается в стек.

2. Если текущий символ − открывающая скобка, то она записывается в стек.

3. Если текущий символ − закрывающая скобка, из стека извлекаются в выходную строку все элементы до открывающей скобки. Открывающая скобка удаляется из стека, но в выходную строку не добавляется.

4. Если текущий символ – знак операции и стек не пуст, то из стека в выходную строку переносятся все операции с большим или равным приоритетом. Знак текущей операции помещается в стек.

5. После просмотра всех символов в строке знаки операций, оставшиеся в стеке, помещаются в выходную строку.

Алгоритм *вычисления выражения, записанного в форме ОПЗ,* основан на использовании стека. При просмотре выражения слева направо значения операндов заносятся в стек. Если найден знак операции, то из стека извлекаются два операнда, к которым применяется найденная операция. Результат заносится в стек. После выполнения всех операций в стеке остается одно значение − результат вычисления арифметического выражения.

Пример 19.1. Программа для вычисления арифметических выражений.

```
int Priority(char ch) { // Вычисление приоритета операции
    switch (ch) {
    case '+' : case '-' : return 1;
    case '*' : case '/' : return 2;
    case '^' : return 3;
     default : return 4;  // Операнд
    }
}

int Op(int ch) {  // Определение типа символа
    if (isdigit(ch)) return 1; // Если цифра
    else
        if (isalpha(ch)) return 2; // Если буква
        else
        if ((ch == '(' || ch == ')' || ch == '+' || ch == '-' ||
            ch == '*' || ch == '/' || ch == '^')) return 3;  // Если знак операции
    return 4;
}

struct TTree // Объявление дерева
{
    char key[9];
    TTree* left = nullptr;
    TTree* right = nullptr;
};

struct TNode  // Объявление стека
{
    double inf;
    TNode* a = nullptr;
};

struct TSA // Синтаксический анализ
{
char Mstr[30][10];  // Массив для хранения операторов и операндов
char st[30];  // Арифметическое выражение
TTree* Tree = nullptr;
```

```
TNode* top = nullptr;
int nel, n;

void push(double inf) {  // Добавление элемента в стек
    TNode* spt = new TNode;
    spt->inf = inf;
    spt->a = top;
    top = spt;
}

double pop() { // Извлечение элемента из стека
    double inf = top->inf;
    TNode* spt = top;
    top = top->a;
    delete spt;
    return inf;
}

void inputstr() { // Ввод уравнения
    cout << "Введите уравнение: ";
    gets_s(st, 30);     n = strlen(st);  // Длина строки
}

TTree* MakeTree(int n, int m) { // Построение дерева
        // Если в строке остался один операнд
    TTree* nl = new TTree;
    if (n == m) {
        strcpy_s(nl->key, _countof(nl->key), Mstr[n]);
        return nl;
    }
    // Если в строке имеется несколько операндов
    int nm=0, prt, priormin = 4, sk = 0;
    for (int i = n; i <= m; i++) {
        char ch = Mstr[i][0];  // Первый символ операнда
        if (ch == '(') sk++; // Если открывающая скобка
        else
            if (ch == ')') sk--; // Если закрывающая скобка
            else
                if (sk == 0) {
```

```
            prt = Priority(ch); // Приоритет операции (4 - не операция)
        // Определ. операции с минимальным приоритетом
                if (prt <= priormin)
                {
                    priormin = prt;
                    nm = i;
                }
}   }
    if (priormin == 4 && Mstr[n][0] == '(' && Mstr[m][0] == ')')
     return MakeTree(n + 1, m - 1); // Обработка выражения в скобках
    strcpy_s(nl->key, _countof(nl->key), Mstr[nm]);
  nl->left = MakeTree(n, nm - 1);
  nl->right = MakeTree(nm + 1, m);
    return nl;
}

void rasAV(TTree* nl) { // Расчет арифметического уравнения
    if (!nl) return; // Выход из рекурсии
      rasAV(nl->left);
      rasAV(nl->right);
    if (Op(nl->key[0]) != 3) { // Если операнд
        cout << "Введите значение переменной:" << nl->key << endl;
        double p;   cin >> p;
        push(p);
    }
    else {   // Если операция
       double x1 = pop();
       double x2 = pop();
       switch (nl->key[0]) {
       case '+':  push(x1 + x2); break;
       case '-':  push(x2 - x1); break;
       case '*':  push(x2 * x1); break;
       case '/':  push(x2 / x1); break;
       case '^':  push(pow(x2, x1)); break;
       };
    }
    delete(nl);
}
```

```
void MakeTerm()  // Получение массива операндов и операций
{
    nel = 0;
    for (int i = 0; i < n; ) {
        Mstr[nel][0] = '\0';
        int nc = 0;
        switch (Op(st[i])) {
        case 1: while (isdigit(st[i]) || st[i] == '.')
                            Mstr[nel][nc] = st[i++];
                            Mstr[nel][++nc] = '\0';
                             break;
        case 2: while (isalpha(st[i]) || isdigit(st[i]))
                                  { Mstr[nel][nc] = st[i++];
                                     Mstr[nel][++nc] = '\0'; }
                                  strcpy_s(Mstr[nel], 10, Mstr[nel]);
                             break;
        case 3: Mstr[nel][nc] = st[i++];
                      Mstr[nel][++nc] = '\0';
                    break;
        default: i++; nel--;
        }
        nel++;
    }
}

void print(TTree* p, int r = 0) {
     if (!p) return;
    cout << setiosflags(ios::right);
    print(p->right, r + 5);
    cout << setw(r) << p->key << endl;
    print(p->left, r + 5);
}

void outd() {
    for (int i = 0; i < nel; i++)
        cout << Mstr[i] << endl;
}
};  // Конец TSA
```

```
int main() {
    system("chcp 1251");
    TSA av;
    av.inputstr();
    av.MakeTerm();
    av.outd();
    av.Tree = av.MakeTree(0,av.nel-1);
    cout << endl << "Дерево : " << endl;
    av.print(av.Tree);
    av.rasAV(av.Tree);  // Расчет
    cout << endl << "Результат = " << av.pop();
}
```

ЛАБОРАТОРНЫЙ ПРАКТИКУМ

Лабораторная работа 1. Программирование линейных алгоритмов

Вычислить значение выражения при заданных исходных данных. Сравнить полученное значение с указанным правильным результатом.

1. $s = \dfrac{2\cos\left(x - \dfrac{2}{3}\right)}{\dfrac{1}{2} + \sin^2 y}\left(1 + \dfrac{z^2}{3 - z^2/5}\right)$

при $x = 14.26$; $y = -1.22$; $z = 3.5 \cdot 10^{-2}$. Ответ: $s = 0.749155$.

2. $s = \dfrac{\sqrt[3]{9 + (x-y)^2}}{x^2 + y^2 + 2} - \mathrm{e}^{|x-y|}\mathrm{tg}^3 z$

при $x = -4.5$; $y = 0.75 \cdot 10^{-4}$; $z = -0.845 \cdot 10^2$. Ответ: $s = -3.23765$.

3. $s = \dfrac{1 + \sin^2(x+y)}{\left|x - \dfrac{2y}{1 + x^2 y^2}\right|} x^{|y|} + \cos^2\left(\mathrm{arctg}\,\dfrac{1}{z}\right)$

при $x = 3.74 \cdot 10^{-2}$; $y = -0.825$; $z = 0.16 \cdot 10^2$. Ответ: $s = 1.05534$.

4. $s = \left|\cos x - \cos y\right|^{\left(1 + 2\sin^2 y\right)}\left(1 + z + \dfrac{z^2}{2} + \dfrac{z^3}{3} + \dfrac{z^4}{4}\right)$

при $x = 0.4 \cdot 10^4$; $y = -0.875$; $z = -0.475 \cdot 10^{-3}$. Ответ: $s = 1.98727$.

5. $s = \ln\left(y^{-\sqrt{|x|}}\right)\left(x - \dfrac{y}{2}\right) + \sin^2(\mathrm{arctg}\, z)$

при $x = -15.246$; $y = 4.642 \cdot 10^{-2}$; $z = 21$. Ответ: $s = -182.038$.

6. $s = \sqrt{10\left(\sqrt[3]{x} + x^{y+2}\right)}\left(\arcsin^2 z - |x - y|\right)$

при $x = 16.55 \cdot 10^{-3}$; $y = -2.75$; $z = 0.15$. Ответ: $s = -40.6307$.

7. $s = 5 \cdot \mathrm{arctg}\, x - \dfrac{1}{4}\arccos x \cdot \dfrac{x + 3|x - y| + x^2}{|x - y| z + x^2}$

при $x = 0.1722$; $y = 6.33$; $z = 3.25 \cdot 10^{-4}$. Ответ: $s = -205.306$.

8. $s = \dfrac{\mathrm{e}^{|x-y|}\,|x-y|^{x+y}}{\mathrm{arctg}\, x + \mathrm{arctg}\, z} + \sqrt[3]{x^6 + \ln^2 y}$

при $x = -2.235 \cdot 10^{-2}$; $y = 2.23$; $z = 15.221$. Ответ: $s = 39.3741$.

9. $s = \left| x^{\frac{y}{x}} - \sqrt[3]{\frac{y}{x}} \right| + (y - x)\frac{\cos y - \dfrac{z}{(y-x)}}{1 + (y-x)^2}$

при $x = 1.825 \cdot 10^2$; $y = 18.225$; $z = -3.298 \cdot 10^{-2}$. Ответ: $s = 1.21308$.

10. $s = 2^{-x}\sqrt{x + \sqrt[4]{|y|}}\sqrt[3]{e^{x-1/\sin z}}$

при $x = 3.981 \cdot 10^{-2}$; $y = -1.625 \cdot 10^3$; $z = 0.512$. Ответ: $s = 1.26185$.

11. $s = y^{\sqrt[3]{|x|}} + \dfrac{\cos^3(y)}{e^{|x-y|} + \dfrac{x}{2}} \cdot |x - y| \left(1 + \dfrac{\sin^2 z}{\sqrt{x+y}} \right)$

при $x = 6.251$; $y = 0.827$; $z = 25.001$. Ответ: $s = 0.712122$.

12. $s = 2^{\left(y^x\right)} + \left(3^x\right)^y - \dfrac{y\left(\operatorname{arctg} z - \dfrac{1}{3}\right)}{|x| + \dfrac{1}{y^2 + 1}}$

при $x = 3.251$; $y = 0.325$; $z = 0.466 \cdot 10^{-4}$. Ответ: $s = 4.23655$.

13. $s = \dfrac{\sqrt[4]{y + \sqrt[3]{x-1}}}{|x - y|\left(\sin^2 z + \operatorname{tg} z\right)}$

при $x = 17.421$; $y = 10.365 \cdot 10^{-3}$; $z = 0.828 \cdot 10^5$. Ответ: $s = 0.330564$.

14. $s = \dfrac{y^{x+1}}{\sqrt[3]{|y-2|} + 3} + \dfrac{x + \dfrac{y}{2}}{2|x+y|}(x+1)^{-1/\sin z}$

при $x = 12.3 \cdot 10^{-1}$; $y = 15.4$; $.z = 0.252 \cdot 10^3$. Ответ: $s = 82.8256$.

15. $s = \dfrac{x^{y+1} + e^{y-1}}{1 + x|y - \operatorname{tg} z|}\left(1 + |y - x|\right) + \dfrac{|y-x|^2}{2} - \dfrac{|y-x|^3}{3}$

при $x = 2.444$; $y = 0.869 \cdot 10^{-2}$; $z = -0.13 \cdot 10^3$. Ответ: $s = -0.498707$.

Пример выполнения лабораторной работы

Условие: написать программу для вычисления линейного арифметического выражения

$$h = \frac{x^{2y} + e^{y-1}}{1 + x|y - \operatorname{tg} z|} + 10 \cdot \sqrt[4]{x} - \ln z\,.$$

При $x = 5.6$, $y = 8.94 \cdot 10^{-2}$, $z = 0.23 \cdot 10^{-3}$. Ответ: $h = 24.9365$.

Текст программы:

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{
        system("chcp 1251");
        double x, y, z, h;
        cout << "Введите x: ";
                cin >> x;
        cout << "Введите y: ";
                cin >> y;
        cout << "Введите z: ";
                cin >> z;
        double a = pow(x, 2 * y) + exp(y - 1);
        double b = 1 + x * fabs(y - tan(z));
        double c = 10 * pow(x, 1 / 4.) - log(z);
                h = a / b + c;
        cout << "Результат = " << h << endl;
}
```

	Данные с клавиатуры вводятся в экспоненциальной форме. Например, число $0.23 \cdot 10^{-3}$ вводится: 0.23e-3

Лабораторная работа 2. Программирование разветвляющихся алгоритмов

Вычислить значение в соответствии с номером варианта. Предусмотреть возможность выбора вида функции f(x): cos(x), x^2 или e^x. Вывести на экран информацию о выполняемой ветви вычислений.

1. $$a = \begin{cases} (f(x)+y)^2 + \sqrt[3]{|f(x)|}, & xy > 0; \\ (f(x)+y)^2 + \sin x, & xy < 0; \\ (f(x)+y)^2 + y^3 & \textit{иначе.} \end{cases}$$

2. $$b=\begin{cases}\ln(f(x))+\sqrt[4]{|f(x)|}, & x/y=0;\\ \ln|f(x)/y|-y^2, & x/y<0;\\ (f(x)^2+y)^3 & \textit{иначе.}\end{cases}$$

3. $$c=\begin{cases}f(x)^2+\sqrt[3]{y}+\sin x, & x-y=0;\\ (f(x)-y)^2+\ln x, & x-y>0;\\ (y-f(x))^2+\operatorname{tg} y & \textit{иначе.}\end{cases}$$

4. $$d=\begin{cases}\sqrt[3]{|f(x)+x|}-tg\ y, & x>y;\\ (y-f(x))^3+\sin y, & x=y;\\ y+x^3-\sqrt{f(x)} & \textit{иначе.}\end{cases}$$

5. $$e=\begin{cases}\sin(f(x))/3, & xy=0;\\ \ln|y-f(x)|, & 7<xy<10;\\ 2tg^2x-y & \textit{иначе.}\end{cases}$$

6. $$g=\begin{cases}\mathrm{e}^{f(x)-y}+\sqrt[3]{x}, & x/y=0;\\ x^2-\ln(y^2+x), & -5<x/y<0;\\ 2f(x)^2-y^3 & \textit{иначе.}\end{cases}$$

7. $$s=\begin{cases}\sin^2 x-f(x), & x+|y|=0;\\ \sqrt[3]{|xy|}, & x+|y|<0;\\ 3f^2(x) & \textit{иначе.}\end{cases}$$

8. $$b=\begin{cases}\sqrt{x}/y, & x^2+y=0;\\ \cos^3 y-f(x), & x^2+y<0;\\ \sin(\cos(2f(x))) & \textit{иначе.}\end{cases}$$

9. $$l=\begin{cases}2\sin^2(\ln|x|), & y=0;\\ \operatorname{tg}(y^2-x), & -5<y<0;\\ x^2+f(x)-9 & \textit{иначе.}\end{cases}$$

10. $k = \begin{cases} \ln(|f(x)| + |y|), & |xy| > 10; \\ e^{f(x)+y}, & |xy| < 10; \\ \sqrt[3]{|f(x)|} + y & \textit{иначе.} \end{cases}$

11. $w = \begin{cases} \operatorname{tg}^2 x - f(x), & xy = 0; \\ e^{2f(x)} - y^2, & 0 < xy < 10; \\ \ln|y| + 2f(x) & \textit{иначе.} \end{cases}$

12. $g = \begin{cases} y^2 \cdot \sin^2 x, & y \cdot f(x) = 0; \\ \operatorname{tg}^2 x + f(x), & y \cdot f(x) < 0; \\ 2f(x) - \sin y & \textit{иначе.} \end{cases}$

13. $q = \begin{cases} \ln x - f^2(x), & y \cdot f(x) = 10; \\ 2y - 10\sin x, & y \cdot f(x) < 10; \\ y^2 + f(x) & \textit{иначе.} \end{cases}$

14. $u = \begin{cases} \sin x + \ln y, & x^2 y = 0; \\ \operatorname{tg}^2(f(x)), & 2 < x^2 y < 7; \\ f(x)^2 / 2 + x & \textit{иначе.} \end{cases}$

15. $w = \begin{cases} \sqrt[3]{f(x)} - xy, & 2 \cdot x / y = 0; \\ \sin^2 x - y, & 2 \cdot x / y < 0; \\ 4y - \operatorname{tg}(x) & \textit{иначе.} \end{cases}$

Лабораторная работа 3. Программирование циклических алгоритмов

Вывести на экран таблицу значений функции y(x) и ее разложения в ряд s(x) для x, изменяющегося от a до b с шагом h = (b – a) / 17. Задание выбрать в соответствии с номером варианта в табл. Л.1.

Таблица Л.1

Вариант	*a*	*b*	Функция	Разложение функции в ряд Тейлора	*k*
1	0.1	1	$y(x) = \sin x$	$s(x) = \sum_{n=0}^{k} (-1)^n \frac{x^{2n+1}}{(2n+1)!}$	160
2	0.1	1	$y(x) = \operatorname{ch} x$	$s(x) = \sum_{n=0}^{k} \frac{x^{2n}}{(2n)!}$	100

Окончание табл. Л.1

Вариант	a	b	Функция	Разложение функции в ряд Тейлора	k
3	0.1	1	$y(x)=\mathrm{e}^{x\sin x}$	$s(x)=\sum_{n=0}^{k}\frac{(x\sin x)^n}{n!}$	120
4	0.1	1	$y(x)=\cos x$	$s(x)=\sum_{n=0}^{k}(-1)^n\frac{x^{2n}}{(2n)!}$	80
5	0.1	1	$y(x)=\frac{\sin x}{x}$	$s(x)=\sum_{n=0}^{k}(-1)^n\frac{x^{2n}}{(2n+1)!}$	140
6	0.1	1	$y(x)=\mathrm{sh}\,x$	$s(x)=\sum_{n=0}^{k}\frac{x^{2n+1}}{(2n+1)!}$	80
7	0.1	1	$y(x)=\mathrm{e}^{-2\mathrm{e}^x}$	$s(x)=\sum_{n=0}^{k}\frac{2^n(-\mathrm{e}^x)^n}{n!}$	120
8	0.1	1	$y(x)=5^x$	$s(x)=\sum_{n=0}^{k}\frac{x^n\ln^n 5}{n!}$	100
9	0.1	1	$s(x)=\mathrm{e}^{2x}$	$s(x)=\sum_{n=0}^{k}\frac{(2x)^n}{n!}$	140
10	0.1	0.5	$y(x)=x^2\mathrm{e}^x$	$s(x)=\sum_{n=2}^{k}\frac{x^n}{(n-2)!}$	150
11	0.1	1	$y(x)=x\sin x$	$s(x)=\sum_{n=0}^{k}(-1)^n\frac{x^{2n+2}}{(2n+1)!}$	100
12	0.1	1	$y(x)=\mathrm{e}^{\cos x}$	$s(x)=\sum_{n=0}^{k}\frac{\cos^n x}{n!}$	80
13	–2	–0.1	$y(x)=x\cos x$	$s(x)=\sum_{n=0}^{k}(-1)^n\frac{x^{2n+1}}{(2n)!}$	140
14	0.2	0.8	$y(x)=3^{x-1}$	$s(x)=\sum_{n=0}^{k}\frac{(x-1)^n\ln^n 3}{n!}$	100
15	0.1	0.8	$y(x)=\cos 2x$	$s(x)=\sum_{n=0}^{k}(-4)^n\frac{x^{2n}}{(2n)!}$	180

Лабораторная работа 4. Использование одномерных массивов

Ввести с клавиатуры размер массива, выделить необходимый объем памяти для хранения элементов массива и ввести исходные данные. Выполнить задание, результат вывести на экран.

1. Задан массив целых чисел. Удалить все элементы, имеющие значение больше, чем среднее арифметическое значение элементов массива.

2. Задан массив целых чисел. Преобразовать массив следующим образом: все отрицательные элементы массива перенести в начало, а все остальные – в конец, сохранив исходное взаимное расположение как среди отрицательных, так и среди положительных элементов.

3. Задан массив целых чисел. Найти число, наиболее часто встречающееся в этом массиве.

4. Задан массив целых чисел. Найти числа, входящие в массив не более одного раза.

5. Задан массив действительных чисел. Сдвинуть элементы массива циклически на n позиций вправо, если n – положительное число, и влево, если n – отрицательное число (значение n задается с клавиатуры).

6. Задан массив целых чисел. Удалить из массива все числа, встречающиеся в массиве более одного раза.

7. Задан массив действительных чисел. Переставить в обратном порядке элементы, расположенные между первым положительным и последним отрицательным элементами массива.

8. Задан массив целых чисел. Удалить все элементы, стоящие до элемента с максимальным значением.

9. Задан массив действительных чисел. Определить количество различных элементов в массиве.

10. Задан массив целых чисел. Найти наименьший положительный элемент среди элементов с четными индексами массива.

11. Задан массив действительных чисел. Перенести максимальный элемент в нулевую позицию, а минимальный – в последнюю позицию массива. Взаимное расположение остальных элементов не должно изменяться.

12. Задан массив действительных чисел. Удалить все положительные элементы, у которых справа находится отрицательный элемент.

13. Задан массив целых чисел. Удалить из массива минимальный и максимальный элементы.

14. Задан массив действительных чисел. Найти сумму элементов, расположенных между минимальным и максимальным элементами массива.

15. Задан массив целых чисел. Найти произведение элементов, расположенных между последним и предпоследним положительными элементами массива.

Лабораторная работа 5. Использование двумерных массивов

Ввести с клавиатуры количество строк и столбцов массива, выделить необходимый объем памяти для хранения элементов массива и ввести исходные данные. Выполнить задание, результат вывести на экран.

1. Задана матрица размером $N \times M$. Определить количество различных элементов матрицы (т. е. повторяющиеся элементы считать один раз).

2. Задана матрица размером $N \times M$. Поменять местами строку, содержащую максимальный элемент, и строку, содержащую минимальный элемент.

3. Задана матрица размером $N \times M$. Вывести все элементы, являющиеся максимальными в своем столбце и одновременно минимальными в своей строке.

4. Задана матрица размером $N \times M$. Получить одномерный массив, каждый элемент которого будет содержать значение 0, если строка матрицы с таким же номером упорядочена по возрастанию, или значение 1 в противном случае.

5. Задана матрица размером $N \times M$. Удалить строку матрицы, содержащую элемент с максимальным значением.

6. Задана матрица размером $N \times M$. Определить количество «особых» элементов матрицы, считая элемент «особым», если он меньше суммы остальных элементов соответствующей строки.

7. Задана матрица размером $N \times M$. Поменять местами столбец, содержащий элемент с минимальным значением, и столбец, содержащий элемент с максимальным значением.

8. Задана матрица размером $N \times M$. Упорядочить ее строки по убыванию их максимальных элементов.

9. Задана матрица размером $N \times M$. Поменять местами строку, содержащую элемент с максимальным значением, и строку, содержащую элемент с минимальным значением.

10. Задана матрица размером $N \times M$. Упорядочить ее столбцы по возрастанию их наименьших элементов.

11. Задана матрица размером $N \times M$. Удалить столбец матрицы, содержащий элемент с минимальным значением.

12. Задана матрица размером $N \times M$. Получить одномерный массив, занося в ячейку значение 0, если строка матрицы с таким же номером содержит хотя бы один нулевой элемент, или 1 в противном случае.

13. Задана матрица размером $N \times M$. Удалить строку с максимальной суммой элементов.

14. Задана матрица размером $N \times M$. Определить количество «особых» элементов матрицы, считая элемент «особым», если он больше суммы остальных элементов соответствующего столбца.

15. Задана матрица размером $N \times M$. Упорядочить строки по возрастанию суммы их элементов.

Лабораторная работа 6. Программирование с использованием строк

Ввести нуль-терминальную строку с клавиатуры. Выполнить задание, результат вывести на экран.

1. Дана строка символов, состоящая из произвольных десятичных цифр. Числа в строке отделены друг от друга одним или несколькими пробелами. Удалить из строки четные числа.

2. Заменить в строке все группы подряд расположенных пробелов на один пробел.

3. Дана строка, состоящая из нулей и единиц. Удалить все группы, состоящие из трех нулей.

4. Вставить слово *Visual* между вторым и третьим словом строки. Слова в строке разделены одним или несколькими пробелами.

5. Поменять местами первое и второе слова строки. Слова в строке разделены одним или несколькими пробелами.

6. Удалить из строки слова, содержащие четное количество символов. Слова в строке разделены одним или несколькими пробелами.

7. Дана строка символов, состоящая из произвольных десятичных цифр. Числа в строке отделены друг от друга одним или несколькими пробелами. Вывести на экран числа этой строки в порядке возрастания их значений.

8. Дана строка, состоящая из нулей и единиц. Вывести группу с максимальным количеством одинаковых символов.

9. Вывести на экран порядковый номер слова максимальной длины и номер позиции в строке, с которой оно начинается. Слова в строке разделены одним или несколькими пробелами.

10. Удалить из строки предпоследнее слово. Слова в строке разделены одним или несколькими пробелами.

11. Вывести слова, которые начинаются и заканчиваются одной и той же буквой. Слова в строке разделены одним или несколькими пробелами.

12. Заменить в строке все слова *C* на *C++*. Слова в строке разделены одним или несколькими пробелами.

13. Дана строка, состоящая из нулей и единиц. Вывести на экран группы единиц с максимальным и минимальным количеством символов.

14. Удалить из строки слова, содержащие символ *'w'*. Слова в строке разделены одним или несколькими пробелами.

15. Дана строка, состоящая из нулей и единиц. Подсчитать количество групп с пятью единицами.

Лабораторная работа 7. Программирование с использованием структур

Объявить структуру с заданными полями. Динамически выделить память для хранения списка. Ввести данные. Выполнить задание, результат вывести на экран.

1. Имеется список участников спортивных соревнований. Каждый элемент списка содержит следующую информацию: название команды, фамилия спортсмена, номер телефона, возраст, рост и вес. Вывести в алфавитном порядке фамилии спортсменов, возраст которых менее 18 лет.

2. У администратора железнодорожных касс хранится информация о свободных местах в поездах. Каждый элемент списка содержит следующую информацию: номер поезда, время отправления, пункт назначения, число свободных мест. Вывести информацию о поездах, которые следуют до Бреста, в порядке убывания количества свободных мест.

3. Имеется список товаров, хранящихся на складе. Каждый элемент списка содержит следующую информацию: наименование, артикул, количество, цена. Вывести в алфавитном порядке информацию о товарах, количество которых на складе больше 10 и меньше 100 шт.

4. В аэропорту имеется список пассажиров, зарегистрировавшихся на рейс. Каждый элемент списка содержит следующую информацию: фамилия, номер билета, вес багажа. Вывести в алфавитном порядке фамилии пассажиров, вес багажа которых превышает 20 кг.

5. Имеется список участников олимпиады. Каждый элемент списка содержит следующую информацию: название учебного заведения, фамилия участника, номер телефона, количество набранных очков. Вывести в порядке убывания количества набранных очков фамилии участников из БГУИР.

6. Имеется список семян овощных культур. Каждый элемент списка содержит следующую информацию: номер партии, название культуры, номера месяцев посева, высадки рассады и уборки урожая. Вывести в алфавитном порядке названия культур, урожай которых убирается в августе.

7. Имеется список студентов. Каждый элемент списка содержит следующую информацию: фамилия, номер телефона, год и место рождения, три экзаменационных оценки за последнюю сессию. Вывести в алфавитном порядке фамилии студентов, которые сдали экзамены без двоек.

8. Имеется список автомобилей. Каждый элемент списка содержит следующую информацию: марка, номер кузова, год выпуска, объем двигателя и расход топлива. Вывести в порядке возрастания расхода топлива информацию об автомобилях, выпущенных после 2010 года.

9. Имеется список студентов. Каждый элемент списка содержит следующую информацию: фамилия, номер телефона, год и место рождения, три экзаменационные оценки за последнюю сессию. Вывести информацию о студентах, проживающих в Минске, в порядке убывания среднего балла.

10. Имеется список сотрудников предприятия. Каждый элемент списка содержит следующую информацию: фамилия, табельный номер, год рождения и год поступления на работу. Вывести информацию о сотрудниках фирмы, родившихся после 1995 года, в порядке убывания стажа работы.

11. Имеется телефонная база данных. Каждый элемент базы содержит следующую информацию: номер телефона, фамилия и адрес абонента. Вывести на

экран в алфавитном порядке фамилии абонентов, номера телефонов которых начинаются с числа 23.

12. Имеется список автомобилей. Каждый элемент списка содержит следующую информацию: марка, номер кузова, год выпуска, объем двигателя и максимальная скорость. Вывести информацию об автомобилях, выпущенных после 2005 года, в порядке убывания их максимальной скорости.

13. Имеется список стран мира. Каждый элемент списка содержит следующую информацию: название страны, код страны, название части света, в которой находится страна, и площадь страны. Вывести информацию о странах, находящихся в Европе, в порядке возрастания их площади.

14. Имеется расписание движения междугородных автобусов. Каждый элемент расписания содержит следующую информацию: номер рейса, время отправления, пункт назначения, время прибытия в пункт назначения. Вывести информацию о рейсах до города Могилев в порядке возрастания времени их отправления.

15. Имеется список книг. Каждый элемент списка содержит следующую информацию: название, фамилия автора, год издания, количество страниц, ISBN книги. Вывести в алфавитном порядке названия книг, изданных до 1990 года.

Лабораторная работа 8. Программирование с использованием файлов

Записать необходимые данные в бинарный файл (задание взять из соответствующего варианта лабораторной работы 7). Прочитать данные и выполнить задание. Результат вывести на экран и в текстовый файл.

Лабораторная работа 9. Использование функций

Вывести на экран таблицу значений функции и ее разложения в ряд для x, изменяющегося от a до b с шагом h = (b – a) / 10. Расчет y(x) и s(x) поместить в функцию. Использовать прототипы функций. Параметры передавать указанным в табл. Л.2 способом. Расчет функции s(x) выполнить с заданной точностью ε.

Таблица Л.2

Вариант	*a*	*b*	Функция	Разложение функции в ряд Тейлора	ε	Способ передачи параметров
1	0.8	1.8	$y(x)=\ln x$	$s(x)=-\sum_{n=1}^{\infty}(-1)^n\frac{(x-1)^n}{n}$	10^{-4}	По ссылке
2	0.1	0.9	$y(x)=\mathrm{ch}^2 x$	$s(x)=1+\sum_{n=1}^{\infty}\frac{2^{2n-1}x^{2n}}{(2n)!}$	10^{-5}	По значению
3	0.1	0.6	$y(x)=\frac{1}{1+x}$	$s(x)=\sum_{n=0}^{\infty}(-1)^n x^n$	10^{-6}	По указателю

Окончание табл. Л.2

Вариант	a	b	Функция	Разложение функции в ряд Тейлора	ε	Способ передачи параметров
4	–0.9	0.9	$y(x) = x\,\mathrm{arctg}\,x$	$s(x) = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n+2}}{1+2n}$	10^{-4}	По ссылке
5	–0.1	1	$y(x) = 2^{-x}$	$s(x) = \sum_{n=0}^{\infty} \frac{x^n(-\ln(2))^n}{n!}$	10^{-5}	По значению
6	–0.9	0.9	$y(x) = \cos(x-4)$	$s(x) = \sum_{n=0}^{\infty} (-1)^n \frac{(4-x)^{2n}}{(2n)!}$	10^{-3}	По указателю
7	–0.5	0.5	$y(x) = \cos(\sin x)$	$s(x) = \sum_{n=0}^{\infty} (-1)^n \frac{\sin^{2n}(x)}{(2n)!}$	10^{-4}	По ссылке
8	–0.3	0.4	$y(x) = e^x + e^{-x}$	$s(x) = \sum_{n=0}^{\infty} \frac{(-x)^n + x^n}{n!}$	10^{-5}	По значению
9	–0.1	1.3	$y(x) = 2^x$	$s(x) = \sum_{n=0}^{\infty} \frac{x^n \ln^n 2}{n!}$	10^{-3}	По указателю
10	–0.5	0.5	$y(x) = e^x$	$s(x) = \sum_{n=0}^{\infty} \frac{x^{2n-1}(2n+x)}{(2n)!}$	10^{-4}	По ссылке
11	0.1	0.8	$y(x) = \ln(1+x^2)$	$s(x) = -\sum_{n=1}^{\infty} (-1)^n \frac{x^{2n}}{n}$	10^{-5}	По значению
12	1	2.5	$y(x) = \sin^2 x$	$s(x) = -\sum_{n=1}^{\infty} (-1)^n \frac{2^{2n-1}x^{2n}}{(2n)!}$	10^{-3}	По указателю
13	–1.5	1.5	$y(x) = \cos^3 x$	$s(x) = \frac{1}{4}\sum_{n=0}^{\infty} (-1)^n \frac{(3+9^n)x^{2n}}{(2n)!}$	10^{-4}	По ссылке
14	–0.8	0.9	$y(x) = \mathrm{ch}(x^2)$	$s(x) = \sum_{n=0}^{\infty} \frac{x^{4n}}{(2n)!}$	10^{-5}	По значению
15	–0.9	0.9	$y(x) = \mathrm{arctg}\,x$	$s(x) = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n+1}}{2n+1}$	10^{-3}	По указателю

Лабораторная работа 10. Программирование рекурсивных алгоритмов

Решить задачу двумя способами: с применением рекурсии и без нее.

1. Вычислить произведение четного значения ($n \geq 2$) сомножителей $y(n) = \frac{2}{1} \cdot \frac{2}{3} \cdot \frac{4}{3} \cdot \frac{4}{5} \cdot \frac{6}{5} \cdot \frac{6}{7} \cdot \ldots \cdot \frac{n}{n-1} \cdot \frac{n}{n+1}$.

2. Проверить, является ли заданная строка палиндромом.

3. Вычислить число сочетаний $C_n^k = \frac{n!}{k!(n-k)!}$ по формуле $C_n^0 = C_n^n = 1$, $C_n^k = C_{n-1}^k + C_{n-1}^{k-1}$ при $n > 1$; $0 < k < n$.

4. Вычислить $y(n) = \sqrt{1 + \sqrt{2 + \ldots + \sqrt{(n-1) + \sqrt{n}}}}$.

5. Вычислить значение $x = \sqrt{a}$, используя формулу $x_n = \frac{1}{2}\left(x_{n-1} + a \,/\, x_{n-1}\right)$, в качестве начального приближения использовать значение $x_0 = (1 + a) / 2$.

6. Вычислить $y(n, k) = 1^k + 2^k + \ldots + n^k$.

7. Вычислить $y(n) = \cfrac{1}{n + \cfrac{1}{(n-1) + \cfrac{1}{(n-2) + \cfrac{1}{\ldots}}}}$

$$\ldots + \frac{1}{1 + \frac{1}{2}}$$

8. Подсчитать количество цифр в заданном числе.

9. Вычислить $y(n) = \cfrac{1}{1 + \cfrac{1}{2 + \cfrac{1}{3 + \cfrac{1}{\ldots}}}}$

$$\ldots + \frac{1}{(n-1) + \frac{1}{n}}$$

10. Написать функцию умножения двух чисел, используя только операцию сложения.

11. В упорядоченном массиве целых чисел a_i, $i = 0 \ldots (n-1)$ найти номер элемента x методом бинарного поиска: если $x \leq a_{n/2}$, тогда $x \in \left[a_1 \ldots a_{n/2}\right]$, иначе

$x \in \left[a_{n/2+1} \ldots a_n \right]$. Если элемент x отсутствует в массиве, то вывести соответствующее сообщение.

12. Написать функцию сложения двух чисел, используя только операцию добавления единицы.

13. Вычислить произведение двух целых положительных чисел $P = a \cdot b$ по следующему алгоритму: если b четное, то $P = 2 \cdot (a \cdot b / 2)$, иначе – $P = a + (a \cdot (b - 1))$. Если $b = 0$, то $P = 0$.

14. Подсчитать сумму цифр в десятичной записи заданного числа.

15. Найти значение функции Аккермана $A(m, n)$, которая определяется для всех неотрицательных целых аргументов m и n следующим образом: $A(0, n) = n + 1$, если $m = 0$; $A(m, 0) = A(m - 1, 1)$, если $n = 0$; $A(m, n) = A(m - 1, A(m, n - 1))$, если и $m > 0$ и $n > 0$.

Лабораторная работа 11. Алгоритмы сортировки

Дополнить программу, написанную при выполнении лабораторной работы 8, функциями упорядочения массива структур по неубыванию заданного ключа. Результат вывести на экран.

1. Ключ: рост спортсмена. Методы сортировки: *QuickSort* и сортировка вставкой.

2. Ключ: время отправления поезда. Методы сортировки: *QuickSort* и метод Шелла.

3. Ключ: цена товара. Методы сортировки: *QuickSort* и сортировка выбором.

4. Ключ: вес багажа пассажира. Методы сортировки: *QuickSort* и сортировка вставкой.

5. Ключ: количество набранных очков участником олимпиады. Методы сортировки: *QuickSort* и метод Шелла.

6. Ключ: номер месяца уборки урожая. Методы сортировки: *QuickSort* и сортировка выбором.

7. Ключ: год рождения студента. Методы сортировки: *QuickSort* и сортировка вставкой.

8. Ключ: объем двигателя автомобиля. Методы сортировки: *QuickSort* и метод Шелла.

9. Ключ: год рождения студента. Методы сортировки: *QuickSort* и сортировка выбором.

10. Ключ: год поступления на работу сотрудника. Методы сортировки: *QuickSort* и сортировка вставкой.

11. Ключ: номер телефона абонента. Методы сортировки: *QuickSort* и метод Шелла.

12. Ключ: год выпуска автомобиля. Методы сортировки: *QuickSort* и сортировка выбором.

13. Ключ: год образования государства. Методы сортировки: *QuickSort* и сортировка вставкой.

14. Ключ: номер рейса автобуса. Методы сортировки: *QuickSort* и метод Шелла.

15. Ключ: количество страниц в книге. Методы сортировки: *QuickSort* и сортировка выбором.

Лабораторная работа 12. Алгоритмы поиска

Дополнить программу, написанную при выполнении лабораторной работы 11, функциями поиска элементов по ключу в массиве структур. Найти элемент с заданным ключом указанным методом поиска (для упрощения предполагается, что в массиве присутствует не более одного такого элемента). Если элемент не найден, то вывести соответствующее сообщение.

1. Вывести на экран фамилию спортсмена, у которого рост равен 197 см. Метод поиска: интерполяционный.

2. Вывести на экран пункт назначения поезда, который отправляется в 11 часов. Методы поиска: линейный с барьером и двоичный.

3. Вывести на экран наименование товара с ценой, равной 265 000 руб. Методы поиска: линейный и двоичный.

4. Вывести на экран фамилию пассажира, у которого багаж весит 58 кг. Метод поиска: интерполяционный.

5. Вывести на экран фамилию участника олимпиады, который набрал 212 очков. Методы поиска: линейный с барьером и двоичный.

6. Вывести на экран название культуры, которую убирают в июне (шестом месяце года). Методы поиска: линейный и двоичный.

7. Вывести на экран средний балл, набранный на экзамене студентом, родившимся в 1991 году. Методы поиска: линейный с барьером и двоичный.

8. Вывести на экран марку автомобиля с объемом двигателя 1998 $см^3$. Метод поиска: интерполяционный.

9. Вывести на экран фамилию студента, родившегося в 1980 году. Методы поиска: линейный с барьером и двоичный.

10. Вывести на экран фамилию сотрудника, который был принят на работу в 1999 году. Метод поиска: интерполяционный.

11. Вывести на экран фамилию абонента, на которого зарегистрирован номер телефона 797-24-74. Методы поиска: линейный и двоичный.

12. Вывести на экран максимальную скорость автомобиля, выпущенного в 1996 году. Методы поиска: линейный с барьером и двоичный.

13. Вывести на экран название государства, образованного в 1927 году. Метод поиска: интерполяционный.

14. Вывести на экран пункт назначения автобуса с номером рейса 295. Методы поиска: линейный с барьером и двоичный.

15. Вывести на экран название книги, в которой 1575 страниц. Методы поиска: линейный и двоичный.

Лабораторная работа 13. Хеширование

Используя данные из лабораторной работы 8, создать хеш-таблицу из M элементов (число M выбирается исходя из количества элементов в массиве структур и особенностей схемы хеширования). Осуществить поиск элемента по заданному ключу в хеш-таблице. Вывести на экран исходный массив, хеш-таблицу и все поля найденной структуры. Задание выбрать в соответствии с номером варианта в таблице Л.3.

Таблица Л.3

Вариант	Ключевое поле	Схема хеширования
1	Номер телефона	С квадратичной адресацией
2	Номер поезда	С произвольной адресацией
3	Артикул товара	С двойным хешированием
4	Номер билета	На основе связанных списков
5	Номер телефона	С квадратичной адресацией
6	Номер партии	С произвольной адресацией
7	Номер телефона	С двойным хешированием
8	Номер кузова автомобиля	На основе связанных списков
9	Номер телефона	С квадратичной адресацией
10	Табельный номер	С произвольной адресацией
11	Номер телефона	С двойным хешированием
12	Номер кузова автомобиля	На основе связанных списков
13	Код страны	С квадратичной адресацией
14	Номер рейса	С произвольной адресацией
15	ISBN книги	С двойным хешированием

Лабораторная работа 14. Использование стеков

Создать стек, состоящий из n целых чисел. Выполнить задание (информационную часть в оперативной памяти не перемещать). Результат вывести на экран. В конце работы освободить всю динамически выделенную память.

1. Добавить элемент со значением 23 перед предпоследним элементом стека.

2. Удалить каждый третий (по порядку) элемент стека.

3. Найти среднее значение всех элементов стека. Удалить из стека все элементы, значение которых меньше среднего значения.

4. Удалить элементы, значение которых больше среднего арифметического всех элементов стека.

5. Удалить из стека все отрицательные числа.

6. Удалить все элементы стека, расположенные перед минимальным элементом стека.

7. Удалить все элементы, расположенные между первым и последним отрицательными элементами стека.

8. Добавить элемент со значением 28 после максимального элемента стека.

9. Поменять местами первый положительный и предпоследний отрицательный элементы стека.

10. Удалить из стека все элементы, значения которых находятся в диапазоне от 0 до 9.

11. Удалить из стека все нечетные числа.

12. Поменять местами минимальный и максимальный элементы стека.

13. Преобразовать стек таким образом, чтобы порядок следования элементов был изменен на обратный.

14. Поменять местами второй и предпоследний элементы стека.

15. Добавить элемент со значением 41 перед каждым отрицательным элементом.

Лабораторная работа 15. Использование двусвязанных списков

Выполнить задание в соответствии с вариантом (информационную часть в оперативной памяти не перемещать). Результат вывести на экран. В конце работы освободить всю динамически выделенную память.

1. Создать двусвязанный список, состоящий из *n* целых чисел. Переместить во второй список элементы, находящиеся между минимальным и максимальным элементами первого списка.

2. Создать два двусвязанных списка, состоящих из *n* целых чисел упорядоченных по неубыванию. Переместить в третий список элементы со значениями, которые встречаются и в первом, и во втором списках.

3. Создать двусвязанный список, состоящий из *n* целых чисел. Отрицательные элементы удалить, а четные перенести во второй список.

4. Создать двусвязанный список, состоящий из *n* символов латинского алфавита и символов арифметических операций. Переместить символы арифметических операций во второй список.

5. Создать два двусвязанных списка, состоящих из *n* символов латинского алфавита. Переместить все данные в третий список таким образом, чтобы строчные символы находились в левой половине списка, а прописные − в правой.

6. Создать двусвязанный список, состоящий из *n* целых чисел. Переместить во второй список элементы, значения которых больше среднего значения элементов первого списка.

7. Создать двусвязанный список, состоящий из *n* символов латинского алфавита. Удалить из списка элементы с повторяющимися более одного раза значениями.

8. Создать два двусвязанных списка, состоящих из *n* целых чисел, упорядоченных по неубыванию. Преобразовать их в третий список, который будет упорядочен по невозрастанию.

9. Создать двусвязанный список, состоящий из *n* символов латинского алфавита. Преобразовать его в два списка: первый список должен содержать прописные символы, второй − строчные.

10. Создать двусвязанный список, состоящий из *n* целых чисел. Извлечь из первого списка и переместить во второй список все отрицательные числа.

11. Создать двусвязанный список, состоящий из *n* целых чисел. Удалить из списка все элементы, находящиеся между его максимальным и минимальным элементами.

12. Создать двусвязанный список, состоящий из *n* целых чисел. Переместить во второй список элементы, повторяющиеся в первом списке более одного раза.

13. Создать двусвязанный список, состоящий из *n* целых чисел. Преобразовать его в два списка: первый список должен содержать только четные числа, второй − нечетные.

14. Создать двусвязанный список, состоящий из *n* действительных чисел. Расположить элементы списка в обратном порядке.

15. Создать два двусвязанных списка, состоящих из *n* целых чисел, упорядоченных по неубыванию. Переместить все данные в третий список, удаляя повторяющиеся значения.

Лабораторная работа 16. Работа с бинарным деревом поиска

Создать сбалансированное дерево поиска, состоящее из целых чисел. Вывести информацию на экран, используя прямой, обратный и симметричный обходы дерева. Выполнить задание, результат вывести на экран. В конце работы освободить всю динамически выделенную память.

1. Найти узел, имеющий значение, ближайшее к среднему значению всех ключей дерева.

2. Удалить из правой ветви дерева узел с минимальным значением ключа и всех его потомков.

3. Удалить в дереве все узлы, имеющие только одного потомка справа.

4. Удалить в дереве все узлы, имеющие четные ключи.

5. Удалить из дерева ветвь с вершиной, имеющей заданный ключ.

6. Удалить из дерева узел с заданным ключом.

7. Поменять местами узлы с минимальным и максимальным ключами в левом поддереве.

8. Поменять местами узел с максимальным ключом и узел, являющийся корнем дерева.

9. Найти количество листьев на каждом уровне дерева.

10. Удалить в дереве все узлы, имеющие отрицательные ключи.

11. Поменять местами узлы с минимальным и максимальным ключами.

12. Удалить в дереве все узлы, имеющие только одного потомка слева.

13. Удалить все узлы дерева, имеющие значение ключа больше 7.

14. Удалить из левой ветви дерева узел с максимальным значением ключа и всех его потомков.

15. Удалить все узлы дерева, имеющие значение ключа, равное 33.

Лабораторная работа 17. Вычисление алгебраических выражений

Ввести заданное арифметическое выражение и необходимые данные. Преобразовать запись арифметического выражения в форму обратной польской записи (для обозначения операции возведения в степень использовать знак ^). Вычислить арифметическое выражение. Результат вывести на экран. Задание выбрать в соответствии с номером варианта.

1. $(x-y)^w \cdot \dfrac{c+k}{f-k}$.
2. $\dfrac{f+s}{f-y} + \dfrac{y+s^w}{x-s}$.
3. $a \cdot x^w - \dfrac{b+x}{y}$.
4. $x \cdot \dfrac{y+a}{y+b^w} - c$.
5. $\dfrac{x^w}{x-y} \cdot s + b^2$.
6. $\dfrac{c+k \cdot s}{f-k \cdot s^w} + a$.
7. $b - s \cdot \dfrac{x}{x^w + y^w}$.
8. $\dfrac{c^w \cdot d^w}{k^w \cdot (k+c)}$.
9. $x^w - y^w + \dfrac{a+y}{a-x}$.
10. $x - y \cdot \dfrac{(x+y)^w}{x+k}$.
11. $(a-b)^w \cdot \dfrac{x^w}{x^w + y}$.
12. $\dfrac{x-c}{c+y^w} \cdot x - y$.
13. $\dfrac{x}{f-k^w} + xy - c$.
14. $ax^w + (cy-a) \cdot y$.
15. $a^w \cdot \dfrac{x+k^w}{y-k} + s$.

ПРИЛОЖЕНИЕ

РАБОТА В СРЕДЕ *MICROSOFT VISUAL C++*

1. Консольный режим работы

Программа, создаваемая в среде *Visual C++*, оформляется в виде отдельного проекта. ***Проект*** (*project*) – набор взаимосвязанных исходных файлов, предназначенных для решения определенной задачи, компиляция и компоновка которых позволяет получить выполняемую программу. В проект входят как файлы, непосредственно создаваемые программистом, так и файлы, которые автоматически создает и редактирует среда программирования.

Для **создания нового проекта** необходимо:

– выбрать **Файл – Создать – Проект**;

– в открывшемся окне выбрать **Пустой проект *C++*** (*выбрать C++, Windows, Консоль*);

– в поле **Имя проекта** ввести имя проекта, например *maylab*1;

– в поле **Расположение** ввести имя каталога, в котором будет размещен проект и полный путь к нему, например *D*:*WORK**mylab*1. Каталог также можно выбрать, используя диалоговое окно **Расположение проекта**, для чего надо щелкнуть мышью по кнопке **…** ;

– щелкнуть мышью по кнопке **Создать**.

Для работы с консольным приложением необходимо создать новый или добавить существующий файл с текстом программы.

Для **создания нового файла** необходимо:

– выбрать **Проект – Добавить новый элемент**;

– выбрать **Файл *C++* (*.cpp*)**, задать имя файла, нажать кнопку **Добавить**.

Для **добавления в проект уже существующего файла** с текстом программы необходимо выбрать **Проект – Добавить существующий элемент** и указать имя добавляемого файла.

2. Выполнение программы

Для создания проекта необходимо выполнить **Сборка – Собрать решение** (**Ctrl + Shift + B**). В окне **Вывод** появятся результаты сборки проекта. Если в программе были обнаружены синтаксические ошибки, то выводится их описание. Все ошибки необходимо исправить.

Если ошибки не были обнаружены, то можно запустить программу на выполнение **Отладка – Запуск без отладки** (**Ctrl + F5**) (для файлов, в которые после последней сборки вносились изменения, автоматически выполняются перекомпилирование и перекомпоновка).

После окончания работы проект можно закрыть, выбрав **Файл – Закрыть решение**, или закрыть приложение *MVC++*.

Для открытия сохраненного ранее проекта выбрать **Файл – Открыть проект или решение**.

3. Отладка программы

Если синтаксических ошибок в программе нет (программа выполняется), но результат неверный, необходимо проверить наличие логических ошибок.

Для поиска логических ошибок имеется встроенный отладчик.

Для построчного выполнения программы используется **Отладка – Шаг с обходом** (**F10**). При каждом нажатии клавиши **F10** выполняется текущая строка и осуществляется переход к следующей строке. Если необходимо проверить текст вызываемой функции, то следует нажать **Отладка – Шаг с обходом** (**F11**). Для того чтобы начать отладку с определенной строки программы, надо установить курсор в эту строку и нажать **Ctrl + F10**.

Имеется возможность установки *точек прерывания* выполнения программы. Для установки точки прерывания необходимо поместить курсор в нужную строку и нажать **F9**. Точка прерывания обозначается красным кружком на специальном поле (слева от текста программы). Для удаления точки прерывания поместить курсор в необходимую строку и повторно нажать **F9**. Количество точек прерывания в программе не ограничено. Для выполнения программы до точки прерывания необходимо нажать **F5**. Для продолжения отладки (выполнения программы до следующей точки прерывания) повторно нажимается клавиша **F5**.

Желтая стрелка на поле слева от окна текста программы указывает на строку, которая будет выполнена на следующем шаге отладки.

Для контроля за значением переменной можно подвести к ней указатель мыши и задержать его на несколько секунд. На экране рядом с именем переменной появится окно, содержащее текущее значение этой переменной. Кроме этого, значения последних измененных, а также добавленных в список пользователем переменных будут отображаться в окне **Видимые** (***Watch***).

СПИСОК РЕКОМЕНДОВАННОЙ ЛИТЕРАТУРЫ

1. Вирт, Н. Алгоритмы и структуры данных / Н. Вирт. – СПб. : Невский Диалект, 2001. – 352 с.

2. Лафоре, Р. Объектно-ориентированное программирование в C++ / Р. Лафоре. – 4-е изд. – СПб. : Питер, 2016. – 928 с.

3. Керниган, Б. Язык программирования С / Б. Керниган, Д. Ритчи. – 2-е изд., перераб. и доп. – М. : Вильямс, 2009. – 304 с.

4. Навроцкий, А. А. Основы алгоритмизации и программирования в среде Visual C++ : учеб.-метод. пособие / А. А. Навроцкий. – Минск : БГУИР, 2014. – 160 с.

5. Шилдт, Г. Искусство программирования на C++ / Г. Шилдт. – СПб. : БХВ-Петербург, 2005. – 496 с.

6. Страуструп, Б. Язык программирования C++ / Б. Страуструп. – М. : Бином, 2012. – 1104 с.

7. Кнут, Д. Искусство программирования. В 3 т. Т. 1–3 / Д. Кнут. – М. : Вильямс, 2004. – 486 с.

Св. план 2025, поз. 2

Учебное издание

Навроцкий Анатолий Александрович

ОСНОВЫ АЛГОРИТМИЗАЦИИ И ПРОГРАММИРОВАНИЯ В СРЕДЕ VISUAL C++

УЧЕБНО-МЕТОДИЧЕСКОЕ ПОСОБИЕ

2-е издание, дополненное и пересмотренное

Редактор *А. Ю. Шурко*
Корректор *Е. Н. Батурчик*
Компьютерная правка, оригинал-макет *Е. Г. Бабичева*

Подписано в печать 24.12.2025. Формат 60×84 1/16. Бумага офсетная. Гарнитура «Таймс».
Отпечатано на ризографе. Усл. печ. л. 9,18. Уч.-изд. л. 9,6. Тираж 50 экз. Заказ 140.

Издатель и полиграфическое исполнение: учреждение образования
«Белорусский государственный университет информатики и радиоэлектроники».
Свидетельство о государственной регистрации издателя, изготовителя,
распространителя печатных изданий №1/238 от 24.03.2014,
№2/113 от 07.04.2014, №3/615 от 07.04.2014.
Ул. П. Бровки, 6, 220013, г. Минск