**Максим Динман**

# C++

## ОСВОЙ НА ПРИМЕРАХ

Санкт-Петербург
«БХВ-Петербург»
2006

УДК 681.3.068+800.92С++
ББК 32.973.26-018.1
Д44

**Динман М. И.**

Д44 С++. Освой на примерах. — СПб.: БХВ-Петербург, 2006. — 384 с.: ил.

ISBN 5-94157-917-9

Подробно и доступно на занимательных примерах рассмотрены синтаксис, семантика и техника программирования на языке С++. Описаны все этапы проектирования программ, приведены подробные комментарии программного кода, проанализированы результаты вычислений, показаны типичные проблемы и пути их решения. Большое внимание уделяется алгоритмам и примерам решения задач при помощи графов, а также алгоритмам шифрования. Каждая глава содержит упражнения для самостоятельной работы.

*Для учащихся и начинающих программистов*

УДК 681.3.068+800.92С++
ББК 32.973.26-018.1

**Группа подготовки издания:**

| | |
|---|---|
| Главный редактор | *Екатерина Кондукова* |
| Зам. главного редактора | *Игорь Шишигин* |
| Зав. редакцией | *Григорий Добин* |
| Редактор | *Анна Кузьмина* |
| Компьютерная верстка | *Натальи Караваевой* |
| Корректор | *Наталия Першакова* |
| Дизайн серии | *Игоря Цырульникова* |
| Оформление обложки | *Елены Беляевой* |
| Зав. производством | *Николай Тверских* |

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 27.06.06.
Формат 60×90 1/16. Печать офсетная. Усл. печ. л. 24.
Тираж 3000 экз. Заказ №
"БХВ-Петербург", 194354, Санкт-Петербург, ул. Есенина, 5Б.

Санитарно-эпидемиологическое заключение на продукцию
№ 77.99.02.953.Д.006421.11.04 от 11.11.2004 г. выдано Федеральной службой
по надзору в сфере защиты прав потребителей и благополучия человека.

Отпечатано с готовых диапозитивов
в ГУП "Типография "Наука"
199034, Санкт-Петербург, 9 линия, 12

ISBN 5-94157-917-9

# Оглавление

# Введение

Данная книга предназначена для тех, кто хочет освоить азы программирования на C++. Здесь каждый сможет найти много интересных и полезных вещей, которые помогут и при написании простых программ, и при разработке сложных проектов.

Программирование на C++ в современном обществе стало правилом хорошего тона. Большинство программ, предназначенных для работы в средах Windows и UNIX, были написаны на C либо C++.

Программируя на C++, вы не только сумеете автоматизировать свою работу, но и научиться думать по-иному, т. е. подходить к задаче любого рода, даже бытовой, с такой стороны, с которой на ее решение вами будет затрачено минимальное количество времени. Причем решение таких задач не будет содержать сложных для понимания фраз или выдержек из кода, а окажется простым и доступным для любого человека, никак не связанного с программированием.

Приведенные далее правила помогут вам в решении любой задачи:

- ищите решение задачи в ее условии;
- проанализируйте входные данные;
- всегда помните, что вам нужно получить в результате;
- не спешите сразу писать код той или иной задачи;
- решение может стать очевидным, если свести задачу к некоторым бытовым случаям;
- решите задачу в общих чертах;

- предусмотрите все частные случаи, если таковые имеются;
- перечислите компоненты и объекты, которые необходимы при решении задачи;
- определите свойства каждого используемого объекта;
- запрограммируйте составленный алгоритм;
- старайтесь код программы сопровождать понятными комментариями;
- если у вас возникли трудности, произведите пошаговое выполнение алгоритма, это поможет найти и устранить ошибку;
- никогда не думайте, что C++ умнее или в чем-то превосходит вас.

# B1. О названии

Название C++ — изобретение совсем недавнее (лето 1983 года). Его придумал Рик Масситти. Название указывает на эволюционную природу перехода к нему от C. "++" — это операция приращения в C. Чуть более короткое имя C+ является синтаксической ошибкой. Кроме того, оно уже было использовано в названии совсем другого языка. Знатоки семантики C находят, что C++ хуже, чем ++C.

Более ранние версии языка использовались, начиная с 1980 года, и были известны как "C с классами". Первоначально язык был придуман потому, что автор хотел написать модели, управляемые прерываниями, для чего был бы идеален Simula67, если не принимать во внимание эффективность. "C с классами" использовался для крупных проектов моделирования, в которых строго тестировались возможности написания программ, требующих минимального пространства памяти и времени на выполнение. В "C с классами" не хватало перегрузки операций, ссылок, виртуальных функций и многих деталей. C++ был впервые введен за пределами исследовательской группы автора в июле 1983 года. Однако тогда многие особенности C++ были еще не придуманы.

# B2. О разработке

Изначально C++ был разработан, чтобы автору языка и его коллегам не приходилось программировать на ассемблере, C или других современных языках высокого уровня. Основным его предназначением было сделать удобным написание сложных программ. Плана разработки C++ на бумаге никогда не было — проект, документация и реализация двигались одновременно.

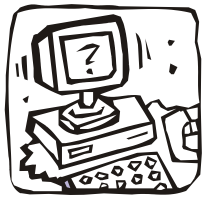
Разумеется, внешний интерфейс среды программирования C++ был написан на C++. Никогда не существовало "Проекта C++" и "Комитета по разработке C++". Поэтому язык развивался и продолжает развиваться во всех направлениях путем преодоления сложностей, с которыми сталкиваются программисты, а также в процессе дискуссий автора с коллегами.

В качестве базового языка для C++ был выбран C, потому что он многоцелевой, лаконичный, имеет относительно низкий уровень, отвечает требованиям большинства задач системного программирования, применим в любой операционной системе.

# B3. В общих чертах о C++

C++ — универсальный язык программирования, наиболее подходящий для серьезного программиста. За исключением второстепенных деталей, C++ является надмножеством языка программирования C. Помимо возможностей, которые дает C, C++ предоставляет гибкие и эффективные средства определения новых типов, точно отвечающих концепциям приложения, благодаря чему программист может разделять разрабатываемую программу на легко поддающиеся контролю части. Такой метод построения программ часто называют абстракцией данных. C++ имеет гораздо лучшие, чем в C, средства выражения модульности программы и проверки типов. В языке есть также усовершенствования, не связанные непосредственно с классами, включающие в себя символические константы.

В C++ сохранены возможности языка C по работе с основными объектами аппаратного обеспечения (биты, байты, слова, адреса и т. п.). Это позволяет весьма эффективно реализовывать типы, определяемые пользователем. C++ и его стандартные библиотеки спроектированы так, чтобы обеспечить переносимость. Имеющаяся на текущий момент реализация языка может работать в большинстве систем, поддерживающих C. При написании программ на C++ можно использовать библиотеки языка C, а также бóльшую часть инструментальных средств, поддерживающих программирование на C.



# Глава 1

# Знакомство с приложением

## 1.1. Установка C++

Для установки среды разработки C++ запустите файл Setup.exe, который находится в папке инсталляции программы. На экране появится диалоговое окно установки (рис. 1.1). Для продолжения нажмите кнопку **Next** (Далее), оставив все значения без изменения.

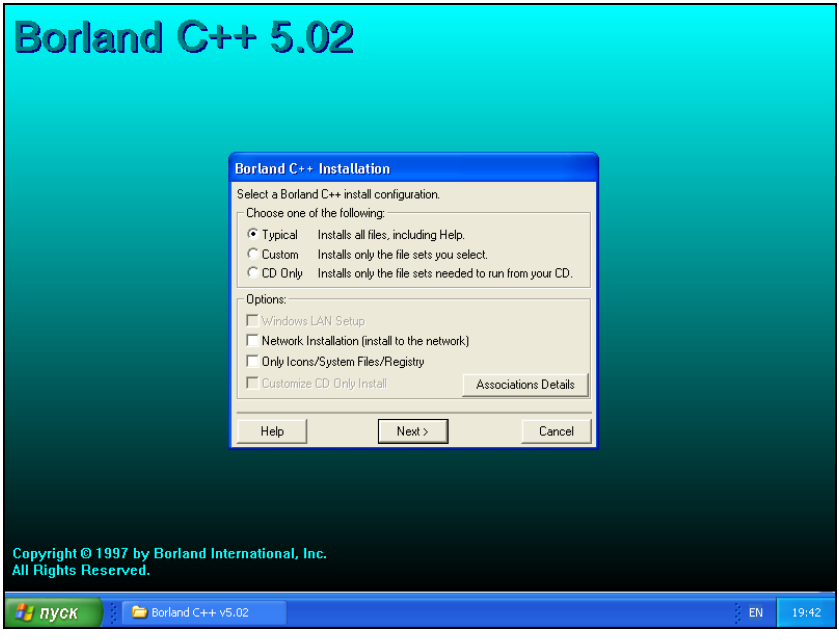



**Рис. 1.1.** Установка среды разработки C++

Затем укажите папку, в которой вы желаете хранить программу, например, C:\BC5, и нажмите кнопку **Next** (Далее). Значения параметров в следующих двух диалоговых окнах также оставьте без изменений.

На этом этап установки приложения завершен. Открыть среду разработки C++ для работы можно с помощью команды меню **Пуск | Все программы | Borland C++ 5.02 | Borland C++** (рис. 1.2).


**Рис. 1.2.** Открытие приложения Borland C++ 5.02

# 1.2. Рабочее окно программы

После открытия на экране вы увидите рабочее окно программы (рис. 1.3), которое состоит из следующих частей:

- меню;
- панель инструментов;
- рабочая область;
- строка состояния.

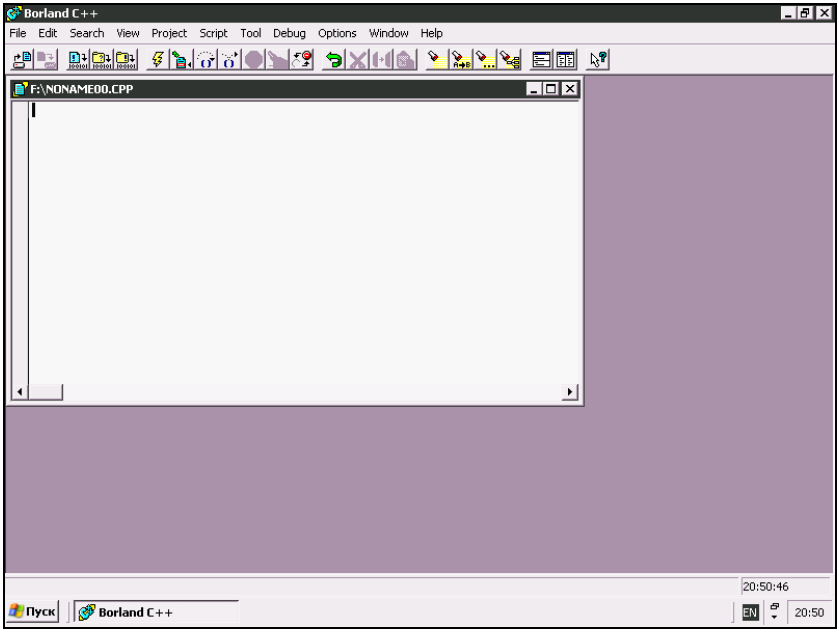



**Рис. 1.3.** Окно C++

В *строке состояния* программа отображает краткие подсказки, связанные с тем или иным объектом.

Приведем назначение некоторых пунктов меню:

- **File** (Файл) — это один из самых главных пунктов меню; он отвечает за открытие и сохранение проекта;

- **Edit** (Правка) — используется для копирования, вставки, вырезания информации, а также для отмены последних операций над текстом;
- **Search** (Поиск) — отвечает за поиск информации в тексте;
- **View** (Вид) — открывает окна, которые помогают при написании программы; подробно об этом пункте меню будет рассказано позже;
- **Project** (Проект) — предназначен для управления созданным проектом;
- **Options** (Опции) — этот пункт меню служит для настройки параметров программы и проекта;
- **Window** (Окно) — помогает расположить окна каскадом, а также переключаться между ними.

## 1.3. Создание и сохранение документа

Чтобы создать новый документ, выберите пункт меню **File | New | Text edit** (Файл | Создать | Текстовая область). В рабочей области появится окно — *редактор кода*, в котором и будет содержаться основной код программы (см. рис. 1.3).

Чтобы сохранить документ, выберите пункт меню **File | Save as...** (Файл | Сохранить как...). В результате откроется окно сохранения документа (рис. 1.4).

В этом окне укажите папку, в которой будет размещаться документ, и имя, под которым он будет храниться. Для подтверждения сохранения нажмите кнопку **Открыть**. Данная команда дублируется нажатием комбинации клавиш <Ctrl>+<K>+<S>.

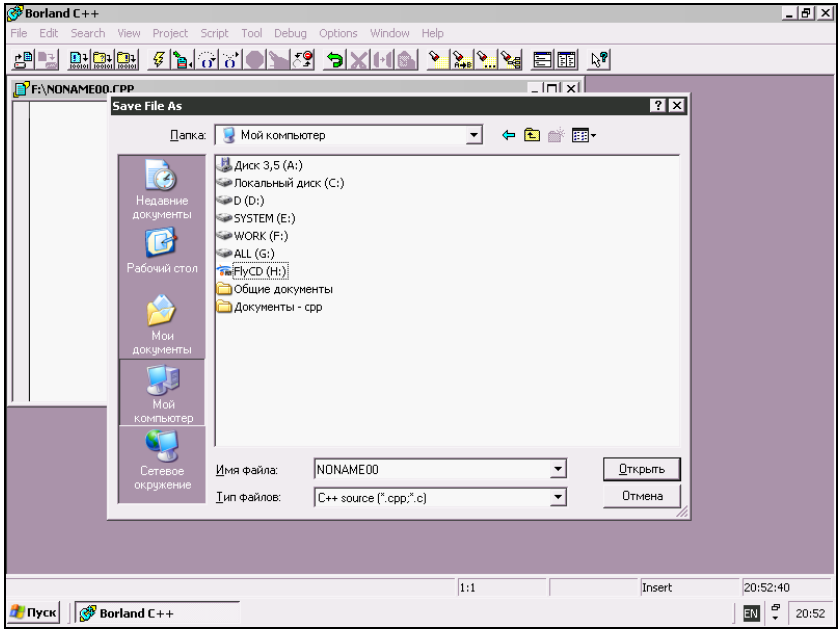



**Рис. 1.4.** Сохранение документа

# 1.4. Компиляция программы

После того как исходный текст набран (это можно сделать в любом текстовом редакторе, хотя существуют и специальные приложения), его необходимо преобразовать в программу, которая и будет выполняться на компьютере. Важно понять, что сам текст только формально описывает алгоритм вычислений — он не является программой.

Дело в том, что процессор понимает только двоичный код, представляющий собой очень простые машинные команды. В машинном коде написать более-менее сложную программу практически невозможно. В обычном языке программирования для операции над парой переменных достаточно одной команды, а процессору на это могут потребоваться десятки машинных команд.

В том, что машинные команды очень просты, а написать приличную программу с их помощью невероятно трудно, нет противоречия. Все детали механической части современного автомобиля совершают очень простые вращательные или поступательные движения, но невозможно создать автомобиль из куска металла с помощью молотка и напильника, нужны специальные инструментальные средства.

Точно так же дело обстоит и в вычислительной технике. Здесь тоже нужны специальные инструментальные средства. Перевод из текста в двоичный код осуществляется специальными программами, которые называются *трансляторами*. Они транслируют, т. е. переводят тексты, написанные на языке программирования, в машинный код.

В вычислительной технике существуют два класса трансляторов: интерпретаторы и компиляторы. *Интерпретатор* работает как синхронный переводчик — просматривает исходный текст строку за строкой, переводит каждую строку в промежуточный или в машинный код и передает его на исполнение. Если ошибок нет, интерпретатор приступает к следующей строке. *Компилятор* (а именно к этому классу относится рассматриваемая нами система программирования Borland C++) работает как литературный переводчик. Сначала он просматривает весь текст, иногда и не один раз, находит общие повторяющиеся места, тщательно готовит стратегию перевода, подбирает самые эффективные аналоги и только после этого переводит весь исходный текст целиком и полностью, создав при этом новый документ, который называется *объектным кодом*. Объектный код можно считать законченной программой, хотя и не вполне.

Итак, *компиляция* — это процесс преобразования исходной программы в исполняемую. Процесс компиляции состоит из двух этапов. На первом этапе выполняется проверка текста программы на отсутствие ошибок, на втором — генерируется исполняемая программа (exe-файл).

После ввода текста функции обработки события и сохранения проекта можно из меню **Project** (Проект) выбрать команду **Compile**

(Компиляция) или выполнить компиляцию, нажав клавишу <F9>. Диалоговое окно компилятора представлено на рис. 1.5.

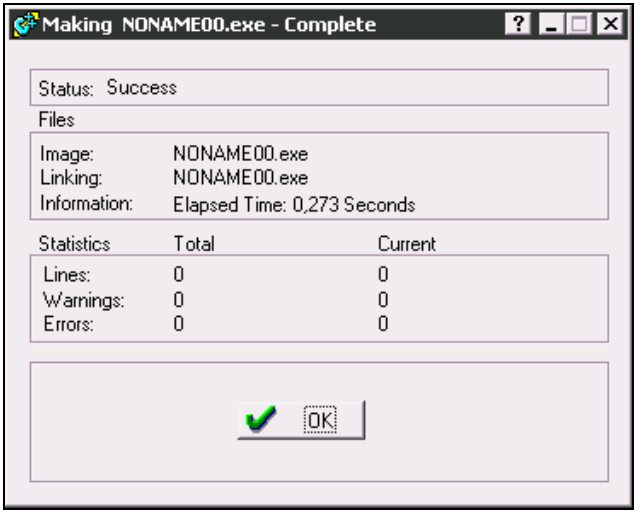



**Рис. 1.5.** Результаты компиляции в диалоговом окне

Если ошибок не найдено (**Errors** = 0), то можно запустить приложение, нажав комбинацию клавиш <Ctrl>+<F9>.

Компиляцию можно осуществлять и в пошаговом режиме с помощью клавиши <F7>. После ее нажатия строка, с которой начинается программа, будет выделена синим цветом. При повторном нажатии клавиши операции, находящиеся в этой строке, будут выполнены, и синим цветом подсветится следующая строка.

Запуск программы можно начать с любой желаемой строки, установив в ней курсор и нажав клавишу <F4>.

# Глава 2

# Основы программирования

В данной главе вы познакомитесь с основами программирования на C++, получите представление о программах и разработке алгоритмов для них.

## 2.1. Этапы разработки программы

Программа, которую выполняет компьютер, нередко отождествляется с самим компьютером, т. к. человек, использующий программу, вводит в компьютер исходные данные (например, при помощи клавиатуры), а компьютер выводит результат на экран, на принтер или в файл. Процессор преобразует исходные данные в результат по определенному алгоритму, который, будучи записан на специальном языке, называется *программой*. Таким образом, чтобы компьютер решил некоторую задачу, необходимо разработать последовательность команд, обеспечивающую решение данной задачи, или, как говорят, написать программу.

Выражение "написать программу" означает только один из этапов создания компьютерной программы, когда разработчик (программист) действительно пишет команды (инструкции) на бумаге или при помощи текстового редактора.

*Программирование* — это процесс создания (разработки) программы, который может быть представлен с помощью последовательности шагов:

1. Спецификация (определение, формулирование требований к программе).

2. Разработка алгоритма.
3. Кодирование (запись алгоритма на языке программирования).
4. Отладка.
5. Тестирование.

## Спецификация

*Спецификация* — определение требований к программе — один из важнейших этапов, на котором подробно описывается исходная информация, формулируются требования к результату, поведение программы в особых случаях (например, при вводе неверных данных).

## Разработка алгоритма

На этапе *разработки алгоритма* необходимо определить последовательность действий, которые надо выполнить для получения результата. Если задача может быть решена несколькими способами, то программист, используя некоторый критерий (например, скорость работы), выбирает наиболее подходящий алгоритм. Результатом этапа разработки алгоритма является его подробное словесное описание или блок-схема.

## Кодирование

После того как определены требования к программе и составлен алгоритм решения, он записывается на выбранном языке программирования. В результате получается исходная программа.

## Отладка

*Отладка* — это процесс поиска и устранения ошибок. Ошибки в программе разделяют на две группы: синтаксические (ошибки в тексте) и алгоритмические. Синтаксические ошибки устранить довольно легко, а алгоритмические ошибки обнаружить труднее. Этап отладки можно считать законченным, если программа правильно работает при любом правильном наборе входных данных.

## Тестирование

*Тестирование* особенно важно, если вы предполагаете, что вашей программой будут пользоваться другие. На этом этапе следует проверить, как ведет себя программа при как можно большем количестве входных наборов данных, в том числе и заведомо неверных.

# 2.2. Элементы языка C++

## Алфавит

Множество символов языка C++ включает прописные и строчные буквы латинского алфавита и цифры.


C++ чувствителен к регистру. Это означает, что "a" и "A" — совсем разные буквы.

Язык программирования, как и любой другой язык, имеет свой словарь. *Словарь языка программирования* представляет собой совокупность зарезервированных или так называемых "ключевых" слов. Все зарезервированные слова содержат только строчные буквы (символы нижнего регистра) и написаны на английском языке.


Пробел используется не только в качестве разделительного знака, но и как символьный знак.

Приведем список ключевых слов языка C++:

| | | | |
|---|---|---|---|
| `asm` | `char` | `delete` | `extern` |
| `auto` | `class` | `do` | `float` |
| `break` | `const` | `double` | `for` |
| `case` | `continue` | `else` | `friend` |
| `catch` | `default` | `enum` | `goto` |

```
if          protected   static      typedef
inline      public      struct      union
int         register    switch      unsigned
long        return      template    virtual
new         short       this        void
operator    signed      throw       volative
private     sizeof      try         while
```

**Примечание**

Использование части ключевых слов будет объяснено в процессе изложения материала, остальные войдут в словарь программиста по мере изучения C++.

Кроме того, в C++ используются следующие символы:

- знаки препинания: `. , : ;`;
- скобки: `( ) [ ] { }`;
- знаки: `| ^ ? _`;
- двойные и одинарные кавычки: `"` и `'`.

Алгоритм заключается в фигурные скобки `{}` после выражения `main()`, в котором круглые скобки показывают, что это программный блок, называемый *функцией*. Обычно программа состоит из нескольких функций, но все же `main()` присутствует всегда, с нее и начинается выполнение программы. Данная функция всегда оканчивается оператором `return 0;`, показывающим, что программа завершена. Вот пример простой программы:

```
main()
{
  return 0;
}
```

**Внимание**

Зарезервированные (ключевые) слова запрещается использовать в качестве имени переменных.

## Комментарии

*Комментарии*, как и в других языках программирования, не обрабатываются компилятором, и поэтому не влияют на выполнение программы. В языке C++ можно записывать комментарии двух видов:

- комментарии можно разместить после двух подряд идущих слэшей `//`, записанных в любом месте строки. Все, что следует после этих символов до конца строки, воспринимается как комментарий. Для продолжения комментария на следующей строке необходимо в ней повторно записать эти два символа;
- перед началом комментария один раз записываются два подряд идущих символа `/*` (открытие комментария), а в конце — два подряд идущих символа `*/` (закрытие комментария). В этом случае комментарии могут занимать несколько строк.

**Совет**

Используйте первый способ для коротких комментариев, а второй — для длинных. Для доступности и облегчения восприятия кода другими программистами и пользователями всегда старайтесь снабжать программу подробными комментариями к коду.

## Переменные

В процессе написания программы программисту приходится использовать *переменные*. Каждая переменная имеет имя, тип, размер и значение.

*Имя переменной* в прямом смысле является ее названием. Имя переменной, или идентификатор, может состоять из латинских букв, цифр и символа подчеркивания. Прописные и строчные буквы в именах различаются. Язык C++ не ограничивает длину идентификатора, однако пользоваться слишком длинными именами типа достаточно неудобно. Чтобы текст вашей программы был более понятным, рекомендуется использовать такие идентификаторы, которые несут какой-либо смысл, поясняя назначение объекта в программе, например, `birth_date` или `salary`.

*Тип* определяет, какие символы или числа записаны в ячейку памяти под этим именем.

*Размер* указывает максимальную величину или точность задания числа.

*Значение* определяет содержимое ячейки памяти.

Более общее определение переменной можно дать следующим образом: переменная — это именованная область памяти, к которой программист имеет доступ из программы.

**Совет**

Как правило, в программах переменные на том или ином шаге принимают разные значения. Для более быстрой и удобной ориентации в их значениях можно использовать окно **Watch** (Часы). Чтобы открыть данное окно, можно воспользоваться пунктами меню **View | Watch** (Вид | Часы). Для добавления некоторой переменной в окно нажмите комбинацию клавиш <Ctrl>+<A> и введите имя переменной. По завершению ввода нажмите клавишу <Enter>. Теперь при пошаговом выполнении программы вы сможете увидеть значение помещенной в окно переменной на данном шаге.

## Типы данных

Основным принципом типизации, принятым в языках программирования и базах данных, является то, что любая константа, переменная, выражение и функция относится к некоторому типу, характеризующему прежде всего множество значений, к которым относятся константы, которые могут принимать переменные и выражения и которые могут формировать функции. При описании любых используемых констант, переменных и функций явно или неявно указывается их тип. В первую очередь это дает возможность компилятору и/или системе управления базами данных выделить для хранения объекта данных ровно тот объем памяти, который определяется допустимым диапазоном значений типа.

Однако концепция типа этим не исчерпывается. Следующим исключительно важным свойством типа данных является инкапсуляция внутреннего представления его значений (*инкапсуляция* — это механизм защиты данных от несанкционированного доступа;

она будет рассмотрена в *главе 8*). К значению типа данных (констант, переменных, выражений и функций) можно обращаться только с помощью операций, предопределенных в описании этого типа. Эти операции могут быть явными (например, арифметические операции `+`, `-`, `*` и `/` для числовых типов) или неявными (например, операция преобразования значения целого типа к значению типа с плавающей запятой). В некоторых языках (в C и C++ тоже) допускаются и явные преобразования типов.

Наличие типовых описаний констант, переменных и функций и предписанные правила определения типов выражений вместе с поддержкой свойства инкапсуляции типов дают возможность компиляторам языков программирования и языков баз данных производить существенный контроль допустимости языковых конструкций на этапе компиляции, что позволяет сократить число проверок на стадии выполнения программ и облегчить их отладку.

Перечислим существующие категории типов.

- *Встроенные типы данных*, т. е. типы, предопределенные в языке программирования. Обычно в языке фиксируются внешнее представление значений этих типов (вид литеральных констант) и набор операций с описанием их семантики. Внутреннее представление и реализация операций выбираются в конкретных компиляторах и подсистемах поддержки выполнения программ.
- *Конструируемые типы* (иногда их называют составными) обладают той особенностью, что в языке предопределены средства спецификации таких типов и некоторый набор операций, дающих возможность доступа к компонентам составных значений.
- *Указательные типы* дают возможность работы с типизированными множествами абстрактных адресов переменных, содержащих значения некоторого типа. В языках с более слабой типизацией (например, C и C++) допускаются практически неограниченные манипуляции указателями.

В C++ также реализован набор базовых типов данных, представленных в табл. 2.1.

***Таблица 2.1.*** *Базовые типы данных*

| Имя типа | Размер в байтах | Допустимое значение |
|---|---|---|
| `char` | 1 байт | От –128 до 127 |
| `int` | Зависит от реализации | |
| `short int` | 2 байта | От –32 768 до 32 767 |
| `long int` | 4 байта | От –2 147 483 648 до 2 147 483 647 |
| `unsigned char` | 1 байт | От 0 до 255 |
| `unsigned int` | Зависит от реализации | |
| `unsigned short int` | 2 байта | От 0 до 65 535 |
| `unsigned long int` | 4 байта | От 0 до 4 294 967 295 |
| `float` | 4 байта | От ~3.4e – 38 до ~3.4e + 38 |
| `double` | 8 байтов | От ~1.7e – 308 до ~1.7e + 308 |
| `long double` | 10 байтов | От ~3.4e – 4932 до ~1.1e + 4932 |

Отметим, что ключевые слова `signed` и `unsigned` необязательны. Они указывают, как интерпретируется нулевой бит объявляемой переменной, т. е. если указано ключевое слово `unsigned`, то нулевой бит интерпретируется как часть числа, в противном случае нулевой бит интерпретируется как знаковый. В случае отсутствия ключевого слова `unsigned` целая переменная считается знаковой. В том случае, если спецификатор типа состоит из ключевого типа `signed` или `unsigned` и далее следует идентификатор переменной, то она будет рассматриваться как переменная типа `int`.

Чтобы назначить переменной тот или иной тип, необходимо указать его перед первым использованием переменной. Например:

```
unsigned int A;
int B;          /* подразумевается signed int B */
unsigned C;     /* подразумевается unsigned int C*/
```

Так же можно по одному типу определять несколько переменных, тогда список переменных следует указать через запятую.

Простое определение переменной не задает ее начального значения. Если объект определен как глобальный, спецификация C++ гарантирует, что он будет инициализирован нулевым значением. Если же переменная локальная либо динамическая, ее начальное значение не определено, т. е. она может содержать некоторое случайное значение.

**Совет**

Довольно часто после написания и исполнения программы полученный результат оказывается неверным, хотя алгоритм правильный. Данная ошибка возможна из-за того, что в начале программы не было задано значение некоторой переменной. Поэтому рекомендуется явно указывать начальное значение переменной, по крайней мере, в тех случаях, когда неизвестно, может ли объект инициализировать сам себя.

## Преобразование типов

При выполнении операций производится автоматическое преобразование типов, чтобы привести операнды выражений к общему типу или чтобы расширить короткие величины до размера целых величин, используемых в машинных командах. Выполнение преобразования зависит от специфики операций и от типа операндов.

Рассмотрим общие арифметические преобразования:

- операнды типа `float` преобразуются к типу `double`;
- если один операнд — `long double`, то второй преобразуется к этому же типу;
- если один операнд — `double`, то второй также преобразуется к типу `double`;
- любые операнды типа `char` и `short` преобразуются к типу `int`;
- любые операнды `unsigned char` или `unsigned short` преобразуются к типу `unsigned int`;
- если один операнд — `unsigned long`, то второй преобразуется к типу `unsigned long`;

- если один операнд — `long`, то второй преобразуется к типу `long`;
- если один операнд — `unsigned int`, то второй операнд преобразуется к этому же типу.

Таким образом, при вычислении выражений операнды преобразуются к типу того операнда, который имеет наибольший размер.

## Константы

*Константы*, в отличие от переменных, не могут изменяться программой. Записываются они по следующим правилам:

- вещественные константы можно записать в обычной форме, используя символ . (точка) для разделения целой и дробной части (`0.54`) или в экспоненциальной форме (`0.5e1`, `-0.12345e+2`, `987e-5`);
- целочисленные константы можно записать в десятичной (`-26`) или шестнадцатеричной системе счисления (`0x1A`);
- символьные константы записываются в одинарных кавычках (`'5'`, `'A'`, `'\t'`, `'\r'`);
- строковые константы записываются в двойных кавычках (`"Line"`).

**Примечание**

Грамматический контроль строк не выполняется.

Константы можно определить одним из следующих способов:

- непосредственно записать в выражении;
- с помощью ключевого слова `const`, например, `const Long=5`. Тогда в выражении вместо константы `5` указывается идентификатор `Long`. Этот способ имеет следующее преимущество. Если одна и та же константа в программе встречается несколько раз, то достаточно изменить ее значение один раз при объявлении;
- с помощью директивы препроцессора `#define`, например, `#define Long 5`. Директива заменяет каждое появление сим-

волов `Long` на `5`. Подробней о директивах препроцессора будет рассказано позже.

## Арифметические и логические операции

В C++ используются *арифметические* и *логические операции*. Их перечень приведен в табл. 2.2.

***Таблица 2.2.*** *Арифметические и логические операции*

| **Название операции** | **Символ, который используется в коде** |
|---|---|
| Арифметическое сложение | `+` |
| Арифметическое вычитание | `-` |
| Умножение | `*` |
| Деление | `/` |
| Отрицание | `!` |
| Присваивание | `=` |
| Вычисление остатка | `%` |
| Логическое сложение | `&&` |
| Логическое умножение | `\|\|` |
| *Проверка на равенство* | |
| Равно | `= =` |
| Не равно | `!=` |
| *Проверка отношения* | |
| Больше | `>` |
| Меньше | `<` |
| Больше или равно | `>=` |
| Меньше или равно | `<=` |

Подробнее остановимся на операции присваивания. Имена переменных соответствуют областям в памяти компьютера. Когда переменной присваивается новое значение, оно замещает значение, ранее хранившееся в ячейке памяти. Например:

```
a=3; b=12; c=a+b; a=1; b=-10;
```

При выполнении операций присваивания переменная `a` получает значение `3`, а переменная `b` — значение `12`.

В результате операции сложения в переменную `c` записывается число `15`. Затем `a` присваивается число `1`, а переменной `b` — число `-10`, при этом старое значение переменных утрачивается.

В C++ имеются сокращенные записи арифметических операций, представленные в табл. 2.3.

***Таблица 2.3.*** *Сокращенные записи арифметических операций*

| Сокращенный вид операции | Полный вид операции |
|---|---|
| `c+=5` | `c=c+5` |
| `c-=5` | `c=c-5` |
| `c*=5` | `c=c*5` |
| `c/=5` | `c=c/5` |
| `c%=5` | `c=c%5` |
| `c++` | `c=c+1` |
| `++c` | `c=c+1` |
| `c-` | `c=c-1` |
| `-c` | `c=c-1` |

**Совет**

Рекомендуется использовать сокращенные записи арифметических операций в целях экономии времени набора кода.

# 2.3. Директивы препроцессора C++

Обработка программы препроцессором происходит перед ее компиляцией. Управление работой препроцессора осуществляется при помощи его директив.

*Директивы препроцессора* представляют собой инструкции, записанные в тексте программы на C++ и выполняемые до трансляции программы. Директивы препроцессора позволяют изменить текст программы, например, заменить некоторые лексемы в тексте, вставить текст из другого файла, запретить трансляцию части текста и т. п. Все директивы препроцессора начинаются со знака `#`. После директив точка с запятой не ставится.

Пустая директива состоит из одного символа `#`. Данная инструкция игнорируется.

## Директива *#include*

Директива `#include` включает в текст программы содержимое указанного файла.

Эта директива имеет две формы:

```
#include "имя_файла"
#include <имя_файла>
```

Имя файла должно соответствовать соглашениям операционной системы и может состоять либо только из имени файла, либо из имени файла с предшествующим ему маршрутом. Если имя файла указано в кавычках, то поиск файла осуществляется в соответствии с заданным маршрутом, а при его отсутствии — в текущем каталоге. Если имя файла задано в угловых скобках, то поиск файла производится в стандартных каталогах операционной системы, задаваемых командой `PATH`.

Директива `#include` имеет следующие свойства:

- она может быть вложенной, т. е. во включаемом файле тоже может содержаться директива `#include`, которая замещается после включения файла, содержащего эту директиву;

- она широко используется для включения в программу так называемых *заголовочных файлов*, содержащих прототипы библиотечных функций, и поэтому большинство программ на C начинаются с этой директивы.

## Директива *#define*

Директива `#define` служит для замены часто использующихся констант, ключевых слов, операторов или выражений некоторыми идентификаторами. Идентификаторы, заменяющие текстовые или числовые константы, называют *именованными константами*. Идентификаторы, заменяющие фрагменты программ, называют *макроопределениями*, причем макроопределения могут иметь аргументы.

Деректива `#define` имеет две синтаксических формы:

```
#define идентификатор текст
#define идентификатор (список параметров) текст
```

Эта директива заменяет идентификатор на текст при всех последующих вхождениях. Такой процесс называется *макроподстановкой*. Текст может представлять собой любой фрагмент программы на C++ либо вообще отсутствовать. В последнем случае все экземпляры идентификатора удаляются из программы.

Во второй синтаксической форме в директиве `#define` имеется список формальных параметров, который может содержать один или несколько идентификаторов, разделенных запятыми. Формальные параметры в тексте макроопределения отмечают позиции, на которые должны быть подставлены фактические аргументы макровызова. Каждый формальный параметр может появиться в тексте макроопределения несколько раз.

При макровызове вслед за идентификатором записывается список фактических аргументов, количество которых должно совпадать с количеством формальных параметров.

## Директива *#undef*

Директива `#undef` используется для отмены действия директивы `#define`.

Директива отменяет действие текущего определения `#define` для указанного идентификатора. Не является ошибкой использование директивы `#undef` для идентификатора, который не был определен директивой `#define`.

## Директива *#error*

Директива препроцессора `#error` имеет следующий синтаксис:

`#error` *`msg`*

Директива печатает в процессе компиляции сообщение об ошибке вида:

`Error:` *`filename`* `line# : Error directive:` *`msg`*

Здесь *`msg`* — сообщение, заданное директивой `#error`.

## Директива *#typedef*

Данная директива позволяет задать синоним для встроенного либо пользовательского типа данных. Например:

```
typedef double Wages;
typedef int Coll;
```

Имена, определенные с помощью директивы `#typedef`, можно использовать точно так же, как спецификаторы типов:

```
typedef double Wages;
main()
{
  Wages i, j, k;
  i=j+k;
}
```

Эта директива начинается с ключевого слова `typedef`, за которым идет спецификатор типа, и заканчивается идентификатором, который становится синонимом для указанного типа.

Применяя мнемонические имена для типов данных, можно сделать программу более легкой для восприятия. Кроме того, принято употреблять такие имена для сложных составных типов, воспринимаемых с трудом.

# 2.4. Ввод/вывод

В ходе работы приложения программисту необходимо вводить некоторые данных и получать ответ — так называемые выходные данные. В C++ ввод и вывод на экран осуществляется при помощи команд `cin` и `cout` соответственно. Чтобы использовать команды `cin` и `cout`, необходимо подключить заголовочный файл:

```
#include <iostream.h>
```

Теперь рассмотрим подробнее команды `cin`, `cout`:

```
cout<<элемент1<<элемент2<<...<<элементN;
```

В качестве элемента могут использоваться следующие величины:

- переменная одного из указанных выше типов, например, `cout<<My_res;`
- строковая константа, т. е. текст, который должен записываться в двойных кавычках. В тексте можно использовать так называемые *управляющие символы*, признаком которых является обратная наклонная черта. Один из них, `\n`, означает, что информация, выводимая после него, будет размещена на следующей строке. Этот символ может располагаться не только в начале строковой константы, но и в середине, в конце, строковая константа может также содержать только этот единственный символ;
- числовая константа, например, `cout<<5;`
- выражение, заключенное в круглые скобки, например, `cout<<(a+b);`
- ключевое слово `endl`, которое называют *манипулятором вывода*, означает, что после этого слова информация будет выводиться с новой строки.

Рассмотрим на примере отличие `/n` от `endl`.

```
#include <iostream.h>   // Заголовочный файл
main()
{
  // Пусть определены две переменные с некоторым значением
```

```
  int a=2, b=5;
  cout<<"\n Сумма "<<(a+b)<<"\n Разность "<<(a-b);
  // или
  cout<<endl<<" Сумма "<<(a+b)<<endl<<" Разность "<<(a-b);
  return 0;              // Завершает программу
}
```

В результате на экране пользователь увидит число 7 (сумма `a+b`), а в следующей строке — число –3 (разность `a-b`).

Результат выполнения вывода обоими способами будет одинаковый. Следовательно, имеет смысл оставить один из них.

Ввод соответственно происходит следующим образом:

```
cin>>элемент1>>элемент2>>...>>элементN;
```

Здесь в качестве элемента может быть переменная, но не выражение и не константа. При выполнении этого оператора программа останавливается и ждет ввода необходимого количества данных.

**Примечание**

Для вывода можно использовать функцию `printf`, а для ввода — функцию `scanf`, подключив заголовочный файл с помощью директивы `#include <stdio.h>`.

# 2.5. Управляющие структуры

Любая программа в C++ может быть написана с использованием трех управляющих структур:

- структуры выбора;
- структуры повторения;
- структуры следования.

## Структуры выбора

В C++ существует несколько структур выбора. Все они могут использоваться при написании программ. При их помощи ставятся те или иные условия, в результате выполнения/невыполнения которых производятся те или иные операции.

## Структура выбора *if/else*

На взгляд многих программистов, самой простой структурой выбора является структура выбора `if`. Рассмотрим два вида этой структуры: полный и сокращенный.

Полная форма выглядит следующим образом:

```
if (условие)
{
  операция1;
  операция2;  // Выполняется, если условие верно
  ...
  операцияN;
} else
{
  операция1;
  операция2;  // Выполняется, если условие не верно
  ...
  операцияN;
}
```

Здесь под условием понимается равенство либо неравенство некоторых переменных или выражений. Если это условие выполняется, то компилятор начинает совершать операции, заключенные в фигурные скобки после самого условия, в противном случае компилятор перейдет к операциям после ключевого слова `else`.

**Примечание**

Если необходимо, чтобы после выполнения условия совершилась одна операция, то заключать ее в фигурные скобки не требуется.

Сокращенная форма имеет следующий вид:

```
if (условие)
{
  оператор1;
  оператор2;
```

```
  ...
  операторN;
}
```

В сокращенной форме отсутствует ключевое слово `else` и, соответственно, вторая последовательность операторов. Если выражение истинно, то последовательность из *N* операторов выполняется. В противном случае она пропускается и выполняется следующий после `if` оператор.

Аналогично решается вопрос с фигурными скобками и в этом случае.

Пример записи условия выглядит так:

```
if (a==b) cout<<"a=b";   // Если a равно b, то выводит...
if (a!=b) cout<<"a!=b";  // Если a не равно b, то выводит...
```

При необходимости задать несколько условий можно воспользоваться логическим сложением `&&`:

```
if (условие1 && условие2 && ... && условиеN) операция;
```

Для более компактной записи структуры выбора `if/else` применяется тернарная операция `? :`. Общий вид операции следующий:

```
выражение1 ? выражение2 : выражение3;
```

Операция выполняется следующим образом: вычисляется *выражение1*, если оно истинно, то вычисляется *выражение2*, иначе вычисляется *выражение3*. Например,

```
if (k>5) My_res=10; else My_res=x*y;
```

Вид этого выражения в тернарном варианте будет следующим:

```
My_res=k>5 ? 10 : x*y;
```

Не для любого оператора `if` можно записать тернарную операцию. Во-первых, ее можно использовать для замены только полной формы условного оператора. Во-вторых, в одной и другой ветви операции (после символов `?` и `:`) можно указать по одному выражению.

## Структура выбора *switch*

Структура выбора `switch` является более удобной для использования. Общая форма структуры выбора `switch` значительно отличается от `if`:

```
char k;  // k типа char
switch(k)
{  // Начало переключателя
  case constant:
    {операция1; операция2; ...; операцияN} break;
  case constant:
    {операция1; операция2; ...; операцияN} break;
  case constant:
    {операция1; операция2; ...; операцияN} break;
  default: {операция1; операция2; ...; операцияN} break;
}   // Конец переключателя
```

Выражение, записываемое после ключевого слова `switch` в скобках, может быть целого, символьного или перечислимого типа. Чаще всего это просто целочисленная или символьная переменная. Нельзя использовать выражение вещественного или строкового типов. В константном выражении можно использовать константы и нельзя записывать выражения, в том числе переменные или вызовы функций. Здесь обычно указывается целая или символьная константа.

Работает оператор следующим образом. Значения константных выражений вычисляются во время компиляции. Во время выполнения вычисляется значение выражения и сравнивается с первой константой на строгое равенство. Если не совпало, оно же сравнивается со второй константой и т. д. Если значение выражения совпало с одной из констант, выполняется соответствующая последовательность операторов, заключенных в фигурные скобки (если операция одна, то фигурные скобки не обязательны), затем команда `break` передает управление оператору, следующему за `switch`. Если `break` отсутствует, то выполняется следующая последовательность операторов, хотя значение выражения и не

совпадает со следующей константой. Обязательность `break` для пропуска остальных ветвей — основная особенность этого оператора. В случае отсутствия совпадений выполняются операторы, записанные после метки `default:`, которая не является обязательной. Если метки `default:` нет в теле переключателя `switch`, то при отсутствии совпадений ничего не выполняется.

**Примечание**

Допускается вложенность операторов `switch`.

**Совет**

При большом количестве вложенных проверок используйте структуру выбора `switch`.

## Примеры структур выбора

### Пример 1

Рассмотрим программу, которая, в зависимости от номера месяца, выводит на экран одно из слов: `Зима`, `Весна`, `Лето` или `Осень`.

```
#include <iostream.h>    // Заголовочный файл
#include <conio.h>       // Заголовочный файл
main()
{
  int n;
  cin>>n;      // Требует ввода целого числа
  if (n>=3 && n<=5) cout<<"Весна"; else
    if (n>=6 && n<=8) cout<<"Лето"; else
      if (n>=9 && n<=11) cout<<"Осень"; else cout<<"Зима";
  getch();     // Задерживает экран
  return 0;    // Завершает программу
}
```

Программа довольно простая, но в то же время имеет несколько нюансов. Для упрощения работы с алгоритмом прочитаем его по-русски.

Это выглядит примерно так:

*Если "Весна", то выводим результат, иначе,*

*если "Лето", то выводим результат, иначе,*

*если "Осень", то выводим результат, иначе выводим "Зима".*

Как только какое-либо условие будет выполнено, дальнейшие проверки производиться не будут благодаря `else`.

Если выполнить эту задачу на компьютере, то после вывода на экран окно с отображением результата будет автоматически закрываться. С помощью функции `getch()` выполнение программы приостанавливается, и можно анализировать выведенный на экран результат, пока не будет нажата любая клавиша. Эта функция использует заголовочный файл conio.h, который подключается следующим образом:

```
#include <conio.h>
```

## Пример 2

Даны три монеты достоинством 1, 5, 10 копеек. Рассмотрим программу, которая определит, можно ли набрать заданную сумму, используя только две монеты из трех.

```
#include <iostream.h>
#include <conio.h>
main()
{
  int n;
  cin>>n;      // Требует ввода целого числа
  switch(n)
  {
    case 1+5:  cout<<"Yes"; break;
    case 1+10: cout<<"Yes"; break;
    case 5+10: cout<<"Yes"; break;
    default:   cout<<"No";
  }
  getch();     // Задерживает экран
  return 0;    // Завершает программу
}
```

В этой задаче с помощью структуры выбора `switch` перечисляются все доступные суммы, и если одна из них совпадет с `n`, то будет выведен результат и программа завершится.

Запишем решение задачи при помощи структуры выбора `if` и отследим различия.

```
#include <iostream.h>
#include <conio.h>
main()
{
  int n;
  cin>>n;      // Требует ввода целого числа
  if (1+5==n) cout<<"Yes"; else
    if (1+10==n) cout<<"Yes"; else
      if (5+10==n) cout<<"Yes"; else cout<<"No";
  getch();     // Задерживает экран
  return 0;    // Завершает программу
}
```

Данное решение аналогично предыдущему, но является менее предпочтительным в плане громоздкости кода.

## Пример 3

Рассмотрим программу, которая определяет и выводит на экран максимальное число из трех введенных.

```
#include <iostream.h>
#include <conio.h>
main()
{
  int a,b,c;
  cin>>a>>b>>c;     // Требует ввода трех чисел
  if (a>b && a>c) cout<<a;   // Если a>b и a>c, то выводит a
  if (b>a && b>c) cout<<b;   // Если b>a и b>c, то выводит b
  if (c>b && c>a) cout<<c;   // Если c>b и c>a, то выводит c
  getch();          // Задерживает экран
  return 0;         // Завершает программу
}
```

Это тривиальное решение. Можно оптимизировать алгоритм и уменьшить количество проверок.

```
#include <iostream.h>
#include <conio.h>
main()
{
  int a,b,c;
  cin>>a>>b>>c;    // Требует ввода трех чисел
  if (a>b && a>c) cout<<a; else
    if (b>c) cout<<b; else cout<<c;
  getch();         // Задерживает экран
  return 0;        // Завершает программу
}
```

## Пример 4

В этом примере приводится программа, которая по двум введенным целым числам и знаку операции над ними (`-`, `+`, `*`, `/`) выполняет соответствующее математическое действие.

Пример ввода:

```
8
*
2
```

Пример вывода:

```
16
```

Рассмотрим решение этой задачи двумя способами — при помощи структур выбора `if` и `switch`.

```
#include <iostream.h>
#include <conio.h>
main()
{
  int a,b;
  char z;
  cin>>a>>z>>b;  // Требует ввода двух чисел и знака операции
```

```
  // Проверяет соответствие знака и выполняет
  // соответствующую операцию
  if (z=='-') cout<<(a-b); else
    if (z=='+') cout<<(a+b); else
      if (z=='*') cout<<(a*b); else cout<<(a/b);
  getch();        // Задерживает экран
  return 0;       // Завершает программу
}
```

**Примечание**

Здесь переменная `z` типа `char` (символ). Этот тип применяется для хранения буквы, цифры или другого символа из множества символов. При присваивании или проверке используются одинарные кавычки.

```
#include <iostream.h>
#include <conio.h>
main()
{
  int a,b;
  char z;
  cin>>a>>z>>b;  // Требует ввода двух чисел и знака операции
  switch(z)      // Связывает переменную z и переключатель switch
  {
    // Ищет нужный знак и выполняет соответствующую операцию
    case '-': cout<<(a-b); break;
    case '+': cout<<(a+b); break;
    case '*': cout<<(a*b); break;
    case '/': cout<<(a/b); break;
  }
  getch();        // Задерживает экран
  return 0;       // Завершает программу
}
```

**Примечание**

В теле переключателя `switch` отсутствует метка `default`, т. к. в нашем случае данные корректны, и ввод произойдет обязательно.

### Пример 5

Даны три числа. Рассмотрим программу, которая из трех чисел найдет и выведет на экран такую пару чисел, сумма которых максимальна.

```
#include <iostream.h>
#include <conio.h>
main()
{
  int a, b, c;
  cin>>a>>b>>c;  // Требует ввода трех чисел
  // Определяет максимальную сумму из a+b, a+c, b+c
  if ((a+b)>(a+c) && (a+b)>(b+c)) cout<<(a+b); else
    if ((b+c)>(a+c)) cout<<(b+c); else cout<<(a+c);
  getch();       // Задерживает экран
  return 0;      // Завершает программу
}
```

# 2.6. Структуры повторения

Структура повторения помогает программисту определить действие, которое должно повторяться при соблюдении некоторых условий.

## Структура повторения *for*

Общий вид оператора цикла `for` следующий:

```
for (выражение1; выражение2; выражение3)
{
 операция1;
 операция2;
 ...
 операцияN;
}
```

Рассмотрим работу цикла на примере:

```
for (i=1; i<=5; i++)
```

Данная запись означает, что цикл будет продолжать свою работу до тех пор, пока `i` меньше либо равно пяти, с шагом, равным единице, т. е. `i` изменяется от 1 до 5 (1, 2, 3, 4, 5):

- `i=1;` — начальное значение `i`;
- `i<=5;` — условие работы цикла;
- `i++;` — означает, что на каждом шаге к `i` будет прибавляться единица.

## Структура повторения *while*

Структура повторения `while` (пока) определяет действие, которое должно продолжаться, пока некоторое условие остается истинным. Особенность этой структуры в том, что сначала проверяется условие, а потом выполняются операции. Если условие ложно на первом шаге, то первый цикл не выполнится.

Структура повторения `while` имеет следующий вид:

```
while (условие)
{
  операция1;
  операция2;
  ...
  операцияN;
}
```


Значения всех переменных, входящих в выражение, должны быть определены до выполнения оператора `while`.

### Оператор цикла *do...while* с условием

Его общий вид такой:

```
do
{
  операция1;
  операция2;
```

```
  ...
  операцияN;
}
while (условие);
```

Работает оператор следующим образом. Сначала выполняется тело цикла, а затем вычисляется и проверяется значение выражения. Если оно истинно, то операции повторяются. В противном случае цикл прекращается.

В отличие от `while`, повторяющаяся часть обязательно выполнится хотя бы один раз.

Вторая особенность в том, что значения переменных выражения не обязательно определять до цикла, это можно сделать внутри него.

## Операторы *continue* и *break*

Оператор `continue` пропускает оставшуюся часть тела цикла. В операторах `while` и `do...while` он передает управление повторной проверке условия цикла, а затем, если условие выполнено, продолжается работа цикла.

В случае оператора `for` производится увеличение, затем проверяется условие и, если оно истинно, выполняется тело цикла. Например,

```
for (i=0; i<N; i++)
{
  // Тело цикла
  cout<<i;
  cout<<endl;
}
```

Таким образом, цикл начнет свою работу при `i=0`, и если условие `i<N` выполнится, то компилятор перейдет к выполнению операций, расположенных в теле данного цикла. Затем значение переменной `i` увеличится на единицу, после чего снова будет проверено выполнение условия `i<N`, и операции будут выполнены снова, в противном случае работа цикла будет завершена.

Оператор `break` обеспечивает прекращение выполнения самого внутреннего из объединяющих его операторов `switch`, `do`, `for`, `while`. После выполнения оператора `break` управление передается оператору, следующему за прерванным.

Рассмотрим на примере эти два оператора. Приведенная программа вычисляет сумму только отрицательных чисел.

```
#include <iostream.h>
#include <conio.h>
main()
{
  int sum, i, n, ch;
  sum=0;
  cin>>n;        // Требует ввода количества чисел
  for (i=0; i<n; i++)
  {
    cin>>ch;     // Требует ввода очередного числа
    if (ch>=0) continue; // Если число положительно либо 0,
                         // подается команда
                         // не выполнять суммирование
    sum+=ch;     // Суммирует
   }
   cout<<sum;    // Выводит результат
   getch();      // Задерживает экран
   return 0;     // Завершает программу
}
```

В качестве примера использования оператора `break` приведем программу, которая уведомляет о том, что пользователь ввел единицу, и прекращает ввод данных после этого.

```
#include <iostream.h>
#include <conio.h>
main()
{
  int i, n, ch;
```

```
  cin>>n;        // Требует ввода количества чисел
  for (i=0; i<n; i++)
  {
    cin>>ch;     // Требует ввода очередного числа
    if (ch==1) break;  // Если данное число равно единице,
                       // то выходит из цикла
  }
  getch();       // Задерживает экран
  return 0;      // Завершает программу
}
```

## Оператор *goto*

Использование оператора безусловного перехода `goto` в практике программирования на языке C++ настоятельно не рекомендуется, т. к. он затрудняет понимание программ и возможность их модификаций.

Формат этого оператора следующий:

```
goto имя_метки;
...
имя_метки: оператор;
```

Оператор `goto` передает управление оператору, помеченному меткой *имя_метки*. Он должен находиться в теле той же функции, что и оператор `goto`, а используемая метка должна быть уникальной, т. е. одно имя метки не может быть использовано для разных операторов программы. Имя метки — это идентификатор.

Любой оператор в составном операторе может иметь свою метку. Используя оператор `goto`, можно передавать управление внутрь составного оператора. Но нужно быть внимательным при входе в составной оператор, содержащий объявления переменных с инициализацией, т. к. объявления располагаются перед выполняемыми операторами, и значения переменных при таком переходе не будут определены.

# Примеры структур повторения

## Пример 1

Представим программу, которая осуществляет вывод на экран таблицы умножения.

```
#include <iostream.h>
#include <conio.h>
main()
{
  int i, j;
  for (i=2; i<=9; i++)
  {
    for (j=1; j<=10; j++)
      cout<<(i*j)<<" ";  // Выводит очередной элемент таблицы
    cout<<endl;          // Перемещает курсор на новую строку
  }
  getch();               // Задерживает экран
  return 0;              // Завершает программу
}
```

Рассмотрим предложенный алгоритм подробнее. Если необходимо вывести таблицу умножения до 9 включительно, то внешний цикл должен перебирать все значения от 0 до 9 с шагом 1, а внутренний в свою очередь — все числа от 0 до 10:

```
for (i=2; i<=9; i++)
{
  for (j=1; j<=10; j++)
```

На каждом шаге будет получено два числа, произведение которых следует вывести:

```
cout<<(i*j)<<" ";
```

После того как внутренний цикл закончил свою работу, необходимо перевести курсор на новую строку:

```
  cout<<endl;
}
```

После того как оба цикла закончат свою работу, таблица умножения будет выведена на экран. Чтобы пользователь смог проверить работу программы и оценить выведенные данные, необходимо использовать функцию задержки экрана:

```
getch();
```

После нажатия любой клавиши программа завершится:

```
return 0;
```

## Пример 2

Рассмотрим программу, которая по введенному числу *n* вычислит *n*!.

**Примечание**

Из математики известно, что $n! = 1 \times 2 \times ... \times n$.

Код данной программы выглядит следующим образом:

```
#include <iostream.h>
#include <conio.h>
main()
{
 int n, My_res, i;
 My_res=1;        // Начальное значение переменной
 cin>>n;          // Требует ввод числа n
 for (i=2; i<=n; i++)
   My_res=My_res*i;
 cout<<My_res;    // Выводит результат на экран
 getch();         // Задерживает экран
 return 0;        // Завершает программу
}
```

Проследим работу алгоритма. Определим такую переменную `My_res`, в которой будет храниться произведение числа `i` на ранее полученное произведение на `i`-ом шаге. Вначале значение переменной равно единице:

```
My_res=1;
```

Для перебора всех чисел отрезка [2; *n*] будем использовать структуру повторений `for`. На `i`-ом шаге будем умножать текущее произведение (`My_res`) на `i`.

```
for (i=2; i<=n; i++)
  My_res=My_res*i;
```

Левая граница отрезка равна не единице, а двойке, т. к. бессмысленно умножать единицу на единицу.

Проследим изменения переменной `My_res` при `n=3`.

```
My_res=1;          // Начальный шаг
i=2;               // Берется первое число из отрезка
My_res=1*2=2;      // Вычисляется произведение
i=3;               // Берется следующее число из отрезка
My_res=2*3=6;      // Вычисляется произведение
```

## Пример 3

Рассмотрим программу, которая выводит на экран все простые числа из отрезка [*a*; *b*].

**Примечание**

*Простым числом* называется число, которое делится лишь на единицу и на само себя без остатка.

Далее приведен код данной программы с комментариями.

```
#include <iostream.h>
#include <conio.h>
main()
{
  int a, b, i, j, p;
  cin>>a>>b;              // Требует ввод концов отрезка
  for (i=a; i<=b; i++)  // Перебор всех элементов отрезка
  {
    for (j=2; j<i/2+1; j++)  //Перебор всех возможных
      // делителей элемента отрезка.
      // Если элемент отрезка делится без остатка на j, то
```

```
      // запоминает, что число уже не простое (p=1).
      if (i%j==0) p=1;
      // Если число простое, то выводит результат на экран,
      // иначе обнуляет признак p.
    if (p==0) cout<<i<<" "; else p=0;
  }
  getch();   // Задерживает экран
  return 0;  // Завершает программу
}
```

Алгоритм решения данной задачи следующий. На первом шаге необходимо перебрать все элементы отрезка (`i` — текущий элемент) и все возможные делители элемента (`j` — текущий делитель). Очевидно, что делителями числа `i` могут являться только те числа, которые находятся на отрезке от `2` до `i/2+1`. Если некоторое число `j` является делителем числа `i`, то флаг `p` становится равным единице. Если по завершению цикла, который перебирает все возможные делители числа `i`, флаг `p` остался равным `0`, то число `i` — простое, в противном случае `i` является составным.

```
for (i=a; i<=b; i++)
{
  for (j=2; j<i/2+1; j++)
    if (i%j==0) p=1;
```

Из вышеуказанной части алгоритма видно, что число является простым, если по завершению работы внутреннего цикла переменная `p` осталась равной нулю. Когда все элементы проверены, программа завершается.

```
    if (p==0) cout<<i<<" "; else p=0;
  }
  getch();
  return 0;
}
```

**Примечание**

Данный алгоритм не очень эффективен для больших чисел. С более эффективным алгоритмом вы сможете познакомиться позднее.

## Пример 4

В данном примере представлена программа, которая по трем введенным числам определит и выведет на экран число, имеющее в своем составе больше всего единиц.

```
#include <iostream.h>
#include <conio.h>
main()
{
  int a, b, c, temp, kol_vo_1, kol_vo_2, kol_vo_3;
  int copy_a, copy_b, copy_c;
  cin>>a>>b>>c;  // Требует ввода трех чисел
  kol_vo_1=0;    // Обнуляет переменную
  kol_vo_2=0;    // Обнуляет переменную
  kol_vo_3=0;    // Обнуляет переменную
  copy_a=a;      // Снимает копию с числа a
  copy_b=b;      // Снимает копию с числа b
  copy_c=c;      // Снимает копию с числа c
  // Запускаем цикл while, который будет работать, пока
  // значение переменной copy_a не станет равным нулю
  while (copy_a!=0)
  {
    temp=copy_a%10;  // Отделяет последнюю цифру числа
    copy_a/=10;      // Удаляет последнюю цифру из числа
    // Если цифра, которую мы отделили, равна единице,
    // то увеличивает счетчик
    if (temp==1) kol_vo_1++;
  }
  // Запускаем цикл while, который будет работать, пока
  // значение переменной copy_b не станет равным нулю
  while (copy_b!=0)
  {
    temp=copy_b%10;  // Отделяет последнюю цифру числа
    copy_b/=10;      // Удаляет последнюю цифру из числа
    // Если цифра, которую мы отделили, равна единице,
    // то увеличивает счетчик
```

```
    if (temp==1) kol_vo_2++;
  }
  // Запускаем цикл while, который будет работать, пока
  // значение переменной copy_c не станет равным нулю
  while (copy_c!=0)
  {
    temp=copy_c%10;  // Отделяет последнюю цифру числа
    copy_c/=10;      // Удаляет последнюю цифру из числа
    // Если цифра, которую мы отделили, равна единице,
    // то увеличивает счетчик
    if (temp==1) kol_vo_3++;
  }
  // Находит максимальное значение среди значений счетчиков
  // и выводит на экран число, соответствующее номеру счетчика
  if (kol_vo_1>kol_vo_2 && kol_vo_1>kol_vo_3) cout<<a; else
    if (kol_vo_2>kol_vo_3) cout<<b; else cout<<c;
  getch();   // Задерживает экран
  return 0;  // Завершает программу
}
```

На первом шаге алгоритм снимает копии с введенных чисел. Это необходимо для того, чтобы не потерять их значения:

```
copy_a=a;
copy_b=b;
copy_c=c;
```

Теперь следует проверить все цифры каждого числа, если некоторая из них равна единице, то будем увеличивать счетчик на единицу. Каждой переменной назначим свой счетчик (`kol_vo1` — первой, `kol_vo2` — второй, `kol_vo3` — третьей). Проверить каждую цифру числа можно при помощи вычисления остатка от деления числа на `10`. После проверки цифры необходимо удалить ее из числа. Это можно сделать, округлив результат деления. Итак, необходим цикл, который на каждом шаге будет выделять последнюю цифру числа, сравнивать ее с единицей и затем удалять:

```
while (copy_a!=0)
{
```

```
  temp=copy_a%10;
  copy_a/=10;
  if (temp==1) kol_vo_1++;
}
```

Цикл продолжает свою работу, пока число не станет равным нулю. Это будет означать, что все цифры числа проверены.

Данную операцию необходимо выполнить еще для двух чисел:

```
while (copy_b!=0)
{
  temp=copy_b%10;
  copy_b/=10;
  if (temp==1) kol_vo_2++;
}
while (copy_c!=0)
{
  temp=copy_c%10;
  copy_c/=10;
  if (temp==1) kol_vo_3++;
}
```

Затем сравниваем переменные `kol_vo1`, `kol_vo2` и `kol_vo3` и выводим на экран число, которое соответствует счетчику с большим значением:

```
  if (kol_vo_1>kol_vo_2 && kol_vo_1>kol_vo_3) cout<<a; else
    if (kol_vo_2>kol_vo_3) cout<<b; else cout<<c;
  getch();
  return 0;
}
```

## Пример 5

Рассмотрим программу, которая определяет, является ли введенное число палиндромом.

**Примечание**

*Палиндромом* называется число, которое читается слева направо и справа налево абсолютно одинаково. Примером такового являются числа 121, 444, 5, 10001 и т. д.

Решение данной задачи достаточно простое — необходимо перевернуть введенное число и сравнить его с оригиналом. Для этого на каждом шаге будем брать остаток от деления числа на 10, получая таким образом последнюю цифру, и добавлять его в конец нового числа (рис. 2.1).

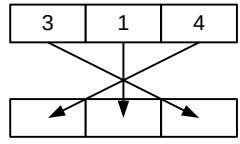



**Рис. 2.1.** Перестановка цифр

Код этой программы выглядит следующим образом:

```
#include <iostream.h>
#include <conio.h>
main()
{
  int a, b, copy_a;
  cin>>a;    // Требует ввода числа
  copy_a=a;  // Снимает копию с a
  b=0;
  // Запускаем цикл while, который будет работать, пока
  // значение переменной copy_a не станет равным нулю
  while (copy_a!=0)
  {
    // Приписывает в конец переменной b последнюю цифру
    // числа copy_a
    b=b*10+copy_a%10;
    copy_a/=10;  // Удаляет последнюю цифру числа copy_a
  }
  // Если число a и "перевертыш" числа a равны, значит, число
  // a — палиндром
  if (a==b) cout<<"Yes"; else cout<<"No";
  getch();   // Задерживает экран
  return 0;  // Завершает программу
}
```

## Пример 6

Напишем программу, которая определит и выведет на экран количество нулей и единиц в двоичном представлении числа.

Вот код данной программы:

```
#include <iostream.h>
#include <conio.h>
main()
{
  int N, kol_zero, kol_one;
  cin>>N;     // Требует ввода числа
  while (N!=0)
  {
    if (N%2==0) kol_zero++; else kol_one++;
    N/=2;
  }
  // Выводит результат
  cout<<"Количество нулей: "<<kol_zero
  cout<<endl<<"Количество единиц: "<<kol_one;
  getch();    // Задерживает экран
  return 0;   // Завершает программу
}
```

Двоичное представление числа можно получить путем выписывания остатков от деления числа `N` на `2`. Например:

```
13=1011
```

Действительно:

```
13%2=1; 13/2=6;
6%2=0;  6/2=3;
3%2=1;  3/2=1;
1%2=1;  1/2=0;
```

Следовательно, необходимо вести вычисления до тех пор, пока число `N` не станет равным нулю, т. е. организовать цикл `while`.

Назначим два счетчика `kol_zero` и `kol_one`, подсчитывающих количество нулей и единиц в двоичной записи числа.

Затем будем проверять остаток от деления текущего значения `N` на `2`, и если окажется, что он равен нулю, то к переменной `kol_zero` прибавляем единицу, в противном случае увеличиваем на единицу значение переменной `kol_one`:

```
if (N%2==0) kol_zero++; else kol_one++;
```

Так как остаток уже найден, разделим значение переменной `N` на `2`:

```
  N/=2;
}
```

Итак, результат определен и готов к выводу:

```
  cout<<"Количество нулей: "<<kol_zero
  cout<<endl<<"Количество единиц: "<<kol_one;
  getch();
  return 0;
}
```

# 2.7. Упражнения

1. Напишите программу, которая проверит, станет ли число *a* палиндромом, если к нему в конец приписать одну из цифр от 1 до 9.
2. Напишите программу, которая определит и выведет на экран `НОД(a; b)`.
3. Напишите программу, которая выводит на экран словесную форму числа. Например: 143 = "Сто сорок три", 10 = "Десять".
4. Напишите программу, которая поможет выучить таблицу умножения. Суть программы в том, что она будет печать на экране:

   ```
   Сколько будет 6*7?
   ```

   Затем вводится ответ. Если ответ правильный, то программа печатает:

   ```
   МОЛОДЕЦ!!!!
   ```

   Если же неверно, то печатает

   ```
   Попробуй еще раз
   ```

   и задает новый вопрос.

5. Напишите программу, которая удалит из введенного числа все цифры, равные $N$ ($N$ вводится с клавиатуры), и выведет результат на экран.
6. Напишите программу, которая определит среднее арифметическое всех чисел из отрезка $[a; b]$.
7. Напишите программу, которая определит и выведет на экран один сомножитель из $N$, который можно вычеркнуть из произведения $1! \times 2! \times 3! \times ... \times N!$ так, чтобы оставшееся произведение было точным квадратом.
8. Напишите программу, которая определит, могут ли числа $a$, $b$ и $c$ являться сторонами треугольника.
9. Пусть дано число $N$. Известно, что его можно разбить на сумму трех, не равных нулю чисел. Напишите программу, которая выведет на экран все такие разложения. Например, число 6 можно получить двумя способами: 1 + 2 + 3, 2 + 2 + 2. При этом следующие разложения считать одинаковыми: 1 + 2 + 3, 1 + 3 + 2, 2 + 3 + 1, 2 + 1 + 3, 3 + 2 + 1, 3 + 1 + 2 и т. п.
10. Будем называть два числа "близнецами", если они простые и разность между ними не превышает двух. Найдите и выведите на экран все пары таких чисел из отрезка $[a; b]$. Примером такой пары могут служить числа 5 и 7, а так же 11 и 13.


Во всех задачах *главы 2* входные данные следует считать корректными.

# Глава 3

# Функции

В данной главе вы познакомитесь с понятием функции, ее свойствами и методами применения в проектах.

Программирование с использованием функций является весьма эффективным и удобным. В программе, которая не содержит отдельных функций, тяжело разобраться, даже если каждое действие снабжено комментарием. При помощи функций можно быстро модифицировать программу, т. к. код сокращается.

## 3.1. Понятие функции

*Функция* — это совокупность объявлений и операторов, обычно предназначенная для решения определенной задачи. Каждая функция должна иметь имя, которое используется для ее объявления, определения и вызова. В любой программе на C++ должна быть функция с именем `main()` (главная функция), именно с этой функции, в каком бы месте программы она не находилась, начинается выполнение программы.

В программах на языке C++ широко используются так называемые *библиотечные функции*, т. е. функции, предварительно разработанные и записанные в библиотеки. Прототипы библиотечных функций находятся в специальных заголовочных файлах, поставляемых вместе с библиотеками в составе систем программирования, и включаются в программу с помощью директивы `#include`.

*Тело функции* — это составной оператор, содержащий операторы, определяющие действие функции.

*Прототип функции* — это явное объявление функции, которое предшествует определению функции. Тип возвращаемого значения при объявлении функции должен соответствовать типу возвращаемого значения в прототипе функции.

Все переменные, объявленные в теле функции без указания класса памяти, имеют класс памяти `auto`, т. е. они являются *локальными*. Управление передается первому оператору тела функции, и начинается выполнение функции, которое продолжается до тех пор, пока не встретится оператор `return` или последний оператор тела функции. Управление при этом возвращается в точку, следующую за точкой вызова, а локальные переменные становятся недоступными. При новом вызове функции для локальных переменных память распределяется вновь, и поэтому старые значения локальных переменных теряются.

## Свойства функции

Перечислим некоторые свойства функций:

- функция является программным блоком;
- функция может быть выполнена как в виде подпрограммы, так и в виде самостоятельной программы;
- функция может иметь или не иметь возвращаемые значения и аргументы;
- прототип функции служит ее объявлением. Функция должна быть объявлена до ее первого использования (вызова);
- в теле функции могут быть вызваны другие функции.

## Встроенные функции

*Встроенные функции* разделены на несколько групп. Прототипы функций размещены в отдельном заголовочном файле.

Компилятор использует прототипы функций для проверки правильности их вызова. Ошибки вызова фиксируются при компиляции.

Прежде чем использовать ту или иную функцию, необходимо проверить, для каких типов данных она разработана и в какой заголовочный файл включена.

## Перегрузка функций

Язык C++ является очень чувствительным к типам данных. Для функций определены типы возвращающих значений и типы аргументов. Если при вызове функции использовать аргумент непредусмотренного типа, то компилятор преобразует тип по правилам приведения типов или зафиксирует ошибку.

C++ позволяет определить несколько функций с одним и тем же именем, но с различным набором аргументов, что называется *перегрузкой функций*.

Например, в стандартной библиотеке C++ есть три отдельные функции для определения абсолютного значения выражений:

```
int abs(int);
double fabs(double);
long labs(long);
```

Гораздо удобнее было бы вызвать одну функцию для выполнения всех вариантов.

Для осуществления перегрузки необходимо записать три варианта прототипа новой функции и три варианта описания функции. Прототипы функций записываются в виде комментария. Для перегруженной функции будем использовать имя `restart`:

```
// int restart(int);        Прототип чисел типа int
// double abs1(double);     Прототип чисел типа double
// long abs1(long);         Прототип чисел типа long
int restart(int i)          // Для чисел типа int
{return abs(i)};
double restart(double i)    // Для чисел типа double
{return fabs(i);}
long restart(long i)        // Для чисел типа long
{return labs(i);}
```

При вызове функции

```
x=restart(y);
```

компилятор определит тип переменной `y` и выполнит нужный вариант.

Перегруженную функцию можно сохранить на жестком диске под некоторым именем, например, rest.h. После этого к программе можно будет подключать заголовочный файл rest.h для использования перегруженной функции `restart`.

## Арифметические и алгебраические функции

Встроенные арифметические и алгебраические функции приведены в табл. 3.1.

***Таблица 3.1.*** *Встроенные арифметические и алгебраические функции*

| Функция | Запись | Описание | Необходимый заголовочный файл |
|---|---|---|---|
| `ceil` | `int ceil(Extended x);` | Округление вверх | math.hpp |
| `exp` | `double exp(double x)` | Экспонента | math.h |
| `fabs` | `double fabs(double x)` | Абсолютное значение | math.h |
| `floor` | `int floor(Extended X);` | Округление вниз | math.hpp |
| `fmod` | `Double fmod(double x, double y)` | Остаток от деления `x` на `y` | math.h |
| `sqrt` | `double sqrt(double x)` | Квадратный корень | math.h |

## Тригонометрические функции

Встроенные тригонометрические функции приведены в табл. 3.2.

***Таблица 3.2.** Встроенные тригонометрические функции*

| Функция | Запись | Описание | Необходимый заголовочный файл |
|---|---|---|---|
| `acos` | `double acos(double x)` | Арккосинус | math.h |
| `asin` | `double asin(double x)` | Арксинус | math.h |
| `atan` | `double atan(double x)` | Арктангенс | math.h |
| `cos` | `double cos(double x)` | Косинус | math.h |
| `sin` | `double sin(double x)` | Синус | math.h |
| `tan` | `double tan(double x)` | Тангенс | math.h |

**Примечание**

Приведенные выше прототипы функций, например, `double tan (double x)`, означают, что функция требует аргумент `x` типа `double` и возвращает значение типа `double`.

# 3.2. Функция-подпрограмма

Каждый программист имеет возможность создавать свои функции, если набор стандартных функций недостаточен для разработки больших и сложных проектов.

Существенной особенностью языка C++ является то, что *функции не могут быть вложенными*. Говоря другими словами, невозможно определить одну функцию внутри другой. В таком случае говорят, что все функции находятся на одном уровне видимости. Вызвать одну функцию из другой, конечно же, можно.

Для использования функций необходимо выполнить следующее:

1. Перед главной функцией `main()` записать прототип (объявление, заголовок) функции. Например,

   ```
   void name_of_function();
   ```

   После прототипа функции обязательно ставить *точку с запятой*. Здесь *`name_of_function`* — это программное имя функции. Круглые скобки означают, что функция не имеет никаких параметров, ни входных, ни выходных.

2. Описание функции разместить после текста функции `main()`. Следует повторить заголовок, но *без точки с запятой* в конце, а затем записать ее текст. При необходимости объявляются так называемые локальные переменные, которые можно использовать только внутри данной функции.

Вот пример описания функции:

```
void name_of_function()
{
  // Тело функции
}
```

При вызове функции ей при помощи аргументов *(формальных параметров)* могут быть переданы некоторые значения *(фактические параметры)*, используемые во время выполнения функции. Она также может возвращать некоторое значение. Допускается использование функций, не имеющих аргументов и не возвращающих никаких значений. Действие их может состоять, например, в изменении значений некоторых переменных, выводе на печать некоторых текстов и т. п.

Выполнение вызова функции происходит в описанном ниже порядке.

1. Вычисляются значения выражений в списке. Затем, если известен прототип функции, тип полученного фактического аргумента сравнивается с типом соответствующего формального параметра. Если они не совпадают, то либо производится преобразование типов, либо формируется сообщение об ошибке.

Число выражений должно совпадать с числом формальных параметров, если только функция не имеет переменного числа параметров. В последнем случае проверке подлежат лишь обязательные параметры. Если в прототипе функции указано, что ей не нужны параметры, а при вызове они указаны, формируется сообщение об ошибке.

2. Происходит присваивание значений фактических параметров соответствующим формальным параметрам.
3. Управление передается на первый оператор функции.
4. Оператор `return` в теле функции передает управление и вычисленное значение в вызывающую функцию. При отсутствии `return` управление возвращается после выполнения последнего оператора тела функции, а возвращаемое значение не определено.


Функция должна быть объявлена до ее первого использования.

Далее приведен пример простой программы, в которой присутствует функция-подпрограмма, вычисляющая сумму введенных чисел `a` и `b`.

```
# include <iostream.h>
# include <conio.h>
void summ(int a, int b);  // Прототип функции
main()
{
  int a, b;
  cin>>a>>b;     // Требует ввода двух чисел
  summ(a,b);     // Подает в функцию с параметрами a и b
  getch();       // Задерживает экран
  return 0;      // Завершает программу
}
void summ(int a, int b)  // Описание функции
{
  cout<<(a+b);             // Выводит сумму
}
```

## Передача параметров функции *main()*

Функция `main()`, с которой начинается выполнение программы, может быть определена с параметрами, которые передаются из внешнего окружения, например, из командной строки. Во внешнем окружении действуют свои правила представления данных, а точнее, все данные представляются в виде строк символов. Для передачи этих строк в функцию `main()` используются два параметра, первый из которых служит для передачи числа передаваемых строк, второй — для передачи самих строк. Общепринятые (но не обязательные) имена этих параметров — `argc` и `argv`.

Функция `main()` может иметь и третий параметр, `argp`, служащий для передачи в функцию `main()` параметров операционной системы, в которой выполняется программа.

# 3.3. Ссылочный тип

Сначала рассмотрим самостоятельное использование переменной этого типа, так называемую *независимую ссылку*.

Например, две переменные объявлены следующим образом:

```
int x;
int &y=x;
```

Здесь символ `&` означает, что переменная `y` — ссылочного типа.

После такого объявления идентификатор `y` назначает другое, альтернативное имя ячейке, названной `x`, т. е. `y` и `x` — равнозначны. Иногда говорят, что ссылочная переменная — это псевдоним переменной, к которой она обращается.

Поэтому при выводе этих чисел `cout<<x<<" "<<y;` на экран будут выведены два одинаковых числа. Если прибавить к значению переменной `x` некоторое число, то оно будет прибавлено и к переменной `y`.

**Примечание**

Аналогичное событие произойдет, если число прибавить к переменной `y`, т. е. переменные `x` и `y` сравняются.

## Возврат значений из функции с помощью переменных ссылочного типа

Чтобы возвратить некоторые значения из функции, можно использовать ссылочный тип следующим образом:

1. В заголовке функции в круглых скобках объявить переменные ссылочного типа.
2. При вызове функции в качестве параметров указать простые переменные соответствующего типа, объявленные обычным образом.

Аналогично ссылочный тип используется и в случае, если значение переданной в функцию переменной надо изменить, и это измененное значение требуется возвратить в точку вызова.

Приведем наглядный пример возврата значений из функции с помощью ссылочного типа.

```
void Summa(int &number1, int &number2)
{
  number1++;
  number2++;
}
```

А вызов функции будет таким:

```
Summa(a, b);
```

Тогда после команды `cout<<a<<" "<<b;` на экран будут выведены исходные числа, увеличенные на единицу.

Так как `number1` и `a` — это одна и та же ячейка, то изменение `number1` автоматически приведет к изменению переменной `a`. Аналогично изменение переменной `number2` приведет к изменению переменной `b`.

# 3.4. Рекурсия

*Рекурсивным* называется объект, частично состоящий или определяемый с помощью самого себя. *Факториал* — это классиче-

ский пример рекурсивного объекта. Факториал числа $n$ — это произведение целых чисел от 1 до $n$. Обозначается факториал числа $n$ так: $n!$.

Согласно определению,

$$n! = 1 \times 2 \times 3 \times ... \times (n-1) \times n.$$

Приведенное выражение можно переписать следующим образом:

$$n! = n \times ((n-1) \times (n-2) \times ... \times 3 \times 2 \times 1) = n \times (n-1)!$$

То есть факториал числа $n$ равен произведению числа $n$ на факториал числа $(n-1)$. В свою очередь факториал числа $(n-1)$ — это произведение числа $(n-1)$ на факториал числа $(n-2)$ и т. д.

Таким образом, если вычисление факториала числа $n$ реализовать как функцию, то в теле этой функции будет инструкция вызова функции вычисления факториала числа $(n-1)$, т. е. функция будет вызывать сама себя. Такой способ вызова называется *рекурсией*, а функция, которая обращается сама к себе, называется рекурсивной функцией. Рекурсивная функция работает, пока значение $n - i$ не станет равным единице.

Далее приведено решение задачи.

```
#include <iostream.h>
#include <conio.h>
#include <process.h>
int i, ch, N, My_res;
void factorial(int ch);   // Прототип функции
main()
{
  cin>>N;          // Требует ввода числа
  My_res=1;        // Начальное значение результата
  factorial(N);    // Вызывает рекурсивную функцию
                   // с начальными параметрами
}
void factorial (int ch)   // Описание функции
{
  if (ch==1)
```

```
  {
    cout<<My_res;  // Выводит результат
    getch();       // Задерживает экран
    exit(1);       // Прерывает работу программы
  }
  My_res*=ch;      // Умножает результат на очередное число
  factorial(ch-1); // Функция вызывает саму себя
                   // с новым параметром
}
```

**Примечание**

В данной программе используется новый заголовочный файл process.h и новая встроенная функция `exit(1)`, которая прерывает работу программы.

## Шаблоны функций

В C++ существуют *шаблоны* (templates) функций, которые позволяют создавать универсальные программы.

Рассмотрим простейший пример шаблона функции. Описанию функции должен предшествовать список параметров шаблона:

```
template <class A, class B, ..., class Z>
```

Здесь `A`, `B`, ..., `Z` — условные обозначения типов переменных.

# 3.5. Примеры программ с использованием функций

## Пример 1

Рассмотрим задачу о Ханойских башнях. Имеются три стержня с номерами 1, 2, 3. На стержень 1 надето *n* дисков различного диаметра так, что они образуют пирамиду (рис. 3.1). Требуется написать программу для печати последовательности перемещений дисков со стержня на стержень, необходимых для переноса пи-

рамиды со стержня 1 на стержень 2 при использовании стержня 3 в качестве вспомогательного. При этом за одно перемещение должен переноситься только один диск, и диск большего диаметра не должен помещаться на диск меньшего диаметра. Доказано, что для $n$ дисков минимальное число необходимых перемещений равно $2^n - 1$.

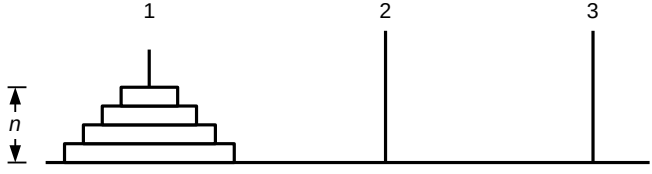



**Рис. 3.1.** Ханойские башни

В простейшем случае, когда пирамида состоит только из одного диска, необходимо выполнить одно действие — перенести диск со стержня 1 на стержень 2. В общем случае будем переносить $n$ дисков со стержня $i$ на стержень $j$, считая стержень $w$ вспомогательным. Сначала следует перенести $n - 1$ диск со стержня $i$ на стержень $w$ при вспомогательном стержне $j$, затем перенести один диск со стержня $i$ на стержень $j$ и, наконец, перенести $n - 1$ диск из $w$ на стержень $j$, используя вспомогательный стержень $i$. Итак, задача о переносе $n$ дисков сводится к двум задачам о переносе $n - 1$ диска и одной простейшей задаче. Схематически это можно записать так:

$$T(n, i, j, w) = T(n - 1, i, w, j) + T(1, i, j, w) + T(n - 1, w, j, i).$$

На рис. 3.2 изображена последовательность вызовов функции `rec(int,int,int,int)`.

Рассмотрим полный код программы с комментариями.

```
#include <iostream.h>
#include <conio.h>
void tn(int, int, int, int);  // Прототип функции
main()
{
```

```
int n;
  cin>>n;          // Требует ввода количества дисков
  rec(n,1,2,3);    // Запускает рекурсивную функцию
                   // с начальными параметрами
  getch();         // Задерживает экран
  return 0;        // Завершает программу
}
void rec(int n, int i, int j, int w) // Описание функции
{
  if (n>1)             // Если дисков больше чем один, то...
  {
    rec (n-1,i,w,j);  // Вызывает функцию с новыми параметрами
    rec (1,i,j,w);    // Вызывает функцию с новыми параметрами
    rec (n-1,w,j,i);  // Вызывает функцию с новыми параметрами
  }
  else cout<<i<<" "<<j<<endl;  // Выводит результат
  return;                      // Завершает работу функции
}
```

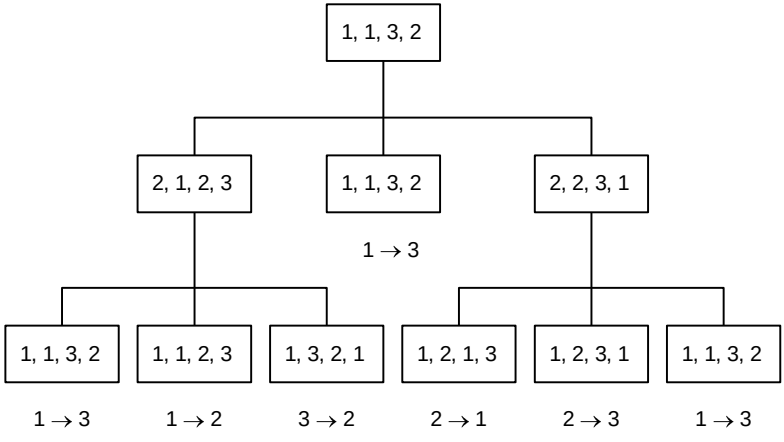



**Рис. 3.2.** Последовательность вызова функции

## Пример 2

Рассмотрим перегруженную функцию `swap(x,y)`, которая меняет местами значения переменных `x` и `y`.

Перед применением перегруженной функции необходимо сначала написать заголовочный файл swap.h, который и будет менять местами значения переменных. Далее приведен его код.

```
void swap(int &x, int &y)          // Для чисел типа int
{int temp=x; x=y; y=temp;};        // Меняет местами значения
                                   // переменных
void swap(double &x, double &y)    // Для чисел типа double
{double temp=x; x=y; y=temp;};
void swap(long &x, long &y)        // Для чисел типа long
{long temp=x; x=y; y=temp;};
```

Этот файл следует сохранить на жестком диске под именем swap.h.

В основной программе необходимо подключить файл swap.h для использования функции `swap(x,y)`. Код основной программы выглядит следующим образом:

```
#include <iostream.h>
#include <conio.h>
#include "F:/Swap.h"  // Созданный заголовочный файл
main()
{
  cout<<"x= ";  // Выводит сообщение о готовности принять
                // первый операнд
  cin>>x;       // Требует ввода первого операнда
  cout<<"y =";  // Выводит сообщение о готовности
                // принять второй операнд
  cin>>y;       // Требует ввода второго операнда
  // Выводит значения переменных до обмена
  cout<<"До обмена:"<<endl<<"x= "<<x<<" y= "<<y;
  swap(x,y);    // Вызывает функцию для обмена
                // значений двух переменных
```

```
  // Выводит значения переменных после обмена
  cout<<"После обмена"<<endl<<"x= "<<x<<" y= "<<y;
  getch();      // Задерживает экран
  return 0;     // Завершает программу
}
```

## Пример 3

Программа с использованием функции, определяющая из двух введенных чисел то, которое имеет максимальную сумму цифр.

Предлагается рекурсивный вариант решения такой задачи.

```
#include <iostream.h>
#include <conio.h>
int rec(int);    // Прототип функции
int Sum_P, Sum_L, Sum;
main()
{
  int K, L;
  cin>>K>>L;     // Требует ввода двух чисел
  Sum_P=rec(K);  // Возвращает сумму цифр числа P
  Sum_L=rec(L);  // Возвращает сумму цифр числа L
  if (Sum_P>Sum_L) cout<<K; else cout<<L;  // Выводит на экран
                                           // максимальную из сумм
  getch();       // Задерживает экран
  return 0;      // Завершает программу
}
int rec(int Number)
{
  Sum+=Number%10;
  Number=Number/10;
  if (Number==0) return Sum;
  rec(Number);
}
```

Данная программа состоит из рекурсивных функций типа `int` и одной главной функции `main()`. Так как в программе используются функции, то необходимо записать их прототип:

```
int rec(int);
```

Главная функция считывает с экрана два операнда, а затем каждый из них передает как параметр в функцию:

```
cin>>K>>L;
Sum_P=rec(K);
Sum_L=rec(L);
```

Когда функция `rec(int)` завершит свою работу и компилятор снова перейдет к главной функции, то последует выбор наибольшей суммы и вывод числа, которое ей соответствует:

```
  if (Sum_P>Sum_L) cout<<K; else cout<<L;
  getch();
  return 0;
}
```

Рассмотрим рекурсивную функцию.

Функция суммирует все цифры числа за несколько шагов. На первом шаге отделяется последняя цифра числа и прибавляется к общей сумме цифр. После того как цифра просуммирована, она удаляется из числа. Значение общей суммы присваивается переменной `Sum`:

```
int rec(int Number)
{
  Sum+=Number%10;
  Number=Number/10;
```

Далее необходимо проверить, все ли цифры числа просуммированы. Если да, то функция завершает свою работу, иначе следует рекурсивный вызов функции с новым параметром. Под новым параметром понимается число без последней цифры на данном шаге. Проверку выполняет оператор `if`:

```
  if (Number==0) return Sum;
  rec(Number);
}
```

## Пример 4

Программа нахождения наибольшего общего делителя (НОД) двух чисел.

```
#include <iostream.h>
#include <conio.h>
void NOD(int, int);  // Прототип функции
main()
{
  int K, L;
  cin>>K>>L;     // Требует ввода двух операндов
  NOD(K,L);      // Запускает функцию с двумя параметрами,
                 // для вычисления НОД(a,b)
  getch();       // Задерживает экран
  return 0;      // Завершает программу
}
void NOD(int a, int b)     // Описание функции
{
  if (a>b) a=a-b; else b=b-a;  // Вычитает из большего меньшее
  if (b==0)
  {
    cout<<a;
    return;      // Завершение работы функции
  }
  NOD(a,b);      // Функция рекурсивно вызывает себя
                 // с новыми значениями параметров
}
```

Сначала записывается прототип функции, затем вводятся два числа, НОД которых требуется найти, а после вызывается сама функция:

```
void NOD(int, int);
cin>>K>>L;
NOD(K,L);
```

Известно, что самым быстрым и удобным способом нахождения наибольшего общего делителя является *алгоритм Евклида*. Суть

его состоит в том, чтобы на каждом шаге вычитать из большего числа меньшее, причем, если числа окажутся равными, то из b вычитается a:

```
void NOD(int a, int b)
{
  if (a>b) a=a-b; else b=b-a;
```

Данная операция продолжается до тех пор, пока число b не станет равным нулю, после чего последуют вывод ответа и завершение работы функции:

```
if (b==0)
{
  cout<<a;
  return;
}
```

На каждом шаге, если b! = 0, то функция вызывает себя с новыми значениями параметров, т. е. числами после вычитания.

```
NOD(a,b);
```

Далее алгоритм повторяется.

Проследим работу алгоритма на некоторых двух числах:

```
a=12; b=8;
```

1. **Шаг 1.** Делаем проверку, какое из чисел больше:

   12 > 8, следовательно, a = 12 – 8 = 4.

   Результаты после шага 1: a = 4; b = 8.

2. **Шаг 2.** Делаем проверку, какое из чисел больше:

   8 > 4, следовательно, b = 8 – 4 = 4.

   Результаты после шага 2: a = 4; b = 4.

3. **Шаг 3.** Делаем проверку, какое из чисел больше:

   4 = 4, следовательно, b = 4 – 4 = 0.

   Результаты после шага 3: a = 4; b = 0.

Так как b = 0, то алгоритм завершает свою работу, а НОД(a, b) = a = 4.

# Пример 5

Рассмотрим программу, которая определяет число Фибоначчи под номером *N* и проверит, является ли число возрастающим. Число называется *возрастающим*, если каждая его последующая цифра равна ли больше всех предыдущих.

*N*-ое число ряда Фибоначчи определяется по формуле:

$$P_N = (F^N - G^N)/2,$$

где

```
F=(1-sqrt(5))/2; G=(1+sqrt(5)/2);
```

Далее приведен код программы:

```
#include <iostream.h>
#include <conio.h>
#include <math.h>
void Fibonachi(int, int&);  // Прототип функции
void Test(int, int&);       // Прототип функции
main()
{
  int N;
  int Number;
  int &My_res=Number;
  int p;
  int &m=p;  // Объявляет ссылку на переменную p
  cin>>N;    // Требует ввода операнда
  Fibonachi(N, Number);  // Вызывает функцию двух параметров,
                         // которая определяет
                         // N-ое число Фибоначчи
  cout<<Number<<endl;    // Выводит N-ое число Фибоначчи
  Test(Number, p);       // Вызывает функцию, которая
                         // проверяет, является ли число
                         // возрастающим
  // Выводит результат
  if (p==1) cout<<"No"; else cout<<"Yes";

  getch();    // Задерживает экран
```

```
  return 0;   // Завершает программу
}
void Fibonachi(int K, int &My_res)  // Описание функции
{
  float x, y, F, G;
  int i;;
  x=(1-sqrt(5))/2;
  y=(1+sqrt(5))/2;
  G=F=1;
  for (i=0; i<K; i++)
  {
    F*=x;  // Возводит число x в степень N
    G*=y;  // Возводит число y в степень N
  }
  My_res=fabs(F-G)/2;  // N-ое число Фибоначчи
}
// Вместо использования цикла можно написать My_res=(x^K+y^K)/2;,
// но важно научиться применять различные способы
// организации вычислений
void Test(int L, int &m)
{
  int last=0, temp=0;
  last=L%10;          // Выделяет последнюю цифру числа
  L/=10;              // Удаляет последнюю цифру из числа
  temp=L%10;          // Выделяет последнюю цифру числа
  if (L==0) return;   // Если все цифры числа просмотрены,
                      // то завершает работу функции
  if (last<temp)      // Если последняя цифра меньше предыдущей,
  {                   // то признак становится равным 1
    m=1;
    return;           // Завершает работу функции
  }
  Test(L,m);          // Функция вызывается рекурсивно
}
```

Передавать число Фибоначчи под номером *N* из функции `Fibonachi(int,int&)` в главную функцию `main()` будем при помощи ссылочного типа переменной.

Рассмотрим функцию, которая определяет *N*-ое число ряда Фибоначчи. Код данной функции достаточно прост, т. к. все операции выполняются по формулам, введенным ранее.

```
void Fibonachi(int K, int &My_res)
{
  float x, y, F, G;
  int i;;
  x=(1-sqrt(5))/2;
  y=(1+sqrt(5))/2;
  G=F=1;
  for (i=0; i<K; i++)
  {
    F*=x;
    G*=y;
  }
  My_res=fabs(F-G)/2;
}
```

Переменная `&My_res` — независимая ссылка переменной `Number`. То есть после изменения `&My_res` изменяется и `Number`. В данном случае `&My_res` — это число ряда Фибоначчи под *N*-ым номером, и оно будет передано в главную функцию.

Ссылочный тип объявляется в функции `main()` следующим образом:

```
int Number;
int &My_res=Number;
```

Функция `main()` считывает *N* с клавиатуры и передает его вместе с начальным значением `Number` в функцию `Fibonachi(int,&int)`. Функция вычисляет значение `&My_res` и передает его в главную функцию, где организован вывод переменной `Number`, которая считается равнозначной переменной `&My_res`:

```
cout<<Number<<endl;
```

Затем осуществляется вызов функции `Test(int,&int)`, которая проверяет, является ли число возрастающим.

```
Test(Number, p);
```

Здесь `p` — переменная, ссылкой на которую является `&m`:

```
int p;
int &m=p;
```

Алгоритм, по которому осуществляется проверка, действует согласно правилам, уже рассмотренным ранее. Сначала выделяется последняя цифра числа (переменная `last`):

```
void Test(int L, int &m)
{
  int last=0, temp=0;
  last=L%10;
```

Затем она удаляется из числа, и алгоритм снова выделяет последнюю цифру (переменная `temp`):

```
L/=10;
temp=L%10;
```

После этого необходимо проверить, все ли цифры числа просмотрены. Если да, то функция завершает свою работу:

```
if (L==0) return;
```

В противном случае сравниваются переменные `last` и `temp`. Если выполняется неравенство `last<temp`, то число — не возрастающее и признак `m` становится равным единице, после чего функция завершает свою работу:

```
if (last<temp)
{
  m=1;
  return;
}
```

Если же значение переменной `last` больше либо равно значению переменной `temp`, то функция вызывает себя рекурсивно с новыми параметрами:

```
  Test(L,m);
}
```

После завершения работы `Test(int,int)` в главной функции осуществляется вывод соответствующего ответа:

```
  if (p==1) cout<<"No"; else cout<<"Yes";
  getch();
  Return 0;
}
```

## 3.6. Упражнения

1. Напишите функцию, которая будет переводить числа из десятичной системы счисления в шестнадцатеричную.
2. Напишите функцию, которая вычислит минимальную степень $Q$ числа $P$, такую, чтобы число $P^Q$ являлось взаимно простым с $K$.
3. Напишите перегруженную функцию, которая будет добавлять цифру $L$ в конец числа $P$.
4. Напишите перегруженную функцию, которая по заданным катетам прямоугольного треугольника будет находить длину его гипотенузы.
5. Напишите перегруженную функцию, которая по заданным координатам двух точек будет находить расстояние между ними.
6. Напишите программу, которая будет удалять из числа рядом стоящие цифры, если их сумма равна $S$.
7. Напишите программу, которая проверит, является ли число $N$ суммой двух простых чисел.
8. Напишите перегруженную функцию, которая будет возвращать единицу, если число $N$ делится на 11, и 0 в противном случае.
9. Напишите перегруженную функцию, которая меняет местами цифру под номером $K$ в числе $A$ с цифрой под номером $L$ в числе $B$.
10. Напишите программу, которая сортирует цифры числа по возрастанию (убыванию).

**Внимание**

Во всех задачах *главы 3* входные данные считать корректными.

# Глава 4

# Массивы

Ранее мы научились использовать те или иные переменные для хранения любых данных. Также было оговорено, что каждая переменная имеет свой тип. Но если необходимо запомнить тысячу чисел, резервировать в памяти тысячу переменных очень неудобно. Для этой цели в языках программирования можно использовать специальную область памяти, в которой можно объединить множество переменных, причем доступ к каждой из них будет быстрым и элементарным. Такое "хранилище" переменных и называется массивом.

## 4.1. Понятие массива

*Массив* — это последовательная группа переменных, имеющих *одно имя* и *один тип*. Массивы широко используются в том случае, если необходимо обработать много данных, которые пользователь вводит с клавиатуры.

Массив назовем *одномерным*, если он состоит из одной строки и *N* столбцов.

Как и любая другая переменная, массив (в данном случае одномерный) должен быть объявлен в следующем виде:

```
тип имя_массива [размерность];
```

Например:

```
int array[10];
```

Данная запись означает, что резервируется память для 10 чисел целого типа с именем `array` и порядковыми номерами от 0 до 9 включительно.

Отдельный элемент массива определяется именем массива и индексом элемента в квадратных скобках. Индекс представляет собой целое число.

**Внимание**

В C++ индексация начинается с нуля.

`a[0]` — первый элемент массива. `a[9]` — последний элемент массива.

Элементы массива в C++, как и переменные, обладают всеми атрибутами заданного типа.

Массив имеет следующие свойства:

- элементы массива имеют одинаковый тип, поэтому занимают одинаковый объем памяти;
- массив располагается в оперативной памяти, а не на внешнем устройстве;
- элементы массива занимают подряд идущие ячейки.

Существуют два варианта объявления массива:

- неинициализированный (например, массив с четырьмя элементами целого типа можно определить как `int array[4]`);
- инициализированный. Определяется следующим образом: `int array[]={2, 3, 5, 7}`.

**Внимание**

При работе с массивами следите за тем, чтобы не выходить за их объявленные границы. Компилятор не предупреждает об этой ошибке.

## Ввод элементов массива

Ввод элементов одномерного массива также можно организовать с клавиатуры, например, следующим образом:

```
cin>>Size;  // Читает с клавиатуры количество элементов,
            // которые пользователь хочет ввести
for (i=0; i<=Size; i++)
  cin>>array[i];  // Считывает с клавиатуры очередной элемент
```

Здесь `Size` — это количество элементов в массиве.

Затем цикл `for` двигает указатель от `0` до `N` и требует ввода некоторого символа.

Для того чтобы не вводить при каждом запуске программы все элементы массива, предлагается заполнить массив автоматически случайными числами. Для этого существует функция `random(K)` — *генератор случайных чисел*.

```
int array[Size];  // Объявляет массив
int i;
K=100;
randomize();      // Инициализация генератора случайных чисел
for (i=0; i<Size; i++)
  array[i]=random(K);  // Каждому элементу массива присваивает
                       // некоторое случайное значение
```

Таким образом, каждый элемент массива станет равным некоторому числу из промежутка $[0; K - 1]$.

Если необходимо, чтобы числа были не только положительными, но и отрицательными, можно записать, например, так:

```
array[i]=random(100)-10;
```

Тогда каждый элемент массива станет равным некоторому числу из промежутка $[-10; K - 11]$.

## Вывод элементов массива

Существует несколько способов вывода одномерного массива в C++.

Приведем некоторые из них.

1. Простой вывод элементов массива в одну строку экрана:

```
for (i=0; i<Size; i++)
  cout<<array[i]<<" ";  // Выводит очередной элемент массива
cout<<endl;             // Переводит курсор
                        // на следующую строку
```

2. Для рассмотрения следующих вариантов необходимо ознакомиться с новой функцией для вывода `printf()`, которая имеет следующий синтаксис:

```
printf(строка, список);
```

   В строке можно записать:

   - любую последовательность символов, включая пробелы, которые точно в таком же виде будут отображаться при выводе на экран;
   - управляющие символы или символьные константы с обратным слэшем:
     - ◊ `\n` — переход на следующую строку;
     - ◊ `\r` — возврат каретки, т. е. переход в начало текущей строки;
     - ◊ `\t` — горизонтальная табуляция, т. е. вывод с определенных стандартных позиций;
     - ◊ `\a` — непродолжительный звуковой сигнал;
   - форматы, или спецификаторы формата, перед которыми записывается символ `%`, никак не связанный ни с соответствующей операцией, ни с вычислением процента. Форматы указывают способ вывода соответствующих элементов списка и зависят от типов выводимых значений:
     - ◊ `%f` — для вывода вещественных чисел;
     - ◊ `%d` — для вывода целых чисел;
     - ◊ `%E` или `%e` — для вывода вещественных чисел в экспоненциальной форме, например, `0.5E-4` или `0.5e-4`;

- ◊ %G — выбирает в зависимости от значения числа формат %f или %E;
- ◊ %g — выбирает в зависимости от значения числа формат %f или %e;
- ◊ %X или %x — для вывода шестнадцатеричных чисел;
- ◊ %c — для вывода символа;
- ◊ %s — для вывода строки.

Перед спецификаторами форматов можно использовать модификаторы форматов для изменения способа представления выводимых значений: %nd, где целая константа n определяет ширину поля для выводимого целого числа. Например, printf("%5d%05d", i, number) выводит справа поля шириной 5 со стоящими впереди пробелами для i и нулями для number.

Вся строка с указанными выше элементами записывается, как правило, в виде текстовой константы. Можно использовать для этих целей переменную строкового типа.

В списке функции printf() можно записать через запятую выражения, значения которых выводятся. Как частный случай выражения можно записать константу или переменную. Между элементами списка и спецификаторами формата с символом % должно быть соответствие в количестве, порядке следования и типе.

**Внимание**

Любой из элементов строки или список могут отсутствовать.

Вернемся к способам вывода массива.

Вывод массива относительно большой размерности N, который не вмещается в одну строку, по K элементов в строке:

```
int K=10;
for (i=0; i<Size; i++)
{
  cout<<array[i]<<" ";    // Вывод очередного элемента массива
```

```
    if ((i+1)%K==0) cout<<endl;
    cout << endl;          // Переводит курсор на новую строку
  }
```

3. В предыдущем варианте числа не выровнены. Поэтому следующий вариант устраняет этот недостаток.

```
for (int i=0; i<18; i++)
{
  printf("%7.2f",A[i]);
  if ((i+1)%K==0) printf("\n");
}
```

## Операция *sizeof*

С помощью операции `sizeof()` можно определить размер памяти, которая соответствует идентификатору или типу. Операция `sizeof()` имеет следующий формат:

```
sizeof(выражение);
```

В качестве выражения может быть использован любой идентификатор либо имя типа, заключенное в скобки. Если в качестве выражения указано имя массива, то результатом является размер всего массива (т. е. произведение числа элементов на длину типа), а не размер указателя, соответствующего идентификатору массива.

## Передача одномерных массивов в функцию

*Массив* — это переменная, которая может содержать множество значений одного и того же типа. Как и все другие переменные, массивы могут использоваться как параметры функций.

Приведем код простой программы, в которой используется функция, выводящая элементы массива на экран, причем в функцию в качестве параметров будет передаваться количество элементов в массиве и сам массив.

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

```
void show_array(int values[], int NOE)  // Описание функции
{
  int i;
  for (i=0; i<NOE; i++)
    printf("%d\n", values[i]);  // Выводит на экран очередной
                                // элемент массива
}
main()
{
  int array[5]={10, 20, 30, 40, 50};
  show_array(array, 5);         // Вызывает функцию show_array
  getch();                      // Задерживает экран
  return 0;                     // Завершает программу
}
```

Как видно из программы, при объявлении массива в качестве формального параметра функции не требуется указывать размер массива. Вместо этого необходимо указать только квадратные скобки `[]`. Отсюда следует, что для функции безразличен размер массива. Массив, передаваемый в функцию, имеет тип `int`. При попытке передать в функцию массив типа `float` компилятор выдаст сообщение об ошибке.

## Максимальный объем памяти, занимаемой массивом

При работе в среде MS-DOS максимальный объем, который может быть отведен под массив, зависит от текущей модели памяти, но, вообще говоря, не может превышать 64 Кбайт. Компиляция следующей программы может быть неудачной из-за того, что массивы требуют слишком много памяти:

```
main()
{
  char string[66000];
  int values[66000];
  float numbers[66000];
}
```

# 4.2. Многомерные массивы

Массивы могут быть *многомерными*. Самые распространенные многомерные массивы — это матрицы (двумерные массивы). Объявляются они следующим образом:

```
int array[n][m];
```

Это означает, что объявлен массив из `n` × `m` целых чисел с именем `array`, где `n` — это количество строк, а `m` — количество столбцов.

Наглядный пример массива, объявленного ниже, приведен в табл. 4.1.

```
int array[4][4];
```

***Таблица 4.1.*** *Пример двумерного массива*

| | **0** | **1** | **2** | **3** |
|---|---|---|---|---|
| **0** | `array[0][0]` | `array[0][1]` | `array[0][2]` | `array[0][3]` |
| **1** | `array[1][0]` | `array[1][1]` | `array[1][2]` | `array[1][3]` |
| **2** | `array[2][0]` | `array[2][1]` | `array[2][2]` | `array[2][3]` |
| **3** | `array[3][0]` | `array[3][1]` | `array[3][2]` | `array[3][3]` |

При объявлении матрицу можно инициализировать, т. е. определить ее элементы:

```
int array[n][m]={{1, 2, 3, 4, 5},
                 {6, 7, 8, 9, 10},
                 {11, 12, 13, 14, 15}};
```

**Внимание**

Если в строке указано меньше элементов, чем требуется, то остальные инициализируются нулями.

## Обработка матриц

### Обработка матрицы по столбцам и строкам

Построчная обработка матрицы применяется в том случае, если для каждой строки матрицы требуется найти некоторый ее пара-

метр, например, сумму, количество всех элементов строки или элементов с некоторым условием, наименьший (наибольший) элемент среди всех или части элементов строки и т. д.

В таких программах организуются два цикла. Внешний цикл отвечает за перебор всех строк матрицы, а внутренний перебирает все элементы строки.

Далее приведен фрагмент программы, которая вычисляет сумму всех элементов каждой строки матрицы и выводит ее значение на экран.

```
for (i=0; i<n; i++)
{
  for (j=0 j<m; i++)
    Sum+=array[i][j];
  cout<<endl<<Sum;
  Sum=0;
}
```

Аналогичные вычисления и анализ можно выполнять не для строк, а для столбцов. В этом случае внешний цикл отвечает за перебор всех столбцов матрицы, а внутренний перебирает все элементы столбца.

## Обработка всей матрицы

Обработка всей матрицы используется в том случае, если необходимо проанализировать матрицу целиком. В таких алгоритмах матрицу можно обрабатывать и по срокам, и по столбцам одновременно.

Далее приведен фрагмент программы, которая находит и выводит на экран максимальное значение из всех элементов матрицы.

```
max=array[0][0];  // Предположим, что левый верхний элемент
                  // матрицы максимальный
for (i=0; i<m; i++)
  for (j=0 j<n; i++)
    if (max<array[i][j]) max=array[i][j];
cout<<endl<<max;
```

## Передача двумерных массивов в функцию

При работе с многомерными массивами может возникнуть необходимость определять функции, имеющие массивы в качестве параметров. При передаче функции в качестве фактического параметра одномерного массива нет необходимости указывать количество его элементов. При передаче функции двумерных массивов необязательным является только количество строк, но количество столбцов должно указываться. В следующей программе определяется функция `show_array` для вывода значений элементов двумерного массива:

```
#include <stdio.h>
#include <conio.h>
// Описание функции
void show_array(int array[][10], int rows)
{
  int i, j;
  for (i = 0; i < rows; i++)
    for (j = 0; j < 10; j++)
      // Выводит очередной элемент матрицы
      printf ("array [%d][%d]=%d\n", i, j, array[i][j]);
}
main()
{
  int array[2][10]={{1, 2, 3, 4, 5, 6, 7, 8, 9, 10},
                    {11, 12, 13, 14, 15, 16, 17, 18, 19,
20}};
  show_array(array,2);  // Вызывает функцию
  getch();              // Задерживает экран
  return 0;             // Завершает программу
}
```

При объявлении функции с массивом в качестве формального параметра необязательно указывать размер массива. В случае с функцией `show_array` квадратные скобки, которые следуют за именем переменной `array`, информируют компилятор лишь о том, что параметр является массивом, причем размер массива для компилятора не имеет значения.

## 4.3. Символьные массивы

При работе с символьными массивами возникает много проблем. Именно поэтому массивы типа `char` заслуживают особого внимания.

Функции обработки строк размещены в заголовочном файле string.h.

Рассмотрим основные функции, которые используются для обработки строк:

- `strlen(ST)` — возвращает целое число, равное длине строки `ST` без нулевого символа;
- `strcpy(ST,CH)` — полностью копирует строку `CH` в строку `ST`;
- `strncpy(ST,CH,N)` — копирует `N` символов из `CH` в `ST` поверх имеющейся записи;
- `strcat(ST,CH,N)` — добавляет `N` символов из `CH` в `ST`. Первый добавляемый символ становится на место нулевого символа строки `ST`;
- `strtok(ST,K)` — разбивает строку `ST` на отдельные лексемы по заданному разделительному знаку `K` типа `char`. Под *лексемами* будем понимать части, на которые разбивается слово;
- `strcmp(ST,CH)` — производит сравнение строк `ST` и `CH` и возвращает 0, если строки равны, положительное число, если `ST > CH`, и отрицательное, если `CH > ST`. Сравнение происходит по кодам символов. Коды символов упорядочены по алфавиту только для латиницы, т. к. для нее введены стандарты ASCII и EBCDIC. К кириллице данное утверждение не относится. Выявление кодов производится при помощи прямого преобразования типа `(int)char`.

## 4.4. Сортировка

Задачи на сортировку данных в прикладном программировании встречаются довольно часто. Как только данные отсортированы, в них становится намного легче ориентироваться, поэтому в программировании исследованы различные методы сортировки.

В данном разделе собраны хорошо известные и наиболее используемые методы сортировки.

*Пузырьковая сортировка* и *выборочная сортировка* сравнительно неэффективны, их среднее время работы пропорционально $n^2$, где $n$ — количество элементов массива.

Два следующих метода, называемые *быстрой сортировкой* и *сортировкой слияниями*, сложнее, но в то же время являются более эффективными. Их среднее время работы пропорционально $n \times \lg(n)$.

## Пузырьковая сортировка

Пузырьковая сортировка характеризуется очень простым кодом и алгоритмом.

Пусть дан массив из четырех элементов. Представьте себе этот массив отображенным вертикально, с нулевым элементом вверху и третьим элементом внизу.

Рассмотрим алгоритм данной сортировки.

Пузырьковая сортировка "пробегает" по массиву снизу вверх. При этом каждый элемент массива сравнивает с тем, что находится над ним, и если последний больше, то меняет их местами. Данные операции продолжаются до тех пор, пока наименьший элемент не "всплывет" на самый верх.

Далее пузырьковая сортировка применяет ту же самую процедуру к подмассиву, состоящему из элементов с индексами от 1 до 3. Сортировка продолжается, пока все подмассивы не будут исчерпаны. В табл. 4.2 показан порядок всплытия элементов массива.

***Таблица 4.2.*** *"Всплытие" элементов массива*

| Индекс | Исходный массив | "Всплытие" 2 | "Всплытие" 4 |
|---|---|---|---|
| 0 | 6 | 2 | 2 |
| 1 | 4 | 6 | 4 |
| 2 | 7 | 4 | 6 |
| 3 | 2 | 7 | 7 |

Вначале алгоритм сравнивает значение элемента 3 со значением элемента 2, и на основе правил, которые описаны выше, решает поменять их местами. Далее сравнение происходит между значениями элементов 2 и 1, и алгоритм также обменивает их. Завершающим шагом будет сравнение значений элементов 1 и 0 и обмен значениями между ними. На этом первый шаг алгоритма будет завершен, т. к. наименьший элемент массива "всплыл" на самый верх.

Далее такой же алгоритм применяется к подмассиву с индексами от 1 до 3, а затем — с индексами от 2 до 3. После этого все подмассивы будут исчерпаны, и все элементы окажутся на должных позициях.

Аналогичную сортировку можно проделать, когда не наименьший элемент будет "всплывать", а наибольший "тонуть".

Далее приведен код пузырьковой сортировки.

```
#include <iostream.h>
#include <conio.h>
main()
{
  int array[100], size, x, y, temp, i;
  // Размер массива берем заведомо больше необходимого,
  // но работать будем только с заданным числом элементов.
  // Динамическое выделение памяти под массивы
  // будет рассмотрено позже
  cin>>size;  // Требует ввода количества элементов массива
  for (x=0; x<size; x++)
    cin>>array[x];  // Требует ввод элементов массива
  for (x=0; x<size-1; x++)
    for (y=x+1; y<size; --y)
      // Если значение элемента с индексом x больше значения
      // элемента с индексом y, то меняет их местами
      if (array[x]>array[y])
      {
        temp=array[x];
        array[x]=array[y];
```

```
        array[y]=temp;
      }
  for (i=0; i<size; i++)
    cout<<array[i]<< " ";   // Выводит отсортированный массив
  getch();   // Задерживает экран
  return 0;  // Завершает программу
}
```

Предел значения переменной `x` во внешнем цикле повторения при сортировке равен `size` – 1, а не `size`, т. к. если требуется сравнить попарно `size` чисел, то необходимо произвести `size` – 1 сравнение.

В этом алгоритме обмен местами элементов с индексами `x` и `y` происходит в случае, если значение элемента с индексом `x` больше значения элемента с индексом `y`. Для реализации сортировки по убыванию следует только изменить знак сравнения, чтобы смена местами элементов с индексами `x` и `y` происходила в случае, если значение элемента с индексом `x` меньше элемента с индексом `y`.

## Выборочная сортировка

Главным недостатком пузырьковой сортировки является количество перестановок элементов массива. В *выборочной сортировке* этот недостаток устранен. Здесь элемент сразу занимает свою конечную позицию.

Как и при пузырьковой сортировке, данный алгоритм будет просматривать подмассивы и отыскивать наименьший элемент в каждом из них. А далее последует единственный обмен в данном подмассиве.

**Примечание**

Подмассивы будут ограничиваться с двух сторон индексами `start` и `size` – 1. Правый индекс равен `size` – 1, т. к., если все прочие элементы заняли свои места, наибольший автоматический оказывается в правильном расположении.

Далее приведен код выборочной сортировки по возрастанию элементов.

```
#include <iostream.h>
#include <conio.h>
main()
{
  int array[100], size, min_pos, i, start, min, temp;
  cin>>size;        // Требует ввода количества
                    // элементов массива
  for (i=0; i<size; i++)
    cin>>array[i];  // Требует ввода элементов массива
  start=0;          // Начальная левая граница подмассива
  // Запускаем цикл повторений, который выполняет операции,
  // пока все подмассивы не будут пройдены
  while (start!=size-1)
  {
    // Предположим, что элемент с индексом левой границы
    // подмассива минимален
    min=array[start];
    min_pos=start;  // Запоминает индекс элемента
                    // с минимальным значением
    // Запускаем цикл поиска минимального элемента в подмассиве
    for (i=start; i<size; i++)
      if (array[i]<min)  // Сравнивает i-ый элемент массива со
                         // значением переменной min
      {
        min=array[i];    // Запоминает минимальное значение
        min_pos=i;       // Запоминает позицию элемента
                         // с минимальным значением
      }
    // Если позиция минимального значения элемента массива
    // не совпадает с предположенной ранее, то меняет местами
    // элементы с данными позициями
    if (min_pos!=start)
    {
```

```
      temp=array[start];
      array[start]=array[min_pos];
      array[min_pos]=temp;
    }
    start++;                // Уменьшает границу подмассива
  }
  for (i=0; i<size; i++)
    cout<<array[i]<< " ";  // Выводит отсортированный массив
  getch();   // Задерживает экран
  return 0;  // Завершает программу
}
```

Приведем код выборочной сортировки, алгоритм которой сортирует массив по убыванию элементов.

```
#include <iostream.h>
#include <conio.h>
main()
{
  int array[100], size, max_pos, i, start, max, temp;
  cin>>size;        // Требует ввода количества элементов
массива
  for (i=0; i<size; i++)
    cin>>array[i];  // Требует ввода элементов массива
  start=0;          // Начальная левая граница подмассива
  // Запускаем цикл повторений, который выполняет операции,
  // пока все подмассивы не будут пройдены
  while (start!=size-1)
  {
    // Предположим, что элемент с индексом левой границы
    // подмассива максимален
    max=array[start];
    max_pos=start; // Запоминает индекс элемента
                   // с минимальным значением
    // Запускаем цикл поиска минимального элемента в подмассиве
    for (i=start; i<size; i++)
      if (array[i]>max)  // Сравнивает i-ый элемент массива
```

```
                        // со значением переменной max
      {
        max=array[i];    // Запоминает максимальное значение
        max_pos=i;       // Запоминает позицию элемента
                         // с минимальным значением
      }
    // Если позиция максимального значения элемента массива
    // не совпадает с предположенной ранее, то меняет местами
    // элементы в данных позициях
    if (max_pos!=start)
    {
      temp=array[start];
      array[start]=array[max_pos];
      array[max_pos]=temp;
    }
    start++;               // Уменьшает границу подмассива
  }
  for (i=0; i<size; i++)
    cout<<array[i]<< " ";  // Выводит отсортированный массив
  getch();   // Задерживает экран
  return 0;  // Завершает программу
}
```

## Быстрая сортировка

Метод *сортировки разделением* был предложен Чарльзом Хоаром в 1962 году. Этот метод является развитием метода простого обмена и настолько эффективен, что его стали называть методом "быстрой сортировки" — Quicksort.

*Быстрая сортировка* хоть и была разработана более 40 лет назад, является наиболее широко применяемым и одним из самых эффективных алгоритмов.

Метод основан на принципе *"разделяй и властвуй"*. Общая схема такова:

1. Из массива выбирается некоторый опорный элемент `p`.

2. Затем необходимо разделить массив на два подмассива. Слева от `p` элементы меньшие либо равные `p`, а справа, большие либо равные `p`.
3. Теперь массив состоит из двух подмножеств, причем левое либо меньше, либо равно правому.

   Для обоих подмножеств, если в них более двух элементов, следует проделать ту же процедуру.

В итоге получится полностью отсортированная последовательность.

Рассмотрим алгоритм подробнее.

1. Введем два указателя: `i` и `j`. В начале алгоритма они указывают, соответственно, на левый и правый конец последовательности.
2. Будем двигать указатель `i` с шагом в один элемент по направлению к концу массива, пока не будет найден элемент `array[i]>=p`. Затем аналогичным образом начнем двигать указатель `j` от конца массива к началу, пока не будет найден `array[j]<=p`.
3. Далее, если `i<=j`, меняем `array[i]` и `array[j]` местами и продолжаем двигать `i` и `j` по тем же правилам.
4. Повторяем шаг 2, пока `i<=j`.

Теперь массив разделен на две части: все элементы левой меньше либо равны `p`, все элементы правой — больше либо равны `p`. Разделение завершено.

Проиллюстрируем работу алгоритма на примере.

Исходный массив:

```
5 2 7 2 13 3 8 15 19
```

Выберем опорный элемент `p`. Например, пусть это будет элемент `13`.

1. **Шаг 1.** Меняем местами элементы в позициях 4 и 6:

   ```
   5 2 7 2 8 3 13 15 19
   ```

   Выбираем опорный элемент `7`.

2. **Шаг 2.** Меняем местами элементы в позициях 2 и 5:

   `5 2 3 2 8 7 13 15 19`

   Выбираем опорный элемент `2`.

3. **Шаг 3.** Меняем местами элементы в позициях 0 и 3:

   `2 2 3 5 8 7 13 15 19`

   Выбираем опорный элемент `8`.

4. **Шаг 4.** Меняем местами элементы в позициях 4 и 5:

   `2 2 3 5 7 8 13 15 19`

Сортировка закончена, т. к. все элементы встали на свои позиции.

Далее приведен код быстрой сортировки.

```
#include <iostream.h>
#include <conio.h>
int array[100000];  // Объявляет массив, в котором будут
                    // храниться числа,
                    // которые необходимо отсортировать
void quicksort(long High, long Low)     // Описание функции,
                    // которая выполняет быструю сортировку
// В качестве параметров функции выступают верхняя и нижняя
// границы обрабатываемого на данном шаге подмассива
{
  long i, j;
  int p, temp;
  i=Low;      // Делает копию нижней границы
  j=High;     // Делает копию верхней границы
  p=array[(Low+High)/2];  // Выбирает опорный элемент
  do
  {
    while (array[i]<p) i++;  // Если выполняется условие цикла,
                             // то передвигает указатель i
                             // на один элемент вправо
      while (array[j]>p) j--;  // Если выполняется условие цикла,
                             // то передвигает указатель j
```

```
                    // на один элемент влево
        if (i<=j)  // Если указатели не пересеклись по позициям
                    // в массиве, то меняет местами элементы i и j
        {
          temp=array[i];
          array[i]=array[j];
          array[j]=temp;
          i++;
          j--;
        }
  }
  while (i<=j);  // Цикл продолжает свою работу до тех пор,
    // пока указатели не пересекутся по позициям в массиве
  if (j>Low) quicksort(j, Low);  // Если в выделенном
    // подмассиве с границами [Low,j] больше одного элемента,
    // то запускает для него функцию quicksort(long,long)
  if (High>i) quicksort(High, i);  // Если в выделенном
    // подмассиве с границами [i,High] больше одного элемента,
    // то запускает для него функцию quicksort(long,long)
}
main()  // Описание главной функции
{
  int size;  // Объявляет переменную, в которой будет
             // храниться количество введенных элементов
  int i;
  cin>>size;      // Требует ввести количество элементов
  for (i=0; i<size; i++)
  cin>>array[i];  // Производит чтение очередного элемента
                  // с клавиатуры
  quicksort(size-1, 0);  // Запускает функцию с начальными
   // параметрами, которая отсортирует указанный массив.
   // Первый параметр — верхняя граница начального подмассива,
   // второй, соответственно, — нижняя.
   // После того как функция quicksort(long,long) отсортировала
```

```
   // массив, он готов к выводу.
  for (i=0; i<size; i++)
    cout<<array[i]<<" ";  // Выводит очередной элемент
                          // отсортированного массива
  getch();   // Задерживает экран
  return 0;  // Завершает программу
}
```

Вышеуказанный алгоритм сортирует массив чисел по возрастанию. Далее приведен алгоритм, который сортирует массив чисел по убыванию.

```
#include <iostream.h>
#include <conio.h>

int array[100000];  // Объявляет массив,
                    // в котором будут храниться числа,
                    // которые необходимо отсортировать
void quicksort(long High, long Low)  // Описание функции,
  // которая выполняет быструю сортировку.
  // В качестве параметров функции выступают верхняя и нижняя
  // границы обрабатываемого на данном шаге подмассива
{
  long i, j;
  int p, temp;
  i=Low;   // Делает копию нижней границы
  j=High;  // Делает копию верхней границы
  p=array[(Low+High)/2];  // Выбирает опорный элемент
  do
  {
    while (array[i]>p) i++;  // Если выполняется условие
      // цикла, то передвигает указатель i
      // на один элемент вправо
    while (array[j]<p) j--;  // Если выполняется условие
      // цикла, то передвигает указатель j
      // на один элемент влево
```

```
    if (i<=j)  // Если указатели не пересеклись по позициям
               // в массиве, то меняет местами элементы i и j
    {
      temp=array[i];
      array[i]=array[j];
      array[j]=temp;
      i++;
      j--;
    }
  }
  while (i<=j);  // Цикл продолжает свою работу до тех пор,
                 // пока указатели не пересекутся
                 // по позициям в массиве
  if (j>Low) quicksort(j, Low);        // Если в выделенном
   // подмассиве с границами [Low,j] больше одного элемента,
   // то запускает для него функцию quicksort(long,long)
  if (High>i) quicksort(High, i);       // Если в выделенном
   // подмассиве с границами [i,High] больше одного элемента,
   // то запускает для него функцию quicksort(long,long)
}
main()  // Описание главной функции
{
  int size;  // Объявляет переменную, в которой будет
             // храниться количество введенных элементов
  int i;
  cin>>size;  // Требует ввести количество элементов
  for (i=0; i<size; i++)
    cin>>array[i];  // Производит чтение очередного элемента
                    // с клавиатуры
  quicksort(size-1, 0);  // Запускает функцию с начальными
    // параметрами, которая отсортирует указанный массив.
    // Первый параметр — верхняя граница начального подмассива,
    // второй, соответственно, — нижняя.
    // После того как функция quicksort(long,long)
    // отсортировала массив, он готов к выводу
```

```
  for (i=0; i<size; i++)
    cout<<array[i]<<" ";  // Выводит очередной элемент
                          // отсортированного массива
  getch();   // Задерживает экран
  return 0;  // Завершает программу
}
```

**Примечание**

В данном алгоритме мы просто разделили массив таким образом, что в левой части оказались все числа, бóльшие либо равные опорному элементу `p`, а в правой части, наоборот, все элементы, меньшие, чем `p`.

## Сортировка методом Шелла

Метод получил свое название от фамилии создателя, Дональда Шелла. Этот метод заключается в сравнении элементов массива, разделенных одинаковым расстоянием таким образом, чтобы элементы были упорядочены по расстоянию. Затем это расстояние делится пополам, и процесс повторяется, и т. д.

Далее приведен код программы, которая сортирует массив по возрастанию значений элементов методом Шелла.

```
#include <stdio.h>
#include <iostream.h>
#include <conio.h>
void shell_sort(int array[], int N)  // Описание функции
{
  int temp, Middle, i, Change;
  Middle=N/2;  // Находит номер центрального элемента массива
  do
  {
    do
    {
      Change=0;  // Введем признак, начальное значение
                 // которого равно нулю. Признак показывает
                 // внесение изменений в массив
```

```
      for (i=0; i<N-Middle; i++)
        if (array[i]>array[i+Middle])
        {
          // Меняет местами элементы массива
          temp=array[i];
          array[i]=array[i+Middle];
          array[i+Middle]=temp;
          Change=1;  // В массив внесены изменения
        }
    } while (Change);
  } while (Middle!=Middle/2);
}
main()
{
  int array[100], i, Size;
  cin>>Size;      // Требует ввода размера массива
  for (i=0; i<Size; i++)
  cin>>array[i];  // Требует ввода очередного элемента массива
 shell_sort(array, Size);  // Вызывает функцию сортировки
 for (i=0; i<Size; i++)
  printf("%d ",array[i]);  // Выводит очередной элемент
                           // отсортированного массива
 getch();   // Задерживает экран
 return 0;  // Завершает программу
}
```

Рассмотрим функцию `shell_sort(int[],int)`, которая выполняет сортировку массива:

```
void shell_sort(int array[], int N)
{
  int temp, Middle, i, Change;
```

Переменной `Middle` будем присваивать позицию центрального элемента подмассива. В начале значение переменной равно `N/2`, где `N` — это количество элементов в массиве.

Далее, определим переменную `Change`, которая отвечает за изменения в массиве. Если `Change=0`, то на текущем шаге элемен-

ты массива не переставлялись местами, в противном случае `Change=1`.

Предположим, что массив уже отсортирован:

```
do
{
  do
  {
    Change=0;
```

Теперь при помощи цикла повторений `for` просмотрим все элементы массива от нулевого до `N-Middle`:

```
for (i=0; i<N-Middle; i++)
```

и проверим, действительно ли массив отсортирован. Если на некотором шаге окажется, что `array[i]>array[i+Middle]`, то поменяем элементы местами:

```
for (i=0; i<N-Middle; i++)
  if (array[i]>array[i+Middle])
  {
    temp=array[i];
    array[i]=array[i+Middle];
    array[i+Middle]=temp;
```

а переменной `Change` присвоим значение 1. Это означает, что в массиве произошли некоторые изменения, и необходимо запустить еще одну такую проверку:

```
  Change=1;
}
```

Алгоритм будет продолжать свою работу до тех пор, пока в массиве будут происходить перестановки некоторых двух элементов, и пока расстояние между элементами не станет равным единице (`Middle=Middle/2`):

```
    } while (Change);
  } while (Middle!=Middle/2);
}
```

## Сортировка массива при известном интервале значений элементов

Данная сортировка применима к массиву только в том случае, если задано некоторое число *K*, и все элементы массива меньше либо равны ему, т. е. для выполнения данной сортировки необходимо знать один общий интервал, в котором лежит каждый элемент массива. Алгоритм такой сортировки достаточно прост в реализации.

Пусть дан массив натуральных чисел, значение каждого элемента которого не превышает девяти (табл. 4.3).

***Таблица 4.3.*** *Массив натуральных чисел*

| **Индекс** | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| **Элемент** | 5 | 2 | 5 | 9 | 8 | 4 |

Для сортировки такого массива будет использоваться вспомогательный массив, число ячеек в котором совпадает с числом *K*. В нашем случае необходимо выделить память для массива из девяти ячеек, которые будут заполнены нулями (табл. 4.4).

***Таблица 4.4.*** *Вспомогательный массив, заполненный нулями*

| **Индекс** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| **Элемент** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Теперь будем поочередно просматривать каждый элемент массива, который необходимо отсортировать, и, если на *i*-ом шаге выделен элемент *M*, то во вспомогательном массиве значение элемента, находящегося в позиции *M*, увеличиваем на единицу.

Совершим вышеописанные действия с заданным массивом чисел. После завершения работы алгоритма вспомогательный массив примет вид, как указано в табл. 4.5.

***Таблица 4.5.*** *Вспомогательный массив после завершения сортировки*

| **Индекс** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| **Элемент** | 0 | 1 | 0 | 1 | 2 | 0 | 0 | 1 | 1 |

Теперь вспомогательный массив обработан, можно выводить результат следующим образом: если элемент вспомогательного массива не равен нулю, то выводим номер его позиции *P* раз, где *P* равно значению данного элемента.

Итак, в нашем случае вывод будет следующим: 2, 4, 5, 5, 8, 9.

Теперь рассмотрим программу, которая сортирует массив натуральных чисел вышеуказанным методом. Чтобы не вводить все элементы массива, сгенерируем их случайным образом при помощи функции `random(int)`. Ниже приведен код данной программы:

```
#include <iostream.h>
#include <conio.h>
#include <stdlib.h>
int i, j, N, K, array[1000], Help_array[1000];
main()
{
  cin>>N>>K;  // Требует ввода двух чисел:
              //   N — количество элементов в массиве,
              //   K — верхняя граница диапазона
              //       для генератора случайных чисел
  randomize();  // Инициализация генератора случайных чисел
  for (i=0; i<N; i++)
    array[i]=random(K);  // Заполняет массив случайными
                         // числами, которые не превышают K-1
   // Заполняет вспомогательный массив
   for (i=0; i<N; i++)
     Help_array[array[i]]+=1;
     // Выводит результат
   for (i=1; i<K; i++)
```

```
    if (Help_array!=0)
      for (j=0; j<Help_array[i]; j++)
        cout<<i<<" ";
  getch();   // Задерживает экран
  return 0;  // Завершает программу
}
```

# 4.5. Поиск заданного элемента в массиве

## Последовательный поиск элемента

Название данного метода раскрывает суть алгоритма, по которому выполняется поиск заданного элемента в массиве. Необходимо запустить цикл повторений, который станет просматривать все элементы один за другим до тех пор, пока нужный элемент не будет найден, либо не будут пройдены все его элементы.

Вот код такой программы:

```
#include <iostream.h>
#include <conio.h>
// Описание функции
int Search(int array[], int N, int Number)
{
  int i=0;
  while (i!=N)
    // Если элемент массива под номером i равен искомому,
    // то функция возвращает
    // позицию данного элемента (переменная i)
    if (array[i]==Number) return i; else i++;
    // Если цикл завершил свою работу, а элемент не найден,
    // то функция возвращает -1
  return -1;
}
main()
{
```

```
  int i, Size, Numerals[1000], Number, P;
  cout<<"Введите размерность массива!"<<endl;
  cin>>Size;           // Требует ввода размера массива
  cout<<"Введите элементы массива!"<<endl;
  for (i=0; i<Size; i++)
    cin>>Numerals[i];  // Требует ввода очередного
                       // элемента массива
  cout<<"Введите число для поиска!"<<endl;
  cin>>Number;  // Требует ввода искомого числа
  P=Search(Numerals,Size,Number);  // Запускает функцию поиска
  // Выводит результат
  if (P==-1) cout<<"No"; else cout<<"Yes "<<P+1;
  getch();   // Задерживает экран
  return 0;  // Завершает работу программы
}
```

## Двоичный поиск

Итак, один из способов поиска значения в массиве — последовательный просмотр всех его элементов. Такой способ поиска приемлем для массивов небольших размеров. Последовательный поиск в большом массиве может потребовать много времени. Если элементы массива отсортированы, то можно использовать *двоичный поиск*. Он называется двоичным, потому что после каждой операции количество просматриваемых значений уменьшается в два раза. Наилучший способ понять механизм двоичного поиска — это представить, как в словаре организуется поиск нужного слова. Например, нужно найти слово "Frame". Для этого открываем словарь на середине и просматриваем слова. Допустим, все слова на этой странице начинаются на букву "K", в этом случае ясно, что нужное слово — в первой половине словаря. Таким образом, исключается вторая половина словаря. Далее открываем словарь на половине оставшейся части и видим там слова, начинающиеся на букву "H". Теперь исключена половина оставшейся части. Повторяя таким образом процесс деления пополам, мы можем быстро найти страницу со словом "Frame".

**Примечание**

При двоичном поиске значения в массиве должны быть отсортированы в порядке возрастания.

Приведем пример программы, которая выполняет поиск в массиве заданного элемента при помощи описанного выше алгоритма, при этом будем считать, что массив уже отсортирован в порядке возрастания значений элементов:

```
#include <iostream.h>
#include <conio.h>
// Описание функции
int Binary_Search(int array[], int N, int Number)
{
  int Left, Right, Middle;
  Left=0;                 // Левая граница
  Right=N-1;              // Правая граница
  Middle=(Left+Right)/2;  // Середина
  while (Left<=Right)
  {
    // Если элемент найден, то возвращает его позицию в массиве
    if (array[Middle]==Number) return Middle;
    // Если элемент левее середины, то уменьшает правую границу
    if (array[Middle]>Number) Right=Middle-1;
    // иначе увеличивает левую границу
    else Left=Middle+1;
    // Находит середину подмассива,
    // ограниченного левой и правой границей
    Middle=(Left+Right)/2;
  }
  return -1;  // Возвращает -1, если элемент в массиве не найден
}
main()
{
  int i, Size, Numerals[100000], Number, Pos;

  cout<<"Введите размер массива!"<<endl;
  cin>>Size;
```

```
  for (i=0; i<Size; i++)
    Numerals[i]=i;
  cout<<"Введите число для поиска!"<<endl;
  cin>>Number;
  Pos=Binary_Search(Numerals,Size,Number);
  if (Pos==-1) cout<<"Число в массиве отсутствует";
  else cout<<"Число находится в позиции: "<<Pos;
  getch();   // Задерживает экран
  return 0;  // Завершает программу
}
```

# 4.6. Арифметика больших чисел

Довольно часто в задачах возникает необходимость проводить некоторые арифметические операции над числами, для которых вещественный тип неприемлем, а типа `long` недостаточно. Для хранения таких чисел используется одномерный массив, каждый элемент которого представляет собой одну десятичную цифру числа.

## Вычитание чисел

В программе, которая будет вычитать два числа, предлагается использовать три массива. В массиве `A` будут храниться все цифры первого числа, в массиве `B` — все цифры второго числа, а в массиве `C` — результат. Алгоритм вычитания двух чисел сводится к обычному алгоритму вычитания двух чисел "столбиком". Для того чтобы не запутаться с разрядами, рекомендуется в массивы записывать цифры чисел с "хвоста" до "головы". Например, вычтем числа 731 и 401 (рис. 4.1).

Таким образом, 731 – 401 = 330.

**Примечание**

При реализации алгоритма необходимо предусмотреть ситуацию, когда первое число меньше второго. В этом случае результат будет отрицательным.

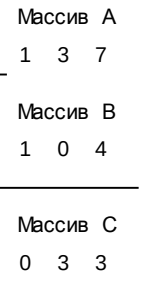



**Рис. 4.1.** Вычитание чисел

Далее приведен код программы, которая реализует алгоритм вычитания положительных чисел "столбиком".

В данной программе мы будем использовать запись `*Temp`. Она необходима для использования встроенной функции `atoi(*char)`. В данном случае следует считать, что это просто переменная типа `char`, а в дальнейшем будет рассказано о способах обращения с переменными, записанными в таком виде.

```
#include <iostream.h>
#include <conio.h>
#include <stdlib.h>
int a[100], b[100], c[100];
void input();       // Прототип функции, которая реализует
                    // ввод данных
void max(int,int);  // Прототип функции, которая определяет,
                    // какое из чисел больше
void done(int,int,int);  // Прототип функции, которая
                         // реализует алгоритм вычитания
void output(int,int);    // Прототип функции, которая выводит
                         // результат действий на экран
void input()  // Описание функции
{
  char *Temp, S[100];
  int i, j, Len_1, Len_2, pos;
  cin>>S;  // Требует ввода первого числа с клавиатуры
```

```
   Len_1=strlen(S); // Возвращает количество цифр, из
                   // которых состоит введенное число
  i=Len_1-1;       // Начальная позиция указателя строки
  Temp="";         // Начальное значение
  pos=0;           // Начальная позиция указателя массива A
  while (i>=0)
  {
    *Temp=S[i];    // Выделяет очередную цифру из числа
    a[pos]=atoi(Temp);  // Помещает ее в массив A
    pos++;              // Сдвигает указатели
    --i;
  }
  cin>>S;          // Требует ввода второго числа
  Len_2=strlen(S);  // Возвращает количество цифр,
                   // из которых состоит введенное число
  i=Len_2-1;       // Начальная позиция указателя строки
  Temp="";
  pos=0;           // Начальная позиция указателя массива B
  while (i>=0)
  {
    *Temp=S[i];    // Выделяет очередную цифру из числа
    b[pos]=atoi(Temp);  // Помещает ее в массив B
    pos++;              // Сдвигает указатели
    --i;
  }
  max(Len_1, Len_2);  // Запускает функцию, которая
                      // определяет, какое из чисел больше
}
void max(int Len_1, int Len_2)  // Описание функции
{
  int i, p;
  // Возвращает единицу, если первое число больше, иначе
  // возвращает двойку
  if (Len_1>Len_2)
  {
```

```
    p=1;
    done(Len_1, Len_2, p);
    return;  // Завершает работу функции
  }
  if (Len_1<Len_2)
  {
    p=2;
    done(Len_1, Len_2, p);
    return;
  }
  p=1;
  for (i=Len_1-1; i>=0; i--)
  {
    if (a[i]<b[i])
    {
      p=2;
      done(Len_1, Len_2, p);
      return;
    }
    if (a[i]>b[i])
    {
      done(Len_1, Len_2, p);
      return;
    }
  }
  done(Len_1, Len_2, p);
  return;
}
void done(int Len_1, int Len_2, int p) // Описание функции
{
  int i, w, K;
  if (Len_1>Len_2) K=Len_1; else K=Len_2;
  if (p==2)
  for (i=0; i<K; i++)
  {
```

```
    w=a[i];
    a[i]=b[i];
    b[i]=w;
  }
  for (i=0; i<K; i++)
  {
    c[i]=a[i]-b[i]+c[i];  // Вычитает очередные две цифры числа
    if (c[i]<0)
    {
      c[i+1]-=1;
      c[i]+=10;
    }
  }
  output(K,p);  // Запускает функцию, которая выводит результат
}
void output(int K, int p)
{
  int i;
  while (c[K]==0 && K>0) K--;
  if (p==2) cout<<"-";  // Если результат отрицателен,
                        // выводит знак - (минус)
  for (i=K; i>=0; i--)
    cout<<c[i];         // Выводит очередной элемент массива

  getch();  // Задерживает экран
  return;   // Завершает работу функции
}
main()
{
  input();
}
```

Рассмотрим алгоритм, по которому работает данная программа.

Как и во всех задачах, на первом шаге необходимо прочитать данные, над которыми потом будут производиться некоторые

операции. Ввод данных в программе реализует функция `input()`. Функция не содержит никаких параметров и используется исключительно для того, чтобы отделить процедуру ввода от остальной программы. Сначала функция принимает число, которое представляет собой строку символов:

```
cin>>S;
```

После того как строка введена, программа заносит каждый ее символ, начиная с последнего, в массив. Так как каждый символ строки определяется типом `char`, то необходимо использование функции `atoi()`, которая будет возвращать элементу массива с номером `pos` очередную цифру числа:

```
while (i>=0)
{
  *Temp=S[i];
  a[pos]=atoi(Temp);
  pos++;
  --i;
}
```

**Внимание**

Для того чтобы использовать функцию `atoi()`, необходимо подключить к программе заголовочный файл stdlib.h.

Данные операции следует проделать и для второго числа.

В результате все цифры первого числа будут занесены в массив `A`, а цифры второго числа — в массив `B`. Таким образом, массивы подготовлены для обработки.

На следующем шаге необходимо определить, какое из чисел больше. Данную подзадачу реализовывает функция `max(int, int)`.

```
max(Len_1, Len_2);
```

Параметрами данной функции являются длины чисел, т. е. количество цифр, из которых состоит каждое число. Функция будет возвращать 1, если первое число больше второго, и 2 — в другом

случае. Данную проверку можно осуществлять по нескольким критериям:

- если длина числа `A` больше длины числа `B`, то очевидно, что первое число больше:

```
if (Len_1>Len_2)
{
  p=1;
  done(Len_1, Len_2, p);
  return;
}
```

- если длина числа `A` меньше длины числа `B`, то очевидно, что второе число больше:

```
if (Len_1<Len_2)
{
  p=2;
  done(Len_1, Len_2, p);
  return;
}
```

- если длины чисел равны, необходимо по очереди сравнивать цифры чисел под номером `i` до тех пор, пока они остаются равными. Если на некотором шаге цифра под номером `i` числа `A` меньше цифры под номером `i` числа `B`, значит, `B` больше `A`. Сравнения могут продолжаться до тех пор, пока `i` не достигнет нуля. В этом случае значение переменной `p` остается равным единице.

```
  p=1;
  for (i=Len_1-1; i>=0; i--)
  {
    if (a[i]<b[i])
    {
      p=2;
      done(Len_1, Len_2, p);
      return;
    }
```

```
      if (a[i]>b[i])
      {
        done(Len_1, Len_2, p);
        return;
      }
    }
    done(Len_1, Len_2, p);
    return;
  }
```

После того как переменной `p` возвращено некоторое значение, происходит вызов функции `done(int,int,int)`, которая выполняет основное действие — вычитание двух чисел. Параметрами данной функции являются длины чисел, а также переменная `p`, которая указывает на большее число.

В данной функции на первом шаге переменной `K` возвращается максимальная длина числа:

```
if (Len_1>Len_2) K=Len_1; else K=Len_2;
```

Если `p=2`, т. е. второе число больше первого, то целесообразно будет поменять элементы массивов местами таким образом, чтобы все элементы массива `B` "перешли" в массив `A`, а элементы массива `A` в `B`.

```
if (p==2)
  for (i=0; i<K; i++)
  {
    w=a[i];
    a[i]=b[i];
    b[i]=w;
  }
```

На каждом следующем шаге будем вычитать из `i`-го элемента массива `A` `i`-ый элемент массива `B` и полученную разность складывать с переносом из предыдущего разряда `C`:

```
for (i=0; i<K; i++)
{
  c[i]=a[i]-b[i]+c[i];
```

Если на некотором шаге окажется, что разность отрицательна, то алгоритм будет действовать по принципу "столбика", т. е. к отрицательной разности прибавляется десять, а из следующего элемента массива вычитается единица:

```
  if (c[i]<0)
  {
    c[i+1]-=1;
    c[i]+=10;
  }
}
```

Данный алгоритм станет продолжать свою работу до тех пор, пока не будут пройдены все `K` элементов.

По завершению работы функции `done(int,int,int)` вызывается функция `output(int,int)`, которая выводит результат вычитания на экран:

```
  output(K,p);
}
```

Вывод происходит следующим образом:

Для корректного вывода результата необходимо убрать все незначащие нули из числа:

```
while (c[K]==0 && K>0) K--;
```

В данном случае условие `K>0` предназначено для худшего случая, когда разность равна нулю.

Если результат отрицательный (`p=2`), то на экран, по условию, требуется вывести знак минус `-`:

```
if (p==2) cout<<"-";
```

после чего следует вывод каждого элемента массива:

```
for (i=K; i>=0; i--)
  cout<<c[i];
```

## Сложение чисел

Алгоритм сложения двух больших чисел во многом схож с алгоритмом вычитания. Их различия состоят в том, что при сложении

чисел не требуется определять, какое из чисел больше, следовательно, функция `max(int,int)` не будет присутствовать в программе, а из этого вытекает отсутствие переменной `p`. Различия также наблюдаются в функции `done(int,int)`, которую мы рассмотрим далее.

Представим код программы, которая складывает два положительных числа:

```
#include <iostream.h>
#include <conio.h>
#include <stdlib.h>
int a[100], b[100], c[100];
void input();
void done(int, int);
void output(int);
void input()
{
  char *Temp, S[100];
  int i, j, Len_1, Len_2, pos;
  cin>>S;
  i=strlen(S)-1;
  Temp="";
  pos=0;
  while (i>=0)
  {
    *Temp=S[i];
    a[pos]=atoi(Temp);
    pos++;
    --i;
  }
  Len_1=pos;
  pos=0;
  cin>>S;
  i=strlen(S)-1;
  Temp="";
  pos=0;
```

```
  while (i>=0)
  {
    *Temp=S[i];
    b[pos]=atoi(Temp);
    pos++;
    --i;
  }
  Len_2=pos;
  done(Len_1, Len_2);
}
void done(int Len_1, int Len_2)
{
  int i, K;
  if (Len_1>Len_2) K=Len_1; else K=Len_2;
  for (i=0; i<K; i++)
  {
    c[i]=a[i]+b[i]+c[i];
    if (c[i]>9)
    {
      c[i+1]+=(c[i]/10);
      c[i]%=10;
    }
  }
  output(K);
}
void output(int K)
{
  int i;
  while (c[K]==0 && K>0) K--;
  for (i=K; i>=0; i--)
    cout<<c[i];
  getch();
  return;
}
```

```
main()
{
  input();
}
```

Рассмотрим функцию `done(int, int)`, которая отличается от функции с таким же именем из предыдущей программы. В данном случае в функцию поступают два параметра — длины каждого из чисел.

Так как два массива уже подготовлены для обработки, то на каждом последующем шаге будем складывать `i`-ый элемент массива `A` с `i`-ым элементом массива `B` и полученную сумму складывать с результирующим `C`:

```
for (i=0; i<K; i++)
{
  c[i]=a[i]+b[i]+c[i];
```

Если на некотором шаге окажется, что сумма превышает 9, то программа будет действовать согласно алгоритму сложения "столбиком", т. е. к следующему элементу массива прибавляется полученная сумма, нацело разделенная на десять, а в текущий элемент `c[i]` записывается остаток от деления суммы на 10:

```
  if (c[i]>9)
  {
    c[i+1]+=(c[i]/10);
    c[i]%=10;
  }
}
```

Данный алгоритм станет продолжать свою работу до тех пор, пока не будут пройдены все `K` элементов.

По завершению работы функции `done(int, int)` будет вызвана функция `output(int)`, которая выводит результат вычитания на экран:

```
  output(K);
}
```

## Умножение чисел

Алгоритм умножения двух чисел, так же как и предыдущие алгоритмы арифметики, реализуется методом "столбика" и, по сути, похож на алгоритм сложения. Небольшие различия программ заключаются в организации функции `done(int, int)`.

Далее представлен код данной программы.

```
#include <iostream.h>
#include <conio.h>
#include <stdlib.h>
int a[100], b[100], c[100];
void input();
void done(int, int);
void output(int);
void input()
{
  char *Temp, S[100];
  int i, j, Len_1, Len_2, pos;
  cin>>S;
  i=strlen(S)-1;
  Temp="";
  pos=0;
  while (i>=0)
  {
    *Temp=S[i];
    a[pos]=atoi(Temp);
    pos++;
    --i;
  }
  Len_1=pos;
  pos=0;
  cin>>S;
  i=strlen(S)-1;
  Temp="";
  pos=0;
```

```
  while (i>=0)
  {
    *Temp=S[i];
    b[pos]=atoi(Temp);
    pos++;
    --i;
  }
  Len_2=pos;
  done(Len_1, Len_2);
}
void done(int Len_1, int Len_2)
{
  int i, K, j;
  if (Len_1>Len_2) K=Len_1; else K=Len_2;
  for (i=0; i<Len_1; i++)
    for (j=0; j<Len_2; j++)
    {
      c[i+j]=c[i+j]+a[i]*b[j];
      if (c[i+j]>9)
      {
        c[i+j+1]+=(c[i+j]/10);
        c[i+j]%=10;
      }
    }
  output(K);
}
void output(int K)
{
  int i;
  K+=K;
  while (c[K]==0 && K>0) K--;
  for (i=K; i>=0; i--)
    cout<<c[i];
  getch();
```

```
  return;
}
main()
{
  input();
}
```

Рассмотрим функцию `done(int, int)`, которая реализует умножение. В данном случае, как и при сложении чисел, в функцию поступают два параметра — длины каждого из чисел.

Так как два массива уже подготовлены для обработки, то на каждом последующем шаге будем умножать `i`-ый элемент массива `A` на `i`-ый элемент массива `B` и полученное произведение складывать с результирующим `C`:

```
for (i=0; i<Len_1; i++)
  for (j=0; j<Len_2; j++)
  {
    c[i+j]=c[i+j]+a[i]*b[j];
```

Необходимо заметить, что программа запускает два вложенных цикла `for`. Первый "пробегает" по всем элементам массива `A`, а второй, соответственно, по всем элементам массива `B`.

Если на некотором шаге окажется, что произведение превышает максимальную оценку 9, то к следующему элементу массива прибавляется полученное произведение, нацело разделенное на десять, а в текущий элемент `c[i+j]` записывается остаток от деления произведения на 10:

```
  if (c[i+j]>9)
  {
    c[i+j+1]+=(c[i+j]/10);
    c[i+j]%=10;
  }
}
```

Данный алгоритм станет продолжать свою работу до тех пор, пока не будут пройдены все `K` элементов.

По завершению работы функции `done(int, int)` вызывается функция `output(int)`, которая выводит результат вычитания на экран:

```
  output(K);
}
```

**Совет**

Модифицируйте рассмотренные программы таким образом, чтобы вычитать, складывать и умножать можно было не только положительные числа, но и отрицательные.

# 4.7. Примеры использования массивов

## Пример 1

Дан массив из *N* элементов. Рассмотрим программу, которая определит количество четных чисел на отрезке, заключенном между максимальным и минимальным элементами массива.

```
#include <iostream.h>
#include <conio.h>
main()
{
  int N, i, min, max, min_pos, max_pos, array[100], p, k,
kol;
  cin>>N;           // Требует ввода количества элементов массива
  for (i=0; i<N; i++)
   cin>>array[i];  // Требует ввода очередного элемента массива
  min=array[0];    // Пусть нулевой элемент массива
                   // является минимальным
  min_pos=0;       // Задает его позицию (0)
  for (i=0; i<N; i++)
    if (array[i]<min)  // Если значение элемента с индексом
    {                  // i меньше min, то запоминает
      min=array[i];    // его значение
```

```
      min_pos=i;        // и позицию
    }
  max=array[0];     // Пусть нулевой элемент массива максимальный
  max_pos=0;        // Задает его позицию (0)
  for (i=0; i<N; i++)
    if (array[i]>max)  // Если значение элемента с индексом
    {                  // i больше max, то
      max=array[i];    // запоминает его значение
      max_pos=i;       // и позицию
    }
  if (min_pos<max_pos)  // Определяет, какой элемент левее —
  {                      // максимальный или минимальный
    p=min_pos;           // Крайний слева
    k=max_pos;           // Крайний справа
  }
  else
  {
    p=max_pos;  // Крайний слева
    k=min_pos;  // Крайний справа
  }
  kol=0;
  for (i=p+1; i<k; i++)  // Поиск продолжает от крайнего слева
                         // до крайнего справа элемента
    if (array[i]%2==0) kol++;  // Если число четное, то
                               // увеличивает счетчик на 1
  cout<<kol;     // Выводит количество четных чисел
  getch();       // Задерживает экран
  return 0;      // Завершает программу
}
```

Из условия следует, что на первом шаге необходимо найти минимальный и максимальный элементы, а также запомнить их позицию в массиве. С задачей нахождения минимального и максимального элемента вы уже знакомы.

Предположим, что нулевой элемент массива минимален. Далее запустим цикл повторений, при помощи которого все элементы массива будут сравниваться с переменной `min`:

```
min=array[0];
min_pos=0;
for (i=0; i<N; i++)
  if (array[i]<min)
  {
    min=array[i];
    min_pos=i;
  }
```

Таким же образом поступим и с максимальным элементом:

```
max=array[0];
max_pos=0;
for (i=0; i<N; i++)
  if (array[i]>max)
  {
    max=array[i];
    max_pos=i;
  }
```

Так как в задаче требуется найти количество четных чисел, находящихся между минимальным и максимальным, то необходимо установить, какой из элементов является левой границей интервала, а какой — правой. Левой границей будет являться минимальный элемент, если его позиция в массиве левее позиции максимального, в противном случае максимальный элемент — левая граница интервала, а минимальный — правая.

```
if (min_pos<max_pos)
{
  p=min_pos;
  k=max_pos;
}
else
{
```

```
  p=max_pos;
  k=min_pos;
}
```

Границы установлены, теперь можно начинать поиск четных чисел в заданном интервале. Поиск будет основан на проверке остатка от деления очередного элемента на 2. Если остаток равен 0, то число четное, и, следовательно, счетчик увеличивается на единицу:

```
kol=0;
for (i=p+1; i<k; i++)
  if (array[i]%2==0) kol++;
```

Затем следует вывод результата и завершение работы программы:

```
  cout<<kol;
  getch();
  return 0;
}
```

## Пример 2

Пусть дан массив из $N$ элементов, значения которых не превышают $N - 1$, причем значение любого элемента массива не равно номеру этого элемента.

Есть некий указатель, который, начиная с элемента массива под номером `K`, переходит к элементу `array[K]` и т. д. Рассмотрим программу, которая подсчитывает сумму элементов за `L` переходов. Предлагается рекурсивное решение данной задачи.

```
#include <iostream.h>
#include <conio.h>
#include <process.h>
int i, sum, pos, N, K, L, array[100];
void rec(int pos) // Объявление функции rec
{
  L--;         // Одна операция выполнена, счетчик уменьшается на 1
  if (L<0)     // Если все операции выполнены, то
  {
    cout<<sum;  // Выводит сумму
```

```
    getch();     // Задерживает экран
    exit(1);     // Выходит из программы
  }
  sum+=array[pos];  // Увеличивает сумму на значение элемента
                    // массива, который находится в позиции pos
  rec(array[pos]);  // Функция вызывает саму себя
                    // с новым параметром
}
main()
{
  cin>>N>>K>>L;   // Требует ввода трех переменных
  for (i=0; i<N; i++)
  cin>>array[i];  // Требует ввода очередного элемента массива
  sum=0;          // Начальное значение суммы
  rec(K);         // Запускает функцию rec
                  // с начальным параметром
}
```

Алгоритм данной задачи является рекурсивным, т. е. имеющим такую функцию `rec(int)`, которая вызывает саму себя при заданном параметре. Очевидно, что начальным параметром функции будет `K` — номер элемента, с которого начинается суммирование:

```
rec(K);
```

Так как алгоритм уже начал работу с некоторого элемента, то счетчик `L` уменьшаем на единицу.

```
void rec (int pos)
{
  L--;
```

После этого необходимо проверить состояние переменной `L`. Если все переходы уже сделаны, т. е. `L<0`, то программа выводит результат и завершается:

```
if (L<0)
{
  cout<<sum;
```

```
  getch();
  exit(1);
}
```

Если же данное условие не выполняется, то элемент под текущим номером суммируется с переменной `sum`, и функция `rec(int)` вызывает саму себя с параметром `array[pos]`:

```
  sum+=array[pos];
  rec(array[pos]);
}
```

## Пример 3

Некоторое издательство имеет $N$ авторов. Для каждого из них известно время, за которое автор пишет ту или иную книгу. Рассмотрим программу, которая определит, сумеют ли авторы написать $K \le N$ книг за время $T$, если каждый не хочет писать более чем одну, а некоторые могут и вообще не писать (при $K < N$).

Далее приведен код данной программы.

```
#include <iostream.h>
#include <conio.h>
main()
{
  int array[100], size, min_pos, i, start, min, temp, K, T;
  int sum;
  cin>>N>>K>>T;      // Требует ввода трех переменных
  for (i=0; i<N; i++)
    cin>>array[i];  // Требует ввода очередного элемента
                    // массива — времени написания книги
  start=0;          // Начальная левая граница подмассива
  // Запускаем цикл повторений, который выполняет операции,
  // пока все подмассивы не пройдены
  while (start!=N-1)
  {
    // Предположим, что элемент с индексом левой границы
    // подмассива минимален
```

```
    min=array[start];
    min_pos=start;      // Запоминает индекс элемента
    // Запускаем цикл поиска минимального элемента в подмассиве
    for (i=start; i<N; i++)
    if (array[i]<min)  // Сравнивает i-ый элемент массива со
                       // значением переменной min
    {
      min=array[i];    // Запоминает минимальное значение
      min_pos=i;       // Запоминает позицию элемента с
                       // минимальным значением
    }
    // Если позиция минимального значения элемента массива не
    // совпадает с предположенной ранее, то меняет местами
    // элементы с данными позициями
    if (min_pos!=start)
    {
      temp=array[start];
      array[start]=array[min_pos];
      array[min_pos]=temp;
    }
    start++;     // Уменьшает границу подмассива
  }
  sum=0;         // Начальное значение суммы
  for (i=0; i<K; i++)
    sum+=array[i];   // Находит сумму K первых элементов массива
  if (sum>T) cout<<"No";
  else cout<<"Yes";  // Выводит нужный результат
  getch();           // Задерживает экран
  return 0;          // Завершает программу
}
```

Данная задача сводится к тому, чтобы на каждом шаге находить минимальный элемент массива, помечать, что элемент уже просмотрен, а также суммировать их. Следует найти столько элементов, сколько книг необходимо написать.

Чтобы не искать на каждом шаге минимальный элемент, отсортируем массив по возрастанию, например, выборочной сортировкой, и найдем сумму первых K элементов:

```
for (i=0; i<K; i++)
  sum+=array[i];
```

Если в конце сумма минимальных элементов будет больше T, то, следовательно, авторы не уложатся в заданный срок, иначе заказчик будет доволен:

```
if (sum>T) cout<<"No";
else cout<<"Yes";
```

## Пример 4

Программа, которая в целочисленном массиве найдет и выведет на экран размер самой длинной последовательности одинаковых, подряд идущих чисел и само это повторяющееся число. Например, в массиве {2, 1, 2, 1, 2, 1, 1, 1, 1, 0, 2, 2, 2, 0, 5, 2, 8} самая длинная последовательность состоит из четырех единиц.

**Примечание**

Будем использовать рекурсивную функцию. Кроме того, в задаче следует предусмотреть наихудший случай, когда все числа различны. Тогда необходимо вывести длину последовательности — 1 и любое число из массива.

Код данной программы таков:

```
#include <iostream.h>
#include <conio.h>
#include <process.h>
int array[100], length, max_long, i, pos, size, number;
void rec(pos)  // Описывает функцию
{
  if (pos==size)  // Если элементы массива закончились, то
  {
    if (max_long==0) number=array[0]; // Проверяет, есть
                        // ли наихудший случай в массиве
```

```
    cout<<max_long<<" "<<number;      // Выводит результат
    getch();  // Задерживает экран
    exit(1);  // Завершает работу программы
  }
  if (array[pos+1]==array[pos]) // Если последующий
                  // элемент равен предыдущему, то
  {
    length++;   // Увеличивает длину последовательности на 1
    rec(pos+1); // Функция вызывает сама себя с новым параметром
  }
  else
    if (length>max_long)  // Сравнивает длину новой
                          // найденной последовательности
                          // с найденной ранее
    {
      max_long=length;    // Новая последовательность
                          // становится максимальной
      number=array[pos];  // Число, из которого состоит
                          // последовательность
      length=1;           // Возвращает начальное значение,
                          // для запуска поиска
                          // новой последовательности
    }
    length=1;    // Возвращает начальное значение для
                 // запуска поиска новой последовательности
    rec(pos+1);  // Функция вызывает сама себя с новым параметром
 }
main()
{
  cin>>size;         // Требует ввода количества элементов массива
  for (i=0; i<size; i++)
    cin>>array[i];  // Требует ввода очередного элемента массива
  max_long=1;  // Начальное значение длины максимальной
               // последовательности (состоящей
```

```
                // в худшем случае только из одного числа)
  length=1;     // Начальное значение длины
                // последовательности из одинаковых чисел
  rec(0);       // Запускает рекурсивную функцию rec с параметром 0.
                // 0 — это индекс массива, с которого
                // начинается поиск последовательности,
                // состоящей из одинаковых чисел
}
```

В программе начальное значение длины последовательности из одинаковых чисел равно единице (`length=1`), т. к. если в массиве существует хотя бы один элемент, то он уже представляет собой последовательность.

Как говорилось выше, будем использовать рекурсивную функцию с одним параметром. На первом шаге необходимо запустить ее с параметром 0 — индексом элемента массива, с которого начинается поиск последовательности, состоящей из одинаковых чисел:

```
rec(0);
```

В начале работы функции будем проверять состояние переменной `pos`. Если `pos=size`, то все элементы массива просмотрены, и требуется проверить наличие наихудшего случая, т. е. когда они различны. Тогда можно выводить длину последовательности, равную 1, и любое число из массива:

```
void rec(pos)
{
  if (pos==size)
  {
    if (max_long==0) number=array[0];
    cout<<max_long<<" "<<number;
    getch();
    exit(1);
  }
```

Если не все элементы массива пройдены, то проверяем, равны ли два элемента в позициях `pos` и `pos+1`. Если да, то увеличиваем

счетчик на единицу и запускаем функцию `rec(int)` с параметром `pos+1` (следующий элемент):

```
if (array[pos+1]==array[pos])
{
  length++;
  rec(pos+1);
}
```

Если же элементы не равны, т. е. цепочка из одинаковых чисел закончена, то сравниваем длину найденной последовательности с `max_long` (максимальная ранее найденная последовательность), и переменной `max_long` присваиваем максимальное из двух значений, а переменной `number` — число, из которой состоит последовательность. Переменной `length`, в свою очередь, возвращается в начальное значение для того, чтобы можно было организовать новый поиск, вызвав функцию с параметрам `pos+1`:

```
else
  if (length>max_long)
  {
    max_long=length;
    number=array[pos];
    length=1;
  }
  length=1;     // Готов для начала поиска
                // новой последовательности
  rec(pos+1);  // Функция вызывает сама себя с новым параметром
}
```

Так как алгоритм приступает к поиску новой последовательности, то переменной `length` присваиваем ее начальное значение и запускаем функцию `rec(int)` с параметром `pos+1`.

## Пример 5

Дан массив из *N* целых чисел. Рассмотрим программу, которая изменит массив таким образом, что в начале будут располагаться все числа, не равные нулю, а конце все нули.

```
#include <iostream.h>
#include <conio.h>
```

```
#include <process.h>
main()
{
  int array[100], i, pos, N, temp;
  cin>>N;          // Требует ввода количества
                   // элементов массива
  for (i=0; i<N; i++)
    cin>>array[i]; // Требует ввода очередного
                   // элемента массива
  pos=N-1;         // Начальное положение правого указателя
  while (array[pos]==0) pos--;  // Двигает указатель влево,
                     // пока не встретит первый ненулевой элемент
  i=0;           // Начальное положение левого указателя
  while (i<pos)  // Выполняет операции, пока указатель i
                 // находится левее указателя pos
  {
    // Если элемент массива под номером i равен нулю, то меняет
    // местами его с ненулевым элементом под номером pos
    if (array[i]==0)
    {
      temp=array[i];
      array[i]=array[pos];
      array[pos]=temp;
      while (array[pos]==0) pos--; // Двигает указатель влево,
                 // пока не встретит первый ненулевой элемент
    }
    i++;  // Передвигает указатель вправо
  }
  for (i=0; i<N; i++)
    cout<<array[i]<<" ";  // Выводит очередной элемент массива
  getch();   // Задерживает экран
  return 0;  // Завершает программу
}
```

Данная программа работает по следующему алгоритму. Сначала вводятся два указателя. Первый указатель `i` привязан к нулевому

элементу массива, а последний элемент массива будет начальным положением второго указателя `pos`:

```
pos=N-1;
```

Будем двигать указатель `pos` от начальной позиции влево, пока не встретим первый ненулевой элемент:

```
while (array[pos]==0) pos--;
```

Указатель `i` будем перемещать вправо и проверять элемент `array[i]`. Если он равен нулю, то поменяем его местами с элементом `array[pos]` и снова будем искать ненулевой элемент, перемещая указатель `pos`:

```
while (i<pos)
{
  if (array[i]==0)
  {
    temp=array[i];
    array[i]=array[pos];
    array[pos]=temp;
    while (array[pos]==0) pos--;
  }
  i++;
}
```

Перемещаем оба указателя до тех пор, пока они не встретятся.

# 4.8. Упражнения

1. Напишите программу, которая отсортирует массив целых чисел по возрастанию (убыванию) между максимальным и минимальным значениями массива.
2. Даны некоторые числа *N* и *M*, количество цифр в каждом из которых не превышает 256. Напишите программу, которая возведет число *N* в степень *M* и выведет результат на экран.
3. Даны два массива. Напишите программу, которая находит такую последовательность максимальной длины, которая содержится и в первом, и во втором массивах.

4. Определена некоторая последовательность цифр. Напишите программу, которая определит, какое максимальное число можно составить из данной последовательности при условии, что оно должно делиться на 15.
5. Дан массив, элементы которого равны либо единице, либо двойке. Напишите программу, которая расставит элементы массива таким образом, чтобы за каждой единицей следовала двойка, причем, если каких-то элементов больше, то они выписываются в конце. Например, последовательность {2, 1, 1, 1, 2, 1} должна преобразоваться в {1, 2, 1, 2, 1, 1}.
6. Напишите программу, которая отсортирует элементы массива по возрастанию (убыванию) их последней цифры.
7. Квадратная матрица размером $N \times N$ заполнена целыми числами из промежутка [–9; 9]. Назовем фигурой всякую часть этой матрицы, состоящую из $K$ последовательных строк и $K$ последовательных столбцов. Сумму всех чисел в этой фигуре назовем суммой фигуры. Напишите программу, которая определит фигуру с максимальной суммой.
8. Напишите программу, которая сформирует такой массив из $N/2$ элементов, сумма которых равна числу $N$.
9. Напишите программу, которая заполняет матрицу $N \times N$ числами от 1 до $N^2$ по спирали, начиная от верхнего левого угла.
10. Напишите программу, которая определит и выведет на экран сумму главной и побочной диагонали матрицы.

**Внимание**

Во всех задачах *главы 4* входные данные считать корректными.

# Глава 5

# Структуры данных

*Структуры данных* представляют собой набор некоторым образом сгруппированных данных. Простейшим примером структуры данных служит массив.

Все элементы массива являются равнодоступными, а при решении некоторых задач возникает необходимость работать с данными в том или ином порядке. Для этого служат такие структуры данных, как очереди, стеки и списки.

## 5.1. Очередь

*Очередью* будем называть структуру данных, для которой определены операции добавления и удаления элементов.

Очередь, в которой нет ни одного элемента, называется *пустой*. В этом случае указатели `Start` и `Last` указывают на одно и то же место.

Очередь можно реализовать несколькими способами. Простейшим способом реализации служит массив и два указателя. Первый указывает на начало очереди, а второй, соответственно, на ее конец.

*Начало очереди* — это номер элемента массива, который будет удаляться.

*Конец очереди* — это номер элемента массива, в который будут заноситься данные.

Если некоторый элемент занесен в очередь раньше других, то он раньше других и подлежит обработке. Данное определение можно сравнить с обычной очередью в магазине — если покупатель пришел первым, то он и обслуживается первым. Другими словами, очередь — эта та структура данных, в которую элемент "заходит" первый и первый же "выходит". Именно поэтому очередь нередко обозначают аббревиатурой *FIFO* (First In, First Out — первый зашел, первый вышел).

**Внимание**

Элементы удаляются из очереди в том же порядке, в котором они были добавлены в очередь.

Пусть у нас есть очередь из трех элементов `A`, `B` и `C`, в которую первым был помещен элемент `A`, а последним — элемент `C` (рис. 5.1).

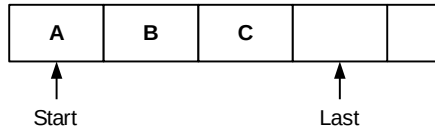



**Рис. 5.1.** Изображение очереди с первым элементом `A` и последним `C`

Здесь указатель `Start` (начало очереди) определяет место в массиве, откуда очередной элемент будет удаляться, а указатель `Last` (конец очереди) определяет место в массиве, куда очередной элемент будет добавляться.

Как было сказано выше, *сначала обрабатывается* элемент, который *первый вошел* в очередь, в нашем случае это элемент `A`. Он будет удален из очереди путем смещения указателя `Start` на один элемент вправо (рис. 5.2).

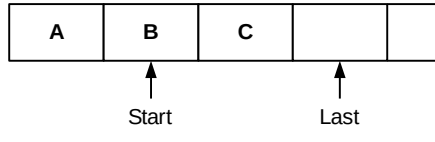



**Рис. 5.2.** Удаление обработанного элемента из очереди

Теперь добавим некоторый элемент `D` в очередь. Для этого необходимо записать `D` в позицию `Last`, а затем передвинуть указатель `Last` на один элемент вправо (рис. 5.3).

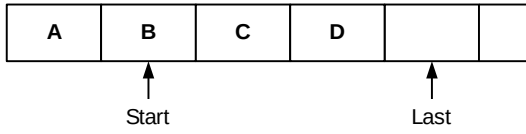



**Рис. 5.3.** Добавление элемента в очередь

## Операции над очередями

Для реализации операций над очередями будем использовать следующие функции:

- добавление элемента в очередь

```
void Add(int Last, int Number)
```

- проверка пустоты очереди

```
void Empty(int Start, int Last)
```

- удаление элемента из очереди

```
void Remove(int Start, int Last)
```

Для моделирования очереди будем использовать массив `Queue` (очередь).

### Добавление элемента в очередь

Пусть размер очереди равен `maxqueue`. Тогда функция добавления элемента в очередь будет выглядеть так:

```
void Add(int Last, int number)
{
  if (Last==maxqueue) exit(1);  // Очередь полна
  Queue[Last]=number;   // Добавляет элемент number в очередь
  Last++;   // Сдвигает указатель Last на один элемент вправо
}
```

### Проверка очереди на наличие элементов

Данная функция возвращает `p=1`, если очередь пуста, и `p=2`, если в очереди есть элементы.

```
void Empty(int Start, int Last)
{
  if (Start==Last) p=1;   // Очередь пуста
  else p=2;               // Очередь не пуста
}
```

### Удаление элемента из очереди

Функция удаления элемента из очереди такова:

```
void Remove (int Start, int Last)
{
  if (Start==Last) exit(1);  // Очередь пуста
  Start++;     // Сдвигает указатель Start на один элемент вправо
}
```

## 5.2. Стеки

*Стеком* называется структура данных, элементы которой организованы таким образом, что их удаление будет осуществляться противоположно тому порядку, в котором производилось добавление. Стек можно представить в виде вертикального массива с одним указателем на верхний элемент, традиционно называемый *вершиной* стека (`Top`). В текущий момент времени вершина стека является доступной. Новый элемент можно добавлять в вершину стека, он же первый подлежит обработке. Стек часто обозначают аббревиатурой *LIFO* (Last In, First Out — последний зашел, первый вышел).

Если стек содержит только один элемент, и этот элемент удаляется, то в результате выполнения операции в данный стек больше не сможет быть занесен ни один элемент. Такой стек называется *пустым*.

Допустим, имеется стек из элементов `A`, `B` и `C`. Если элемент `A` был первым занесен в стек, а элемент `C` — последним, то стек будет выглядеть так, как показано на рис. 5.4.

Если удалить из стека элемент `C`, доступный для операций, переместив указатель `Top` на один элемент вниз, то стек будет выглядеть, как показано на рис. 5.5.

Теперь добавим в стек некоторый элемент `D`. Для этого запишем его в вершину стека, сместив указатель `Top` на один элемент вниз (рис. 5.6).

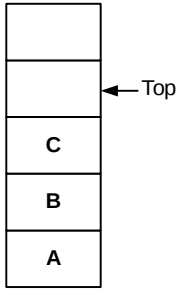



**Рис. 5.4.** Содержимое стека

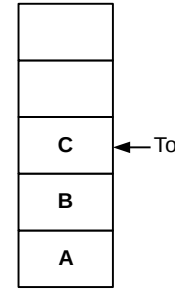



**Рис. 5.5.** Удаление элемента из стека

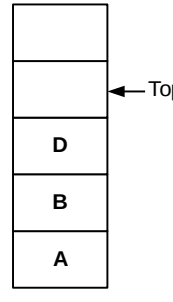



**Рис. 5.6.** Добавление элемента в стек

## Операции над стеками

Для реализации операций над стеками будем использовать следующие функции:

- добавление элемента в стек

```
void Push(LIFO, Top, Number)
```

- проверка пустоты стека

  `void Empty(Top)`

- удаление элемента из стека

  `void Pop(Top)`

## Добавление элемента в стек

Пусть размер стека равен `maxstack`. Чтобы поместить элемент в стек, необходимо указать сам элемент и индекс (адрес) вершины стека.

```
void Push(int Top, int Element)
{
  if (Top==maxstack) exit(1);  // Стек заполнен
  Stack[Top]=Element;     // Добавляет элемент Element в стек
  Top++;      // Сдвигает указатель Top на один элемент вверх
}
```

## Проверка стека на наличие элементов

Данная процедура возвращает `p=1`, если стек пуст, и `p=2`, если в стеке есть элементы.

```
void Empty(int Top)
{
  if (Top==0) p=1;   // Стек пуст
  else p=2;          // Стек не пуст
}
```

## Удаление элемента из стека

Функция удаления элементов из стека такова:

```
void Remove(int Top)
{
  if (Top==0) exit(1);  // Стек пуст
  Top--;                // Сдвигает указатель Top
                        // на один элемент вниз
}
```

# 5.3. Списки

*Списком* называется некоторая последовательность элементов, связаных при помощи указателей. Список, который не содержит ни одного элемента, называется *пустым*.

Каждый элемент списка (иногда называемый *узлом*) состоит из двух частей. Первая часть содержит данные, принадлежащие элементу, и является информационной. Вторая часть содержит указатель и является справочной. Указатель определяет местоположение элемента списка, связанного с данным элементом. На рис. 5.7 приведен пример списка с элементами `A`, `B`, `C` и `D`.

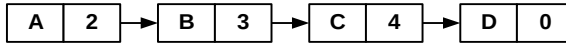



**Рис. 5.7.** Связь элементов списка

Стрелка в данном списке обозначает связь элементов списка. Указатель (т. е. вторая часть элемента списка) элемента `A` определяет номер следующего элемента списка. Указатель элемента `B` показывает, что за ним следует элемент `C` и т. д. Отсутствие или нулевое значение указателя означает, что данный элемент является *последним* в списке.

Элементы в списке могут быть связаны посредством разных указателей. Например, есть списки, в которых для каждого элемента указатель задает местоположение следующего. Существуют списки, в которых указатель задает местоположение предыдущего элемента. Такие списки называют *однонаправленными*. В отличие от однонаправленных списков, в *двунаправленных* списках для каждого элемента задаются два указателя, которые определяют местоположение как предыдущего элемента, так и последующего.

При работе со списками используется, как правило, еще один указатель, который определяет *первый элемент списка*.

Для представления однонаправленного списка в памяти компьютера можно использовать двумерный массив `List[2][`*`maxlist`*`]`,

где *maxlist* — это количество элементов в списке. В данном двумерном массиве любой элемент List[0][i] содержит значение элемента списка, а в List[1][i] находится указатель на следующий элемент.

Двунаправленный список так же представляется в виде двумерного массива List[3][*maxlist*]. В этом случае элемент списка расположен в List[0][i], а в List[1][i] и List[2][i] находятся указатели, которые указывают соответственно на предыдущий и последующий элементы списка.

## Реализация операций над списками

Основными операциями над списками являются:

- переход к очередному элементу списка;
- добавление в список нового элемента;
- поиск заданного элемента;
- удаление элемента из списка.

Выполнение этих операций основывается на использовании и изменении указателей.

**Примечание**

В отличие от очередей и стеков, элемент в список может быть добавлен в любое место.

### Создание пустого списка

Определим значение указателей для пустого списка. Пусть переменная US будет задавать адрес первого элемента списка свободных мест, а переменная UN — адрес первого элемента списка. Для пустого списка US=1, UN=0.

Приведем пример создания пустого списка при помощи функции New.

```
void New()
{
  for (i=1; i<maxlist-1; i++)
```

```
    List[1][i]=i+1;
  List[1][maxlist]=0;
  US=1;
  UN=0;
}
```

**Примечание**

Для точного и правильного использования операций над списками будем считать, что индексация элементов массива начинается с единицы, а не с нуля.

Вид пустого списка приведен в табл. 5.1.

***Таблица 5.1.*** *Вид пустого списка*

| **Характеристика** | **Значения** | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Индекс массива | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| Элемент списка | | | | | | | | | | |
| Указатели | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 0 |

## Добавление в список нового элемента

Для добавления элемента `X` в список необходимо в переменную `Pos` запомнить адрес первого элемента списка свободных мест. Затем нужно поместить по этому адресу элемент `X` и в качестве указателя на следующий элемент списка записать номер первого элемента списка. Теперь указателем первого элемента списка будет указатель `US`, т. к. по его адресу записан новый элемент. Он из разряда свободных элементов переходит в разряд занятых. Указателем списка свободных мест будет `Pos`, т. е. элемент, который следует в списке свободных мест за только что исключенным из него элементом.

Приведем пример кода функции `Insert(int, int, int)`, которая добавляет элемент `X` в список.

```
void Insert(int US, int UN, int X)
{
```

```
  if (Sp==0) exit(1);  // Список не содержит элементов
  Pos=List[1][US];     // Находит адрес первого элемента
                       // списка свободных мест
  List[0][US]=X;       // Помещает по этому адресу значение X
  List[1][US]=UN;      // В качестве указателя
                       // на следующий элемент помещает
                       // номер первого элемента списка
  UN=US;
  US=Pos;
}
```

Рассмотрим подробно операцию добавления элемента `X` в список.

Сначала указатель списка свободных мест `US=1`, а первый элемент списка `UN=0`. Первоначальный вид списка приведен в табл. 5.1.

Добавим в список элемент `A`, используя функцию `Insert(int, int, int)`. Вид обновленного списка приведен в табл. 5.2. Значит, `US=2`, а `UN=A`.

***Таблица 5.2.*** *Добавление в список элемента* `A`

| **Характеристика** | **Значения** | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Индекс массива | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| Элемент списка | A | | | | | | | | | |
| Указатели | 0 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 0 |

Добавим в список элемент `B`, используя функцию `Insert(int, int, int)`. `US=3`, `UN=B` (табл. 5.3).

***Таблица 5.3.*** *Добавление в список элемента* `B`

| **Характеристика** | **Значения** | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Индекс массива | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| Элемент списка | A | B | | | | | | | | |
| Указатели | 0 | 1 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 0 |

Аналогичным образом добавим в таблицу элементы C и D (табл. 5.4): US=5, а UN=D.

***Таблица 5.4.*** *Список после добавления в него элементов C и D*

| Характеристика | Значения | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Индекс массива | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| Элемент списка | A | B | C | D | | | | | | |
| Указатели | 0 | 1 | 2 | 3 | 6 | 7 | 8 | 9 | 10 | 0 |

Теперь список свободных позиций содержит элементы с индексами 5, 6, 7, 8, 9, 10, а первым элементом списка является элемент D. Логическое расположение элементов в списке будет D, C, B, A. Этот порядок определяется указателями.

## Поиск элемента в списке

Поиск элемента X в списке осуществляется следующим образом. Переменной Lop присваиваем значение указателя первого элемента списка. Далее просматриваем список до тех пор, пока не найдем нужный элемент, либо не достигнем конца списка. Если по окончанию процедуры Lop=0, значит, в списке нет элемента со значением X.

Далее приведен пример функции Find(int, int, int), которая организовывает поиск в заданном списке элемента со значением X.

```
void Find(int X, int UN)
{
  Lop=UN;
  while (List[0][Lop]!=X && Lop!=0)
    Lop=List[1][Lop];     // Присваивает переменной Lop ссылку
                          // на следующий элемент
}
```

## Удаление элемента из списка

Чтобы удалить элемент X из списка, необходимо сначала найти его в списке, например, при помощи функции Find(int, int, int). Если элемент в списке найден, то происходит его удаление.

При удалении изменяется указатель записи элемента, идущего в списке перед элементом X таким образом, чтобы он указывал на элемент, идущий в списке после элемента X.

Пусть Pos — индекс элемента массива, который необходимо удалить, а K — индекс элемента массива, который следует за элементом List[1][Pos].

Функция Delete(int, int, int) удаляет элемент X из списка. Далее приведен ее код.

```
void Delete(int K, int Pos, int US)
{
  List[1][K]=List[1][Pos];
  List[1][Pos]=US;
  US=Pos;
}
```

Проследим удаление элемента X на примере списка из табл. 5.4.

Удалим элемент C из списка при помощи функции Delete(int, int, int).

***Таблица 5.5.*** *Удаление элемента из списка*

| Характеристика | Значения | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Индекс массива | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| Элемент списка | A | B | C | D | | | | | | |
| Указатели | 0 | 1 | 5 | 2 | 6 | 7 | 8 | 9 | 10 | 0 |

После проделанных операций указатель списка свободных мест US=3, а первым элементом списка UN=D.

## Добавление элемента в список после заданного элемента

Необходимо сначала найти заданный элемент X, а затем вставить после него новый элемент E. При добавлении нового элемента в список после элемента X по адресу первого элемента списка сво-

бодных мест записывается значение нового элемента. Указатель нового элемента принимает значение, равное адресу следующего за ним элемента списка, а указатель предшествующего ему элемента списка принимает значение адреса нового элемента.

Эти операции реализует функция `Insert_after(int, int, int)`. Далее приведен ее код.

```
void Insert_after(int Pos, int US, int E)
{
  List[0][US]=E;
  List[1][US]=List[1][Pos];
  List[1][Pos]=US;
}
```

Здесь `Pos` — это индекс элемента `X`.

После добавления элемента `E` после элемента `B` в предыдущей таблице получим список, приведенный в табл. 5.6.

***Таблица 5.6.*** *Добавление элемента* `D` *после элемента* `A`

| **Характеристика** | **Значения** | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Индекс массива | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| Элемент списка | A | B | E | D | | | | | | |
| Указатели | 3 | 1 | 0 | 2 | 5 | 6 | 7 | 8 | 9 | 0 |

## Сортировка списков

При работе со списками очень часто возникает необходимость перестановки элементов списка в определенном порядке. Такая задача называется *сортировкой списка*, и для ее решения существуют различные методы.

Если в задаче требуется отсортировать все элементы списка, то он сортируется как обычный массив, любым из ранее предложенных методов (например, табл. 5.7, 5.8).

***Таблица 5.7.*** *Исходный список*

| **Характеристика** | **Значения** | | | | |
|---|---|---|---|---|---|
| Индекс массива | 1 | 2 | 3 | 4 | 5 |
| Элемент списка | 3 | 1 | 6 | 3 | 9 |
| Указатели | 2 | 3 | 4 | 5 | 0 |

***Таблица 5.8.*** *Отсортированный список*

| **Характеристика** | **Значения** | | | | |
|---|---|---|---|---|---|
| Индекс массива | 1 | 2 | 3 | 4 | 5 |
| Элемент списка | 1 | 3 | 3 | 6 | 9 |
| Указатели | 2 | 3 | 4 | 5 | 0 |

В некоторых задачах возникает необходимость сортировки списка таким образом, чтобы по указателям можно было восстановить исходный список (например, табл. 5.9, 5.10).

***Таблица 5.9.*** *Исходный список*

| **Характеристика** | **Значения** | | | | |
|---|---|---|---|---|---|
| Индекс массива | 1 | 2 | 3 | 4 | 5 |
| Элемент списка | 3 | 1 | 6 | 3 | 9 |
| Указатели | 2 | 3 | 4 | 5 | 0 |

***Таблица 5.10.*** *Отсортированный список*

| **Характеристика** | **Значения** | | | | |
|---|---|---|---|---|---|
| Индекс массива | 1 | 2 | 3 | 4 | 5 |
| Элемент списка | 1 | 3 | 3 | 6 | 9 |
| Указатели | 3 | 2 | 5 | 4 | 0 |

В данном случае при сортировке необходимо менять местами не только элементы списка, но и их указатели. Обмен значениями можно организовать функцией `Swap(x,y)`, которая будет менять местами элементы и указатели списка в позициях `x` и `y`. Далее приведен код данной функции.

```
void Swap(int x, int y)
{
  int Temp;
  Temp=List[0][x];
  List[0][x]=List[0][y];
  List[0][y]=Temp;
  Temp=List[1][x];
  List[1][x]=List[1][y];
  List[1][y]=Temp;
}
```

Данная сортировка списка необходима для того, чтобы на некотором `i`-ом шаге можно было вернуться на шаг под номером `j` и выполнить другие действия.

## Слияние списков

Упорядоченные списки `List_1` и `List_2` длиной `M` и `N` сливаются в один упорядоченный список `List_res` длины `M` + `N`, если каждый элемент из `List_1` и `List_2` входит в `List_res` ровно один раз. В табл. 5.11—5.13 приведен пример слияния списков.

***Таблица 5.11.*** *Упорядоченный список* `List_1`

| **Характеристика** | **Значения** | | | | |
|---|---|---|---|---|---|
| Индекс массива | 1 | 2 | 3 | 4 | 5 |
| Элемент списка | 6 | 17 | 23 | 40 | 80 |
| Указатели | 2 | 3 | 4 | 5 | 0 |

***Таблица 5.12.*** *Упорядоченный список* `List_2`

| Характеристика | Значения | | | | |
|---|---|---|---|---|---|
| Индекс массива | 1 | 2 | 3 | 4 | 5 |
| Элемент списка | 2 | 3 | 34 | 1 | 32 |
| Указатели | 2 | 5 | 0 | 1 | 3 |

***Таблица 5.13.*** *Упорядоченный список* `List_res`

| Характеристика | Значения | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Индекс массива | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| Элемент списка | 1 | 2 | 3 | 6 | 17 | 23 | 32 | 34 | 40 | 80 |
| Указатели | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 0 |

Алгоритм слияния двух списков прост. Его основная идея выглядит следующим образом. Для слияния списков `List_1` и `List_2` список `List_res` сначала полагается пустым, а затем к нему последовательно приписывается первый узел из `List_1` или `List_2`, оказавшийся меньшим и отсутствующий в `List_res`.

# 5.4. Примеры использования структур данных

## Пример 1

Дана матрица размером $M \times N$, заполненная 0 и –1. Рассмотрим программу, которая определяет количество нулевых островов в матрице. Островом назовем совокупность смежных клеток матрицы, содержащих нули.

Далее приведен полный код программы с комментариями.

```
#include <iostream.h>
#include <conio.h>
```

```
const N=4;  // Количество строк в матрице
const M=4;  // Количество столбцов в матрице
// Инициализированный способ объявления массива
int array[N+2][M+2]= { {-1, -1, -1, -1, -1, -1},
                       {-1,  0,  0, -1, -1, -1},
                       {-1,  0, -1,  0,  0, -1},
                       {-1, -1,  0, -1,  0, -1},
                       {-1,  0, -1, -1, -1, -1},
                       {-1, -1, -1, -1, -1, -1}};
// Для того чтобы не выйти за пределы матрицы
// при проверке смежных клеток, необходимо объявить матрицу
// (N+2)*(M+2) и края матрицы заполнить числами -1
int Queue[2][M*N];  // Объявляет очередь
int Start, Last;    // Объявляет переменные, которые служат
                    // указателями начала и конца очереди
                    // соответственно
int Label, i, j, p, a, b;
void Add(int x, int y);    // Прототип функции, которая будет
                           // добавлять элемент в очередь
void Find()                // Объявление функции
{
  p=0;  // Пусть есть нулевые элементы в матрице
  for (i=1; i<=N && p==0; i++)
    for (j=1; j<=M && p==0; j++)
      if (array[i][j]==0)  // Если какой-то элемент матрицы
                           // равен нулю, то...
      {
        p=1;
        Add(i,j);  // вызывает функцию Add(int,int) для
                   // занесения координат в очередь
      }
}
void Test() // Объявление функции
{
  Label++;  // Увеличивает метку, т. е. говорит о начале
```

```
            // поиска нового острова
  while (Start!=Last)  // Пока очередь не пуста...
  {
    a=Queue[0][Start];  // Запоминает координату X текущей клетки
    b=Queue[1][Start];  // Запоминает координату Y текущей клетки
    // Проверяет все клетки, смежные с текущей
    if (array[a+1][b]==0) Add(a+1,b);
    if (array[a-1][b]==0) Add(a-1,b);
    if (array[a][b+1]==0) Add(a,b+1);
    if (array[a][b-1]==0) Add(a,b-1);
    Start++;            // Удаляет первый элемент из очереди
  }
}
main()      // Объявление функции
{
  Start=0;  // Начальная позиция указателя Start
  Last=0;   // Начальная позиция указателя Last
  Label=0;
  p=1;
  do
  {
    Find();  // Вызывает функцию, которая проверяет, есть ли
             // нулевой элемент в матрице
    if (p==1) Test();
    // Если такой элемент найден, то вызывает функцию, которая
    // проверяет смежные клетки с данным элементом и заносит их
    // координаты в очередь, если в них значение равно нулю
  } while (p==1);
  cout<<Label;  // Выводит результат на экран
  getch();      // Задерживает экран
  return 0;     // Завершает программу
}
void Add(int x, int y)   // Описание функции
{
  Queue[0][Last]=x;      // Заносит в очередь координату X
```

```
  Queue[1][Last]=y;       // Заносит в очередь координату Y
  array[x][y]=Label;      // Помечает текущую клетку меткой
  Last++;                 // Сдвигает указатель Last вправо
}
```

Данная задача является наглядным классическим способом использования очереди. Здесь мы будем рассматривать очередь как двумерный массив, в котором две строки и $M \times N$ столбцов. В первой строке будет находиться координата `X` той или иной клетки, а во второй — координата `Y` этой же клетки.

Для решения задачи необходимо выполнить ряд следующих действий.

На первом шаге работы алгоритма следует найти такие `i` и `j`, чтобы выполнялось равенство `array[i][j]=0`. Это можно сделать при помощи функции `Find`:

```
void Find()
{
  p=0;
  for (i=1; i<=N && p==0; i++)
    for (j=1; j<=M && p==0; j++)
      if (array[i][j]==0)
      {
        p=1;
        Add(i,j);
      }
}
```

Переменной `p` присваивается значение `1`, если нашлись такие `i` и `j`, что `array[i][j]=0`, в противном случае значение переменной остается равным нулю. В начале работы функции переменная `p` обнуляется.

Если `p=1`, то координаты (`X=i`, `Y=j`) клетки заносятся в очередь, а указатель `Last` сдвигается на один элемент вправо. Данные действия выполняет функция `Add(int, int)`. Затем функция `Find` прекращает свою работу. Каждый нулевой остров помечается меткой `Label`, поэтому при добавлении координат очередной

клетки она помечается этой меткой. Далее приведен код функции `Add(int,int)`.

```
void Add(int x, int y)
{
  Queue[0][Last]=x;
  Queue[1][Last]=y;
  array[x][y]=Label;
  Last++;
}
```

Затем необходимо взять клетку с координатами (`Queue[0][Start]`, `Queue[1][Start]`) и проверить все смежные с ней клетки. Если какая-нибудь из них имеет значение, равное нулю, то ее координаты заносятся в очередь. Затем указатель `Start` сдвигается на один элемент вправо, т. е. удаляется из очереди элемент в позиции `Start`, и берутся следующие координаты из очереди. К ним применяется та же операция. Данные действия выполняются до тех пор, пока очередь не станет пустой, т. е. пока `Start` не равно `Last`. Таким образом, будет найден очередной нулевой остров, все клетки которого будут помечены меткой `Label`.

Данные операции выполняет функция `Test()`:

```
void Test()
{
  Label++;
  while (Start!=Last)
  {
    a=Queue[0][Start];
    b=Queue[1][Start];
    if (array[a+1][b]==0) Add(a+1,b);
    if (array[a-1][b]==0) Add(a-1,b);
    if (array[a][b+1]==0) Add(a,b+1);
    if (array[a][b-1]==0) Add(a,b-1);
    Start++;
  }
}
```

Все действия повторяются до тех пор, пока в матрице остаются элементы, равные нулю. Когда таких элементов не найдется, то функция `main()` выведет результат на экран.

```
main()
{
  Start=0;
  Last=0;
  Label=0;
  p=1;
  do
  {
    Find();
    if (p==1) Test();
  } while (p==1);
  cout<<Label;
  getch();
  return 0;
}
```

## Пример 2

Даны координаты двух клеток шахматного поля размером 8×8. Рассмотрим программу, которая определит и выведет на экран минимальное количество ходов, за которое конь переместится из одной клетки в другую.

Ниже приведен код программы с комментариями.

```
#include <iostream.h>
#include <conio.h>
int array[8][8], x, y, x1, y1, i, j, a, b, Start, Last;
int Queue[2][64];
main()
{
  cin>>x>>y>>x1>>y1;   // Требует ввода двух пар координат
  for (i=0; i<8; i++)
  for (j=0; j<8; j++)
```

```
  array[i][j]=-1;    // Заполняет матрицу значением -1
array[x][y]=0;       // Клетка начала пути
Queue[0][0]=x;  // Заносит в очередь координату X
                // начальной клетки
Queue[1][0]=y;  // Заносит в очередь координату Y
                // начальной клетки
Start=0;        // Начальная позиция левого указателя
Last=1;         // Начальная позиция правого указателя
// Запускаем цикл, который продолжает свою работу до тех пор,
// пока очередь не пуста или искомый конечный элемент не помечен
while (Start!=Last && array[x1][y1]==-1)
{
  a=Queue[0][Start];  // Запоминает координату X текущей клетки
  b=Queue[1][Start];  // Запоминает координату Y текущей клетки
  // Следующие 8 операторов if перебирают все возможные
  // ходы коня, предусматривая возможность выхода за пределы
  // поля, и если клетка, в которую можно пойти,
  // не помечена, то ее координаты заносятся в очередь,
  // а она сама помечается увеличенной на единицу меткой
  // клетки, из которой мы попали в текущую
  if (array[a+2][b-1]==-1 && a+2<8 && b-1<8)
  {
    Queue[0][Last]=a+2;
    Queue[1][Last]=b-1;
    Last++;
    array[a+2][b-1]=array[a][b]+1;
  }
  if (array[a+2][b+1]==-1 && a+2<8 && b+1<8)
  {
    Queue[0][Last]=a+2;
    Queue[1][Last]=b+1;
    Last++;
    array[a+2][b+1]=array[a][b]+1;
  }
  if (array[a-2][b+1]==-1 && a-2<8 && b+1<8)
```

```
{
  Queue[0][Last]=a-2;
  Queue[1][Last]=b+1;
  Last++;
  array[a-2][b+1]=array[a][b]+1;
}
if (array[a-2][b-1]==-1 && a-2<8 && b-1<8)
{
  Queue[0][Last]=a-2;
  Queue[1][Last]=b-1;
  Last++;
  array[a-2][b-1]=array[a][b]+1;
}
if (array[a-1][b+2]==-1 && a-2<8 && b+2<8)
{
  Queue[0][Last]=a-1;
  Queue[1][Last]=b+2;
  Last++;
  array[a-1][b+2]=array[a][b]+1;
}
if (array[a+1][b+2]==-1 && a+1<8 && b+2<8)
{
  Queue[0][Last]=a+1;
  Queue[1][Last]=b+2;
  Last++;
  array[a+1][b+2]=array[a][b]+1;
}
if (array[a+1][b-2]==-1 && a+1<8 && b-2<8)
{
  Queue[0][Last]=a+1;
  Queue[1][Last]=b-2;
  Last++;
  array[a+1][b-2]=array[a][b]+1;
}
if (array[a-1][b-2]==-1 && a-1<8 && b-2<8)
```

```
    {
      Queue[0][Last]=a-1;
      Queue[1][Last]=b-2;
      Last++;
      array[a-1][b-2]=array[a][b]+1;
    }
    Start++;  // Удаляет первый элемент из очереди
  }
  cout<<array[x1][y1];  // Выводит результат
  getch();              // Задерживает экран
  return 0;             // Завершает программу
}
```

Идея решения основывается на использовании очереди. Вначале в очередь помещается элемент, определяющий исходное положение шахматного коня, а соответствующая клетка поля помечается как посещенная:

```
array[x][y]=0;
Queue[0][0]=x;
Queue[1][0]=y;
```

Алгоритм продолжает свою работу до тех пор, пока очередь не становится пустой или не помечена конечная клетка:

```
while (Start!=Last && array[x1][y1]==-1)
{
```

Затем из очереди извлекается очередной элемент (он всегда первый), который определяет некоторую позицию (`a`, `b`):

```
a=Queue[0][Start];
b=Queue[1][Start];
```

После этого отыскиваются клетки в пределах поля, которые достижимы из (`a`, `b`) одним ходом шахматного коня и которые еще не помечены, и заносятся в очередь, а клетки помечаем как посещенные:

```
if (array[a+2][b-1]==-1 && a+2<8 && b-1<8)
{
```

```
  Queue[0][Last]=a+2;
  Queue[1][Last]=b-1;
  Last++;
  array[a+2][b-1]=array[a][b]+1;
}
```

И так далее, все восемь вариантов хода коня (см. выше в коде программы).

После того как первый элемент очереди обработан, его можно удалить:

```
  Start++;
}
```

Далее алгоритм продолжит свою работу, взяв из очереди следующий элемент. Либо, если очередь пуста, выведет результат, и программа завершится:

```
  cout<<array[x1][y1];
  getch();
  return 0;
}
```

## Пример 3

Вокруг ведущего стоят *N* человек, которые пронумерованы по часовой стрелке числами от 1 до *N*. Ведущий, начиная с первого, отсчитывает *M* человек, и тот, на котором остановился счет, выходит из круга. Счет возобновляется со следующего человека, и так до тех пор, пока не останется один. Ниже приведена программа, которая определяет номер оставшегося человека.

```
#include <iostream.h>
#include <conio.h>
int List[100], before_start, i, N, M, Start;
void Find();      // Прототип функции
main()
{
  cin>>N>>M;      // Требует ввода двух переменных
```

```
  for (i=0; i<N-1; i++)
    List[i]=i+1;  // Каждый элемент массива равен
                  // индексу следующего за ним элемента
  List[N-1]=0;    // Последний элемент ссылается на нулевой
  while (Start!=List[Start])
  {
    Find(); // Запускает функцию, которая отсчитывает M человек
    // Удаляет человека под номером Start из круга
    List[before_start]=List[Start];
    Start=List[Start];
  }
  cout<<(Start+1);    // Выводит результат
  getch();            // Задерживает экран
  return 0;           // Завершает программу
}
void Find()       // Описание функции
{
  for (i=0; i<M-1; i++)
  {
    before_start=Start;  // Запоминает номер человека на
                         // данном шаге
    Start=List[Start];   // Номер человека, на котором
                         // остановился счет
  }
}
```

Задача довольно известна в определенных кругах. Алгоритм решения прост, но в то же время демонстрирует классическое использование списков.

В качестве списка будем использовать массив `List[N]`. Данный массив назовем массивом указателей. Каждый элемент массива равен номеру человека, стоящего за ним, т. е. индексу следующего за ним элемента. Заполнение массива выполняется при помощи цикла `for` следующим образом:

```
for (i=0; i<N-1; i++)
  List[i]=i+1;
```

При этом последний человек ссылается на первого: `List[N-1]=0`.

На каждом шаге будем искать `M`-го человека при помощи указателей. Данную операцию демонстрирует функция `Find()`:

```
void Find()
{
  for (i=0; i<M-1; i++)
  {
    before_start=Start;
    Start=List[Start];
  }
}
```

В переменной `before_start` будем хранить номер человека, который находится перед тем, который должен выбыть из круга. После выполнения этой функции будет известен номер человека, который должен выбыть (переменная `Start`), теперь необходимо удалить его из списка. Это можно сделать следующим образом:

```
List[before_start]=List[Start];
Start=List[Start];
```

Суть этой операции состоит в том, чтобы элемент, предшествующий удаляемому, ссылался на элемент, который следует за удаляемым.

## Пример 4

Рассмотрим программу, которая определит, является ли некоторая последовательность *P* палиндромом группы *B*. *Палиндромом группы B* называется последовательность, в которой можно выделить одно либо несколько подмножеств палиндромов, причем элементы подмножеств с номером *i* не являются элементами подмножества с номером *j*. Например, последовательность {1, 2, 3, 3, 2, 1, 4, 4} является палиндромом группы B, чего нельзя сказать про последовательность {1, 2, 3, 3, 2, 2, 1} или {1}.

Далее приведен код данной программы.

```
#include <iostream.h>
#include <conio.h>
char LIFO[100], ch[100];
int Empty, i, Top;
void Push(char number);  // Прототип функции
main()
{
  cin>>ch;      // Требует ввода последовательности элементов
  Top=0;        // Вершина стека
  for (i=0; i<strlen(ch); i++)
  {
    Push(ch[i]);   // Вызывает функцию, которая добавляет
                   // элемент последовательности в стек
    if (LIFO[Top-1]==LIFO[Top-2] && Top>1)
      Top-=2;      // Удаляет два элемента из стека, если они
                   // равны и в стеке более чем один элемент
  }
  // Выводит результат
  if (Top==0) cout<<"Yes"; else cout<<"No";
  getch();         // Задерживает экран
  return 0;        // Завершает программу
}
void Push (char number)  // Описание функции
{
  LIFO[Top]=number;      // Заносит в стек очередной элемент
                         // последовательности
  Top++;                 // Увеличивает указатель стека
}
```

Решение данной задачи состоит в том, что мы будем заносить в стек (добавляя элемент в вершину стека) поочередно каждый элемент последовательности при помощи функции `Push(char)`:

```
void Push(char number)
{
```

```
  LIFO[Top]=number;
  Top++;
}
```

Если элемент, который мы заносим в стек, равен элементу, находящемуся в вершине стека, то будем удалять эти два элемента из нашей структуры данных:

```
if (LIFO[Top-1]==LIFO[Top-2] && Top>1) Top-=2;
```

Данные операции выполняются, пока все элементы последовательности не пройдут через стек. Если по завершению этих операций стек пуст, т. е. `Top=0`, то последовательность является палиндромом группы B, иначе ответ будет отрицательный:

```
  if (Top==0) cout<<"Yes"; else cout<<"No";
  getch();
  return 0;
}
```

## Пример 5

Дано поле размером $N \times M$ клеток, в которые помещены нули и единицы. Назовем его лабиринтом, а единицы стенами. Рассмотрим программу, которая определит и выведет на экран кратчайший путь фишки из точки $(x, y)$ лабиринта в точку $(x_1, y_1)$. Причем фишка может ходить только по нулевым ячейкам поля.

Алгоритм этой задачи почти аналогичен алгоритму, приведенному в примере 2 данной главы. Суть его состоит в том, чтобы перебрать все возможные пути из одной ячейки в другую. Так как при реализации используется структура данных "очередь", то кратчайший путь будет найдет автоматически.

Код программы выглядит следующим образом:

```
#include <iostream.h>
#include <conio.h>
const N=5;
const M=5;
main()
```

```
{
  int x, y, x1, y1, x2, y2, p, i, j, Start, Last;
  int FIFO[3][N*M], result[2][N*M];
  int array[N][M] = { {0, 1, 0, 1, 0},
                      {0, 1, 0, 1, 0},
                      {0, 0, 0, 0, 0},
                      {0, 0, 0, 0, 1},
                      {0, 0, 0, 0, 1} };
  cout<<"Введите стартовые координаты"<<endl;
  cin>>x1>>y1;
  cout<<"Введите конечные координаты"<<endl;
  cin>>x2>>y2;
  array[x1][y1]=-1;  // Стартовую клетку помечает
                     // как пройденную
  FIFO[0][0]=x1;     // Заносит в очередь координату X
                     // стартовой клетки
  FIFO[1][0]=y1;     // Заносит в очередь координату Y
                     // стартовой клетки
  FIFO[2][0]=0;
  Start=0;   // Начальное положение левого указателя очереди
  Last=1;    // Начальное положение правого указателя очереди
             // Цикл продолжает работу,
             // пока не будет достигнута конечная клетка
  while (x2!=FIFO[0][Start] || y2!=FIFO[1][Start])
  {
    x=FIFO[0][Start];  // Извлекает из очереди
                       // первую координату X
    y=FIFO[1][Start];  // Извлекает из очереди
                       // первую координату Y
    // Проверяет все клетки, смежные с текущей
    if (array[x+1][y]==0 && array[x+1][y]==0 && x+1<N)
    {
      FIFO[0][Last]=x+1;    // Заносит в очередь координату X
      FIFO[1][Last]=y;      // Заносит в очередь координату Y
      FIFO[2][Last]=Start;  // Заносит в очередь
```

```
                        // координаты клетки,
                        // из которой мы попали в текущую
    Last++;             // Сдвигает указатель
                        // на один элемент вправо
    array[x+1][y]=-1;   // Помечает клетку как пройденную
  }
  if (array[x-1][y]==0 && array[x-1][y]==0 && x-1>=0)
  {
    FIFO[0][Last]=x-1;  // Заносит в очередь координату X
    FIFO[1][Last]=y;    // Заносит в очередь координату Y
    FIFO[2][Last]=Start; // Заносит в очередь
                        // координаты клетки,
                        // из которой мы попали в текущую
    Last++;             // Сдвигает указатель
                        // на один элемент вправо
    array[x-1][y]=-1;   // Помечает клетку как пройденную
  }
  if (array[x][y+1]==0 && array[x][y+1]==0 && y+1<N)
  {
    FIFO[0][Last]=x;    // Заносит в очередь координату X
    FIFO[1][Last]=y+1;  // Заносит в очередь координату Y
    FIFO[2][Last]=Start; // Заносит в очередь
                        // координаты клетки,
                        // из которой мы попали в текущую
    Last++;             // Сдвигает указатель
                        // на один элемент вправо
    array[x][y+1]=-1;   // Помечает клетку как пройденную
  }
  if (array[x][y-1]==0 && array[x][y-1]==0 && y>0)
  {
    FIFO[0][Last]=x;    // Заносит в очередь координату X
    FIFO[1][Last]=y-1;  // Заносит в очередь координату Y
    FIFO[2][Last]=Start; // Заносит в очередь
                        // координаты клетки,
                        // из которой мы попали в текущую
```

```
      Last++;                // Сдвигает указатель
                             // на один элемент вправо
      array[x][y-1]=-1;      // Помечает клетку как пройденную
    }
    Start++;                 // Удаляет элемент из очереди
  }
  p=Start;  // Порядковый номер в очереди конечного элемента
            // с координатами (x2, y2)
  i=0;
  while (p!=0)
  {
    result[0][i]=FIFO[0][p];
    result[1][i]=FIFO[1][p];
    p=FIFO[2][p];        // Переходит к элементу,
                         // который предшествовал текущему
    i++;
  }
  result[0][i]=x1;       // Заносит в массив результатов
                         // координаты стартовой клетки
  result[1][i]=y1;
  for (j=i; j>=0; j--)   // Выводит результат
    cout<<result[0][j]<<" "<<result[1][j]<<endl;
  getch();               // Задерживает экран
  return 0;              // Завершает программу
}
```

Алгоритм начинает работу со стартовой клетки с координатами ($x_1$, $y_1$). В очередь будем помещать координаты клеток, которые фишка прошла. Если фишка побывала в некоторой клетке поля, то в пройденной ячейке будем ставить –1. На первом этапе помечаем, что фишка стартует из клетки ($x_1$, $y_1$) и заносим ее координаты в очередь:

```
array[x1][y1]=-1;
FIFO[0][0]=x1;
FIFO[1][0]=y1;
FIFO[2][0]=0;
```

Начальные значения указателей очереди:

```
Start=0;
Last=1;
```

Мы имеем такие начальные значения указателей, потому что на первом шаге мы занесли начальные координаты фишки в очередь в позицию 0, и следующий элемент будет добавляться в позицию 1, на текущее положение указателя конца очереди. Это следует из определения очереди.

Все основные операции выполняет цикл `while`, который продолжает свою работу до тех пор, пока не будет достигнута конечная клетка с координатами ($x_2$, $y_2$). Первые две строки тела цикла запоминают очередные координаты клетки, т. е. первый элемент в очереди:

```
while (x2!=FIFO[0][Start] || y2!=FIFO[1][Start])
{
  x=FIFO[0][Start];
  y=FIFO[1][Start];
```

Далее необходимо проверить все ячейки, смежные с выбранной выше. Если некоторая из них не помечена, то ее координаты заносятся в очередь, следовательно, указатель `Last` перемещается на один элемент вперед, а клетка на поле помечается как пройденная. Так как условие задачи ориентировано на то, чтобы вывести путь, необходимо для каждой пройденной ячейки запомнить координаты той клетки, из которой мы в нее попали, т. е. поместить ее координаты в массив предшествующих координат.

```
  if (array[x+1][y]==0 && array[x+1][y]==0 && x+1<N)
  {
    FIFO[0][Last]=x+1;
    FIFO[1][Last]=y;
    FIFO[2][Last]=Start;
    Last++;
    array[x+1][y]=-1;
  }
  if (array[x-1][y]==0 && array[x-1][y]==0 && x-1>=0)
```

```
  {
    FIFO[0][Last]=x-1;
    FIFO[1][Last]=y;
    FIFO[2][Last]=Start;
    Last++;
    array[x-1][y]=-1;
  }
  if (array[x][y+1]==0 && array[x][y+1]==0 && y+1<N)
  {
    FIFO[0][Last]=x;
    FIFO[1][Last]=y+1;
    FIFO[2][Last]=Start;
    Last++;
    array[x][y+1]=-1;
  }
  if (array[x][y-1]==0 && array[x][y-1]==0 && y>0)
  {
    FIFO[0][Last]=x;
    FIFO[1][Last]=y-1;
    FIFO[2][Last]=Start;
    Last++;
    array[x][y-1]=-1;
  }
  Start++;
}
```

Для вывода пути будем использовать массив координат предшествующих клеток. В последнюю очередь будут выведены координаты конечной клетки, перед ними — координаты предыдущей клетки и т. д. Данные будем заносить в массив результатов `result`:

```
p=Start;
i=0;
while (p!=0)
{
```

```
  result[0][i]=FIFO[0][p];
  result[1][i]=FIFO[1][p];
  p=FIFO[2][p];
  i++;
}
```

Последним элементом массива будет стартовая клетка:

```
result[0][i]=x1;
result[1][i]=y1;
```

Чтобы получить последовательность координат пути в прямом порядке (от исходной точки к конечной), а не в обратном, будем выводить элементы массива на экран, начиная с последнего:

```
  for (j=i; j>=0; j--)
    cout<<result[0][j]<<" "<<result[1][j]<<endl;
  getch();
  return 0;
}
```

# 5.5. Упражнения

1. Напишите программу, которая определит, правильно ли расставлены скобки в выражении, если оно состоит из скобок типа: `( ) [ ] { }`.
2. Напишите программу, которая определит значение арифметического выражения. Например, (2 + 7) × 5 + 4. Арифметическое выражение должно вводиться с клавиатуры в одну строку без пробелов (используйте тип данных `char`).
3. Задана некоторая плоскость $N \times M$. Каждой клетке поля соответствует некоторое число. Фишка, начиная движение из угла поля с координатами (0, 0), может перемещаться влево, вправо, вверх и вниз. Напишите программу, которая определит, какую максимальную сумму может набрать фишка за $K$ ходов, при условии, что закончить свое движение она должна в любой клетке $N$-ой строки поля.

4. Определено поле $N \times M$, в пределах которого можно размещать корабли. Кораблями будем называть совокупность подряд идущих клеток. Если клетка `array[i][j]` является клеткой корабля, то `array[i][j]=1`. Напишите программу, которая определит правильность расстановки кораблей. Расстановка считается верной, если выполняются следующие условия:
   - стенки кораблей не соприкасаются;
   - на поле расположены ровно четыре одноклеточных, три двухклеточных, два трехклеточных и один четырехклеточный корабль;
   - по краям поля корабли не располагаются.
5. На шахматном поле $N \times M$ расположены две фишки с начальными координатами $(x, y)$ и $(x_1, y_1)$ соответственно. Первая фишка может двигаться только по белым клеткам, а вторая — исключительно по черным. Пусть некоторые клетки с координатами $(i, j)$ вырезаны из поля, т. е. ни одна из фишек не может перейти на это клеточное поле размером $1 \times 1$. Напишите программу, которая определит, какая из двух фишек попадет первой в клетку с координатами $(K, L)$.
6. В ряд стоят $N$ человек. Ведущий считает от 1 до $M$, начиная с человека под номером $K$. Человек, на котором закончился счет, становится в конец ряда. Напишите программу, которая определит стартовый номер человека оказавшегося в конце ряда после $L$-го счета.
7. Напишите программу, которая в таблице `array` размера `Size` за один просмотр каждый элемент заменит ближайшим следующим за ним элементом, который больше его. Если такого элемента нет, то заменит его нулем.
8. По кругу расположено $N$ монет гербами вверх и $M$ монет гербами вниз. Обходя круг по ходу часовой стрелки, каждая $S$-ая монета переворачивается. В первый раз счет начинается с монеты, лежащей гербом вверх. Напишите программу, которая определит, в каком порядке надо расставить монеты, чтобы после $K$ ходов стало $L$ монет, лежащих гербами вверх.

9. Имеется $N$ черных и белых карточек, сложенных в стопку. Карточки раскладываются на столе в одну линию следующим образом: первая кладется на стол, вторая — вниз стопки, третья — на стол, четвертая — вниз стопки и т. д., пока все карточки не будут выложены на стол. Напишите программу, которая определит, каким должно быть исходное расположение карточек в стопке, чтобы разложенные на столе карточки чередовались по цвету: белая, черная, белая, черная и т. д.
10. $N$ серых и $M$ белых мышей сидят в кругу. Кошка ходит по кругу по часовой стрелке и съедает каждую $S$-ую мышку. В первый раз счет начинается с серой мышки. Составьте алгоритм, определяющий порядок, в котором сидели мышки, если через некоторое время осталось $K$ серых и $L$ белых мышей.

**Внимание**

Во всех задачах *главы 5* входные данные считать корректными.

# Глава 6

# Файлы

С развитием программных технологий мы получаем возможность обрабатывать и пересылать все более объемные и сложные данные. 90% из них хранятся в файлах.

Для использования файлов в приложении разработчику приходится решать множество задач. Главные из них — поиск необходимого файла и выполнение с ним операций ввода/вывода.

Основные принципы организации файловой системы мало изменились со времен MS-DOS. Файловые системы FAT32 и NTFS, появившиеся в Windows 98 и Windows 2000, не изменяют главного понятия файла и способов обращения к нему.

В этой главе мы изучим все основные способы работы с файлами в приложениях C++ на конкретных примерах создания программного кода.

*Файл* — это именованная структура данных, представляющая собой последовательность элементов данных одного типа, причем количество элементов последовательности практически не ограничено. В первом приближении файл можно рассматривать как массив переменной длины неограниченного размера.

У любого файла есть имя, что дает возможность программе работать одновременно с несколькими файлами. В файле могут содержаться данные любых типов, за исключением файлов, т. е. нельзя создавать текстовый файл и хранить в нем файл такого же рода.

Чтобы обеспечить безошибочную работу с файлами, пользователю требуется включить в программу заголовочный файл fstream.h.

# 6.1. Обращение к файлам

Файл, как и любую другую переменную, необходимо описать. В C++ это можно осуществить следующим образом:

```
ofstream filename("c:/input.txt",ios::app);
```

Здесь в двойных кавычках указано имя файла на диске, а `filename` — это программное имя файла, т. е. имя, при помощи которого пользователь будет обращаться к файлу в коде программы. Если файл не существует в указанном каталоге, то он будет создан. Далее, после запятой необходимо указать режим обращения к файлу. Перечень режимов обращения к файлам приведен в табл. 6.1.

***Таблица 6.1.*** *Режимы обращения к файлам*

| Запись режима | Функция режима |
|---|---|
| `ios::in` | Считывание из файла, для потока `ifstream` устанавливается по умолчанию. Курсор устанавливается в начало файла по умолчанию |
| `ios::out` | Запись поверх имеющейся, для потока `ofstream` устанавливается по умолчанию. Курсор устанавливается в начало файла по умолчанию |
| `ios::ate` | Запись и считывание. Запись поверх первоначальной. Курсор устанавливается в конец файла по умолчанию |
| `ios::app` | Запись. Добавление в конец файла после уже имеющейся записи |

Например, команда `ifstream filename("input.txt")` открывает файл input.txt с программным именем `filename` в режиме считывания. Аргумент `ios::in` опущен, т. к. он будет установлен по умолчанию.

Чтобы *открыть* файл как для считывания, так и для записи, необходимо использовать следующую команду:

```
fstream filename("input.txt", ios::in | ios::out);
```

Здесь оператором `|` объединяются два аргумента (`ios::in` и `ios::out`).

Чтобы проверить после открытия файла исполнение данной операции, можно использовать проверку вида:

```
if (!filename)
{
  cout<<"File not open";
  exit(1);
}
```

После использования файла необходимо *закрыть* его. Это можно сделать следующим образом:

```
filename.close();
```

## Последовательная запись в файл

Для записи некоторой информации в файл используют операцию, которая является аналогом операции `cout<<`:

```
filename<<блок1<<блок2<<...<<блокN;
```

Рассмотрим простейшую программу, которая создает файл и дописывает в него данные некоторого клиента, введенные с клавиатуры.

```
#include <iostream.h>
#include <fstream.h>   // Заголовочный файл
                       // для использования файлов
#include <conio.h>
main()
{
  char num[10], name[10];
  ofstream output("Base.txt", ios::app);  // Открытие файла
                                          // для записи
```

```
  cin>>num>>name;                    // Ввод данных
  cout<<"Данные занесены в файл. Ожидаю новых данных"<<endl;
  output<<num<<" "<<name<<endl;      // Записывает в файл данные
  output.close();                    // Закрывает файл
  getch();                           // Задерживает экран
  return 0;                          // Завершает программу
}
```

## Последовательное чтение из файла

Для чтения из файла некоторой информации используют операцию, которая является аналогом операции `cin>>`:

`filename>>`*блок1*`>>`*блок2*`>>...>>`*блокN*`;`

Рассмотрим простейшую программу, которая читает из файла некоторые данные и выводит их на экран.

```
#include <iostream.h>
#include <fstream.h> // Заголовочный файл для использования файлов
#include <conio.h>
main()
{
  char num[10], name[10];
  ifstream input("Base.txt");     // Открывает файл для
                                  // считывания информации
  if (!input) exit(1);
  while (!input.eof())            // Пока не конец файла
  {
    input>>num>>name;             // Чтение данных из файла
    cout<<num<<" "<<name<<endl;   // Выводит на экран
                                  // данные, прочитанные из файла
  }
  input.close();                  // Закрывает файл
  getch();                        // Задерживает экран
  return 0;                       // Завершает программу
}
```

При открытии файла для считывания информации аргумент `ios::in` опущен, т. к. он устанавливается по умолчанию.

В программе выполняется проверка процедуры открытия файла на корректность следующим образом:

```
if (!input) exit(1);
```

В данном случае, если файл не будет открыт, то программа завершает свою работу.

Чтение данных выполняется до тех пор, пока не будет достигнут конец файл, т. е. пока не будет прочитан его последний символ. Структура повторений `while` завершит свою работу по достижении конца файла, и операции в теле цикла перестанут выполняться.

## Произвольная запись в файл

Ранее мы рассматривали последовательную запись в файл, теперь рассмотрим произвольную, т. е. запись некоторой информации в любое место файла.

Позиционирование (установка курсора на некоторое место) будет производиться при помощи функции `seekp()`. В круглых скобках указывается номер позиции в байтах. Отсчет от начала задается по умолчанию.

Запись реализуется при помощи функции `write` (писать):

```
filename.seekp(N);
filename.write((char*)&M,sizeof(M));
```

Данная запись означает, что указатель ставится на `N`-й байт от начала файла `filename` и на это место записывается значение переменной `M` с размером `sizeof(M)` байтов.

## Произвольное чтение из файла

Позиционирование (установка курсора на некоторое место) будет производиться при помощи функции `seekg()`. В круглых скобках указывается номер позиции в байтах. Отсчет от начала задается по умолчанию.

Чтение реализуется при помощи функции `read` (читать):

```
filename.seekg(K);
filename.read((char*)&Z,sizeof(Z));
```

Данная запись означает, что указатель следует установить на `K`-й байт от начала файла `filename` и присвоить переменной `Z` значение, записанное от позиции `K` на `sizeof(Z)` позиций вперед.

## 6.2. Поиск в файле

Поиск некоторой информации в файле является стандартной задачей при работе с файлами. Рассмотрим программу, которая определяет все вхождения текста из файла Base_1.txt в файл Base_2.txt. Далее приведен код программы с комментариями:

```
#include <iostream>
#include <fstream.h>
#include <conio.h>
main()
{
  char ch[100], buffer;
  int w, i, K, pos, p, SK;
  bool test;       // Объявляется переменная логического типа
  ifstream input("Base_1.txt");   // Открывает файл для чтения
  ifstream input_1("Base_2.txt"); // Открывает файл для чтения
  input_1>>ch;          // Читает из файла строку, все
                        // вхождения которой необходимо найти
  SK=p=0;
  K=strlen(ch);         // Возвращает длину строки
  while (!input.eof()) // Пока не просмотрен весь файл
                        // eof (end of file) — конец файла
  {
    pos=input.tellg(); // Определяет текущее положение указателя
    // Читает из файла один символ, позиция которого равна
    // позиции указателя
    input.read((char*)&buffer,sizeof(buffer));
```

```
    // Если это не символ, который обозначает переход на новую
    // строку, то увеличивает счетчик SK на единицу
    if (buffer!='\n') SK++;
    if (buffer==ch[0])
    {
      test=true;
      i=1;
      while (i<K && test)
      {
        // Читает из файла очередной символ
        input.read((char*)&buffer, sizeof(buffer));
        // Символ не входит в строку поиска, и этот символ не '\n'
        if (buffer!=ch[i] && buffer!='\n')
        {
          test=false;
          input.seekg(pos);  // Перемещает указатель
                             // на начальное значение
          // Читает из файла один символ, позиция которого
          // равна позиции указателя
          input.read((char*)&buffer, sizeof(buffer));
        } if (buffer!='\n') i++;
      }
      if (test)  // Если строка поиска и строка из файла
      {          // совпали, то выводит место вхождения
        p=1;
        cout<<SK<<endl;
        SK+=(K-1);
      }
    }
  }
  input.close();          // Закрывает файл
  if (p==0) cout<<"No";  // Выводит уведомление, что данная
                          // строка в файле не содержится
  getch();                // Задерживает экран
  return 0;               // Завершает программу
}
```

Рассмотрим представленный алгоритм поиска по частям.

Сначала файл Base_1.txt открывается для чтения следующим образом:

```
ifstream input("Base_1.txt");
```

Аналогично открывается и файл Base_2.txt:

```
ifstream input_1("Base_2.txt");
```

Основные действия выполняются в цикле `while`. Данный цикл продолжает работу до тех пор, пока указатель не достигнет конца файла. В ходе работы цикла переменной `pos` присваивается текущее положение указателя:

```
while (!input.eof())
{
  pos=input.tellg();
```

Далее программа читает из файла символ в позиции, равной текущему положению указателя, и, если этот символ не равен символу перехода на новую строку, то происходит сравнение прочитанного символа с первым символом строки поиска. Если они оказываются равными, то выполняется блок сравнений `if`.

```
input.read((char*)&buffer,sizeof(buffer));
if (buffer!='\n') SK++;
if (buffer==ch[0])
  {
```

Внутри этого блока программа должна прочитать все символы, которые будут совпадать с символами текущей строки. Здесь используются счетчик `i` и цикл `while`, который следит за длиной прочитанной строки и значением булевой переменной `test`. Оператор повторений прерывает работу в том случае, если `i` становится равной длине строки поиска или значение `test` станет `false`:

```
test=true;
i=1;
while (i<K && test)
{
```

В теле цикла `while` происходит чтение из файла очередного символа. Затем он сравнивается с соответствующим символом строки поиска. Если символы совпадают, то счетчик `i` увеличивается на единицу. Если нет, то файл возвращается к состоянию, предшествующему этим сопоставлениям, т. е. указатель файла должен вернуться на ту позицию, с которой началось сравнивание. Это происходит при помощи функции `seekg()` с переменной `pos` в качестве параметра:

```
  input.read((char*)&buffer, sizeof(buffer));
  if (buffer!=ch[i] && buffer!='\n')
  {
    test=false;
    input.seekg(pos);
    input.read((char*)&buffer, sizeof(buffer));
  } else i++;
}
```

Если строка поиска обнаружена в файле, то на экран выводится соответствующий результат:

```
    if (test)
    {
      p=1;
      cout<<SK<<endl;
      SK+=(K-1);
    }
  }
}
```

Если признак `p` становится равным единице, то строка найдена в файле, по крайней мере, один раз.

Если весь файл Base_1 прочитан, а искомая строка не найдена, то программа выводит соответствующий результат и завершается:

```
  input.close();
  if (p==0)cout<<"No";
  getch();
  return 0;
}
```

## Поиск и замена в файле

Данная задача усложнена тем, что необходимо не только проверить, существует ли некоторая строка в файле, но и при положительном результате заменить ее на некоторую другую строку.

Далее приведен код программы с комментариями:

```
#include <iostream>
#include <fstream.h>
#include <conio.h>
main()
{
  char ch[100], replace[100], buffer;
  int i, K, T, pos;
  bool test;
  ifstream input("Base.txt");    // Открывает файл для чтения
  ofstream output("Result.txt"); // Открывает файл для записи
  cin>>ch>>replace;      // Читает строку, которую необходимо
                         // найти, а также строку замены
  K=strlen(ch);          // Определяет длину строки поиска
  T=strlen(replace);     // Определяет длину строки замены
  while (!input.eof())   // Пока не конец файла
  {
    pos=input.tellg();   // Определяет текущее положение
    // Читает из файла один символ, позиция которого равна
    // позиции указателя
    input.read((char*)&buffer,sizeof(buffer));
    if (buffer==ch[0])
    {
      test=true;
      i=1;
      while (i<K && test)
      {
        // Читает из файла очередной символ
        input.read((char*)&buffer, sizeof(buffer));
        if (buffer!=ch[i])   // Символ не входит в строку поиска
```

```
        {
          test=false;
          input.seekg(pos);  // Перемещает указатель
                             // на начальное значение
          // Читает из файла один символ, позиция которого
          // равна позиции указателя
          input.read((char*)&buffer, sizeof(buffer));
        } else i++;
      }
      if (test)  // Если строка поиска и строка из файла
      {          // совпали, то записывает в файл output
                 // вместо строки поиска строку замены
        output.write((char*)&replace,T);
        test=false;
      } else output.write((char*)&buffer,sizeof(buffer));
    } else output.write((char*)&buffer,sizeof(buffer));
  }
  input.close();   // Закрывает файл
  output.close();  // Закрывает файл
  return 0;        // Завершает программу
}
```

Основные действия выполняются в цикле `while`. Данный цикл продолжает работу до тех пор, пока указатель не достигнет конца файла. В ходе работы цикла переменной `pos` присваивается текущее положение указателя следующим образом:

```
while (!input.eof())
{
  pos=input.tellg();
```

Далее программа читает из файла символ в позиции, равной текущему положению указателя, и сравнивает его с первым символом строки поиска. Если оба символа равны, то выполняется блок сравнений `if`.

```
input.read((char*)&buffer,sizeof(buffer));
if (buffer==ch[0])
{
```

Если нет, то символ, хранящийся в переменной `buffer`, записывается в файл с результатом без изменений.

```
else output.write((char*)&buffer,sizeof(buffer));
```

Внутри блока `if` программа должна прочитать все символы, которые будут совпадать с символами текущей строки. В программе используются счетчик `i` и цикл `while`, который следит за длиной прочитанной строки и значением булевой переменной `test`. Оператор повторений прерывает выполнение в том случае, если `i` становится равной длине строки поиска или значение `test` станет `false`:

```
test=true;
i=1;
while (i<K && test)
{
```

В теле цикла `while` происходит чтение из файла очередного символа. Затем он сравнивается с соответствующим символом строки поиска. Если символы совпадают, то счетчик `i` увеличивается на единицу. Если нет, то файл возвращается к состоянию, предшествующему этим сопоставлениям, т. е. указатель файла должен вернуться на ту позицию, с которой началось сравнивание. Это происходит при помощи функции `seekg()` с переменной `pos` в качестве параметра:

```
  input.read((char*)&buffer,sizeof(buffer));
  if (buffer!=ch[i])
  {
    test=false;
    input.seekg(pos);
    input.read((char*)&buffer,sizeof(buffer));
  } else i++;
}
```

Если строка поиска обнаружена в файле, то в файл-результат записывается строка замены вместо строки поиска, а булевой переменной `test` присваивается значение `false`. Если строка не

была обнаружена, то в файл-результат выводится просто символ, содержащийся в переменной `buffer`:

```
    if (test)
    {
      output.write((char*)&replace,T);
      test=false;
    } else output.write((char*)&buffer,sizeof(buffer));
  }
}
```

После завершения работы цикла `while` программа закрывает входной и выходной файлы и завершается:

```
  input.close();
  output.close();
  return 0;
}
```

# 6.3. Примеры программ обработки файлов

## Пример 1

Рассмотрим программу, которая определит, сколько раз встречается та или иная буква (вводится с клавиатуры) в тексте. Тест находится в файле `input`.

Далее приведен полный код программы:

```
#include <iostream.h>
#include <conio.h>
#include <fstream.h>
main()
{
  char buffer, K;
  int result;
  // Открывает файл с программным именем input для чтения
```

```
  ifstream input("String.txt");
  cin>>K;                // Требует ввода символа
  while (!input.eof())   // Пока не достигнут конец файла
  {
    input.get(buffer);   // Читает следующий символ из файла
    if (buffer==K) result++;
  }
  input.close();         // Закрывает файл
  cout<<result;          // Выводит результат
  getch();               // Задерживает экран
  return 0;              // Завершает программу
}
```

Решение данной задачи заключается в чтении из файла всех символов поочередно и сравнении с заданным. Если они равны, то счетчик увеличивается на единицу. Для считывания символов будем использовать функцию `get()`, где в круглых скобках указывается параметр — переменная, в которую следует прочитать символ. По достижении конца строки указатель файла автоматически переходит на следующую строку:

```
while (!input.eof())
{
  input.get(buffer);
  if (buffer==K) result++;
}
```

Затем следует вывод результата и закрытие программы:

```
  input.close();
  cout<<result;
  getch();
  return 0;
}
```

## Пример 2

В данном примере приведена программа, которая определяет максимальное число для каждой строки файла. Последний элемент строки равен нулю, причем никакие другие элементы

нулю не равны. Результаты (максимальные числа строк) записываются в файл.

```
#include <iostream.h>
#include <conio.h>
#include <fstream.h>
main()
{
  int max, number;
  ifstream input("input.txt");      // Открывает файл
                                    // для чтения
  ofstream output("output.txt");    // Открывает файл
                                    // для записи
  max=0;            // Начальное значение максимального
                    // значения первой строки
  while (!input.eof())
  {
    input>>number;          // Читает из файла очередное число
    if (number>max) max=number;  // Сравнивает его
                                 // с максимальным
    // Если достигнут конец строки, т. е. прочитан элемент 0,
    // то выводит в выходной файл максимальное значение строки
    // и переводит указатель на другую строку
    if (number==0) output<<max<<endl;
    max=0;          // Обнуляет для начала нового поиска
  }
  input.close();    // Закрывает файл
  output.close();   // Закрывает файл
  return 0;         // Завершает программу
}
```

Операции станут выполняться до тех пор, пока не будет достигнут конец файла:

```
while (!input.eof())
{
```

Суть решения состоит в том, чтобы последовательно читать по одному числу из файла и сравнивать его с максимальным. Если

считанное число больше максимального, то последнему присваивается значение считанного:

```
input>>number;
if (number>max) max=number;
```

Сравнения происходят до тех пор, пока не будет считан ноль: тогда в выходной файл выводится переменная `max`, а указатель переводится на следующую строку. Это означает, что некоторая строка файла пройдена, и максимальное значение для нее найдено и выведено. Затем переменная `max` становится равной нулю:

```
if (number==0) output<<max<<endl;
max=0;
```

После завершения работы цикла `while` необходимо закрыть оба файла:

```
  input.close();
  output.close();
  return 0;
}
```

## Пример 3

Представим программу, которая определяет и отправляет в выходной файл сумму каждой строки матрицы $M \times N$. Матрица находится в файле `input`. В первой строке файла указан размер исходной матрицы.

Далее приведен полный код программы с комментариями:

```
#include <iostream.h>
#include <conio.h>
#include <fstream.h>
int M, N, i, j, buffer, Row_summa;
main()
{
  ifstream input("input.txt");   // Открывает файл для чтения
  ofstream output("output.txt"); // Открывает файл для записи
  input>>N>>M;                   // Считывает из файла размер матрицы
```

```
  for (i=0; i<N; i++)      // Перебирает все строки матрицы
  {
    for (j=0; j<M; j++)    // Перебирает все элементы
                           // строки матрицы
    {
      input>>buffer;       // Читает из файла очередной
                           // элемент строки
      Row_summa+=buffer;   // Суммирует считанный элемент с
    }                      // общей суммой строки
    output<<Row_summa<<endl;  // Выводит общую сумму строки
                              // в файл
    Row_summa=0;              // Обнуляет значение общей суммы
  }
 input.close();      // Закрывает входной файл
 output.close();     // Закрывает выходной файл
 return 0;           // Завершает программу
}
```

В данной программе используются два файла: входной `input` и выходной `output` (программные имена файлов). Откроем `input` для чтения, а `output` для записи:

```
ifstream input("input.txt");
ofstream output("output.txt");
```

На первом шаге необходимо прочитать из файла размер матрицы. Это можно сделать при помощи последовательного считывания:

```
input>>N>>M;
```

Для нахождения суммы каждой строки будем использовать два цикла `for`. Первый (внешний) отвечает за перебор всех строк файла от 0 до $N - 1$. Внутренний цикл отвечает за перебор всех элементов $i$-ой строки и совершает операции суммирования до $M - 1$:

```
for (i=0; i<N; i++)
{
  for (j=0; j<M; j++)
  {
```

Во внутреннем цикле необходимо последовательно читать из файла каждый элемент и суммировать его с переменной `Row_summa` (сумма строки):

```
  input>>buffer;
  Row_summa+=buffer;
}
```

Когда все элементы строки просуммированы, внутренний цикл передает управление внешнему, но перед этим необходимо записать результат в выходной файл и обнулить переменную суммы:

```
  output<<Row_summa<<endl;
  Row_summa=0;
}
```

Затем оба файла закрываются, и программа завершает свою работу:

```
  input.close();
  output.close();
  return 0;
}
```

## Пример 4

Рассмотрим программу, которая читает из входного файла $L$ чисел и определяет в записи числа количество всех содержащихся цифр в отдельности.

Например, если в файле `input` будет находиться единственное число 1342 и $L = 1$ соответственно, то выходной файл `output` будет таким:

```
1342
0 0
1 1
2 1
3 1
4 1
5 0
```

```
6 0
7 0
8 0
9 0
```

Далее приведен полный код программы с комментариями.

```
#include <iostream.h>
#include <conio.h>
#include <fstream.h>
main()
{
  int N, K, L, c, temp, kol_vo;
  // Открывает файл с программным именем input для чтения
  ifstream input("input.txt");
  // Открывает файл с программным именем output для записи
  ofstream output("output.txt");
  input>>L;      // Считывает из файла
  while (L!=0)
  {
    input>>N;    // Считывает из файла очередное число
    K=N;         // Делает копию числа N
    output<<N<<endl;  // Выводит в файл число N и переводит
                      // курсор на следующую строку
    // Начальное значение цифры (int c), количество повторений
    // которой в заданном числе необходимо подсчитать
    c=0;
    kol_vo=0;
    while (c!=10)
    {
      K=N;
      while (K!=0)
      {
        temp=K%10;    // Выделяет последнюю цифру числа
        K/=10;        // Удаляет из числа последнюю цифру
        // Если выделенная цифра совпала с текущей проверяемой
```

```
        // (int c), то увеличивает счетчик на единицу
        if (temp==c) kol_vo++;
      }
      output<<c<<" "<<kol_vo<<endl; // Выводит в файл
                                    // результат данного шага
      kol_vo=0;
      c++;          // Определяет следующую цифру для проверки
    }
    L--;            // Количество проверенных чисел
  }
  input.close();    // Закрывает файл
  output.close();   // Закрывает файл
  cout<<"Нажмите любую клавишу для выхода.";
  getch();          // Задерживает экран
  return 0;         // Завершает работу программы
}
```

Алгоритм продолжает работу до тех пор, пока все числа входного файла не будут проверены:

```
while (L!=0)
{
```

В начале данного цикла очередное число считывается из файла, а затем делается его копия, чтобы предотвратить потерю числа. Так как, согласно формату вывода, необходимо, чтобы сначала было записано само число `N`, а затем уже количество цифр в нем, то выведем это число в выходной файл:

```
input>>N;
K=N;
output<<N<<endl;
```

После этого устанавливаются начальные параметры для подготовки работы с новым числом:

```
c=0;
kol_vo=0;
```

Здесь переменная `c` — это очередная цифра, количество которой алгоритм ищет в записи текущего числа `N`. Очевидно, что `c` будет

изменяться от 0 до 9. В свою очередь переменная `kol_vo` — это счетчик, значение которого будет увеличиваться на единицу, если `i`-я цифра числа совпала со значением переменной `c`.

Далее запускаем цикл, который будет изменять значение переменной `c` от 0 до 9:

```
while (c!=10)
{
```

В начале цикла необходимо переменной `K` передать значение `N` для того, чтобы подсчитать в записи числа количество цифр `c`:

```
K=N;
```

Итак, на каждом шаге будем выделять последнюю цифру числа, затем удалять ее из числа и сравнивать с переменной `c`. Если две переменные окажутся равными, то счетчик `kol_vo` будет увеличен на единицу. Данные операции будут выполняться до тех пор, пока все цифры числа не будут просмотрены, т. е. пока число не станет равным нулю:

```
while (K!=0)
{
  temp=K%10;
  K/=10;
  if (temp==c) kol_vo++;
}
```

Таким образом, подсчет количества повторений очередной цифры в числе выполнен. Результат необходимо вывести в файл `output`:

```
output<<c<<" "<<kol_vo<<endl;
```

Далее алгоритм приступит к новой проверке. Для этого переменная `kol_vo` обнуляется, а значение переменной `c` увеличивается на единицу:

```
  kol_vo=0;
  c++;
}
```

Если в очередном числе уже подсчитано и выведено в файл количество повторений каждой цифры, то алгоритм вернется в начало, к первому циклу while, предварительно уменьшив значение L на единицу. Это будет означать, что проверено еще одно число:

```
  L--;
}
```

После того как все числа входного файла будут проверены, последует закрытие файлов, а затем программа завершит свою работу:

```
  input.close();
  output.close();
  cout<<"Нажмите любую клавишу для выхода.";
  getch();
  return 0;
}
```

## Пример 5

Программа, которая для каждой пары чисел проверяет, являются ли они взаимно простыми. Каждая пара чисел находится в отдельной строке файла input. Результат программа должна выводить в файл output.

**Примечание**

Два числа называются *взаимно простыми*, если их наибольший общий делитель равен единице.

Далее приведен код программы:

```
#include <iostream.h>
#include <conio.h>
#include <fstream.h>
int NOD(int, int);     // Прототип функции
// Прототип функции помещен без ключевого слова void потому,
// что функция void не может возвращать значение явно,
// а функция int может
main()
{
```

```
  int K, L, W, N, i;
  // Открывает файл с программным именем input для чтения
  ifstream input("input.txt");
  // Открывает файл с программным именем output для записи
  ofstream output("output.txt");
  input>>N;    // Читает из файла число, равное
  // количеству пар, которые содержатся в файле
  for (i=0; i<N; i++)
  {
    input>>K>>L;  // Читает из файла очередную пару чисел
    W=NOD(K,L);   // Запускает функцию, которая возвращает
                  // переменной W НОД(K,L)
    output<<K<<" "<<L;  // Выводит результат в файл
    if (W==1) output<<" Yes"; else output<<" No";
    output<<endl;    // Перемещает курсор на следующую строку
  }
  cout<<"Работа выполнена. Нажмите любую клавишу.";
  getch();          // Задерживает экран
  return 0;         // Завершает программу
}
int NOD(int a, int b)  // Описание функции
{
  if (a>b) a=a-b; else b=b-a;
  if (b==0) return a;
  NOD(a,b);       // Функция вызывает саму себя рекурсивно
                  // с новыми параметрами
 }
```

Алгоритм решения задачи следующий. На первом шаге из файла считывается число `N`, которое отвечает за количество пар, содержащихся в файле:

```
input>>N;
```

Далее, на каждом последующем шаге из файла считывается очередная пара чисел:

```
for (i=0; i<N; i++)
{
  input>>K>>L;
```

Затем при помощи функции `NOD(int,int)` находится наибольший общий делитель пары. С данной функцией вы уже знакомились ранее. Вот ее код:

```
int NOD(int a, int b)
{
  if (a>b) a=a-b; else b=b-a;
  if (b==0) return a;
  NOD(a,b);
}
```

Функция `NOD(int,int)` целого типа, и возвращает она значение такого же целого типа.

После того как НОД пары подсчитан, необходимо сравнить полученный результат с единицей и вывести соответствующий ответ.

```
  if (W==1) output<<" Yes"; else output<<" No";
  output<<endl;
}
```

# 6.4. Упражнения

1. Напишите программу, которая находит максимальное (минимальное) значение в файле.
2. Дан файл `input`, элементы которого располагаются столбцом. Напишите программу, которая отсортирует элементы файла по возрастанию и выведет отсортированный вариант в файл `output`.
3. Напишите программу, которая создаст новый файл `output`, содержащий все простые числа файла `input`.
4. Существует файл `input`, в котором расположены некоторые фамилии, по одной в строке. Напишите программу, выводящую в файл `output` каждую *i*-ую фамилию, все символы которой входят в некоторую *j*-ую фамилию, т. е. если файл `input` имеет вид:

   ```
   бхурцилова
   хурцилова
   ```

то в `output` будет выведена фамилия `хурцилова`, т. к. все символы данной фамилии присутствуют в первой.

5. В файле `input` непрерывно размещены некоторые цифры. Непрерывную совокупность цифр назовем числом. Напишите программу, которая возведет данное число в степень *N*. Количество цифр в числе не больше тысячи.
6. Напишите программу, которая определит и выведет на экран все совпадающие элементы файлов Num_1 и Num_2.
7. Напишите программу, которая определит и выведет на экран количество пар взаимно простых чисел. Число `A` берется из файла `input`, а число `B` — из `output`.
8. Напишите программу, которая определит сумму чисел, располагающихся на главной диагонали матрицы $N \times M$. Матрица находится в файле.
9. Преобладающим элементом в файле будем называть элемент, встречающийся в нем более $N/2$ раз. Напишите программу, которая определит и выведет на экран все существующие в файле преобладающие элементы.
10. Файл представляет собой последовательность из *N* натуральных чисел. Напишите программу, которая определит и допишет в существующий файл наименьшее натуральное число, отсутствующее в последовательности.

**Внимание**

При решении задач из упражнения использовать массивы, как одномерные, так и многомерные, запрещается.

Во всех задачах *главы 6* входные данные считать корректными.

# Глава 7

# Техника указателей

## 7.1. Понятие указателя

*Указатель* — это адрес памяти, распределяемой для размещения идентификатора (в качестве идентификатора может выступать имя переменной, массива, структуры, строкового литерала). Если переменная объявлена как указатель, то она содержит адрес памяти, по которому может находиться скалярная величина любого типа. При объявлении переменной типа указатель необходимо определить тип объекта данных, адрес которых будет содержать переменная, и имя указателя с предшествующей звездочкой (или группой звездочек). Формат объявления указателя следующий:

*Спецификатор_типа* [*модификатор*] **описатель*

Спецификатор типа задает тип объекта и может быть любого основного типа. Задавая вместо спецификатора типа ключевое слово `void`, можно отсрочить спецификацию типа, на который ссылается указатель. Переменная, объявляемая как указатель на тип `void`, может быть использована для ссылки на объект любого типа. Однако для того чтобы можно было выполнить арифметические и логические операции над указателями или над объектами, на которые они указывают, необходимо при выполнении каждой операции явно определить тип объектов. Такое определение типов может быть выполнено с помощью операции приведения типов.

Конкретное объявление указателя выглядит следующим образом:

```
int *iP i;
```

Данная запись означает, что `i` — переменная типа `int`, а `iP` — указатель. Символ `*` означает, что объявляется указатель, который относится только к следующему за ним идентификатору.

Так как объявлен указатель, то у него есть адрес. Адреса распределяет компьютер. *Операция адресации*, т. е. присваивание объявленному указателю адреса переменной, производится следующим образом:

```
int *iP, i;
i=1;
iP=&i;
```

Унарная операция `&` означает, что переменной возвращается ее адрес. Для того чтобы получить адрес переменной, необходимо провести операцию разыменования. Знак `*` перед указателем возвращает значение переменной, на которую указатель ссылается.

Указатель может указывать на константные данные и сам иметь атрибут `const`. Например:

- `const float *cP;` — объявление указателя на константный символ; сам указатель не является константой;
- `float *const cP;` — объявление константного указателя на некоторое число;
- `const float *const cP;` — объявление константного указателя на константное число.

## 7.2. Указатели на массивы

В языке программирования C++ между указателями и массивами существует тесная связь. Например, когда объявляется массив в виде `int array[10]`, то это означает не только выделение памяти для десяти элементов массива, но и для указателя с именем `array`, значение которого равно адресу первого по счету (нулевого) элемента массива, т. е. сам массив остается безымянным, а доступ к элементам массива осуществляется через указатель с именем `array`. С точки зрения синтаксиса языка указатель `array` является константой, значение которой можно использовать в выражениях, но изменить это значение нельзя.

*Указателем на массив* является имя данного массива, а также указатель на его первый элемент. Например:

```
int array[10], *aP;
aP=array;      // Или aP=&array[0]
```

Изменение указателя переводит на другой элемент массива.

```
aP++;          // Указатель переведен на следующий элемент
массива
--aP;          // Указатель переведен на предыдущий элемент
массива
```

Для доступа к элементам массива существуют два различных способа. Первый способ связан с использованием обычных индексных выражений в квадратных скобках, например, `array[3]=1` или `array[i+2]=7`. При таком способе доступа записываются два выражения (`array` и `i+2`), причем второе заключается в квадратные скобки. Одно из них должно быть указателем, а второе — выражением целого типа. Последовательность записи этих выражений может быть любой, но в квадратных скобках записывается выражение, следующее вторым. Поэтому записи `array[3]` и `3[array]` эквивалентны и обозначают элемент массива `array` под номером три. Указатель, используемый в индексном выражении, не обязательно должен быть константой, указывающей на какой-либо массив, это может быть и переменная. В частности, после выполнения присваивания `aP=array` доступ к десятому элементу массива можно получить с помощью указателя `aP` в форме `aP[10]` или `10[aP]`.

Второй способ доступа к элементам массива связан с использованием адресных выражений и операции разадресации в форме `*(array+10)=3` или `*(array+i+2)=7`. При таком способе доступа адресное выражение, равное адресу десятого элемента массива, тоже может быть записано разными способами: `*(array+10)` или `*(10+array)`.

При реализации на компьютере первый способ приводится ко второму, т. е. индексное выражение преобразуется к адресному. Для приведенных примеров `array[10]` и `10[array]` преобразуются в `*(array+10)`.

Для доступа к начальному элементу массива (к элементу с нулевым индексом) можно использовать просто значение указателя `array` или `aP`. Любая из операций

```
*array=2;
array[0]=2;
*(array+0)=2;
*aP=2;
aP[0]=2;
*(aP+0)=2;
```

присваивает начальному элементу массива значение 2, но быстрее всего выполнятся операции `*array=2` и `*aP=2`, т. к. в них отсутствует сложение.

**Совет**

Указатели на один и тот же тип могут быть собраны в так называемый *массив указателей*.

## Операции над указателями

Над указателями можно выполнять унарные операции — инкремент и декремент. При выполнении операций `++` и `--` значение указателя увеличивается или уменьшается на длину типа, на который ссылается используемый указатель. Например:

```
int *aP, array[10];
aP=&a[5];
aP++;  // Становится равным адресу элемента array[6]
--aP;  // Становится равным адресу элемента array[5]
```

В бинарных операциях сложения и вычитания могут участвовать указатель и величина типа `int`. При этом результатом операции будет указатель на исходный тип, а его значение будет на заданное число элементов больше или меньше исходного. Например:

```
int *aP1, *aP2, array[10];
int i=2;
aP1=array+(i+4); // Становится равным адресу элемента array[6]
aP2=aP1-i;       // Становится равным адресу элемента array[4]
```

В операции вычитания могут участвовать два указателя на один и тот же тип. Результат такой операции имеет тип `int` и равен числу элементов исходного типа между уменьшаемым и вычитаемым, причем если первый адрес младше, то результат имеет отрицательное значение. Например:

```
int *aP1, *aP2, array[10];
int i;
aP1=array+4;
aP2=array+9;
i=aP1-aP2;   // Возвращено значение 5
i=aP2-aP1;   // Возвращено значение -5
```

Значения двух указателей на одинаковые типы можно сравнивать в операциях ==, !=, <, <=, >, >=, при этом значения указателей рассматриваются просто как целые числа, а результат сравнения равен 0 (ложь) или 1 (истина). Например:

```
int *aP1, *aP2, array[10];
aP1=array+5;
aP2=array+7;
if (aP1>aP2) array[3]=4;
```

В данном примере значение `aP1` меньше значения `aP2`, и поэтому оператор `array[3]=4` не будет выполнен.

## Доступ к значениям, адресуемым указателями

Используя адрес, содержащийся в указателе, можно определить значение, содержащееся по этому адресу. Для этого предназначена операция `*aP`. Например, нижеприведенным оператором выводится значение, содержащееся по указателю `aP`:

```
printf("Значение, содержащееся по указателю aP, ", *aP);
```

В следующем операторе переменной `count` присваивается значение, содержащееся по адресу указателя `aP`:

```
count=*aP;
```

Аналогично в память по указателю `aP` заносится значение 7:

```
*aP=7;
```

# 7.3. Указатели на строку

При обработке строк указатели используются достаточно часто. *Строка* — это массив символов, оканчивающийся NULL-символом. В нижеприведенной программе определяется функция show_string() для вывода символов строки с помощью указателя на строку.

```
#include <stdio.h>
#include <conio.h>
void show_string(char *String)  // Описание функции
{
  while (*String)               // Пока строка не пуста
    putchar(*String++);  // Выбирает первый символ из строки
                         // и выводит на экран
}
main()
{
  show_string("Exaple");  // Передает в функцию некоторую строку
  getch();                // Задерживает экран
  return 0;               // Завершает программу
}
```

Переменная String объявлена в функции как указатель. Указатель увеличивается в простом цикле до тех пор, пока не встретится конец строки. Перед тем как вывести очередной символ, он определяется в функции посредством операции *. Затем указатель увеличивается, тем самым ссылаясь на следующий символ.

# 7.4. Указатели на функцию

В языке C++ сами функции не являются переменными, но имеется возможность определить указатель на функцию, который можно обрабатывать, передавать другим функциям, помещать в массивы и т. д.

Наиболее общее использование указателей на функции — возможность передать адрес функции в качестве фактического па-

раметра другой функции. Указатели на функции объявляются следующим образом:

```
int (*F1)();
long (*F2)();
float (*F3)();
```


Обратите внимание на скобки, охватывающие имена переменных. Без них приведенные объявления — это прототипы функций, которые возвращают указатель на указанный тип.

## Функции, возвращающие указатель

По мере усложнения программы приходится создавать функции, которые возвращают указатели на определенные типы. Например, в следующей программе создается функция `point_return()`, которая преобразует все строчные латинские буквы в прописные и возвращает указатель на строку:

```
#include <ctype.h>
#include <conio.h>
#include <iostream.h>
char *point_return(char *String)
{
  int Length, i;
  Length=strlen(String); // Возвращает длину переданной строки
  for (i=0; i<Length; i++)
    String[i]=toupper(String[i]);
  return(String);          // Возвращает строку
}
main()
{
  char *Line="Example";
  char *String;
  String=point_return(Line);
  cout<<String;  // Выводит результат
```

```
  getch();        // Задерживает экран
  return 0;       // Завершает программу
}
```

Для преобразования всех строчных латинских букв в прописные вызывается встроенная функция `toupper(char)`. Для использования функции необходимо подключить к программе заголовочный файл ctype.h.

**Примечание**

Если функция возвращает указатель, то перед ее именем необходимо поставить символ `*`.

# 7.5. Динамическое распределение памяти

Ранее мы рассматривали область действия и время жизни переменных. В C++ предусмотрена возможность динамически выделять и удалять память для переменных. Эти действия становятся доступными при помощи специальных операторов `new` и `delete` в сочетании с указателями.

## Динамическое выделение памяти

Выражение, содержащее операцию `new`, имеет следующий вид:

*указатель_на_тип*=new *имя_типа* (*инициализатор*)

*Инициализатор* — это необязательное выражение, которое задает начальное значение переменной (может использоваться для всех типов, кроме массивов).

При выполнении оператора

```
int *N=new int;
```

создаются 2 объекта:

- динамический безымянный объект;
- указатель на этот объект с именем `N`, значением которого является адрес динамического объекта.

Можно создать и другой указатель на тот же динамический объект:

```
int *other_point=N;
```

Если указателю `N` присвоить другое значение, то можно потерять доступ к динамическому объекту:

```
int *N=new (int);
int i=0;
N=&i;
```

В результате динамический объект по-прежнему будет существовать, но обратиться к нему будет уже нельзя.

При выделении памяти объект можно инициализировать:

```
int *N=new int(3);
```

Динамически распределять память можно не только под обычные переменные, но и под массивы:

```
int *array=new int[50];
```

Теперь с этой динамически выделенной памятью можно работать как с обычным массивом:

```
*(array+1) = 10;
array[6]=array[5]+11;
```

В случае успешного завершения операция `new` возвращает указатель со значением, отличным от нуля.

Результат операции, равный 0, т. е. нулевому указателю `NULL`, говорит о том, что не найден непрерывный свободный фрагмент памяти нужного размера.

Теперь рассмотрим, как выделить динамическую память под двумерный массив.

Для начала опишем его следующим образом:

```
main()
{
  int **array, i, j, N, M;
```

Далее создаем матрицу:

```
array=new int *[N];  // Выделяет память для одномерного массива
```

```
for (int i=0; i<N; i++)
  array[i]=new int[M]; // При помощи цикла повторений
                       // выделяет память для двумерного массива
```

Затем необходимо задать начальное значение элементам двумерного массива при помощи цикла (по умолчанию они не будут равны нулю, т. к. двумерный массив определен не глобально).

```
for (int i=0; i<N; i++)
  for (int j=0; i<M; i++)
    array[i][j]=0;
```

Теперь все готово для ввода элементов созданного двумерного массива.

## Освобождение динамически выделенной памяти

Операция `delete` освобождает для дальнейшего использования в программе участок памяти, ранее выделенный операцией `new`:

```
delete N;       // Удаляет динамический объект типа int,
                // если было N=new int
delete array;   // Удаляет динамический массив длиной 50,
                // если было int *mas = new int[50]
```

Совершенно безопасно применять операцию к указателю `NULL`. Результат же повторного применения операции `delete` к одному и тому же указателю не определен. Обычно происходит ошибка, приводящая к зацикливанию.

Чтобы избежать подобных ошибок, можно применять следующую конструкцию:

```
int *N=new int[10];
if (N)
{
  delete N;
  N=NULL;
} else cout<<"Память уже освобождена"<<endl;
```

# 7.6. Примеры использования указателей

## Пример 1

Приведем программу, которая подсчитывает частоту повторений каждого символа в строке.

```
#include <iostream.h>
#include <conio.h>
int array[2][256];
void Test(char *String)   // Описание функции
{
  while (*String) // Пока указатель не на последнем символе строки
  {
    if (array[0][*String]==0)
      array[0][*String]=int(*String);
    array[1][*String]++;  // Увеличивает счетчик повторений
                          // символа на единицу
    *String++;            // Перемещает указатель
  }
}
main()
{
  char *String;
  int i;
  Test("abcbca^^^&&&");  // Вызывает функцию. В качестве
                         // параметра используется любая строка
 // Выводит результат
 for (i=0; i<=256; i++)
   if (array[0][i]!=0)
     cout<<char(array[0][i])<<" "<<array[1][i]<<endl;
 getch();   // Задерживает экран
 return 0;  // Завершает программу
}
```

В строке может содержаться до 256 неповторяющихся символов, каждый из которых имеет свой код ASCII. Чтобы решить данную задачу, будем использовать двумерный массив из двух строк и 256 столбцов.

На каждом этапе будем выделять очередной символ строки, находить его ASCII-код и в массиве элемент `array[1][ASCII(Symbol)]` увеличивать на единицу. А в ячейку `array[0][ASCII(Symbol)]` будем записывать сам ASCII-код. Таким образом, мы получим массив, в первой строке которого будут находиться встречаемые в строке символы, а во второй — их частота. Заполнять массив таким образом будет функция `void Test(char *String)`:

```
void Test(char *String)
{
  while (*String)
  {
    if (array[0][*String]==0)
      array[0][*String]=int(*String);
    array[1][*String]++;
    *String++;
  }
}
```

**Замечание**

Запись `A=int(B)` означает, что переменной `A` будет возвращен ASCII-код символа `B`.

После этого функция обрабатывает некоторую строку:

```
Test("abcbca^^^&&&");
```

Выводить результат будем следующим образом. Если элемент массива `array[0][i]` не равен нулю, то выводим символ, ASCII-код которого равен значению данного элемента. Затем выводим его частоту.

```
  for (i=0; i<=256; i++)
    if (array[0][i]!=0)
```

```
      cout<<char(array[0][i])<<" "<<array[1][i]<<endl;
  getch();   // Задерживает экран
  return 0;  // Завершает программу
}
```

## Пример 2

Рассмотрим программу, которая вычисляет и выводит на экран простое число Мерсенна, содержащее не менее 100 цифр.

Необходимо напомнить, что число называется *простым*, если оно делится без остатка только на само себя и на единицу.

Числа Мерсенна определяются следующим образом:

$$M = 2^{Power} - 1$$

где *Power* — простое число. Любое число, полученное таким способом, будет являться простым. Таким образом, задача сводится к возведению 2 в степень *Power*.

По условию задачи необходимо вывести простое число, содержащее не менее 100 цифр. Чтобы найти количество цифр числа $2^{Power}$, вычислим $\lg 2^{Power} = Power \times \lg 2 = p \times 0.30103$. Итак, подставим исходное значение *Power* и получим:

$$100 = Power \times 0.30103 \Leftrightarrow Power = 100/0.30103 \Leftrightarrow Power = 332.$$

Из этого следует, что 2 необходимо возводить в степень не меньше, чем 332.

Далее приведен код программы:

```
#include <iostream.h>
#include <conio.h>
#include <math.h>
void Done(int);       // Прототип функции
int a[100000], *aP, c[100000], *cP, Len=1;
main()
{
  int Simple, i;
  aP=a;  // Возвращает указатель на нулевой элемент
         // массива, в который на каждом шаге будет
```

```
        // передаваться текущее значение степени двойки
  Simple=pow(2,11)-1;  // Возвращает простое число 2047
  *aP=1;
  Done(Simple);
  --*aP;     // Вычитает единицу из нулевого элемента массива
  aP+=Len-1; // Переводит указатель на последний элемент массива
  // Выводит результат
  for (i=Len-1; i>=0; i--)
    cout<<*aP--;
  getch();
  return 0;
}
void Done(int Power)  // Описание функции
{
  int i;
  cP=c;  // Возвращает указатель на нулевой элемент массива,
         // в котором будет храниться результат возведения
         // в степень
  // Перемножает массив и const
  for (i=0; i<Len; i++)
  {
    *cP=*cP+*aP*2;
    if (*cP>9)
    {
      *(cP+1)+=(*cP/10);
      *cP%=10;
    }
    // Выполняет последовательное смещение указателей
    // на один элемент
    aP++;
    cP++;
  }
  // Определяет длину получившегося числа. Под длиной
  // понимается количество цифр, содержащихся в числе
  while (c[Len]==0 && Len>0) Len--;
  Len++;
```

```
  // Возвращает указатели на нулевой элемент
  // соответствующих им массивов
  aP=a;
  cP=c;
  for (i=0; i<=Len; i++)
  {
    *aP++=c[i];  // При помощи цикла передает
                 // значения элементов массива c в массив a
    *cP++=0;
  }
  aP=a;    // Возвращает указатель на нулевой элемент массива
  --Power; // Уменьшает степень на единицу
  // Функция завершит свою работу, если 2 уже возведено
  // в нужную степень, в противном случае функция будет
  // вызвана рекурсивно
  if (Power==0) return; else Done(Power);
}
```

Как говорилось ранее, двойку необходимо возвести в некоторую степень `Power` и вычесть из результата единицу, причем `Power` — простое число. Получим такое простое число при помощи встроенной функции `pow(Number, Power)`. Где `Number` — это число, которое необходимо возвести в степень `Power`. Итак, передадим переменной `Simple` некоторое простое число:

```
Simple=pow(2,11)-1;
```

Взято число 11, потому что это наименьшее простое число, позволяющее получить число Мерсенна, содержащее более ста символов.

**Примечание**

Для того чтобы использовать функцию `pow()`, необходимо подключить к программе заголовочный файл math.h.

Теперь необходимо возвести 2 в степень `Simple`. Для этого будем использовать алгоритм умножения арифметики больших чисел, который был разобран ранее. Только теперь будем перемножать не два массива, а массив и число 2. Поэтому ранее разобранный

алгоритм необходимо модифицировать. Функция `Done(int)` умножает столбиком массив на число 2:

```
void Done(int Power)  // Описание функции
{
  int i;
  cP=c;
  for (i=0; i<Len; i++)
  {
    *cP=*cP+*aP*2;
    if (*cP>9)
    {
      *(cP+1)+=(*cP/10);
      *cP%=10;
    }
    aP++;
    cP++;
  }
}
```

Затем, когда результат умножения известен, необходимо определить длину полученного числа:

```
while (c[Len]==0 && Len>0) Len--;
Len++;
```

После этого результат, содержащийся в массиве `c`, переписывается в массив `a`, причем в конце нужно получить пустой массив `c`. Это необходимо, т. к. далее мы будем работать с ним в функции `Done(int)`, к его элементам будут добавляться некоторые значения, и, следовательно, результат может оказаться неправильным.

```
aP=a;
cP=c;
for (i=0; i<=Len; i++)
{
  *aP++=c[i];
  *cP++=0;
}
```

Так как мы возводим 2 в степень `Power`, то на первом шаге 2 уже возведено в степень 1. Поэтому после каждого вызова функции необходимо уменьшать значение переменной `Power` на единицу и возвращать указатель `aP` на нулевой элемент массива:

```
--Power;
aP=a;
```

Рекурсивный вызов будет продолжаться до тех пор, пока `Power` ≠ 0:

```
  if (Power==0) return; else Done(Power);
}
```

После того как функция завершит свою работу, управление будет передано функции `main()`, которая вычтет из числа, полученного последним, единицу, установит указатель `aP` на конец массива и выведет результат:

```
  *aP--;
  aP+=Len-1;
  for (i=Len-1; i>=0; i--)
    cout<<*aP--;
  getch();
  return 0;
}
```

## Пример 3

Напишите программу, которая проверит, является ли число $N$ простым, если $3 \le N \le 200\,000\,000$.

Очевидно, что если мы будем решать данную задачу способом, рассмотренным ранее, т. е. проверять, делится ли $N$ хоть на одно число из промежутка $[2; N/2+1]$, то на проверку уйдет много времени. Поэтому будем использовать более эффективный алгоритм:

1. Сгенерируем массив `Simple` всех простых чисел из промежутка [2; `sqrt(N)` + 1].
2. Если `N` делится без остатка хотя бы на одно число из массива `Simple`, то `N` — не простое.

Вот код данной программы с комментариями:

```
#include <iostream.h>
#include <conio.h>
#include <math.h>
long N, i, j, Simple[1000000], *sP, *tempP, p, pos;
main()
{
  cin>>N;      // Требует ввода числа, для которого необходимо
               // выполнить проверку
  sP=Simple;  // Возвращает указатель на нулевой элемент массива
  // Генерирует простые числа, которые находятся в промежутке
  // от 2 до sqrt(N)+1
  for (i=2; i<sqrt(N)+1; i++)
  {
    for(j=2; j<i/2+1; j++)
      if (i%j==0)
      {
        p=1;
        break;
      }
    // Если выбранное число простое, то заносит его в массив
    if (p==0)
    {
      *sP=i;  // В элемент массива по адресу sP
              // записывает число i
      sP++;   // Перемещает указатель на следующий элемент массива
    } else p=0;
  }
  tempP=sP--;  // Запоминает позицию указателя sP. В данный момент
               // он находится на последнем элементе массива
  sP=Simple;   // Перемещает указатель на нулевой элемент массива
  // Проверяет, делится ли без остатка хотя бы одно число из массива
  // Simple на N.
  // Запускает цикл, который работает до тех пор, пока указатели sP
```

```
  // и tempP не совпадут
  while (sP!=tempP)
    if (N%*sP==0)
    {
      p=1;
      break;
    }
    else sP++; // Перемещает указатель на следующий элемент массива
  // В соответствии со значением переменной p выводит результат
  if (p==1) cout<<"Число не является простым.";
  else cout<<"Число является простым.";
  getch();      // Задерживает экран
  return 0;     // Завершает программу
 }
```

## Пример 4

Приведем программу, которая сортирует массив выборочной сортировкой при помощи указателей.

```
#include <iostream.h>
#include <conio.h>
main()
{
  int array[100], *aP, *startP, *minP, *point, size, i;
  int min temp;
  cin>>size;  // Требует ввода количества элементов массива
  aP=array;   // Устанавливает указатель на нулевой
              // элемент массива
  for (i=0; i<size; i++)
    cin>>*aP++;  // Требует ввода элементов массива
  startP=array;  // Устанавливает указатель
                 // на нулевой элемент массива
                 // Левая начальная граница подмассива
  // Запускаем цикл, который выполняет операции,
  // пока все подмассивы не пройдены, т. е. пока указатель
  // startP и aP не совпадут
```

```
while (startP!=aP)
{
  // Предположим, что элемент с индексом левой границы
  // подмассива минимален
  min=*startP;
  minP=startP;   // Запоминает адрес элемента с минимальным
                 // значением
  point=startP;  // Делает копию указателя startP
  // Запускаем цикл поиска минимального элемента в подмассиве
  while (point!=aP)
  {
    if (*point<min)  // Сравнивает элемент массива,
                     // находящийся по адресу указателя
                     // point, со значением переменной min
    {
      min=*point;   // Запоминает минимальное значение
      minP=point;   // Запоминает адрес элемента с
                    // минимальным значением
    }
    point++; // Переводит указатель на следующий элемент массива
  }
  point--; // Возвращает указатель на предыдущий элемент массива
  // Если адрес минимального значения элемента массива
  // не совпадает с предположенным ранее, то меняет местами
  // элементы с данными адресами
  if (minP!=startP)
  {
    temp=*startP;
    *startP=*minP;
    *minP=temp;
  }
  startP++; // Переводит указатель на следующий элемент массива
}
aP=array; // Устанавливает указатель на нулевой элемент массива
// Выводит результат
```

```
  for (i=0; i<size; i++)
    cout<<*aP++<<" ";
  getch();      // Задерживает экран
  return 0;     // Завершает программу
}
```

## Пример 5

Дана строка *S*. Длина *S* не превышает 2 000 000 символов. Найти и вывести на экран символ, число вхождений которого в *S* максимально, т. е. символ, который повторяется в исходной строке наибольшее количество раз.

Вот первый способ решения задачи:

```
#include <iostream.h>
#include <conio.h>
main()
{
  char *S, sign;
  int max, i, j, repeat;
  S=new char [2000000];  // Динамически выделяет память для
                         // символьного массива S
  cin>>S;  // Требует ввода строки
  max=0;   // Начальное максимальное вхождение
  // Два цикла, которые перебирают все пары
  // символов. Счетчик repeat увеличивается на единицу,
  // если символы в позициях i и j совпали
  for (i=0; i<strlen(S)-1; i++)
  {
    for (j=i+1; j<strlen(S); j++)
      if (S[i]==S[j]) repeat++;
    if (repeat>max)
    {
      max=repeat;  // Запоминает максимальное число вхождений
                   // на данном шаге
      sign=S[i];   // Запоминает символ, которому соответствует
```

```
                    // максимальное число вхождений
                    // на данном шаге
    }
  }
  delete S;     // Освобождает для дальнейшего использования в
                // программе участок памяти, ранее выделенный
                // операцией new
  cout<<sign;  // Выводит результат на экран
  getch();      // Задерживает экран
  return 0;     // Завершает программу
}
```

Данный метод решения не является эффективным, т. к. он последовательно перебирает все пары символов. Сложность такого решения $O(n^2)$, т. е. чтобы получить результат, необходимо выполнить $n^2$ операций, где $n$ — количество символов в строке.

Рассмотрим другое решение данной задачи, которое будет заключаться в заполнении массива из 256 элементов. Известно, что каждый символ имеет свой ASCII-код и всего таких кодов 256. На основе данных утверждений предложим решение, которое справляется с задачей за $n$ операций. Алгоритм такого решения следующий:

1. Выделяется очередной символ строки и в массиве `Code` к элементу с индексом ASCII-кода данного символа прибавляется единица.
2. Находится максимальное значение в массиве `Code`. Позиция этого числа и будет являться ASCII-кодом символа с максимальной частотой повторений.

Ниже приведен код данного алгоритма:

```
#include <iostream.h>
#include <conio.h>
int Code[256];
main()
{
  char *S;
```

```
  int max, max_pos, i;
  S=new char [2000000];  // Динамически выделяет память для
                         // символьного массива S
  cin>>S;                // Требует ввода строки
  // Запускает цикл, который определяет частоту
  // вхождений каждого символа в строку
  for (i=0; i<strlen(S); i++)
    Code[int(S[i])]++;
  delete S;   // Освобождает для дальнейшего использования
              // в программе участок памяти, ранее выделенный
              // операцией new
  max=Code[0];  // Пусть элемент массива
                // под индексом 0 максимальный
  max_pos=0;    // Запоминает начальную позицию
                // максимального элемента
  for (i=0; i<256; i++)
    // Если значение некоторого элемента массива больше, чем
    // значение переменной max, то
    if (Code[i]>max)
    {
      max=Code[i];  // Запоминает максимальное значение и
      max_pos=i;    // позицию максимального значения
    }
  cout<<char(max_pos);  // Выводит результат
  getch();              // Задерживает экран
  return 0;             // Завершает программу
}
```

Такой способ решения задачи более оригинальный и быстрый. Сложность решения равна O(2*n*).

## Пример 6

Рассмотрим программу расшифровки сообщения, закодированного по описанному ниже принципу. Пусть дан шифр (набор цифр 43512) и текст — `examplemode`. Текст состоит из строчных латинских букв и не содержит пробелов. Записываем под ним

цифры шифра. Каждая буква алфавита заменяется буквой, номер которой равен номеру исходной буквы в алфавите плюс цифра, стоящая под ней. Буква не заменяется и выписывается в исходном состоянии, если при суммировании ее текущей позиции и цифры кода получается результат, превосходящий количество букв в алфавите.

Таким образом, если на вход поступит строка `examplemod` и код 43512, то:

- исходная строка: `examplemode`;
- исходный код: `43512435124`;
- зашифрованный текст: `ixfnrphrpfi`.

Для решения данной задачи воспользуемся ASCII-кодами букв. Латинский алфавит задается ASCII-кодами, которые принимают значения от 97 (буква a) до 122 (буква z).

Далее приведен код программы с комментариями:

```
#include <iostream.h>
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
main()
{
  char S[1000], *sP, Code[1000], *C;
  int i, Temp;
  cin>>S;     // Требует ввода строки,
              // которую необходимо закодировать
  cin>>Code; // Требует ввода строки кода
  // Объединяет значение переменной Code с самим собой до тех
  // пор, пока длина кода не станет больше длины строки
  while (strlen(Code)<strlen(S))
    strcat(Code,Code);  // Функция, объединяющая две строковых
                        // переменных в одну
  sP=S;  // Устанавливает указатель на нулевой элемент массива
  C="";  // Определяет строку как пустую
```

```
  for (i=0; i<strlen(S); i++)
  {
    *C=Code[i];              // Выделяет очередную цифру кода
    Temp=(*sP++)+atoi(C);    // Выполняет условие кодировки
    // Если получившееся ASCII-значение выходит за
    // установленные рамки, то выводит символ без изменений
    if (Temp>122) cout<<char(Temp-atoi(C));
    else cout<<char(Temp);  // Выводит символ, получившийся
                            // путем сдвига ASCII-значения
  }
  getch();   // Задерживает экран
  return 0;  // Завершает программу
}
```

# 7.7. Упражнения

1. Напишите программу, которая отсортирует массив по возрастанию количества единиц в двоичной записи чисел.
2. Задан список некоторых фамилий. Две фамилии назовем похожими, если одна из них может быть получена из другой путем:
   - перестановки каких-либо двух соседних букв;
   - замены одной какой-либо буквы некоторой другой;
   - добавления какой-либо буквы в начало или конец фамилии;
   - удаления какой-либо одной буквы.
3. На экспертизу поступает некоторая фамилия. Напишите программу, которая выберет и выведет на экран похожие фамилии из заданного списка фамилий.
4. Назовем число $N$ наивно простым, если $N$:
   - простое число;
   - любое число, полученное при перестановке цифр простого числа.

   Напишите программу, которая определит и выведет на экран все такие числа из отрезка $[a; b]$.

5. Напишите программу построения последовательности, состоящей из $N$ простых чисел. Пусть $N = 3$, тогда результат будет следующим:

```
3
5
7
```

6. Составьте алгоритм и напишите программу, которая будет организовывать поиск и вывод на экран всех двузначных чисел, сумма цифр которых не изменится при умножении числа на 2, 3, 4, 5, 6, 7, 8, 9.

7. Дана некоторая строка, представляющая собой арифметическое выражение. Например, `String="12+4*56+(12*2+5)*13"`. Напишите программу, которая выведет на экран постфиксную запись такого выражения.

   Постфиксная форма записи выражения — это последовательность выполнения арифметических действий. Чтобы перемножить два числа, необходимо знать эти числа, в данном случае — 4 и 56. Затем нужно сложить получившийся результат с 12. Последовательность действий будет продолжаться до тех пор, пока вся строка не будет "превращена" в единственный результат.

   Для приведенного выражения ответ таков:

```
12 4 56 * + 12 2 * 5 + 13 * +
```

8. Напишите программу, которая вычислит значение выражения, записанного в постфиксной форме.

9. Составьте алгоритм и напишите программу, которая заполнит матрицу $M \times N$ ($M > N$) вложенными рамками, каждая из которых изображается числом, равным разности между $N$ и номером рамки. Причем внешнюю рамку считать первой. Например, для $N = 5$ и $M = 8$ результат будет следующим:

```
44444444
43333334
43222234
43333334
44444444
```

10. Разработайте алгоритм и напишите программу, которая будет осуществлять перевод римских чисел в арабские.
11. Даны две непустые строки символов `X` и `Y`. Строчными буквами обозначаются переменные. Переменные могут быть только в строке `X`. Значением переменной может быть непустая строка символов без строчных букв. Напишите программу, которая определит значения переменных так, чтобы после подстановки их в строку `X` она совпала со строкой `Y`. Например, если исходные данные такие:

```
X="tGqtt"
Y="RIGPORYRIRI"
```

то результат:

```
t="RI", q="PORY"
```

**Примечание**

Все задачи из упражнения необходимо решать с использованием указателей.

Во всех задачах *главы 7* входные данные считать корректными.

# Глава 8

# Объектно-ориентированное программирование

В этой главе рассматриваются способы реализации основных механизмов объектно-ориентированного программирования (ООП) в C++.

*Объектно-ориентированное программирование* — это методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию наследования.

*Объект* — это некоторая уникальная единица, имеющая свои данные и функции, эти данные обрабатывающие.

Чтобы создать объект, необходимо определить некий тип данных (в терминах ООП — *класс*), который будет использоваться в программе. Каждая переменная этого типа — экземпляр класса — представляет собой объект. Любой тип данных можно представить как объект.

По сравнению с традиционными способами программирования ООП обладает рядом преимуществ. Главное из них заключается в том, что эта концепция в наибольшей степени соответствует внутренней логике функционирования операционной системы Windows. Программа, состоящая из отдельных объектов, отлично приспособлена к реагированию на события, происходящие в ОС. К другим преимуществам ООП можно отнести большую надежность кода и возможность повторного использования отработанных объектов.

# 8.1. Структуры

До сих пор мы пользовались базовыми типами данных: `float`, `int`, `char` и т. д., а также познакомились с понятием массива. Стоит напомнить, что *массив* — это последовательная группа переменных, имеющих *одно имя* и *один тип*.

В отличие от массива структуры позволяют объединять данные различных типов.

*Структура* — производный тип данных, определенный программистом. Имя структуры — спецификатор типа.

Рассмотрим конструкцию структуры.

```
struct client {int number; float D;} c1,c2;
```

В данной записи можно выделить несколько определений:

- `struct` — это ключевое слово;
- `client` — имя структурного типа;
- `number`, `D` — элементы структуры;
- `c1`, `c2` — объекты структурного типа `client`.

Перечень элементов структуры заключается в фигурные скобки. Объявление структуры, как и любой другой переменной, заканчивается знаком `;`.

Объект можно объявить отдельно от структуры. Например, следующим образом:

```
client G;  // Будет объявлен объект типа client
```

Доступ к отдельным элементам структуры осуществляется при помощи знака `.` (точка) для объекта и символа `->` для указателя на объект.

Рассмотрим использование структур на примере программы, которая выводит на экран все неповторяющиеся разложения числа `array[i]` на сумму трех чисел, а также их количество. При этом для чисел `i`, `j`, `k`, при помощи которых будет составлена сумма, выполняется неравенство: `i>j>k`.

```
#include <iostream.h>
#include <conio.h>
```

```
struct client {int Number;};
void input(client &);  // Прототип функции,
                       // которая считывает данные
void done(client &);   // Прототип функции, которая находит
                       // все разложения числа
int N;
main()
{
  int i;
  cout<<"N=";
  cin>>N;              // Требует ввести количество чисел
  // Выделяет память под массив array из N элементов динамически
  client *array;
  array=new client[N];
  for (i=0; i<N; i++)
    input(array[i]);   // Запускает функцию, которая вводит
                       // очередной элемент массива
  for (i=0; i<N; i++)
    done(array[i]);    // Запускает функцию, которая находит
                       // все разложения числа
  getch();             // Задерживает экран
  return 0;            // Завершает программу
}
void input(client &c)
{
  cout<<"Введите следующее число"<<endl;
  cin>>c.Number;  // Требует ввода с клавиатуры следующего
                  // числа, которое будет занесено в массив
}
void done(client &c)
{
  int i, j, k, p;
  p=0;
  for (i=1; i<c.Number/3; i++)
    for (j=i+1; j<c.Number/2; j++)
```

```
      for (k=j+1; k<c.Number; k++)
        if (i+j+k==c.Number)
        {
          // Если разложение найдено, то происходит
          // его вывод на экран
          cout<<c.Number<<"="<<i<<"+"<<j<<"+"<<k<<endl;
          p++;
        }
  if (p==0) cout<<"Разложений не найдено"<<endl;
  else cout<<"Найдено разложений: "<<p;
}
```

Так как в программе используются функции, то необходимо записать их прототип:

```
void input(client &);
void done(client &);
```

Затем опишем структуру типа `client` с одним элементом `Number`:

```
struct client {int Number;};
```

На первом шаге работы программы необходимо ввести число `N`, задающее количество элементов в массиве, т. е. количество чисел, все разложения которых надо найти:

```
cout<<"N=";
cin>>N;
```

После этого необходимо организовать ввод каждого элемента массива. В данном случае эту операцию реализует функция `input(client &)`:

```
void input(client &c)
{
  cout<<"Введите следующее число"<<endl;
  cin>>c.Number;
}
```

После того как все элементы массива введены, можно выполнять разложение каждого из них. В программе разложения выполняются при помощи функции `done(client &)`. В данную функцию,

как и в функцию `input(client &)`, поступает один параметр типа `client` — число.

Чтобы найти такие `i`, `j`, `k`, которые в сумме равны `c.Number`, необходимо перебрать все возможные тройки данных чисел при условии, что `c.Number>i>j>k`:

```
void done(client &c)
{
  int i, j, k, p;
  p=0;
  for (i=1; i<c.Number/3; i++)
    for (j=i+1; j<c.Number/2; j++)
      for (k=j+1; k<c.Number; k++)
        if (i+j+k==c.Number)
        {
          cout<<c.Number<<"="<<i<<"+"<<j<<"+"<<k<<endl;
          p++;
        }
```

Число `c.Number` равно, как минимум, `i` + (`i` + 1) + (`i` + 1 + 1), следовательно, можно организовать внешний цикл (по `i`) не до `c.Number`, а до `c.Number/3`. Затем, если `i` минимально (равно 1), то `c.Number` равно, как минимум, 1 + `j` + (`j` + 1), следовательно, можно организовать цикл по `j` не до `c.Number`, а до `c.Number/2`. Программа станет работать в 3 × 2 = 6 раз быстрее.

Так как по условию задачи требуется найти еще и количество таких разложений, то при выполнении равенства `i + j + k` = = `c.Number` будем увеличивать переменную `p` на единицу.

Если после завершения работы трех циклов окажется, что `p=0`, то, следовательно, число `c.Number` нельзя представить в виде суммы трех чисел. В противном случае разложения существуют, и они уже выведены на экран.

```
  if (p==0) cout<<"Разложений не найдено"<<endl;
  else cout<<"Найдено разложений: "<<p<<endl;
}
```

После выполнения программы при `N=1` и `array[0]=12` на экран будет выведен следующий ответ:

```
12=1+2+9
12=1+3+8
12=1+4+7
12=1+5+6
12=2+3+7
12=2+4+6
12=3+4+5
Найдено разложений: 7
```

Если же запустить программу при `N=1` и `array[0]=4`, то ответ будет следующим:

```
Разложений не найдено
```

## 8.2. Классы

*Класс* — это определяемый пользователем тип. В отличие от структуры, класс может объединять в себе разнородные по типам переменные и функции. Рассмотрим объявление класса.

```
class Name
{
  private: int a, b[N];
           float c, d;
           int F();
  public:  char ST;
           void Sort(int);
} object;
```

Объявление класса состоит из нескольких частей. Рассмотрим каждую из них:

- `class` — зарезервированное слово;
- `Name` — имя класса;
- `private` — метка, за которой следует группа закрытых элементов. Элемент называется *закрытым*, если он может использоваться только другими элементами класса;

- `public` — метка, за которой следует группа открытых элементов. Элемент называется *открытым*, если к нему разрешен доступ из программы при помощи объектов соответствующего типа (так же, как к элементам структуры);
- `object` — объект с именем `object` типа `Name`.

**Примечание**

Перечень элементов класса заключен в фигурные скобки. Объявление заканчивает знаком `;`.

В объявлении класса указаны прототипы входящих в него функций-элементов. После объявления их следует описать. Правило описания функций-элементов следующее:

```
void Name::Sort(int)
{
  // Тело функции
}
```

Здесь `Name` — это имя класса, `void` — тип функции, `Sort` — имя функции.

Принадлежность функции к классу определяет его имя и нотация `::`.

**Внимание**

При попытке обратиться к закрытому элементу компилятор укажет на ошибку.

Рассмотрим использование класса на примере программы, которая сортирует массив в порядке возрастания и выводит его на экран. Далее приведен код программы:

```
#include <iostream.h>
#include <conio.h>
class Task  // Объявление класса
{
  // Закрытые переменные
  private: void swap(int,int); // Функция обмена
             // значениями двух элементов массива
```

```
          int array[100];
  // Открытые переменные
  public:  void input();       // Функция ввода данных
           void output();      // Функция вывода результата
           void sort(int);     // Функция сортировки массива
};          // Завершение объявления класса
int N;
void Task::input()  // Описание функции
{
  int i;
  for (i=0; i<N; i++)
    cin>>array[i];
}
void Task::sort(int size)  // Описание функции
{
  int start, i, min_pos, min ;
  start=0;           // Начальная левая граница подмассива
  // Запускает цикл повторений, который выполняет операции,
  // пока все подмассивы не пройдены
  while (start!=size-1)
  {
    // Предположим, что элемент с индексом левой границы
    // подмассива минимален
    min=array[start];
    min_pos=start;   // Запоминает индекс элемента с
                     // минимальным значением
    // Запускаем цикл поиска минимального элемента в подмассиве
    for (i=start; i<size; i++)
      if (array[i]<min)  // Сравнивает i-ый элемент массива
со
                         // значением переменной min
      {
        min=array[i];    // Запоминает минимальное значение
        min_pos=i;       // Запоминает позицию элемента
                         // с минимальным значением
```

```
      }
    // Если позиция минимального значения элемента массива не
    // совпадает с предположенной ранее, то меняет местами
    // элементы с данными позициями
    if (min_pos!=start)  // Запускает функцию, которая меняет
                         // местами два элемента массива
      start++;           // Уменьшает границу подмассива
  }
}
void Task::swap(int i, int j)  // Описание функции
{
  int temp;
  temp=array[i];
  array[i]=array[j];
  array[j]=temp;
}
void Task::output()            // Описание функции
{
  int i;
  for (i=0; i<N; i++)
    cout<<array[i]<<" ";  // Требует ввода очередного
                          // элемента массива
   getch();               // Задерживает экран
}
main()
{
  cin>>N;           // Требует ввода одного операнда
  Task object;      // Объявляет объект типа Task
  object.input();   // Запускает функцию, которая
                    // осуществляет ввод элементов массива
  object.sort(N);   // Запускает функцию, которая осуществляет
                    // сортировку массива
  object.output(); // Запускает функцию, которая осуществляет
                    // вывод отсортированного массива
}
```

Начнем разбор программы с объявления класса:

```
class Task
{
  private: void swap(int,int);
           int array[100];
  public:  void input();
           void done();
           void output();
           void sort(int);
};
```

Таким образом, будет объявлен класс с именем `Task` и с переменными закрытого и открытого типов.

За объявлением класса следует описание функций.

Функция `input()` используется для ввода элементов массива. Данный метод ввода элементов использовался раньше.

```
void Task::input()  // Описание функции
{
  int i;
  for (i=0; i<N; i++)
    cin>>array[i];
}
```

Строка `void Task::input()` говорит о том, что функция с именем `input` является элементом класса `Task`.

Следующая функция — это функция, организовывающая сортировку массива. Сортировка в данном случае выполнена методом вставок. Алгоритм этой сортировки разбирался в *главе 4*.

Функция с именем `swap` также принадлежит классу `Task`. Она организовывает обмен значениями между двумя элементами массива. В функцию передаются два целых числа, которые напрямую связаны с номерами элементов массива, значения которых необходимо поменять местами. Обратите внимание на то, что функция `swap(int,int)` является закрытым элементом-функцией,

т. к. она вызывается не из основной программы, а из другого элемента класса — функции `sort(int)`. Вот код функции `swap(int,int)`:

```
void Task::swap(int i, int j)  // Описание функции
{
  int temp;
  temp=array[i];
  array[i]=array[j];
  array[j]=temp;
}
```

Переменная `i` изменяет свое значение от 0 до `N` – 1, при этом на экран выводится элемент массива под номером `i` — `array[i]`.

```
void Task::output()        // Описание функции
{
  int i;
  for (i=0; i<N; i++)
    cout<<array[i]<<" ";  // Вывод очередного элемента массива
  getch();                // Задерживает экран
}
```

Теперь рассмотрим главную функцию `main()`, которая присутствует в любой программе. Первый шаг данной функции — считывание с клавиатуры числа `N`, которое отвечает за количество элементов в массиве:

```
main()
{
  cin>>N;
```

Затем функции ввода данных, сортировки массива и вывода результата вызываются одна за другой. Вызов функций, принадлежащих классу `Task`, осуществляется при помощи объекта `object` типа `Task`, который объявляется перед вызовом функций:

```
  Task object;
  object.input();
  object.sort(N);
  object.output();
}
```

## Вложенные классы

Описание класса может быть *вложенным*. Например:

```
class first
{
  struct second
  {
    int Num;
    float Neg;
  };
  public: int array[N];
};
```

Если только вложенный класс не является очень простым, то в таком описании трудно разобраться. Кроме того, вложение классов — это не более чем соглашение о записи, поскольку вложенный класс не является скрытым в области видимости лексически охватывающего класса.

## Дружественные функции

Ранее оговаривалось, что элементы класса могут быть *открытыми* и *закрытыми*. Для доступа к *закрытым* элементам класса в C++ используются *дружественные функции*.

Согласно правилам записи функций в C++, необходимо записать прототип дружественной функции, включаемый в объявление класса, с которым она "дружит". Доступ к закрытым элементам класса осуществляется при помощи объекта этого класса.


Прототип дружественной функции записывается в объявлении всех классов, с которыми она "дружит".

Прототип дружественной функции оформляется при помощи спецификатора `friend`.

Пусть определены два класса: `vector` (вектор) и `matrix` (матрица). Каждый из них скрывает свое представление, но дает полный

набор операций для работы с объектами его типа. Допустим, надо определить функцию, умножающую матрицу на вектор. Для простоты предположим, что вектор имеет четыре элемента с индексами от 0 до 3, а в матрице — четыре вектора тоже с индексами от 0 до 3. Доступ к элементам вектора обеспечивается функцией `elem()`, и аналогичная функция есть для матрицы. Можно определить глобальную функцию умножения `multiply()` следующим образом:

```
vector multiply(const matrix& m, const vector& v);
{
  vector r;
  for (int i=0; i<3; i++)
  {
    // r[i]=m[i]*v;
    r.elem(i)=0;
    for (int j=0; j<3; j++)
      r.elem(i)+=m.elem(i,j)*v.elem(j);
  }
  return r;
}
```

Это решение является не очень эффективным. При каждом вызове `multiply()` функция `elem()` будет выполняться $4 \times (1 + 4 \times 3)$ раз. Если в `elem()` проводится контроль границ массива, то на него будет потрачено значительно больше времени, чем на выполнение самой функции, и в результате она окажется непригодной для пользователей. С другой стороны, если `elem()` представляет собой некий специальный вариант доступа без контроля, то этой функцией доступа, которая нужна только для обхода контроля, усложняется интерфейс с вектором и матрицей. Если можно было бы сделать `multiply` членом обоих классов `vector` и `matrix`, можно было бы обойтись без контроля индекса при обращении к элементу матрицы, но в то же время не вводить специальной функции `elem()`. Однако функция не может быть членом двух классов. Надо иметь в языке возможность предоставлять функции, не являющейся членом, право доступа к частным членам класса. Такое право получает дружественная функция.

Как говорилось ранее, функция может стать другом класса, если она описана как `friend` (друг). Например:

```
class matrix;
class vector
{
  float v[4];
  // Здесь вы можете объявить нужные вам переменные
  friend vector multiply(const matrix&, const vector&);
};
class matrix
{
  vector v[4];
  // ...
  friend vector multiply(const matrix&, const vector&);
};
```

Функция-друг не имеет никаких особенностей, за исключением права доступа к закрытой части класса. Описания могут быть следующих видов:

- описание `friend` вводит имя функции в область видимости класса, в котором она была описана, и при этом происходят обычные проверки на наличие других описаний такого же имени в этой области видимости;
- описание `friend` может находиться как в открытой, так и в закрытой части класса.

Теперь можно написать функцию `multiply`, используя элементы вектора и матрицы непосредственно:

```
vector multiply(const matrix& m, const vector& v)
{
  vector r;
  for (int i=0; i<3; i++)
  {
    // r[i]=m[i]*v;
    r.v[i]=0;
    for (int j=0; j<3; j++)
```

```
    r.v[i]+=m.v[i][j]*v.v[j];
  }
  return r;
}
```

Подобно функции-члену, дружественная функция явно указывается в описании класса, с которым дружит. Поэтому она является неотъемлемой частью интерфейса класса наравне с функцией-членом.

Функция-член одного класса может быть другом какого-либо иного класса:

```
class First
{
  // Здесь вы можете описать нужные вам переменные
  void Test();
};
class Second
{
  // ...
  friend void First::Test();
};
```

Вполне возможно, что все функции одного класса являются друзьями какого-либо иного класса. Для этого существует краткая форма записи:

```
class First
{
  friend class Second;
  // ...
};
```

В результате такого описания все функции-члены класса `Second` становятся друзьями класса `First`.

## Производные классы

*Производные классы* дают простой, гибкий и эффективный аппарат задания для класса альтернативного интерфейса и определения класса посредством добавления возможностей к уже имею-

щемуся классу без перепрограммирования или перекомпиляции. С помощью производных классов можно также обеспечить общий интерфейс для нескольких различных классов так, чтобы другие части программы могли работать с объектами этих классов таким же образом. При этом обычно в каждый объект помещается информация о типе, чтобы эти объекты могли обрабатываться соответствующим образом в ситуациях, когда их тип нельзя узнать во время компиляции. Для элегантной и надежной обработки таких динамических зависимостей типов имеется понятие виртуальной функции. По своей сути производные классы существуют для того, чтобы облегчить программисту формулировку общности.

## Построение производного класса

Рассмотрим программу, которая имеет дело с людьми, служащими в некоторой фирме. Структура данных в этой программе может быть, например, такой:

```
struct employee  // Класс служащих
{
  char* name;
  short age;
  short department;
  int salary;
  employee* Next;
};
```

Список аналогичных служащих будет связываться через поле `Next`. Теперь определим менеджера:

```
struct manager   // Класс менеджеров
{
  employee emp;
  employee* group;
  // Список описаний может быть продолжен
};
```

Менеджер также является служащим. Относящиеся к служащему `employee` данные хранятся в члене `emp` объекта `manager`. Для че-

ловека это может быть очевидно, но нет ничего, что выделяло бы член `emp` для компилятора. Указатель на менеджера (`manager*`) не является указателем на служащего (`employee*`), поэтому просто использовать один там, где требуется другой, нельзя. В частности, нельзя поместить менеджера в список служащих, не написав для этого специальную программу. Можно либо применить к `manager*` явное преобразование типа, либо поместить в список служащих адрес члена `emp`, но и то, и другое довольно неясно. Корректный подход состоит в том, чтобы установить, что менеджер является служащим, при помощи некоторой добавочной информации:

```
struct manager: employee
{
  employee* group;
  // ...
};
```

Класс `manager` является производным от `employee`, а `employee` является базовым классом для `manager`. Класс `manager` дополнительно к члену `group` имеет члены класса `employee`.

Имея определения `employee` и `manager`, можно создать список служащих, которые являются менеджерами. Например:

```
void List()
{
  manager m1, m2;
  employee e1, e2;
  employee* elist;
  elist = &m1;
  m1.Next = &e1;
  e1.Next = &m2;
  m2.Next = &e2;
  e2.Next = 0;
}
```

Поскольку менеджер является служащим, `manager*` может использоваться как `employee*`. Однако служащий необязательно

является менеджером, поэтому использовать `employee*` как `manager*` нельзя.

## Инкапсуляция

Данные (переменные) и функции для работы с ними объединяются вместе с помощью специального структурированного типа данных, который называется классом. Кроме того, предполагается использование механизма защиты данных от несанкционированного доступа. Другими словами, с информацией, включенной в класс, разрешается работать только функциям данного класса. Этот механизм защиты называется *инкапсуляцией*.

Пусть члену класса требуется защита от несанкционированного доступа. Для этого следует ограничить множество функций, которым он будет доступен. Сделать это можно, предоставив доступ к данному члену класса только тем операциям, которые определены для этого объекта, иными словами, всем функциям-членам. Например:

```
class window
{
  // ...
  protected: int inside;
  // ...
};
class Set: public window
{
  // ...
  public: void Summa ();
  // ...
};
```

Здесь в базовом классе `window` член `inside` типа `int` описывается как защищенный (`protected`), но функции-члены производных классов, например, `Set::Summa()`, могут обратиться к нему и выяснить, с какого вида окном они работают. Для всех других функций член `window::inside` недоступен.

В таком подходе сочетается высокая степень защищенности (низка вероятность случайным образом определить производный класс) с гибкостью, необходимой для программ, которые создают классы и используют их иерархию (всегда можно в производных классах предусмотреть доступ к защищенным членам).

Из этого следует, что нельзя составить полный и окончательный список всех функций, которым будет доступен защищенный член, поскольку всегда можно добавить еще одну, определив ее как функцию-член в новом производном классе. Для метода абстракции данных такой подход зачастую неприемлем. Если язык ориентируется на метод абстракции данных, то очевидное для него решение — это требование указывать в описании класса список всех функций, которым нужен доступ к члену. В C++ для этой цели используется описание частных (`private`) членов.

Важность инкапсуляции, т. е. заключения членов в защитную оболочку, резко возрастает с ростом размеров программы и увеличивающимся разбросом областей приложения.

## Статические члены класса

Класс — это тип, а не данные, и для каждого объекта класса создается своя копия членов, представляющих данные. Однако наиболее удачная реализация некоторых типов требует, чтобы все объекты этого типа имели некоторые общие данные, которые можно описать как часть класса. Например, в операционных системах или при моделировании управления задачами часто нужен список задач:

```
class task
{
  // ...
  static task* Coled;
  // ...
};
```

Описав член `Coled` как статический, мы получаем гарантию, что он будет создан в единственном числе, т. е. не будет создаваться для каждого объекта `task`. Но он находится в области видимости

класса `task` и может быть доступен вне этой области, если описан в открытой части. В этом случае имя члена должно уточняться именем класса. В функции-члене его можно обозначать просто `Coled`.

Использование статических членов класса может заметно сократить потребность в глобальных переменных. Описывая член как статический, мы ограничиваем его область видимости и делаем его независимым от отдельных объектов его класса. Это свойство полезно как для функций-членов, так и для членов, представляющих данные:

```
class task
{
  // ...
  static task* task_Coled;
  static void Example(int);
  // ...
};
```

Но описание статического члена — это только описание, и где-то в программе должно быть единственное определение описываемого объекта или функции, например, такое:

```
task* task::task_Coled = 0;
void task::Example(int p)
{
  // Тело функции
}
```

Закрытые члены также могут определяться подобным образом. Служебное слово `static` не следует использовать в определении статического члена класса. Если бы оно присутствовало, было бы неясно, указывает ли оно на то, что член класса является статическим, или используется для описания глобального объекта или функции.

Зарезервированное слово `static` — одно из самых перегруженных служебных слов в C++. К статическому члену, представляющему данные, относятся оба основных его значения: статически

размещаемый, т. е. противоположный объектам, размещаемым в стеке или свободной памяти, и статический в смысле ограничения области видимости, т. е. противоположный объектам, подлежащим внешнему связыванию. К функциям-членам относится только второе значение `static`.

## Класс *ios*

Класс `ios` является базовым для `ostream` и `istream`. В нем находится управление связью между `istream` или `ostream` и буфером, используемым для операций ввода/вывода. Именно класс `ios` контролирует попадание символов в буфер и извлечение их оттуда. Так, в данном классе есть член, содержащий информацию об используемой при чтении или записи целых чисел системе счисления (десятичная, восьмеричная или шестнадцатеричная), о точности вещественных чисел и т. п., а также функции для проверки и установки значений переменных, управляющих потоком.

## Указатель *this*

Рассмотрим программу, которая выводит на экран два сообщения: "Пример!" и "Example!" При написании будем использовать класс `Str`.

```
class Str
{
  char *String;
  public: void set(char *text) {String=text;}
          void write() {cout<<String<<endl;}
};
main()
{
  Str str1, str2;
  str1.set("Пример!");
  str2.set("Example!");
  str1.write();
  str2.write();
}
```

В результате выполнения этой программы на экране появится следующее сообщение:

```
Пример!
Example!
```

Функция-член `write` самостоятельно определяет, для какого объекта она вызвана, потому что ей в качестве неявного аргумента передается адрес этого объекта. В данном случае — указатель типа `str*`.

Внутри функции-члена класса можно явно использовать этот указатель при помощи зарезервированного слова `this`.

Перед началом выполнения кода функции указатель `this` инициализируется значением адреса объекта, для которого вызвана данная функция-член. Таким образом, приведенное выше определение функции `Str::write()` представляет собой сокращенную форму следующей записи:

```
void write()
{
  cout<<this->String<<endl;
}
```

**Примечание**

Указатель `this` является константным указателем. Это означает, что непосредственное изменение его значений недопустимо, он с самого начала настраивается на определенный объект.

# 8.3. Конструктор и деструктор

Когда происходит объявление класса, в C++ никакое действие не производится, и, следовательно, не выделяется память. Эта процедура похожа на описание новой переменной, поэтому при объявлении инициализация запрещена. В то же время объявление объекта есть некоторое действие, при котором выделяется определенный объем памяти и вызывается конструктор. Другими словами, *конструктор* — это метод, который инициализирует объект.

*Деструктор* действует по противоположным правилам. Он уничтожает созданный объект.

Перечислим некоторые свойства конструктора и деструктора:

- конструктор может иметь аргументы;
- конструктор может быть перегружен, как и обычная функция;
- деструктор, в отличие от конструктора, не может иметь аргументов и не может быть перегружен.

В языке программирования C++ предусмотрен *конструктор* с аргументами по умолчанию. Аргументы по умолчанию могут быть константами или глобальными переменными, так же как и для обычной функции.

Если у класса есть конструктор, то он вызывается всегда, когда создается объект класса. Если у класса есть деструктор, то он вызывается всегда, когда объект класса уничтожается.

Рассмотрим класс, в котором демонстрируется использование конструктора и деструктора на примере добавления и удаления элемента из стека.

```
class char_stack
{
  int size;
  char* top;
  char* s;
  public: char_stack(int sz) {top=s=new char[size=sz];}
         ~char_stack() {delete s;}      // Деструктор
          void push(char c) {*top++=c;}
          char pop() {return *--top;}
};
void function()
{
  char_stack s1(100);
  char_stack s2(99);
}
```

Здесь конструктор `char_stack::char_stack()` и деструктор `char_stack::~char_stack()` вызываются дважды: для `s1` и для `s2`.

Приведем еще один пример использования конструктора и деструктора:

```
#include <iostream.h>
#include <conio.h>
#include <fstream.h>
ofstream ConDes("ConDes.txt");
class Example     // Начало объявления класса
{
  int x, y;
  public:
    Example(int i=5, int j=4)  // Конструктор
  {
    x=i;
    y=j;
    ConDes<<"Constructor: "<<x<<" "<<y<<endl;
  }
  ~Example()                   // Деструктор
  {
    ConDes<<endl<<"Destructor: "<<x<<" "<<y;
  }
};
main()
{
  // Объявление объектов класса
  Example ob1, ob2(15), ob3(15,2);
  return 0;    // Завершает работу программы
}
```

В данной программе объявляются следующие три объекта класса:

- `ob1` — использует заданные по умолчанию значения аргументов (в качестве аргументов выступают переменные `x` и `y`);
- `ob2` — явно задает значение первого аргумента, второй аргумент используется по умолчанию;
- `ob3` — явно задает значение для двух аргументов.

Конструктор и деструктор данного класса информируют о своем существовании путем вывода надписи в файл.

**Примечание**

Конструкторы в данном случае вызываются в порядке объявления объектов, а деструкторы в обратном порядке.

После того как программа завершит свою работу, в файл будут записаны следующие сообщения:

```
Constructor: 5 4
Constructor: 30 4
Constructor: 30 6
Destructor: 30 6
Destructor: 30 4
Destructor: 5 4
```

Конструктор — это специальная функция, которая служит для инициализации, т. е. задания начальных значений всех или некоторых полей класса.

Конструктор обладает следующими особенностями:

- он не имеет никакого типа возвращаемого значения, в том числе `void`;
- имя конструктора совпадает с именем класса;
- вызывается конструктор автоматически при объявлении объекта. Благодаря последним двум особенностям конструктор обязательно будет выполнен, т. к. мы не можем забыть объявить объект, а значит, вызвать конструктор. В противном случае компилятор напомнит об этом.

Например, вместо того чтобы описывать несколько функций, можно описать конструктор, который из параметра `double` создает `complex`:

```
class complex
{
  // ...
  complex(double r) {re=r; im=0;}
};
```

# 8.4. Наследование

*Наследование* — механизм, который реализует использование элементов одних классов другими. Данный механизм действует по принципу создания иерархии классов: base class (базовый) ← ← derived class (производный).

Учитывая сложность и важность отношений наследования, нет ничего удивительного в том, что часто их неправильно понимают и используют сверх меры. Если класс `D` описан как общий производный от класса `B`, то часто говорят, что `D` есть `B`:

```
class B
{
  // ...;
  class D: public B
  // ...;
};
```

Определение наследования можно сформулировать следующим образом.

Наследование означает такое соотношение между классами, когда один класс использует структурную или функциональную часть одного или нескольких других классов (соответственно простое и множественное наследование). Говоря простым языком, наследование — это отношение "есть", или, например, для классов `D` и `B` наследование — это отношение `D` сорта `B`. В отличие от этого, если класс `D` содержит в качестве члена другой класс `B`, то говорят, что `D` "имеет" `B`.

Принадлежность — это отношение "иметь". Например, для классов `D` и `B` это означает, что `D` содержит `B`.

Для примера рассмотрим классы `клавиатура` и `клавиша`. Класс `клавиатура` не может быть производным от класса `клавиша`, т. к. `клавиатура` не "есть" `клавиша`, а "имеет" `клавишу`. Поскольку `клавиатура` может иметь две или больше `клавиши`, следует использовать принадлежность, а не наследование.

# 8.5. Упражнения

1. Составьте класс для работы с целочисленной матрицей, включив в него матрицу, ее текущий размер и следующие методы:
   - конструктор для ввода и проверки текущих размеров матрицы;
   - ввод матрицы;
   - вывод матрицы;
   - поиск наибольшего и наименьшего значений матрицы и номеров строк, где они находятся;
   - нахождение количества чисел-палиндромов.

   В головной функции нужно ввести размер матрицы, создать объект и проверить составленные методы.
2. Составьте класс для работы с двумя строками `S1` и `S2`, включив в него сами строки, их длины и следующие методы:
   - ввод строк;
   - поиск в `S1` подстроки максимальной длины, которая содержится и в `S2`;
   - вывод подстроки.
3. Составьте класс для работы со стеком, предусмотрев следующие методы:
   - создание пустого стека;
   - установление максимального размера;
   - добавление элемента, причем необходимо предусмотреть случай, когда стек заполнен;
   - выделение элемента из стека;
   - удаление элемента, причем необходимо предусмотреть случай, когда стек пуст.
4. Составьте класс для работы с очередью, предусмотрев следующие методы:
   - создание пустой очереди;
   - установление максимального размера;

- добавление элемента, причем необходимо предусмотреть случай, когда очередь полна;
- выделение элемента из очереди;
- удаление элемента, причем необходимо предусмотреть случай, когда очередь пуста.

5. Составьте класс для работы со списком, предусмотрев следующие методы:
   - создание пустого списка;
   - установление максимального размера;
   - добавление элемента в конец списка или в любое место, причем необходимо предусмотреть случай, когда список заполнен;
   - сортировку списка;
   - удаление элемента, причем необходимо предусмотреть случай, когда список пуст.
6. Составьте класс для работы с массивом, включив в него сам массив, его размерность, а также предусмотрите следующие методы:
   - ввод массива;
   - генерацию всех перестановок элементов массива;
   - вывод каждой перестановки в файл.
7. Составьте класс для работы с файлом. В класс необходимо включить путь к файлу, его размер в байтах, а также предусмотреть следующие методы:
   - открытие файла для чтения;
   - последовательное чтение данных из файла;
   - сортировку символов файла по возрастанию ASCII-кода;
   - вывод на экран одинаковых слов, содержащихся в файле.
8. Составьте класс для работы с числами, включив в него количество введенных чисел, а также предусмотрев следующие методы:
   - нахождение чисел, которые входят в ряд Фибоначчи;

- вывод каждого числа из ряда;
- вывод порядкового номера числа в ряду Фибоначчи.

9. Составьте класс для работы с большими числами предусмотрев следующие методы:
   - ввод двух чисел;
   - нахождение суммы чисел;
   - нахождение разности чисел;
   - нахождение произведения чисел;
   - вывод соответствующего результата в файл.
10. Составьте класс для работы с небольшой базой данных, включив в него элементы базы, а также предусмотрев следующие методы:
    - чтение учетных записей базы данных из файла;
    - добавление указанного элемента в базу;
    - удаление указанного элемента из базы;
    - сортировку элементов базы данных по алфавитному возрастанию учетных записей;
    - сохранение базы данных на жесткий диск.

**Примечание**

Во всех задачах *главы 8* входные данные считать корректными.

# Глава 9

# Основы теории графов

Теория графов началась с задачи про семь кенигсбергских мостов. Вопрос задачи звучит следующим образом: можно ли пройти по всем семи мостам города Кенигсберг и при этом вернуться в исходную точку так, чтобы по каждому мосту пройти только один раз. Знаменитый петербургский математик швейцарского происхождения Леонард Эйлер блестяще решил эту задачу при помощи теории графов.

В этой главе вы познакомитесь с основными алгоритма и понятиями теории графов. После ознакомления с данным материалом вы сами легко решите задачу о семи кенигсбергских мостах, а также сможете решать огромное количество других задач, потому что любую задачу можно свести к графам.

## 9.1. Понятие графа

Чтобы дать определение *графа*, необходимо ввести несколько промежуточных определений.

Пусть задано некоторое конечное множество точек. Данное множество будем называть *вершинами графа* и обозначать VG.

Любая пара элементов из множества VG будет называться *ребрами графа* и обозначаться EG.

Вершины графа будем обозначать так: $VG=\{V_1, V_2, V_3, \ldots, V_n\}$ или цифрами VG={1, 2, 3, ..., n}.

Ребра графа будем обозначать парами следующим образом: EG={(V1,V2), (V3,V4)} или EG=((1,2), (3,4)).

При выборе пары вершин из множества VG не имеет значения, в каком порядке они выписаны, т. е. пары вершин (2, 3) и (3, 2) задают одно и то же ребро.

Итак, дадим определение графу.

*Граф* G — это множество вершин VG и множество некоторых пар вершин, т. е. множество ребер EG.

Если некоторая пара вершин соединена несколькими ребрами, то такой граф будет называться *мультиграфом*.

Так как для графа G не имеет значения, в каком порядке выписаны пары вершин из множества VG, то назовем такой граф *неориентированным*.

## Представление неориентированного графа

На бумаге граф удобно представлять в виде рисунков, на которых кружок (или точка) — это *вершина* графа, а прямая, соединяющая два любые кружка (точки) — *ребро* (рис. 9.1).

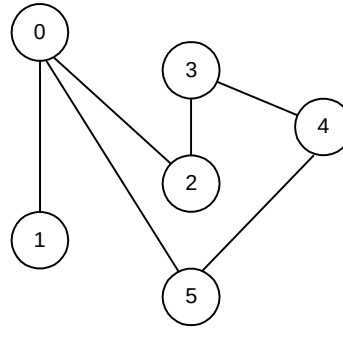



**Рис. 9.1.** Неориентированный граф G1

Граф G1 (рис. 9.1) состоит из шести вершин VG={0, 1, 2, 3, 4, 5} и шести ребер EG={(0,1), (0,2), (0,5), (2,3), (3,4), (4,5)}.

## Представление неориентированного графа в памяти компьютера

Чтобы воспользоваться графом при решении задач на компьютере, необходимо представить его в понятном для компьютера виде.

Существует несколько способов представления графа в памяти компьютера. Например, следующие:

- ❒ матрица смежности;
- ❒ список смежности.

Рассмотрим представление графа при помощи *матрицы смежности*. В качестве примера возьмем граф, изображенный на рис. 9.1.

В данном графе определено 6 вершин и 6 ребер, поэтому для представления графа понадобится матрица размером 6×6, т. е. таблица из шести строк и шести столбцов.

В ячейках $a_{i,j}$ и $a_{j,i}$ матрицы будем ставить единицу, если вершины графа `i` и `j` соединены ребром, в противном случае заполняем ячейки нулями.

С учетом введенных правил граф (см. рис. 9.1) представится в виде матрицы следующим образом (рис. 9.2).

| | | **Столбцы** | | | | | |
|---|---|---|---|---|---|---|---|
| | | **0** | **1** | **2** | **3** | **4** | **5** |
| **Строки** | **0** | 0 | 1 | 1 | 0 | 0 | 1 |
| | **1** | 1 | 0 | 0 | 0 | 0 | 0 |
| | **2** | 1 | 0 | 0 | 1 | 0 | 0 |
| | **3** | 0 | 0 | 1 | 0 | 1 | 0 |
| | **4** | 0 | 0 | 0 | 1 | 0 | 1 |
| | **5** | 1 | 0 | 0 | 0 | 1 | 0 |

**Рис. 9.2.** Матрица смежности графа `G1`

**Внимание**

При изменении нумерации вершин графа меняется и его матрица смежности.

Матрица смежности графа является симметричной матрицей, на главной диагонали которой стоят нули.

Задание графа с помощью матрицы смежности является очень удобным: чтобы узнать, соединены ли вершины `i` и `j` ребром, достаточно только проверить, чему равен элемент $a_{i,j}$.

Если говорить о недостатке такого представления, то он состоит в том, что для хранения графа из $n$ вершин требуется $n^2$ ячеек памяти.

Этот недостаток исправляется при задании графа *списком смежности*. При таком задании вершине `i` графа ставятся в соответствие только вершины, которые находятся в паре с `i`. Это означает, что если в графе определяются ребра `(i,j)`, `(i,k)` и `(i,d)`, то в соответствие вершине `i` будут поставлены вершины `j` и `k`.

Для графа `G1` (см. рис. 9.1) список смежности будет выглядеть так, как представлено на рис. 9.3.

```
0  1  2  5
1  0
2  0  3
3  2  4
4  3  5
5  0  4
```

**Рис. 9.3.** Список смежности графа `G1`

У списка смежности также есть свои недостатки: чтобы выяснить, соединены ли вершины `i` и `j` ребром, необходимо проделать большое количество операций.

Рассмотрим небольшую программу, которая представляет граф в памяти компьютера в виде матрицы смежности. Граф задан списком ребер, который находится в файле.

**Примечание**

Если граф задан списком ребер, то это означает, что в файле содержится некоторое количество строк, в каждой из которых рас-

полагается пара чисел `i` и `j`, обозначающих вершины, соединенные данным ребром.

```
#include <fstream.h>
main()
{
  int **Graf, i, j, N;
  ifstream input("Graf.txt");  // Открывает файл для чтения
  input>>N;    // Читает из файла количество вершин в графе
  // Выделяет динамическую память для матрицы смежности графа
  Graf=new int *[N];
  for (i=0; i<N; i++)
    Graf[i]=new int [N];
  // Обнуляет все элементы матрицы
  for (i=0; i<N; i++)
    for (j=0; j<N; j++)
      Graf[i][j]=0;
  while (!input.eof())
  {
    input>>i>>j;
    Graf[i][j]=1;
    Graf[j][i]=1;
  }
  return 0;
}
```

Теперь напишем программу, которая представит граф в виде списка смежности.

```
#include <fstream.h>
main()
{
  int **Graf, i, j, pos, N;
  ifstream input("Graf.txt");  // Открывает файл для чтения
  input>>N;    // Читает из файла количество вершин в графе
  // Выделяет динамическую память для матрицы смежности графа
  Graf=new int *[N];
  for (i=0; i<N; i++)
```

```
    Graf[i]=new int [2];
  // Обнуляет все элементы матрицы
  for (i=0; i<2; i++)
    for (j=0; j<N; j++)
      Graf[i][j]=0;
  pos=0;
  while (!input.eof())
  {
    input>>i>>j;
    Graf[0][pos]=i;
    Graf[1][pos]=j;
    pos++;
  }
  return 0;
}
```

Двумерная матрица в коде программы отличается от вида списка смежности, приведенного на рис. 9.3, т. к. таблицу визуально легче воспринимать, чем список смежности. Если потом полученный список в программе отсортировать по возрастанию вершин, то получится та же таблица.

## Представление ориентированного графа

Граф будем называть *ориентированным* или *орграфом*, если пара $(V_i, V_j)$ задается однозначно, т. е. связь между вершинами однонаправленная — вершина $V_i$ соединена *дугой* с $V_j$, а $V_j$ не соединена с $V_i$ (рис. 9.4).

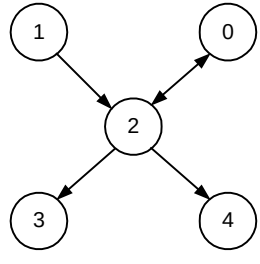



**Рис. 9.4.** Ориентированный граф

Если граф ориентированный, то пары элементов из множества `VG` называются не ребрами, а *дугами*.

Из рис. 9.4 видно, что если вершина `i` соединена ребром с вершиной `j`, то дуга `(i,j)` изображается линией, исходящей из вершины `i` и заканчивающейся стрелкой в вершине `j`.

Если дуга соединят некоторою вершину с собой, то она называется *петлей*.

## Представление ориентированного графа в памяти компьютера

Ориентированный граф, так же, как и неориентированный, можно представлять в памяти компьютера при помощи матрицы смежности и списка смежности.

Рассмотрим представление графа `G2` (рис. 9.5) при помощи матрицы смежности.

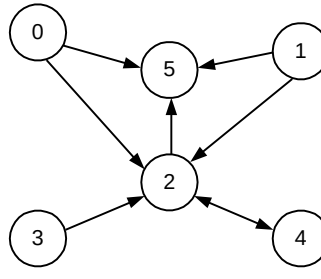



**Рис. 9.5.** Граф `G2`

В данном графе определено шесть вершин и семь ребер. Чтобы задать граф `G` матрицей смежности, необходима матрица 6×6.

В ячейке $a_{i,j}$ матрицы будем ставить единицу, если вершины графа `i` и `j` соединены ребром. В противном случае заполняем ячейки нулями.

С учетом введенных правил граф `G2` с рис. 9.5 представится в виде матрицы так, как показано на рис. 9.6.

| | | Столбцы | | | | | |
|---|---|---|---|---|---|---|---|
| | | **0** | **1** | **2** | **3** | **4** | **5** |
| Строки | **0** | 0 | 0 | 1 | 0 | 0 | 1 |
| | **1** | 0 | 0 | 1 | 0 | 0 | 1 |
| | **2** | 0 | 0 | 0 | 0 | 1 | 1 |
| | **3** | 0 | 0 | 1 | 0 | 0 | 0 |
| | **4** | 0 | 0 | 1 | 0 | 0 | 0 |
| | **5** | 0 | 0 | 0 | 0 | 0 | 0 |

**Рис. 9.6.** Матрица смежности направленного графа `G2`

**Внимание**

При изменении нумерации вершин графа меняется и его матрица смежности.

Ориентированный граф задается при помощи списка смежности таким же образом, как и неориентированный граф. Вершине `i` графа ставятся в соответствие вершины, которые находятся в паре с `i`. Это означает, что если в графе определяются ребра `(i,j)`, `(i,k)`, `(i,d)`, то в соответствие вершине `i` будут поставлены вершины `j` и `k`.

Пусть задан некоторый направленный граф `G3` (рис. 9.7).

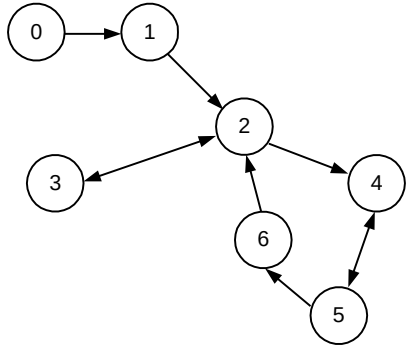



**Рис. 9.7.** Граф `G3`

```
0  1
1  2
2  3  4  6
3  2
4  5
5  4  6
6
```

**Рис. 9.8.** Список смежности направленного графа

Список смежности для этого графа будет выглядеть так, как представлено на рис. 9.8.

Далее приведен код программы, которая представляет ориентированный граф в виде матрицы смежности.

```
#include <fstream.h>
main()
{
  int Graf[100][100], i, j, N;
  ifstream input("Graf.txt");  // Открывает файл для чтения
  input>>N;    // Читает из файла количество вершин в графе
  // Выделяет динамическую память для матрицы смежности графа
  Graf=new int *[N];
  for (i=0; i<N; i++)
    Graf[i]=new int [N];
  // Обнуляет все элементы матрицы
  for (i=0; i<N; i++)
    for (j=0; j<N; j++)
      Graf[i][j]=0;
  while (!input.eof())
  {
    input>>i>>j;
    Graf[i][j]=1;
  }
  return 0;
}
```

Теперь рассмотрим программу, которая представит ориентированный граф в виде списка смежности.

```
#include <fstream.h>
main()
{
  int **Graf, i, j, pos;

  ifstream input("Graf.txt");    // Открывает файл для чтения
  input>>N;      // Читает из файла количество вершин в графе
  // Выделяет динамическую память для матрицы смежности графа
  Graf=new int *[N];
  for (i=0; i<N; i++)
    Graf[i]=new int [2];
  // Обнуляет все элементы матрицы
  for (i=0; i<2; i++)
    for (j=0; j<N; j++)
      Graf[i][j]=0;
  pos=0;
  while (!input.eof())
  {
    input>>i>>j;
    Graf[0][pos]=i;
    Graf[1][pos]=j;
    pos++;
  }
  return 0;
}
```

## Смежность и инцидентность

Две вершины `i` и `j` называются *смежными*, если они соединены ребром. Вершины `i` и `j` называются *концами* ребра. В этом случае говорят, что вершина `i` — инцидентна ребру (0, 1), а ребро (0, 1) инцидентно вершине `i`.

Число ребер, инцидентных некоторой вершине, называется *степенью* (валентностью) данной вершины и обозначается `deg i` или `d(i)`, где `i` — это некоторая вершина графа `G`.

Список степеней вершин графа называется его *степенной последовательностью*.

Если не найдется ни одного ребра, которые будут инцидентны вершине `i`, то эта вершина будет называться *изолированной* (рис. 9.9).

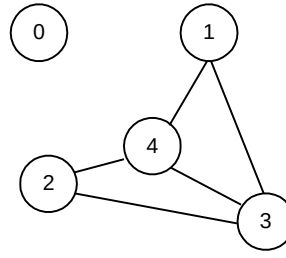



**Рис. 9.9.** Граф с изолированной вершиной

Если только одно ребро графа инцидентно вершине `i`, то эта вершина будет называться *висячей* (концевой).

Ребро, инцидентное висячей вершине, тоже называется *висячим*.

Если найдется такая вершина $V_0$ графа `G`, которая будет являться смежной с любой другой вершиной графа, то такая вершина будет называться *доминирующей* (рис. 9.10).

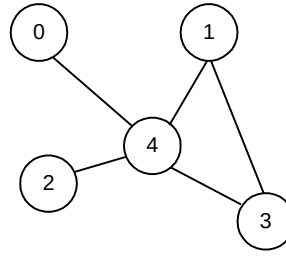



**Рис. 9.10.** Граф с доминирующей вершиной

Вершины $v_1$, $v_2$, ..., $v_n$, смежные с вершиной $v_0$, называются *окружением* вершины $v_0$.

Два ребра называются *смежными*, если они имеют один общий конец (рис. 9.11).

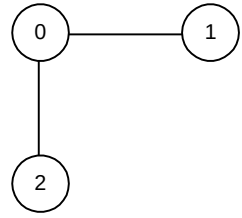



**Рис. 9.11.** Смежные ребра

## Цепи и циклы

*Маршрутом* в графе называется чередующаяся последовательность вершин и ребер, таких, что ребро $e_i$ соединяет вершины $v_i$ и $v_{i+1}$. Другими словами, если некоторый граф $G$ является схемой района, причем его вершины обозначают населенные пункты, а ребра — дороги, то маршрут в графе есть маршрут автомобилиста, путешествующего по району.

Маршрут можно задать последовательностью вершин, например, $v_1$, $v_2$, ..., $v_{i+1}$, а также последовательностью ребер $e_1$, $e_2$, ..., $e_i$.

*Длиной маршрута* называется число ребер, которые в нем содержатся.

*Расстоянием* между двумя вершинами $v_i$ и $v_j$ называется длина кратчайшего маршрута из $v_i$ в $v_j$. Например, расстояние между вершинами 0 и 4 графа $G$ (см. рис. 9.7) равно трем. Маршрут будет следующим: 0 — 1 — 2 — 4.

*Цепью* будем называть маршрут, все ребра которого различны, а *простой цепью* назовем цепь, в которой все вершины, возможно, кроме крайних, различны.

**Примечание**

Вершины должны быть различны, но для первой и последней это условие может не выполняться. Это означает, что последовательность вершин будет являться простой цепью, если все вершины, входящие в цепь, различны, кроме, возможно, крайних.

Если первая и последняя вершины совпадают, то такая цепь будет являться *циклом*.

Например, в графе (рис. 9.12) цепь из вершин 0 — 1 — 2 — 4 — 3 — 2 — 0 будет являться циклом.

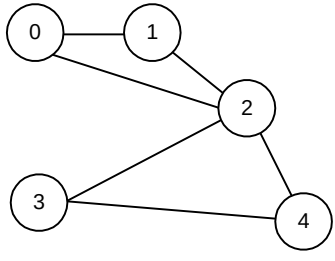



**Рис. 9.12.** Цепь, образующая цикл

Цикл, у которого все вершины различны (за исключением первой и последней), называется *простым циклом*. На рис. 9.12 простым циклом является цикл 0 — 1 — 2 — 0.

Аналогичные определения верны и для ориентированных графов. Только цепь, у которой все вершины различны, называется *путем*, а цикл, у которого все вершины различны, — *контуром*.

# 9.2. Обходы графов

## Обход в ширину

*Обход* (*поиск) в ширину* является алгоритмом, который необходим при решении многих задач. Поиск в ширину эффективно просматривает вершины и ребра графа, задавая им некоторые метки.

Рассмотрим действие данного алгоритма на примере некоторого графа `G` (рис. 9.13). Так как алгоритм будет некоторым образом помечать вершины, то на первом шаге сделаем метки всех вершин нулевыми.

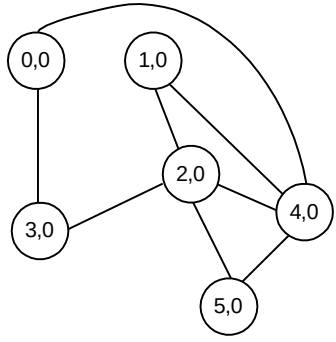



**Рис. 9.13.** Шаг 1 алгоритма поиска в ширину

Теперь необходимо выбрать стартовую вершину поиска. Пусть это будет вершина `0`. Каждой вершине из окружения вершины `0` присвоим метку `1` (рис. 9.14).

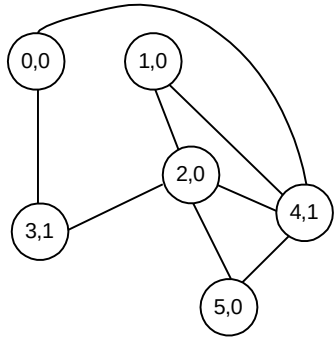



**Рис. 9.14.** Шаг 2 алгоритма поиска в ширину

**Примечание**

Присваивание происходит только в том случае, если вершина не помечена.

Таким образом, вершины 3 и 4 получили метку 1.

Теперь рассмотрим поочередно окружение всех вершин с меткой 1, и каждой вершине из этого окружения присвоим метку 2 (рис. 9.15).

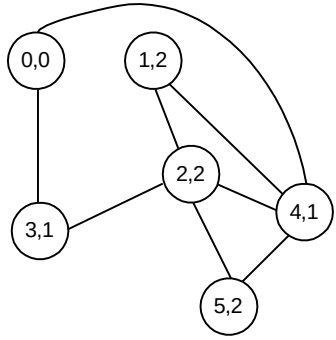



**Рис. 9.15.** Шаг 3 алгоритма поиска в ширину

Таким образом, вершины 2, 1 и 5 получили метку 2.

Далее, рассматривая окружение вершин 2, 1 и 5, алгоритм не найдет ни одной непомеченной вершины. Данный результат означает, что поиск в ширину выполнен, и вершины получили свои метки.

Процесс расстановки меток можно сравнить с процессом распространения волны. Поэтому часто поиск в ширину называют *волновым алгоритмом*.

После того как все вершины получили свои метки, можно определить расстояние от стартовой вершины $V_0$ до любой другой. Оно будет равно метке вершины, расстояние до которой необходимо определить. Следовательно, расстояние от вершины 0 до вершин 3 и 4 равно единице, а расстояние до вершин 1, 2 и 5 равно двум.

## Реализация поиска в ширину

Для реализации волнового алгоритма зададим граф G4 с помощью матрицы смежности (рис. 9.16).

| | | Столбцы | | | | | |
|---|---|---|---|---|---|---|---|
| | | **0** | **1** | **2** | **3** | **4** | **5** |
| Строки | **0** | 0 | 0 | 0 | 1 | 1 | 0 |
| | **1** | 0 | 0 | 1 | 0 | 1 | 0 |
| | **2** | 0 | 1 | 0 | 1 | 1 | 1 |
| | **3** | 1 | 0 | 1 | 0 | 0 | 0 |
| | **4** | 1 | 1 | 1 | 0 | 0 | 1 |
| | **5** | 0 | 0 | 1 | 0 | 1 | 0 |

**Рис. 9.16.** Матрица смежности графа `G4`

Для каждой вершины зарезервируем в массиве `Label` ячейку, в которую будет помещена метка данной вершины. При изложении алгоритма в виде рисунков на первом шаге всем вершинам были присвоены метки `0`, однако при реализации на практике в самом начале алгоритма *N* ячеек массива заполним большим числом, например, 32 767 (табл. 9.1), чтобы затем удобнее было производить сравнение.

***Таблица 9.1.*** *Создание массива меток перед началом волнового алгоритма*

| **Индекс** | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| **Элементы** | 32 767 | 32 767 | 32 767 | 32 767 | 32 767 | 32 767 |

Кроме того, на каждом шаге будем последовательно создавать очереди вершин, имеющих метки `1`, `2` и т. д. В качестве начальной (стартовой) вершины возьмем вершину `0`, т. е. с этой вершины мы будем начинать обход. В качестве начальной можно взять любую из множества вершин графа. После того как мы объявили вершину `0` стартовой, она будет занесена в очередь и получит метку `0` (табл. 9.2, 9.3).

***Таблица 9.2.*** *Создание очереди на первом шаге волнового алгоритма*

| **Индекс** | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| **Элементы** | 0 | — | — | — | — | — |

***Таблица 9.3.*** *Массив меток на первом шаге волнового алгоритма*

| **Индекс** | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| **Элементы** | 0 | 32 767 | 32 767 | 32 767 | 32 767 | 32 767 |

Далее просмотрим нулевую строку матрицы. Если некоторый элемент матрицы `array[0][i]` не равен нулю, то `i` помещается в очередь. В нашем случае вершины `3` и `4` будут занесены в очередь и помечены меткой `1`. Таким образом, очередь и массив меток примут следующий вид (табл. 9.4, 9.5).

***Таблица 9.4.*** *Очередь на втором шаге волнового алгоритма*

| **Индекс** | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| **Элементы** | 0 | 3 | 4 | — | — | — |

***Таблица 9.5.*** *Массив меток на втором шаге волнового алгоритма*

| **Индекс** | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| **Элементы** | 0 | 32 767 | 32 767 | 1 | 1 | 32 767 |

После данных действий вершина `0` будет удалена из очереди путем перемещения указателя начала очереди на один элемент вправо.

Теперь рассмотрим окружение вершин `3` и `4`, т. е. поочередно просмотрим четвертую и пятую строки матрицы.

Если некоторый элемент матрицы `array[3][i]` не равен нулю, то `i` помещается в очередь. В нашем случае вершина `2` будет зане-

сена в очередь и получит метку 2. Таким образом, очередь и массив меток примут вид, как представлено в табл. 9.6 и 9.7.

***Таблица 9.6.*** *Очередь на третьем шаге волнового алгоритма*

| **Индекс** | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| **Элементы** | 0 | 3 | 4 | 2 | — | — |

***Таблица 9.7.*** *Массив меток на третьем шаге волнового алгоритма*

| **Индекс** | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| **Элементы** | 0 | 32 767 | 2 | 1 | 1 | 32 767 |

После данных действий вершина 3 будет удалена из очереди путем перемещения указателя начала очереди на один элемент вправо.

Рассмотрим четвертую строку матрицы. Если некоторый элемент матрицы `array[4][i]` не равен нулю, то `i` помещается в очередь. В нашем случае вершины 1 и 5 будут занесены в очередь и получат метку 2. Таким образом, очередь и массив меток примут вид, как представлено в табл. 9.8 и 9.9.

***Таблица 9.8.*** *Очередь на четвертом шаге волнового алгоритма*

| **Индекс** | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| **Элементы** | 0 | 3 | 4 | 2 | 1 | 5 |

***Таблица 9.9.*** *Массив меток на четвертом шаге волнового алгоритма*

| **Индекс** | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| **Элементы** | 0 | 2 | 2 | 1 | 1 | 2 |

После данных действий вершина 4 будет удалена из очереди путем перемещения указателя начала очереди на один элемент вправо.

Рассматривая окружение вершин 1 и 5, алгоритм не найдет ни одной непомеченной вершины, следовательно, вершины последовательно будут удалены из очереди.

Итак, все вершины графа оказались помечены, а очередь пуста. На этом поиск в ширину окончен.

Рассмотрим теперь программу, которая выполняет поиск в ширину в некотором графе.

```
#include <iostream.h>
#include <conio.h>
#include <fstream.h>
int i, j, k, p, cur;
int Start, N, M;
main()
{
  int *Label;  // Объявляет массив меток
  int *FIFO;   // Объявляет структуру данных "очередь"
  int **Graf;  // Объявляет матрицу смежности графа
  ifstream input ("input.txt"); // Открывает файл для
                                // чтения данных
  // Читает из файла N — количество вершин в графе,
  //                  M — количество ребер в графе,
  //              Start — стартовая вершина обхода
  input>>N>>M>>Start;
  Label=new int [N];  // Выделяет динамическую память для
              // массива меток, состоящего из N элементов
  FIFO=new int [N];   // Выделяет динамическую память для
                      // очереди, состоящей из N элементов
  // Выделяет динамическую память для матрицы N*N
  Graf=new int *[N];
  for (i=0; i<N; i++)
    Graf[i]=new int[N];
  for (k=0; k<M; k++)
  {
    input>>i>>j;   // Читает номера смежных вершин
```

```
    Graf[i][j]=1;  // Задает матрицу смежности графа
  }
  for (i=0; i<M; i++)
  {
    FIFO[i]=0;
    Label[i]=32767;
  }
  p=0;  // Указатель на начало очереди
  k=1;  // Указатель на конец очереди
  FIFO[p]=Start;   // Заносит стартовую вершину в очередь
  Label[Start]=0;  // и помечает ее меткой 0

  while (p!=k)
  {
    cur=FIFO[p];    // Выделяет очередную вершину из очереди
    p++;            // Сдвигает указатель начала на единицу
    for (i=0; i<N; i++)
      if (Graf[cur][i]==1 && Label[i]>Label[cur]+1)
      {
        FIFO[k]=i;  // Заносит вершину в очередь
        k++;        // Сдвигает указатель конца очереди
        Label[i]=Label[cur]+1;      // Помечает вершину
      }
  }
  for (i=0; i<N; i++)
    cout<<Label[i]<<" ";  // Выводит результат
  getch();                // Задерживает экран
  return 0;               // Завершает программу
}
```

Рассмотрим код программы поиска в ширину подробнее.

Начнем с представления графа в виде матрицы смежности. Так как граф задан списком ребер, который находится в файле, то будем последовательно читать из файла очередные две смежные вершины и помечать в матрице, что они соединены ребром:

```
for (k=0; k<M; k++)
{
```

```
  input>>i>>j;
  Graf[i][j]=1;
}
```

После этого определим положение указателей начала и конца очереди:

```
p=0;
k=1;
```

Затем занесем стартовую вершину обхода в очередь и пометим ее нулевой меткой:

```
FIFO[p]=Start;
Label[Start]=0;
```

Все промежуточные преобразования выполнены, теперь алгоритм будет работать до тех пор, пока очередь не станет пустой:

```
while (p!=k)
{
```

Выбираем первый элемент в очереди и сдвигаем указатель `p` на единицу вправо:

```
cur=FIFO[p];
p++;
```

Просматриваем строку матрицы с номером `cur`, т. е. с номером выбранной вершины. Если некоторый ее элемент не равен нулю, и вершина, номер которой соответствует номеру столбца ненулевого элемента, не помечена, то заносим ее в очередь и помечаем:

```
  for (i=0; i<N; i++)
    if (Graf[cur][i]!=0 && Label[i]>Label[cur]+1)
    {
      FIFO[k]=i;
      k++;
      Label[i]=Label[cur]+1;
    }
}
```

## Обход в глубину

Обход (поиск) в глубину, так же, как и обход в ширину, помечает вершины и ребра графа некоторым образом.

Рассмотрим алгоритм поиска на примере некоторого графа `R` (рис. 9.17). Так как алгоритм будет некоторым образом помечать вершины, то на первом шаге сделаем метки всех вершин нулевыми.

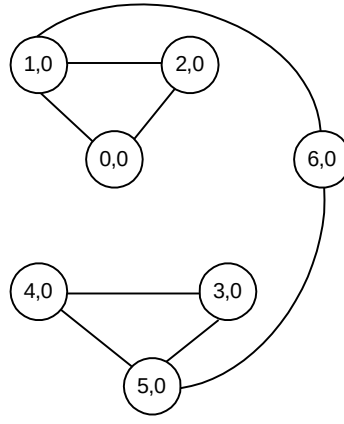



**Рис. 9.17.** Шаг 1 алгоритма поиска в глубину

Выберем в качестве стартовой вершину `1`. Так как с нее начат поиск, то она уже пройдена, и следовательно, получает метку `k`, причем начальное значение `k` равно единице (рис. 9.18). Такое начальное значение выбрано потому, что вершину `1` мы исследуем в первую очередь.

На третьем шаге выберем ребро, которое соединяет вершину `1` и непомеченную вершину с минимальным номером. В данном случае это будет вершина с номером `0`. Увеличиваем `k` на единицу и помечаем вершину `0` меткой `k`. При этом ребро `(1,0)` считается пройденным прямым ребром. Такие ребра будем обозначать толстой линией (рис. 9.19).

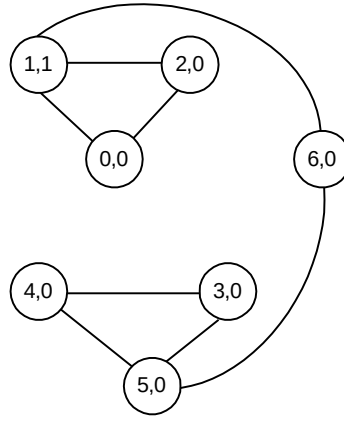



**Рис. 9.18.** Шаг 2 алгоритма поиска в глубину

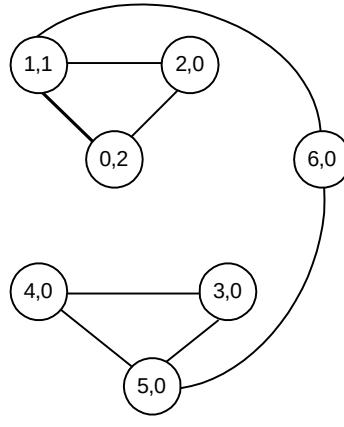



**Рис. 9.19.** Шаг 3 алгоритма поиска в глубину

Затем снова выбираем непомеченную вершину с минимальным номером и помечаем меткой `k` + 1, т. е. вершина `2` будет помечена меткой `3`, а ребро `(0,2)` будет отмечено как пройденное (рис. 9.20).

Продолжим поиск из вершины `2` (шаг 5). Из нее выходит одно непомеченное ребро — это ребро `(2,1)`, но вершина `1` уже помечена.

В данной ситуации необходимо пометить ребро `(2,1)` как обратное. Такие ребра буем помечать штриховой линией (рис. 9.21).

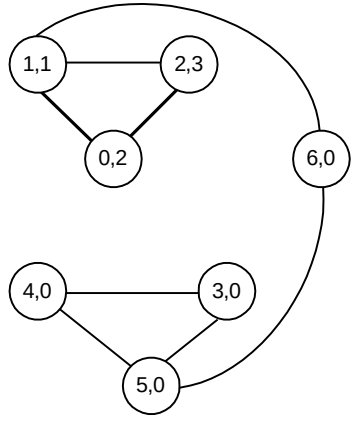



**Рис. 9.20.** Шаг 4 алгоритма поиска в глубину

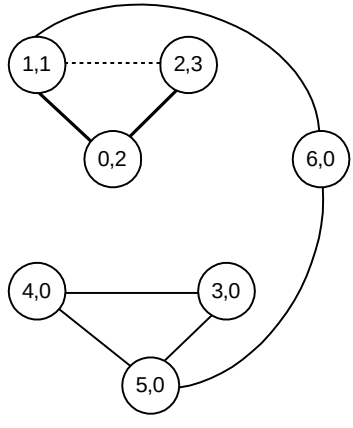



**Рис. 9.21.** Шаг 5 алгоритма поиска в глубину

Вернемся в вершину `0`, из которой мы "пришли" в вершину `2`, и продолжим поиск из нее. Из данной вершины не выходит ни одно непомеченное ребро, следовательно, возвращаемся в вершину, из которой мы "пришли" в `0`, — вершину `1`.

**Внимание**

При возвращении в вершину `i` ее метка не изменяется.

Продолжаем поиск из вершины `1` (шаг 6). Из нее выходит одно непомеченное ребро `(1,6)`, причем вершина `6` не помечена. Итак, ребро `(1,6)` становится помеченным прямым ребром, а вершина `6` получает метку `4` (рис. 9.22).

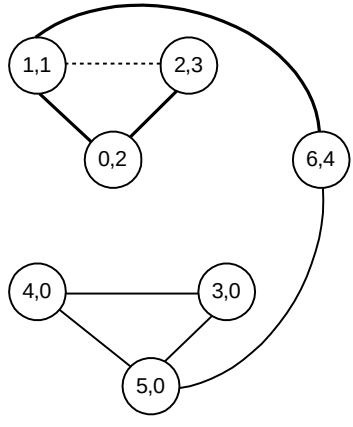



**Рис. 9.22.** Шаг 6 алгоритма поиска в глубину

Поиск продолжается из вершины `6` (шаг 7). Из данной вершины выходит единственное непомеченное ребро `(6,5)`, причем вершина `5` также не помечена. Следовательно, ребро `(6,5)` становится помеченным прямым ребром, а вершина `5` получает метку `5` (рис. 9.23).

Продолжим поиск из вершины `5` (шаг 8). Из данной вершины выходит два непомеченных ребра — это `(5,3)` и `(5,4)`. Ни одна из вершин `3` и `4` не помечена, следовательно, "идем" в вершину с минимальным номером. Таковой будет являться вершина `3`, значит, она получает метку `6`, а ребро `(5,3)` становится помеченным прямым ребром (рис. 9.24).

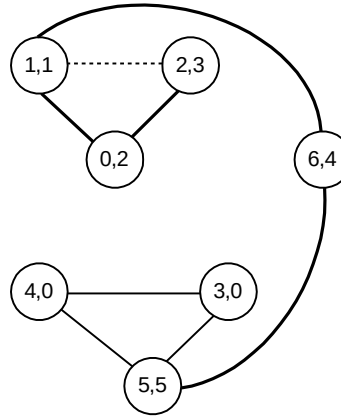



**Рис. 9.23.** Шаг 7 алгоритма поиска в глубину

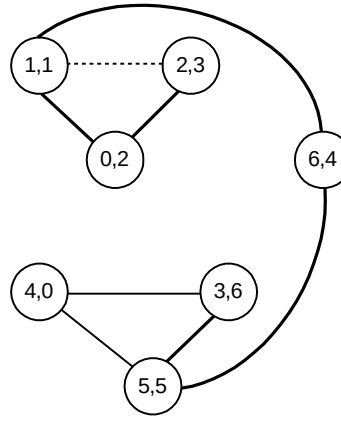



**Рис. 9.24.** Шаг 8 алгоритма поиска в глубину

Продолжим поиск из вершины `3` (шаг 9). Из данной вершины выходит одно непомеченное ребро `(3,4)`, причем вершина `4` также не помечена. Следовательно, ребро `(3,4)` становится помеченным прямым ребром, а вершина `4` получает метку `7` (рис. 9.25).

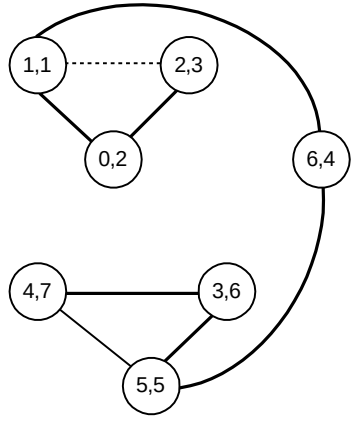



**Рис. 9.25.** Шаг 9 алгоритма поиска в глубину

Продолжим поиск из вершины `4` (шаг 10). Из данной вершины выходит одно непомеченное ребро `(4,5)`, но вершина `5` уже помечена. Следовательно, ребро `(4,5)` становится помеченным обратным ребром (рис. 9.26).

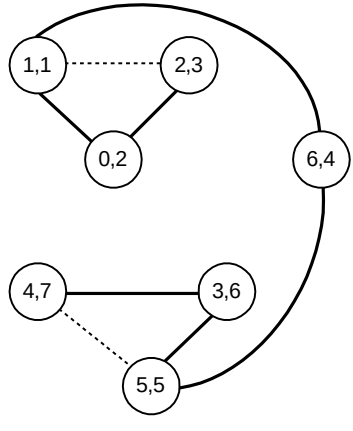



**Рис. 9.26.** Шаг 10 алгоритма поиска в глубину

Так как из вершины `4` больше не выходит ни одного непомеченного ребра, то алгоритм вернется в вершину `3`. Аналогичная ситуация и в этой вершине, поэтому приходится вернуться

в вершину 5. Из вершины 5 также не выходит ни одного непомеченного ребра. Двигаясь таким образом, алгоритм возвращается в вершину 1, из которой был начат обход, и завершает свою работу.

## Реализация поиска в глубину

Для реализации обхода или поиска в глубину зададим граф G с помощью матрицы смежности (рис. 9.27).

| | | Столбцы | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | **0** | **1** | **2** | **3** | **4** | **5** | **6** |
| | **0** | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| | **1** | 1 | 0 | 1 | 0 | 0 | 0 | 1 |
| | **2** | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| **Строки** | **3** | 0 | 0 | 0 | 0 | 1 | 1 | 0 |
| | **4** | 0 | 0 | 0 | 1 | 0 | 1 | 0 |
| | **5** | 0 | 0 | 0 | 1 | 1 | 0 | 1 |
| | **6** | 0 | 1 | 0 | 0 | 0 | 1 | 0 |

**Рис. 9.27.** Матрица смежности графа G

Для реализации обхода в глубину будем использовать стек LIFO, массив меток Label, а также будем вносить некоторые изменения в матрицу смежности.

На первом шаге необходимо создать стек и массив меток (рис. 9.28, табл. 9.9).


**Рис. 9.28.** Пустой стек на первом шаге алгоритма обхода в глубину

***Таблица 9.9.*** *Массив меток на первом шаге алгоритма обхода в глубину*

| **Индекс** | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| **Элементы** | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

**Примечание**

Символ **–** в ячейках стека обозначает, что они не заполнены.

Итак, ранее нами была выбрана стартовая вершина `1`. С нее и начнем поиск, поместив в стек и задав начальную метку `1`. Таким образом, стек и массив меток примут следующий вид (рис. 9.29, табл. 9.10).


**Рис. 9.29.** Стек на втором шаге алгоритма обхода в глубину

***Таблица 9.10.*** *Массив меток на втором шаге алгоритма обхода в глубину*

| **Индекс** | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| **Элементы** | 0 | 1 | 0 | 0 | 0 | 0 | 0 |

На третьем шаге выберем ребро, которое соединяет вершину `1` и непомеченную вершину с минимальным номером, т. е. найдем в строке `1` матрицы первый элемент, равный единице. В данном случае это будет элемент в нулевом столбце. Заносим вершину `0` в стек и помечаем меткой `2` (рис. 9.30, табл. 9.11).

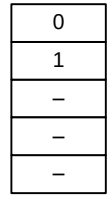



**Рис. 9.30.** Стек на третьем шаге алгоритма обхода в глубину

***Таблица 9.11.*** *Массив меток на третьем шаге алгоритма обхода в глубину*

| **Индекс** | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| **Элементы** | 2 | 1 | 0 | 0 | 0 | 0 | 0 |

При этом ребро `(1,0)` считается пройденным прямым ребром. Чтобы обозначить это, на пересечении первой строки матрицы и нулевого столбца заменим единицу двойкой. Таким образом, если `Graf[i][j]=2`, то ребро является прямым.

Продолжим поиск из вершины `0`. Просмотрим нулевую строку матрицы и найдем первый элемент, равный единице. Такой элемент будет находиться во втором столбце. Вершина `2` не помечена, следовательно, она заносится в стек и помечается меткой `3`, а в матрице смежности, на пересечении нулевой строки и второго столбца ставится `2`, это будет означать, что ребро пройдено и является прямым. При этом стек и массив меток будут выглядеть следующим образом (рис. 9.31, табл. 9.12).

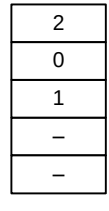



**Рис. 9.31.** Стек на четвертом шаге алгоритма обхода в глубину

***Таблица 9.12.*** *Массив меток на четвертом шаге алгоритма обхода в глубину*

| **Индекс** | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| **Элементы** | 2 | 1 | 3 | 0 | 0 | 0 | 0 |

Затем просмотрим вторую строку матрицы. Элемент `Graf[2][1]=1`, т. е. существует непомеченное ребро, соединяющее вершины `2` и `1`, но вершина `1` уже помечена, следовательно, необходимо удалить вершину `2` из стека, переместив указатель вершины стека на единицу вниз (рис. 9.32).

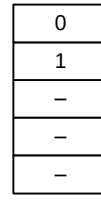



**Рис. 9.32.** Стек после возвращения в вершину `0`

В этом случае ребро `(0,2)` оказывается обратным. Если ребро, соединяющее вершины `i` и `j`, обратное, то в матрице смежности на пересечении `i`-ой строки и `j`-го столбца устанавливается метка `3`.

Таким образом, продолжаем поиск из вершины `0`. Но из вершины `0` так же не выходит ни одного непомеченного ребра, следовательно, возвращаемся в вершину `1`. На данном шаге стек освободится от еще одного элемента (рис. 9.33).

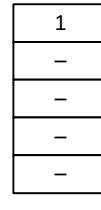



**Рис. 9.33.** Стек при возвращении в вершину `1` после пятого шага алгоритма

Просмотрим первую строку матрицы. Элемент `Graf[1][6]` равен единице, и вершина `6` не помечена. Следовательно, вершина `6` заносится в стек и помечается меткой `4`, а ребро `(1,6)` помечается как прямое: `Graf[1][6]=2` (рис. 9.34, табл. 9.13).

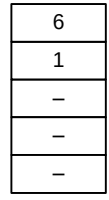



**Рис. 9.34.** Стек на шестом шаге алгоритма обхода в глубину

***Таблица 9.13.*** *Массив меток на шестом шаге алгоритма обхода в глубину*

| **Индекс** | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| **Элементы** | 2 | 1 | 3 | 0 | 0 | 0 | 4 |

В шестой строке матрицы элемент в пятом столбце равен единице, и вершина `5` не помечена. Добавляем вершину `5` в стек и помечаем меткой `5`, а ребро `(6,5)` помечаем как прямое: `Graf[6][5]=2`. Это седьмой шаг алгоритма (рис. 9.25, табл. 9.14).

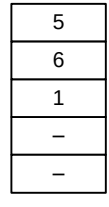



**Рис. 9.35.** Стек на седьмом шаге алгоритма обхода в глубину

***Таблица 9.14.*** *Массив меток на седьмом шаге алгоритма обхода в глубину*

| **Индекс** | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| **Элементы** | 2 | 1 | 3 | 0 | 0 | 5 | 4 |

На следующем шаге аналогичным образом будет занесена в стек вершина `3`. Она получит метку `6`, а ребро `(5,3)` станет прямым: `Graf[5][3]=2` (рис. 9.36, табл. 9.15).

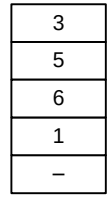



**Рис. 9.36.** Стек на восьмом шаге алгоритма обхода в глубину

***Таблица 9.15.*** *Массив меток на восьмом шаге алгоритма обхода в глубину*

| **Индекс** | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| **Элементы** | 2 | 1 | 3 | 6 | 0 | 5 | 4 |

Далее занесем в стек вершину `4`, предварительно определив, что `Graf[3][4]=1`. Вершина получит метку `7`, а ребро `(3,4)` станет прямым: `Graf[3][4]=2` (рис. 9.37, табл. 9.16).

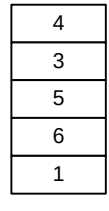



**Рис. 9.37.** Стек на девятом шаге алгоритма обхода в глубину

***Таблица 9.16.*** *Массив меток на девятом шаге алгоритма обхода в глубину*

| **Индекс** | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| **Элементы** | 2 | 1 | 3 | 6 | 7 | 5 | 4 |

Просмотрев четвертую строку матрицы, найдем еще один элемент, равный единице, он будет находиться в пятом столбце, но вершина `5` уже помечена, следовательно, ребро `(4,5)` становится обратным: `Graf[4][5]=3`. Это десятый шаг алгоритма

Таким образом, вершина `4` будет удалена из стека и поиск продолжится из вершины `3`. Затем и она будет удалена из стека (рис. 9.38).

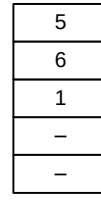



**Рис. 9.38.** Стек после возвращения в вершину `3`

Из вершин `5`, `6` и `1` так же не выходит ни одного непомеченного ребра, следовательно, и они удаляются из стека (рис. 9.39). Массив меток будет выглядеть так же, как и на девятом шаге.

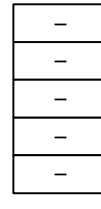



**Рис. 9.39.** Пустой стек после окончания работы алгоритма обхода в глубину

Итак, стек становится пустым, а последняя вершина, в которую мы вернулись, являлась стартовой, это означает, что поиск в глубину окончен, и вершины и ребра получили свои пометки.

Окончательный вид матрицы смежности такой, как представлено на рис. 9.40.

| | | **Столбцы** | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | **0** | **1** | **2** | **3** | **4** | **5** | **6** |
| | **0** | 0 | 2 | 2 | 0 | 0 | 0 | 0 |
| | **1** | 2 | 0 | 3 | 0 | 0 | 0 | 2 |
| | **2** | 2 | 2 | 0 | 0 | 0 | 0 | 0 |
| **Строки** | **3** | 0 | 0 | 0 | 0 | 2 | 2 | 0 |
| | **4** | 0 | 0 | 0 | 2 | 0 | 3 | 0 |
| | **5** | 0 | 0 | 0 | 2 | 2 | 0 | 2 |
| | **6** | 0 | 2 | 0 | 0 | 0 | 2 | 0 |

**Рис. 9.40.** Матрица смежности графа `G` по окончанию работы алгоритма поиска в глубину

Если `Graf[i][j]=2`, то ребро пройдено и является прямым.

Если `Graf[i][j]=3`, то ребро пройдено и является обратным.

Рассмотрим код программы, которая реализует алгоритм поиска в глубину.

```
#include <iostream.h>
#include <conio.h>
#include <fstream.h>
main()
{
  int N, M, Start, m, p, i, j, k;
  int *LIFO;   // Объявляет структуру данных "стек"
  int **Graf;  // Объявляет матрицу смежности графа
  int *Label;  // Объявляет массив меток
  ifstream input("Graf.txt");  // Открывает файл для чтения
  // Читает из файла N — количество вершин в графе,
```

```
//                  M — количество ребер в графе,
//                  Start — стартовая вершина обхода
input>>N>>M>>Start;
LIFO=new int [N];     // Выделяет динамическую память для
                      // стека, состоящего из N элементов
// Выделяет динамическую память для матрицы N*N
Graf=new int *[N];
for (i=0; i<N; i++)
  Graf[i]=new int[N];
Label=new int [N];    // Выделяет динамическую память
                      // для массива меток, состоящего
                      // из N элементов
for (k=0; k<M; k++)
{
  input>>i>>j;    // Считывает из файла очередные смежные
                  // вершины
  Graf[i][j]=1;   // Определяет в матрице, что вершины i и j
                  // соединены ребром
}
LIFO[0]=Start;    // Заносит в стек стартовую вершину
Label[Start]=1;   // Присваивает стартовой вершине метку 1
k=1;              // Позиция вершины стека
m=1;              // Начальное значение метки
while (k!=0)
{
  p=0;
  // Находит вершину с минимальным номером, смежную с
  // вершиной, номер которой находится в стеке в позиции k
  for (i=0; i<N; i++)
    if (Graf[LIFO[k-1]][i]==1)
    {
      p=1;
      break;
    }
  if (p!=0)
```

```
    {
      if (Label[i]==0)
      {
        LIFO[k]=i;   // Заносит вершину в стек
        m++;         // Увеличивает значение метки
        Label[i]=m;  // Помечает вершину
        // Помечает ребро как прямое
        Graf[LIFO[k-1]][i]=2;
        Graf[i][LIFO[k-1]]=2;
        k++;         // Сдвигает позицию вершины стека
                     // на единицу вверх
      } else
      {
        // Помечает ребро как обратное
        Graf[i][LIFO[k-1]]=3;
        Graf[LIFO[k-1]][i]=3;
      }
    } else k--;      // Сдвигает позицию вершины стека
                     // на единицу вниз, тем самым удаляя
                     // вершину из стека
  }
  // Выводит результат
  for (i=0; i<N; i++)
    cout<<Label[i]<<" ";
  getch();   // Задерживает экран
  return 0;  // Завершает программу
}
```

Рассмотрим код данной программы подробнее.

На первом шаге программа должна представить граф в виде матрицы смежности:

```
for (k=0; k<M; k++)
{
  input>>i>>j;
  Graf[i][j]=1;
}
```

Затем необходимо занести в стек вершину, с которой начнется обход, и пометить ее меткой `m`, причем начальное значение `m=1`:

```
LIFO[0]=Start;
Label[Start]=1;
m=1;
```

Установим указатель вершины стека на первый элемент, т. к. в нулевой ячейке находится стартовая вершина:

```
k=1;
```

Так как алгоритм продолжает свою работу до тех пор, пока стек не станет пустым, запишем условие

```
while (k!=0)
{
```

Теперь необходимо найти смежную непомеченную вершину с минимальным номером либо убедиться, что такой не существует. Будем действовать от противного — предположим, что такой вершины не существует:

```
p=0;
```

Просматриваем соответствующую строку матрицы. Если некоторый ее элемент равен единице, то признак `p` становится равным 1, а оператор `break` завершает работу цикла:

```
for (i=0; i<N; i++)
  if (Graf[LIFO[k-1]][i]==1)
  {
    p=1;
    break;
  }
```

Итак, если существует такая вершина, которая еще не помечена, то заносим ее в стек и помечаем:

```
if (p!=0)
{
  if (Label[i]==0)
  {
```

```
    LIFO[k]=i;
    m++;
    Label[i]=m;
```

После того как вершина стала помеченной, необходимо пометить ребро как пройденное и прямое:

```
Graf[LIFO[k-1]][i]=2;
Graf[i][LIFO[k-1]]=2;
```

Сдвигаем позицию вершины стека на единицу вверх:

```
k++;
```

Если же оказалось, что ребро не пройдено, а вершина помечена, то необходимо пометить, что ребро является обратным:

```
} else
  {
    Graf[i][LIFO[k-1]]=3;
    Graf[LIFO[k-1]][i]=3;
  }
```

Если же из вершины вообще не выходит ни одного непомеченного ребра, то номер вершины графа, которая находится в вершине стека, удаляется из него.

```
  } else k--;
}
```

После того как алгоритм завершит свою работу и вершины получат свою метку, программа приступит к выводу результата:

```
  for (i=0; i<N; i++)
    cout<<Label[i]<<" ";
  getch();
  return 0;
}
```

# 9.3. Примеры графов

Граф, несмотря на свою достаточно простую структуру, имеет много разновидностей и свойств. Для более четкой ориентации в теории графов необходимо рассмотреть некоторые дополнитель-

ные понятия, на основе которых будет легче делать предположения и разрабатывать алгоритмы.

## Связные и несвязные графы

Граф называется *связным* тогда и только тогда, когда существует путь любой длины из любой вершины `i` в любую вершину `j`.

Примером *несвязного* графа может служить граф с изолированной вершиной (рис. 9.41).

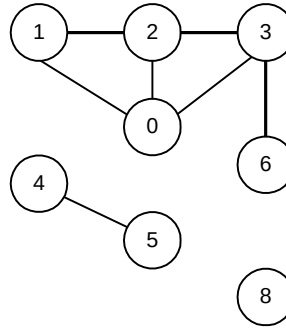



**Рис. 9.41.** Пример несвязного графа

На приведенном рисунке не существует пути из вершин `0`, `1`, `2`, `3` и `6` в вершины `4`, `5` и `8` и обратно. Несвязным является также граф, приведенный на рис. 9.16.

Необходимо отметить, что графы, которые использовались в описаниях алгоритмов обхода, являлись связными.

Чтобы определить, является ли граф связным, можно использовать любой алгоритм обхода графа, т. к. обходы в ширину и в глубину предполагают процедуру прохождения и пометки всех вершин графа. Таким образом, после выполнения любого обхода достаточно проверить, все ли вершины помечены. Если это выполняется, то граф является связным, в противном случае — граф несвязный.

Рассмотрим программу, которая будет проверять, является ли граф связным. Для проверки будем использовать любой из обходов графа, например, поиск в ширину.

Далее приведен код данной программы:

```
#include <iostream.h>
#include <conio.h>
#include <fstream.h>
int i, j, k, p, cur;
int Start, N, M;
main()
{
  ifstream input ("input.txt");  // Открывает файл
                                 // для чтения данных
  // Читает из файла N — количество вершин в графе,
  //                  M — количество ребер в графе,
  //              Start — стартовая вершина обхода
  input>>N>>M>>Start;
  int *FIFO;   // Объявляет структуру данных "очередь"
  int **Graf;  // Объявляет матрицу смежности графа
  int *Label;  // Объявляет массив меток
  input>>N>>M>>Start;
  FIFO=new int [N];   // Выделяет динамическую память для
                      // стека, состоящего из N элементов
  // Выделяет динамическую память для матрицы N*N
  Graf=new int *[N];
  for (i=0; i<N; i++)
    Graf[i]=new int[N];
  Label=new int [N];  // Выделяет динамическую память для
                      // массива меток, состоящего из N элементов
  for (k=0; k<M; k++)
  {
    input>>i>>j;      // Читает номера смежных вершин
    Graf[i][j]=1;     // Задает матрицу смежности графа
  }
  for (i=0; i<M; i++)
  {
```

```
    FIFO[i]=0;
    Label[i]=32767;
  }
  p=0;              // Указатель на начало очереди
  k=1;              // Указатель на конец очереди
  FIFO[p]=Start;    // Заносим стартовую вершину в очередь
  Label[Start]=0;   // и помечаем ее меткой 0
  while (p!=k)
  {
    cur=FIFO[p];    // Выделяет очередную вершину из очереди
    p++;            // Сдвигает указатель начала очереди на единицу
    for (i=0; i<N; i++)
      if (Graf[cur][i]==1 && Label[i]>Label[cur]+1)
      {
        FIFO[k]=i; // Заносит вершину в очередь
        k++;       // Сдвигает указатель конца очереди
        Label[i]=Label[cur]+1;     // Помечает вершину
      }
  }
  p=0;  // Пусть граф связный
  for (i=0; i<N; i++)
    if (Label[i]==32767)
    {
      p=1;
      break;
    }
  // Выводит результат
  if (p==0) cout<<"Граф — связный!";
  else cout<<"Граф — несвязный!";
  getch();      // Задерживает экран
  return 0;     // Завершает программу
}
```

**Примечание**

Алгоритм поиска в ширину уже объяснялся ранее, именно поэтому не будем рассматривать его в очередной раз.

Предположим, что граф связный и все вершины помечены:

```
p=0;
```

Далее будем последовательно просматривать метку каждой вершины графа, и если некоторая из них равна начальному значению, то признаку `p` присваивается единица, а оператор `break` завершает работу цикла:

```
for (i=0; i<N; i++)
  if (Label[i]==32767)
  {
    p=1;
    break;
  }
```

Проверка выполнена, следовательно, можно выводить результат в зависимости от `p`:

```
  if (p==0) cout<<"Граф — связный!";
  else cout<<"Граф — несвязный!";
  getch();
  return 0;
}
```

## Регулярные графы

Граф называется *регулярным (однородным)*, если степени всех его вершин равны между собой (рис. 9.42).

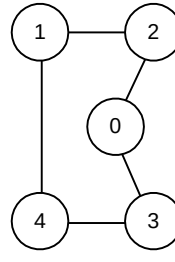



**Рис. 9.42.** Пример регулярного графа

На рисунке видно, что степень каждой вершины графа равна двум, т. к. каждая вершина графа соединена ребрами только с двумя другими.

*Степенью* регулярного графа называется степень его вершин. Из данного определения следует, что степень графа на рис. 9.42 равна двум.

Чтобы определить, является ли граф регулярным, необходимо проверить каждую строку матрицы смежности, которой задан граф. Если количество ненулевых ячеек каждой строки совпадает, то граф является регулярным, в противном случае утверждение не верно.

Рассмотрим программу, которая определяет, является ли граф регулярным.

```
#include <iostream.h>
#include <fstream.h>
#include <conio.h>
int Regular(int);    // Прототип функции
int **Graf;
main()
{
  int i, j, k, N, R;
  ifstream input("input.txt");   // Открывает файл для чтения
  input>>N>>R;      // Читает из файла количество вершин графа
                    // и количество ребер соответственно
  // Выделяет динамическую память для матрицы смежности графа
  Graf=new int *[N];
  for (i=0; i<N; i++)
    Graf[i]=new int [N];
  // Обнуляет все элементы матрицы
  for (i=0; i<N; i++)
    for (j=0; j<N; j++)
      Graf[i][j]=0;
  for (k=0; k<R; k++)
  {
    input>>i>>j;   // Читает из файла номера очередных
```

```
                    // вершин,которые соединены ребром
    Graf[i][j]=1;  // Помечает, что вершины i и j
                    // являются смежными (заполняет
                    // матрицу смежности графа)
  }
  // Выводит результат в зависимости от возвращенного
  // функцией Regular(int) значения
  if (Regular(N)==0) cout<<"Граф не является регулярным!";
  else cout<<"Граф регулярный!";
  getch();   // Задерживает экран
  return 0;  // Завершает программу
}
int Regular(int N)        // Описание функции
{
  int Temp=0, Kol=0, i, j;
  for (j=0; j<N; j++)     // Определяет степень нулевой вершины
    if (Graf[0][j]!=0) Kol++;
  for (i=1; i<N; i++)
  {
    // Определяет степень каждой последующей вершины.
    // Если степень некоторой из них не равна степени
    // нулевой вершины, то возвращает 0
    for (j=0; j<N; j++)
      if (Graf[i][j]!=0) Temp++;
    if (Temp!=Kol) return 0;
    Temp=0;
  }
  return 1;  // Если степени всех вершин равны, то возвращает 1
}
```

## Двудольные графы

Граф называется *двудольным* тогда и только тогда, когда множество его вершин можно разбить на две доли (части) таким образом, что каждое ребро будет соединять вершины различных долей (рис. 9.43). Чтобы граф был двудольным, необходимо и доста-

точно, чтобы он не содержал циклов нечетной длины. Поскольку цикл — это маршрут, у которого стартовая и конечная вершины совпадают, то будем называть граф двудольным тогда и только тогда, когда число ребер, которые входят в тот или иной цикл графа, является числом четным. Данное условие должно выполняться для всех циклов графа.

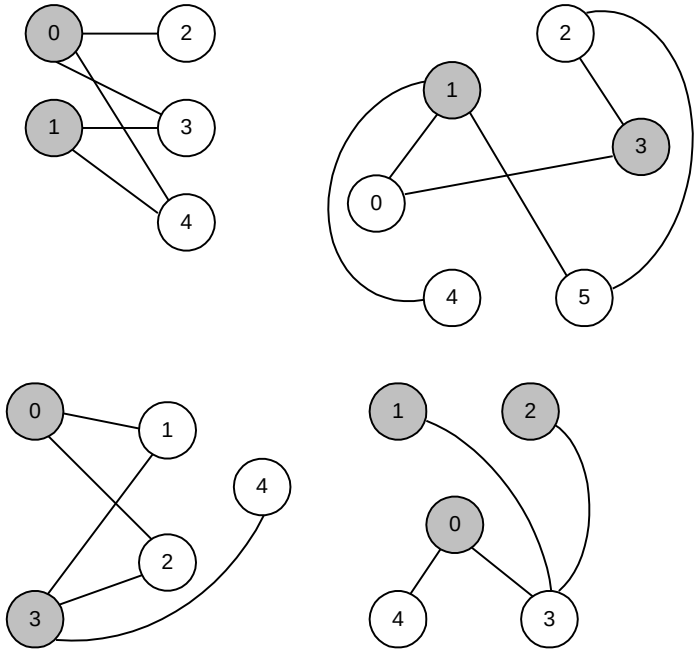



**Рис. 9.43.** Примеры двудольных графов

Если каждая вершина одной доли соединена с каждой вершиной другой доли, то такой граф будем называть *полным двудольным* (рис. 9.44). На рис. 9.43 вершины, закрашенные серым цветом, принадлежат одной доле, а остальные, соответственно, другой.

Далее представлен алгоритм определения, является ли граф двудольным.

1. Произведем поиск в ширину из любой стартовой вершины.

2. К доле A отнесем все вершины с нечетной меткой.
3. К доле B отнесем все вершины с четной меткой.
4. Если некоторые вершины из доли A смежные, то граф не является двудольным.
5. Если некоторые вершины из доли B смежные, то граф не является двудольным.
6. Если пункты 3 и 4 не выполняются, т. е. нет такой пары смежных вершин из доли A и нет ни одной пары смежных вершин из доли B, то граф является двудольным.

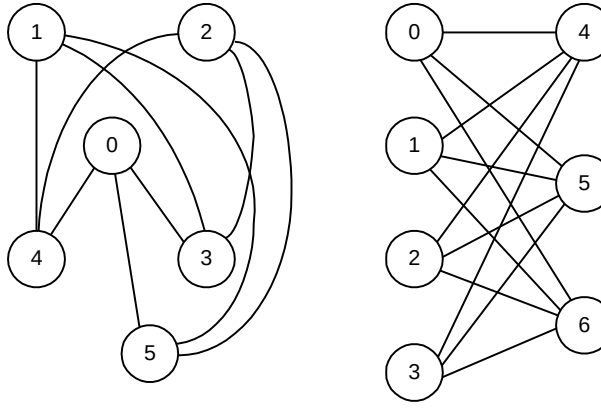



**Рис. 9.44.** Примеры полных двудольных графов

Рассмотрим код программы, реализующей данный алгоритм на C++:

```
#include <iostream.h>
#include <conio.h>
#include <fstream.h>
#include <math.h>
int i, j, k, p, cur;
int Start, N, M;
main()
```

```
{
  ifstream input ("input.txt");   // Открывает файл для
                                  // чтения данных
  // Читает из файла N — количество вершин в графе,
  //                 M — количество ребер в графе
  input>>N>>M;
  int *LIFO;          // Объявляет структуру данных "стек"
  int **Graf;         // Объявляет матрицу смежности графа
  int *Label;         // Объявляет массив меток
  input>>N>>M>>Start;
  LIFO=new int [N];   // Выделяет динамическую память для
                      // стека, состоящего из N элементов
  // Выделяет динамическую память для матрицы N*N
  Graf=new int *[N];
  for (i=0; i<N; i++)
    Graf[i]=new int[N];
  Label=new int [N];  // Выделяет динамическую память для
                      // массива меток, состоящего из N элементов
  Start=0;
  for (k=0; k<M; k++)
  {
    input>>i>>j;      // Читает номера смежных вершин
    Graf[i][j]=1;     // Заполняет матрицу смежности
  }
  for (i=0; i<M; i++)
  {
    FIFO[i]=0;
    Label[i]=32767;   // Заполним массив меток числами,
                      // большими, чем N
  }
  p=0;                // Указатель на начало очереди
  k=1;                // Указатель на конец очереди
  FIFO[p]=Start;      // Заносим стартовую вершину в очередь
  Label[Start]=0;     // и помечаем ее меткой 0
  while (p!=k)
  {
```

```
    cur=FIFO[p];       // Выделяет очередную вершину из очереди
    p++;               // Сдвигает указатель начала на единицу
    for (i=0; i<N; i++)
      if (Graf[cur][i]==1 && Label[i]>Label[cur]+1)
      {
        FIFO[k]=i;     // Заносит вершину в очередь
        k++;           // Сдвигает указатель конца очереди
        Label[i]=Label[cur]+1;        // Помечает вершину
      }
  }
  for (i=0; i<N-1; i++)
    for (j=i+1; j<N; j++)
      if (labs(Label[i]-Label[j])%2==0 && Graf[i][j]!=0)
      {
        cout<<"Исходный граф не является двудольным";
        getch();       // Задерживает экран
        return 0;      // Завершает программу
      }
  cout<<"Исходный граф является двудольным";
  getch();             // Задерживает экран
  return 0;            // Завершает программу
}
```

Согласно описанному выше алгоритму, на первом шаге запускаем процедуру поиска в ширину, в качестве начальной вершины возьмем вершину `0`. Не будем останавливаться на поиске в ширину, т. к. его алгоритм был рассмотрен ранее. Перейдем к финальной части программы, т. е. проверим, существует ли такая пара смежных вершин, метки у которых либо четные, либо нечетные одновременно. Для данной проверки необходимо организовать два цикла повторения:

```
for (i=0; i<N-1; i++)
  for (j=i+1; j<N; j++)
```

Если окажется, что разность меток очередной пары без остатка делится на 2, то граф не двудольный, и алгоритм перейдет к вы-

воду соответствующего ответа, после чего программа будет завершена.

```
if (labs(Label[i]-Label[j])%2==0 && Graf[i][j]!=0)
{
  cout<<"Исходный граф не является двудольным";
  getch();
  return 0;
}
```

Если программа не была завершена ранее, т. е. граф — двудольный, то:

```
  cout<<"Исходный граф является двудольным";
  getch();
  return 0;
}
```

## Пустые и полные графы

Граф называется *полным* тогда и только тогда, когда любая пара его вершин соединена ребром. Такой граф имеет $N \times (N - 1)/2$ ребер, где $N$ — это количество вершин в графе (рис. 9.45).

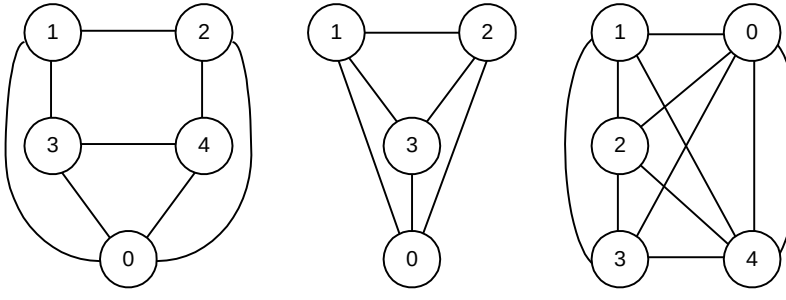



**Рис. 9.45.** Примеры полных графов

Если граф задан матрицей смежности, то, чтобы проверить, является ли граф полным, достаточно убедиться, что количество ненулевых элементов в матрице равно $N \times (N - 1)$, т. е. нули расположены только на диагонали матрицы.

Далее приведена часть кода программы, которая проверяет, является ли граф полным.

```
  for (i=0; i<N; i++)
    for (j=0; j<N; j++)
      if (Graf[i][j]!=0) K++;
  if (K==(N*(N-1)/2)) cout<<"Граф является полным";
  else cout<<"Граф не является полным";
  getch();              // Задерживает экран
  return 0;             // Завершает программу
}
```

Если ребра у графа отсутствуют, он называется *пустым* (рис. 9.46).

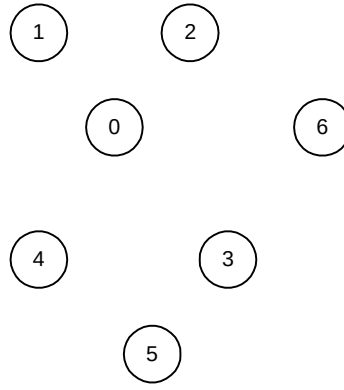



**Рис. 9.46.** Пример пустого графа

Если граф задан матрицей смежности, то, чтобы проверить, является ли граф пустым, достаточно определить, равны ли нулю все элементы матрицы.

Далее приведена часть кода программы, которая проверяет, является ли граф пустым.

```
  for (i=0; i<N; i++)
    for (j=0; j<N; j++)
      if (Graf[i][j]==1)
      {
```

```
        cout<<"Граф не является пустым";
        getch();       // Задерживает экран
        return 0;      // Завершает программу
      }
  cout<<"Граф является пустым";
  getch();             // Задерживает экран
  return 0;            // Завершает программу
}
```

# 9.4. Эйлеровы графы

Связный граф является *эйлеровым* тогда и только тогда, когда степени всех его вершин четны (рис. 9.47).

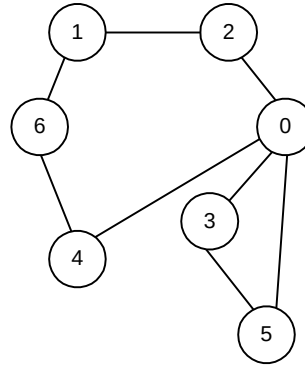



**Рис. 9.47.** Пример эйлерова графа

Степени вершин `1`, `2`, `3`, `4`, `5` и `6` равны двум, а степень вершины `0` равна четырем, поэтому данный граф является эйлеровым.

Цикл в графе называется *эйлеровым циклом* тогда и только тогда, когда он содержит все вершины данного графа. Эйлеров цикл — это цикл, который проходит ровно один раз по каждому ребру.

Не все графы имеют эйлеровы циклы, но если эйлеров цикл существует, то это означает, что, следуя вдоль этого цикла, можно нарисовать граф на бумаге, не отрывая от нее карандаша.

Связный граф называется *эйлеровым*, если он содержит *эйлеров цикл*.

Связный граф содержит *эйлерову цепь* тогда и только тогда, когда он имеет не более двух вершин нечетной степени.

Чтобы определить, является ли граф эйлеровым, достаточно представить его в виде матрицы смежности и подсчитать количество единиц в каждой ее строке. Если в какой-либо строке находится нечетное количество единиц, то степень этой вершины нечетная и, следовательно, граф не является эйлеровым.

Далее приведен код программы, выполняющей данную проверку:

```
#include <iostream.h>
#include <conio.h>
#include <fstream.h>
int i, j, k, N, M, one;
int **Graf;    // Объявляется матрица смежности графа
main()
{
  ifstream input ("input.txt");  // Открывает файл для чтения
  input>>N>>M;           // Читает из файла количество вершин
                         // и ребер в графе соответственно
  // Выделяет динамическую память для матрицы смежности графа
  Graf=new int *[N];
  for (i=0; i<N; i++)
    Graf[i]=new int [N];
  // Обнуляет все элементы матрицы
  for (i=0; i<N; i++)
    for (j=0; j<N; j++)
      Graf[i][j]=0;
  for (k=0; k<M; k++)
  {
    input>>i>>j;       // Читает номера смежных вершин
    Graf[i][j]=1;      // Заполняет матрицу смежности
  }
  for (i=0; i<N; i++)
```

```
  {
    for (j=0; j<N; j++)
      if (Graf[i][j]==1) one++;
    // Если количество единиц в строке матрицы нечетное,
    // то выводится результат и программа завершается
    if (one%2==1)
    {
      cout<<"Граф не является эйлеровым";
      getch();          // Задерживает экран
      return 0;         // Завершает программу
    }
  }
  cout<<"Граф является эйлеровым";
  getch();              // Задерживает экран
  return 0;             // Завершает программу
}
```

## Алгоритм и реализация построения эйлерова цикла

Пусть задан граф `G`, являющийся эйлеровым. Требуется найти эйлеров цикл. В данной задаче используется алгоритм поиска в глубину.

Мы уже строили эйлеров цикл, не подозревая об этом, когда помечали вершины и ребра, выполняя процедуру поиска.

Ребро `(i,j)` мы помечали как обратное, если не было ни одного непомеченного ребра, выходящего из вершины `j`. Если такие ребра записывать по мере нахождения в некоторый стек `L`, то по завершению обхода в глубину мы получим тот самый эйлеров цикл.

Итак, модифицируем код поиска в глубину.

```
#include <iostream.h>
#include <conio.h>
#include <fstream.h>
main()
```

```
{
  int N, M, Start, pos, p, i, j, k, LIFO[100];
  int C[2][100], Graf[100][100];
  ifstream input("Graf.txt");  // Открывает файл для чтения
  // Читает из файла N — количество вершин в графе,
  //                  M — количество ребер в графе
  input>>N>>M;
  Start=0;
  for (k=0; k<M; k++)
  {
    input>>i>>j;   // Считывает из файла очередные смежные вершины
    Graf[i][j]=1;  // Задает в матрице признак того,
                   // что вершины i и j соединены ребром
  }
  LIFO[0]=Start;   // Заносит в стек стартовую вершину
  k=1;             // Позиция вершины стека
  while (k!=0)
  {
    p=0;
    // Находит вершину с минимальным номером, смежную с
    // вершиной, номер которой находится в стеке в позиции k
    for (i=0; i<N; i++)
      if (Graf[LIFO[k-1]][i]==1)
      {
        p=1;
        break;
      }
    if (p!=0)
    {
      LIFO[k]=i;   // Заносит вершину в стек
      // Помечает ребро как прямое
      Graf[LIFO[k-1]][i]=2;
      Graf[i][LIFO[k-1]]=2;
      k++;         // Сдвигает позицию вершины стека
```

```
                    // на единицу вверх
    } else
    {
      // Заносит в стек обратное ребро
      C[0][pos]= LIFO[k-1];
      C[1][pos]= LIFO[k-2];
      pos++;        // Сдвигает на единицу вверх вершину стека C
      k--;          // Сдвигает на единицу вниз вершину стека LIFO
    }
  }
  // Выводит результат
  for (i=0; i<pos-1; i++)
    cout<<C[0][i]<<" "<<C[1][i]<<endl;
  getch();          // Задерживает экран
  return 0;         // Завершает программу
}
```

**Примечание**

В данном алгоритме мы не помечаем вершины, а помечаем только ребра.

# 9.5. Построение кратчайших путей

Задачи на построение кратчайших путей в графе довольно часто используются в бытовых ситуациях. Например, известно, что на карте расположены города, причем некоторые из них соединены дорогами. Водитель едет из города `A` в город `B`. Требуется определить кратчайший маршрут поездки.

Для решения данной задачи необходимо ввести понятие взвешенного графа. Граф называется *взвешенным*, ели каждому его ребру присвоено некоторое число, называемое *длиной ребра* (рис. 9.48).

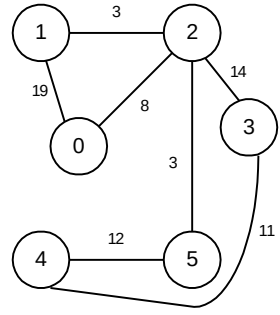



**Рис. 9.48.** Взвешенный граф `G`

Карта представляется в виде графа следующим образом:

- каждый город на карте изображен в виде вершины графа;
- каждая дорога между городами `i` и `j` изображена ребром графа, соединяющим вершины `i` и `j`.

*Кратчайшим путем* между вершинами назовем путь минимального веса.

Взвешенный граф хранится почти так же, как и невзвешенный. Например, если задать для графа матрицу смежности, то на пересечении `i`-ой строки и `j`-го столбца будет помещен вес ребра `(i,j)`. Матрица смежности графа `G` (рис. 9.48) будет выглядеть так, как представлено на рис. 9.49.

| | | **Столбцы** | | | | | |
|---|---|---|---|---|---|---|---|
| | | **0** | **1** | **2** | **3** | **4** | **5** |
| | **0** | 0 | 19 | 8 | 0 | 0 | 0 |
| | **1** | 19 | 0 | 3 | 0 | 0 | 0 |
| | **2** | 8 | 3 | 0 | 14 | 0 | 3 |
| **Строки** | **3** | 0 | 0 | 14 | 0 | 11 | 0 |
| | **4** | 0 | 0 | 0 | 11 | 0 | 12 |
| | **5** | 0 | 0 | 3 | 0 | 12 | 0 |

**Рис. 9.49.** Матрица смежности взвешенного графа

# Алгоритм Дейкстры

Алгоритм Дейкстры использует те же приемы, что и поиск в ширину. При обходе некоторого графа при помощи поиска в ширину мы помечали вершины таким образом, что после завершения работы алгоритма получали длину пути из `i` в `j`. При этом мы считали, что вес каждого ребра равен единице. Так как в данном случае граф взвешенный, т. е. каждое его ребро пропорционально некоторому числу (весу), то будем помечать каждую вершину таким образом, чтобы после завершения работы алгоритма получить кратчайший путь из `i` в `j`.

Рассмотрим действие данного алгоритма на взвешенном графе `W` (рис. 9.50).

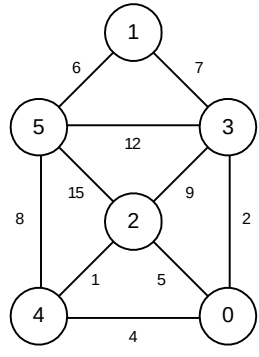



**Рис. 9.50.** Взвешенный граф `W`

Допустим, надо найти длину кратчайшего пути из вершины `0` в вершину `1`.

Для данного алгоритма необходимо использование двух массивов — `Label` и `Active`. В каждой ячейке `Label[i]` будем хранить текущее расстояние от стартовой вершины `Start` до вершины `i`, в массиве `Active` будем отмечать, какие из вершин пройдены.

Так как вершина `0` является стартовой, то отметим ее в массиве `Active` как пройденную, а в массиве меток присвоим ей метку `0` (табл. 9.17, 9.18).

***Таблица 9.17.*** *Массив* `Active` *на первом шаге алгоритма Дейкстры*

| **Индекс** | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| **Элементы** | 1 | 0 | 0 | 0 | 0 | 0 |

***Таблица 9.18.*** *Массив меток* `Label` *на первом шаге алгоритма Дейкстры*

| **Индекс** | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| **Элементы** | 0 | 32 767 | 32 767 | 32 767 | 32 767 | 32 767 |

Из вершины `0` выходят три ребра: `(0,2)`, `(0,3)` и `(0,4)`. Если для некоторой из найденных вершин справедливо, что `Label[j] > > Label[i] + Graf[i][j]`, то пометим ее как пройденную и заполним ячейки массива меток для каждой такой вершины следующим образом:

```
Label[j]=Label[i]+Graf[i][j];
```

В нашем случае имеем:

- `Label[2] > Label[0] + Graf[0][2]`, т. е. 32 767 > 0 + 5. Следовательно, `Label[2] = Label[0] + Graf[0][2] = 5`;
- `Label[3] > Label[0] + Graf[0][3]`, т. е. 32 767 > 0 + 2. Следовательно, `Label[3] = Label[0] + Graf[0][3] = 2`;
- `Label[4] > Label[0] + Graf[0][4]`, т. е. 32 767 > 0 + 4. Следовательно, `Label[4] = Label[0] + Graf[0][4] = 4`.

Таким образом, на данном этапе массивы имеют вид, как показано в табл. 9.19 и 9.20.

***Таблица 9.19.*** *Массив* `Active` *на втором шаге алгоритма Дейкстры*

| **Индекс** | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| **Элементы** | 1 | 0 | 1 | 1 | 1 | 0 |

***Таблица 9.20.*** *Массив меток* `Label` *на втором шаге алгоритма Дейкстры*

| **Индекс** | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| **Элементы** | 0 | 32 767 | 5 | 2 | 4 | 32 767 |

Изменения, внесенные в граф после двух шагов, показаны на рис. 9.51.

Поиск продолжается из вершины, которая имеет минимальную метку в массиве `Label` и является пройденной. Таковой будет вершина `3`.

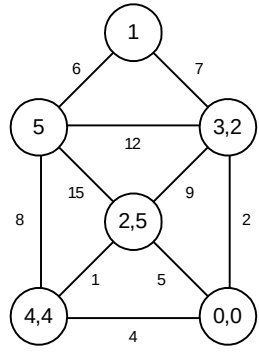



**Рис. 9.51.** Граф `W` на втором шаге алгоритма Дейкстры

Отметим вершину `3` как непройденную в массиве `Active`. Из данной вершины выходят четыре ребра: `(3,0)`, `(3,1)`, `(3,2)` и `(3,5)`. Проверим выполнение условия `Label[j] > Label[i] + Graf[i][j]` для каждой из вершин `0`, `1`, `2`, и `5`:

- `Label[0] > Label[3] + Graf[3][0]`, т. е. 0 < 2 + 2. Так как условие не выполнилось, то метка вершины 0 не изменяется;
- `Label[1] > Label[3] + Graf[3][1]`, т. е. 32 767 > 2 + 7. Следовательно, `Label[1] = Label[3] + Graf[3][1] = 9`;

- `Label[2]` > `Label[3]` + `Graf[3][2]`, т. е. 5 < 2 + 9. Так как условие не выполнилось, то метка вершины `2` не изменяется;
- `Label[5]` > `Label[3]` + `Graf[3][5]`, т. е. 32 767 > 2 + 12. Следовательно, `Label[5]` = `Label[3]` + `Graf[3][5]` = `14`.

Таким образом, на данном этапе массивы имеют вид, представленный в табл. 9.21 и 9.22.

***Таблица 9.21.*** *Массив* `Active` *на третьем шаге алгоритма Дейкстры*

| **Индекс** | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| **Элементы** | 1 | 1 | 1 | 0 | 1 | 1 |

***Таблица 9.22.*** *Массив меток* `Label` *на третьем шаге алгоритма Дейкстры*

| **Индекс** | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| **Элементы** | 0 | 9 | 5 | 2 | 4 | 14 |

Изменения, внесенные в граф после третьего шага, показаны на рис. 9.52.

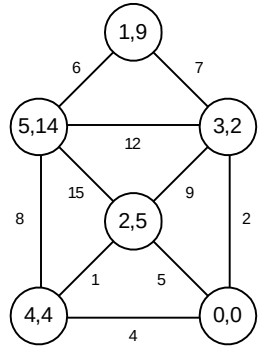



**Рис. 9.52.** Граф `W` на третьем шаге алгоритма Дейкстры

Затем продолжим поиск из вершины 0, т. к. она имеет минимальную метку в массиве Label и является пройденной.

Пометим вершину 0 как не пройденную в массиве Active. Из данной вершины выходят три ребра: (0,2), (0,3) и (0,4). Проверим выполнение условия Label[j] > Label[i] + Graf[i][j] для каждой из вершин 2, 3 и 4:

- Label[2] > Label[0] + Graf[0][2], т. е. 5 = 0 + 5. Так как условие не выполнилось, то метка вершины 2 не изменяется;
- Label[3] > Label[0] + Graf[0][3], т. е. 2 = 0 + 2. Так как условие не выполнилось, то метка вершины 3 не изменяется;
- Label[4] > Label[0] + Graf[0][4], т. е. 4 = 0 + 4. Так как условие не выполнилось, то метка вершины 4 не изменяется.

На данном этапе в массиве Label не произошло никаких изменений. Массив Active будет иметь вид, как показано в табл. 9.23.

***Таблица 9.23.*** *Массив* Active *на четвертом шаге алгоритма Дейкстры*

| **Индекс** | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| **Элементы** | 0 | 1 | 1 | 0 | 1 | 1 |

Найдем вершину с минимальной меткой в массиве Label при условии, что она должна быть пройденной, и продолжим из нее поиск. Таким образом, поиск продолжим из вершины 4, предварительно отметив ее как непройденную в массиве Active. Из вершины 4 выходят три ребра: (4,0), (4,2) и (4,5). Проверим выполнение условия Label[j] > Label[i] + Graf[i][j] для каждой из вершин 0, 2 и 5:

- Label[0] > Label[4] + Graf[4][0], т. е. 0 < 4 + 4. Так как условие не выполнилось, то метка вершины 0 не изменяется;
- Label[2] > Label[4] + Graf[4][2], т. е. 5 > 4 + 1. Так как условие не выполнилось, то метка вершины 2 не изменяется;
- Label[5] > Label[4] + Graf[4][5], т. е. 14 > 4 + 8. Следовательно, Label[5] = Label[4] + Graf[4][5] = 12.

Таким образом, на данном этапе массивы имеют вид, представленный в табл. 9.24 и 9.25.

***Таблица 9.24.*** *Массив* `Active` *на пятом шаге алгоритма Дейкстры*

| **Индекс** | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| **Элементы** | 0 | 1 | 1 | 0 | 0 | 1 |

***Таблица 9.25.*** *Массив меток* `Label` *на пятом шаге алгоритма Дейкстры*

| **Индекс** | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| **Элементы** | 0 | 9 | 5 | 2 | 4 | 12 |

Изменения, внесенные в граф после пятого шага, показаны на рис. 9.53.

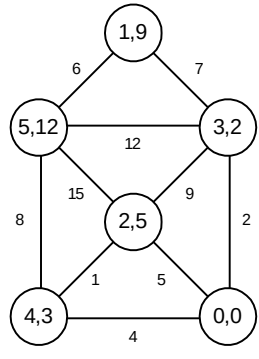



**Рис. 9.53.** Граф `W` на пятом шаге алгоритма Дейкстры

Продолжим поиск из вершины `2`, т. к. она имеет минимальную метку в массиве `Label`, и `Active[2]=1` (вершина является пройденной).

Пометим вершину `2` как непройденную. Из вершины `2` выходят четыре ребра: `(2,0)`, `(2,3)`, `(2,4)` и `(2,5)`. Проверим выполнение

условия `Label[j] > Label[i] + Graf[i][j]` для каждой из вершин `0`, `3`, `4` и `5`:

- `Label[0] > Label[2] + Graf[2][0]`, т. е. 0 < 4 + 5. Так как условие не выполнилось, то метка вершины `0` не изменяется;
- `Label[3] > Label[2] + Graf[2][3]`, т. е. 2 < 4 + 9. Так как условие не выполнилось, то метка вершины `3` не изменяется;
- `Label[4] > Label[2] + Graf[2][4]`, т. е. 3 < 4 + 1. Так как условие не выполнилось, то метка вершины `4` не изменяется;
- `Label[5] > Label[2] + Graf[2][5]`, т. е. 11 < 4 + 15. Так как условие не выполнилось, то метка вершины `5` не изменяется.

На данном этапе в массиве `Label` не произошло никаких изменений. Массив `Active` будет иметь вид, представленный в табл. 9.26.

***Таблица 9.26.*** *Массив* `Active` *на шестом шаге алгоритма Дейкстры*

| **Индекс** | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| **Элементы** | 0 | 1 | 0 | 0 | 0 | 1 |

Продолжим поиск из вершины `1`, т. к. `Label[1] < Label[5]` и `Active[1]=1` (вершина является пройденной).

Пометим вершину `1` как непройденную, т. е. `Active[1]=0`. Проверим выполнение условия `Label[j] > Label[i] + Graf[i][j]` для каждой вершины из окружения `1`:

- `Label[3] > Label[1] + Graf[1][3]`, т. е. 2 < 9 + 7. Так как условие не выполнилось, то метка вершины `3` не изменяется;
- `Label[5] > Label[1] + Graf[1][5]`, т. е. 11 < 9 + 6. Так как условие не выполнилось, то метка вершины `5` не изменяется.

На данном этапе в массиве `Label` не произошло никаких изменений. Массив `Active` будет иметь вид, как показано в табл. 9.27.

***Таблица 9.27.*** *Массив* `Active` *на седьмом шаге алгоритма Дейкстры*

| **Индекс** | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| **Элементы** | 0 | 0 | 0 | 0 | 0 | 1 |

В массиве `Active` осталась единственная вершина `5`, для которой `Active[5]=1`.

Продолжим поиск из нее, проверив выполнение условия `Label[j] > Label[i] + Graf[i][j]` для каждой вершины из окружения `5`:

- `Label[1] > Label[5] + Graf[5][1]`, т. е. 9 < 11 + 6. Так как условие не выполнилось, то метка вершины `3` не изменяется;
- `Label[2] > Label[5] + Graf[5][2]`, т. е. 4 < 11 + 15. Так как условие не выполнилось, то метка вершины `5` не изменяется;
- `Label[3] > Label[5] + Graf[5][3]`, т. е. 2 > 11 + 12. Так как условие не выполнилось, то метка вершины `5` не изменяется;
- `Label[4] > Label[5] + Graf[5][4]`, т. е. 3 > 11 + 8. Так как условие не выполнилось, то метка вершины 5 не изменяется.

На данном шаге также не произошло никаких изменений. Алгоритм завершит свою работу, т. к. в массиве `Active` не осталось ни одного элемента, равного единице. Конечный результат алгоритма представлен на рис. 9.53.

Таким образом, длина кратчайшего пути из вершины `0` в вершину `1` равна 9, т. к. `Label[1]=9`.

Далее представлен код программы, которая вычисляет длину кратчайшего пути из одной вершины графа в другую по алгоритму Дейкстра.

```
#include <iostream.h>
#include <conio.h>
#include <fstream.h>
int min(int[]);  // Прототип функции
int **Graf;      // Объявляет матрицу смежности графа
int *Label;      // Объявляет массив меток
```

```
int *Active;      // Объявляет массив, в котором мы будем
                  // помечать,что некоторая вершина графа
                  // пройдена на данном шаге
int i, j, k;
int Start, N, M, V, Last;
main()
{
  ifstream input ("input.txt");  // Открывает файл для чтения
  // Читает из файла N — количество вершин в графе,
  //                  M — количество ребер в графе,
  //       Start и Last — вершины, между которыми необходимо
  //                      найти длину кратчайшего пути
  input>>N>>M>>Start>>Last;
  // Выделяет динамическую память для матрицы смежности графа
  Graf=new int *[N];
  for (i=0; i<N; i++)
    Graf[i]=new int [N];
  // Обнуляет все элементы матрицы
  for (i=0; i<N; i++)
    for (j=0; j<N; j++)
      Graf[i][j]=0;
  Label=new int [N];   // Выделяет динамическую память для
                       // массива меток
  Active=new int [N];  // Выделяет динамическую память для
                       // массива, в котором мы будем помечать,
                       // что некоторая вершина графа пройдена
                       // на данном шаге
  // Обнуляем элементы массивов, память для которых была
  // выделена динамически
  for (i=0; i<N; i++)
  {
    Label[i]=0;
    Active[i]=0;
  }
  for (k=0; k<M; k++)
```

```
  {
    input>>i>>j>>V;     // Читает номера смежных вершин и вес
                        // ребра, которое их соединяет
    Graf[i][j]=V;       // Задает матрицу смежности
                        // взвешенного графа
  }
  for (i=0; i<M; i++)
    Label[i]=32767;
  Active[Start]=1;      // Помечает стартовую вершину
                        // как пройденную
  Label[Start]=0;
  i=Start;
  do
  {
    for (j=0; j<N; j++)
      if (Graf[i][j]!=0 && Label[j]>Label[i]+Graf[i][j])
      {
        Active[j]=1;    // Помечает вершину как пройденную
        Label[j]=Label[i]+Graf[i][j];
      }
    Active[i]=0;
    i=min(Label);       // Возвращает номер вершины
                        // с минимальной меткой
  } while (i!=-1);
  cout<<Label[Last];    // Выводит результат
  getch();              // Задерживает экран
  return 0;             // Завершает программу
}
int min (int array[])  // Описание функции
{
  int min, k, min_pos=-1;
  min=32767;
  // Находит минимальный элемент в массиве и его позицию
  for (k=0; k<M; k++)
    if (array[k]<min && Active[k]==1)
```

```
    {
      min=array[k];
      min_pos=k;
    }
  return min_pos;       // Возвращает позицию минимального
                        // элемента массива
}
```

## Вывод кратчайшего пути

В предыдущем разделе мы вычислили длину кратчайшего пути, теперь получить сам путь.

Для того чтобы получить путь, необходимо для каждой вершины `j` запоминать номер той вершины, из которой мы попали в `j`, т. е. зарезервировать память еще под один массив — массив "предков".

Теперь, изменяя метку вершины `j`, в которую мы "пришли" по ребру `(i,j)`, необходимо записать `i` в `pred[j]`. Таким образом, каждая вершина будет иметь свое значение в массиве "предков", и нам не составит труда вывести кратчайший путь из `Start` в `Last`. Чтобы получить его в прямом, а не в обратном порядке, необходимо начать вывод с конца массива.

Итак, модифицируем предыдущую программу, чтобы она выводила не только длину пути, но и сам путь.

Вот код модифицированной программы:

```
#include <iostream.h>
#include <conio.h>
#include <fstream.h>
int min(int[]);  // Прототип функции
int **Graf;      // Объявляет матрицу смежности графа
int *Label;      // Объявляет массив меток
int *Active;     // Объявляет массив, в котором мы будем
                 // помечать, что некоторая вершина графа
                 // пройдена на данном шаге
int *pred;       // Объявляет массив предков, в котором мы
```

```
                // будем хранить для каждой вершины i
                // некоторую вершину j, из которой мы попали в i
int i, j, k;
int Start, N, M, V, Last;
main()
{
  ifstream input ("input.txt");  // Открывает файл для чтения
  // Читает из файла N — количество вершин в графе,
  //                  M — количество ребер в графе,
  //      Start и Last — вершины, между которыми необходимо
  //                     найти длину кратчайшего пути
  input>>N>>M>>Start>>Last;
  // Выделяет динамическую память для матрицы смежности графа
  Graf=new int *[N];
  for (i=0; i<N; i++)
    Graf[i]=new int [N];
  // Обнуляет все элементы матрицы
  for (i=0; i<N; i++)
    for (j=0; j<N; j++)
      Graf[i][j]=0;
  Label=new int [N];   // Выделяет динамическую память для
                       // массива меток
  Active=new int [N];  // Выделяет динамическую память для
                       // массива, в котором мы будем
                       // помечать, что некоторая вершина
                       // графа пройдена на данном шаге
  pred=new int [N];    // Выделяет динамическую память для
                       // массива предков
  // Обнуляем элементы массивов, память для которых была
  // выделена динамически
  for (i=0; i<N; i++)
  {
    Label[i]=0;
    Active[i]=0;
    pred[i]=0;
```

```
}
for (k=0; k<M; k++)
{
  input>>i>>j>>V;     // Читает номера смежных вершин и вес
                      // ребра, которое их соединяет
  Graf[i][j]=V;       // Задает матрицу смежности
                      // взвешенного графа
}
for (i=0; i<M; i++)
  Label[i]=32767;
Active[Start]=1;      // Помечает стартовую вершину
                      // как пройденную
Label[Start]=0;
i=Start;
do
{
  for (j=0; j<N; j++)
    if (Graf[i][j]!=0 && Label[j]>Label[i]+Graf[i][j])
    {
      Active[j]=1;    // Помечает вершину как пройденную
      Label[j]=Label[i]+Graf[i][j];
      pred[j]=i;
    }
  Active[i]=0;
  i=min(Label);       // Возвращает номер вершины
                      // с минимальной меткой
} while (i!=-1);
cout<<Label[Last]<<endl;  // Выводит длину кратчайшего пути
int *Temp;
Temp=new int [N];
j=Last;
i=1;
Temp[0]=Last;
while (j!=Start)
{
```

```
    Temp[i]=pred[j];
    j=pred[j];
    i++;
  }
  // Выводит кратчайший путь между вершинами Start и Last
  for (j=i-1; j>=0; j--)
    cout<<Temp[j]<<" ";
  getch();   // Задерживает экран
  return 0;  // Завершает программу
}
int min(int array[])  // Описание функции
{
  int min, k, min_pos=-1;
  min=32767;
  // Находит минимальный элемент в массиве и его позицию
  for (k=0; k<M; k++)
    if (array[k]<min && Active[k]==1)
    {
      min=array[k];
      min_pos=k;
    }
  return min_pos;  // Возвращает позицию
                   // минимального элемента массива
}
int min(int array[])  // Описание функции
{
  int min, k, min_pos=-1;
  min=32767;
  // Находит минимальный элемент в массиве и его позицию
  for (k=0; k<M; k++)
    if (array[k]<min && Active[k]==1)
    {
      min=array[k];
      min_pos=k;
    }
```

```
  return min_pos;  // Возвращает позицию
                   // минимального элемента массива
}
```

# 9.6. Раскраска вершин графа

*Раскраской графа* называется функция `f: V → {1, 2, ..., k}`, которая ставит каждой вершине в соответствие некоторое число (здесь `k` принадлежит множеству натуральных чисел).

Раскраска называется *правильной* тогда и только тогда, когда для любых смежных вершин `i` и `j` выполняется неравенство `f(i)!=f(j)`.

Граф, для которого существует правильная раскраска, называется `k`-раскрашиваемым. Минимальное число `k`, при котором граф `G` является `k`-раскрашиваемым, называется *хроматическим числом*.

При раскрашивании вершин графа им будет задаваться некоторая метка, которая называется *цветом вершины*. Рассмотрим алгоритм раскрашивания вершин.

Пусть на первом шаге все вершины раскрашены в нулевой цвет. Теперь будем постепенно раскрашивать каждую вершину в цвета от 1 до `k`.

На каждом шаге будем закрашивать очередную вершину в минимально возможный цвет, т. е. если вершина `i` не имеет цвета, то она закрашивается минимальным цветом `k`, причем ни одна вершина из окружения `i` не должна быть раскрашена таким цветом.

Данный алгоритм раскраски называется *последовательным*. Необходимо отметить, что при последовательной раскраске вершин графа количество использованных цветов не будет минимальным.

Алгоритм последовательной закраски очень похож на обход в ширину. Далее представлен код программы, реализующей его на C++.

```
#include <iostream.h>
#include <conio.h>
#include <fstream.h>
int Find(int, int[]);  // Прототип функции
```

```
int i, j;
int Start, N, M, k, p, cur, pos;
main()
{
  int *Colors;   // Объявляет массив для хранения цветов,
                 // в которые окрашены вершины на данном шаге
  int *FIFO;     // Объявляет структуру данных "очередь"
  int **Graf;    // Объявляет матрицу смежности графа
  int *Set;      // Объявляет массив для хранения множества
                 // цветов, в которые уже окрашены вершины
  ifstream input ("input.txt");  // Открывает файл для
                                 // чтения данных
  input>>N>>M;   // Читает из файла
  // N — количество вершин в графе,
  // M — количество ребер в графе.
  // Выделяет динамическую память для матрицы N*N
  Graf=new int *[N];
  for (i=0; i<N; i++)
    Graf[i]=new int [N];
  // Обнуляет все элементы матрицы
  for (i=0; i<N; i++)
    for (j=0; j<N; j++)
      Graf[i][j]=0;
  for (k=0; k<M; k++)
  {
    input>>i>>j;       // Читает номера смежных вершин
    Graf[i][j]=1;      // Задает матрицу смежности
  }
  Colors=new int [N];  // Выделяет динамическую память
                       // для массива, предназначенного
                       // для хранения цветов,
                       // в которые окрашиваются вершины
                       // на данном шаге
  FIFO=new int [N];    // Выделяет динамическую память для
                       // очереди, состоящей из N элементов
```

```
Set=new int [N];      // Выделяет динамическую память для
                      // массива, хранящего множество цветов,
                      // в которые уже окрашены вершины
// Обнуляет все элементы массивов, для которых память была
// выделена динамически
for (i=0; i<N; i++)
{
  Colors[i]=0;
  FIFO[i]=0;
  Set[i]=0;
}
Start=0;              // Выбирает начальную вершину
Colors[Start]=1;      // и раскрашивает ее в цвет 1
p=0;                  // Указатель на начало очереди
k=1;                  // Указатель на конец очереди
FIFO[p]=Start;        // Заносим стартовую вершину в очередь
while (p!=k)
{
  cur=FIFO[p];        // Выделяет очередную вершину из очереди
  p++;                // Сдвигает указатель начала на единицу
  for (i=0; i<N; i++)
    if (Graf[cur][i]==1 && Colors[i]==0)
    {
      FIFO[k]=i;      // Заносит вершину в очередь
      k++;            // Сдвигает указатель конца очереди
      pos=0;          // Формируем множество цветов,
                      // в которые окрашены смежные вершины
      for (j=0; j<N; j++)
        if (Graf[FIFO[k-1]][j]==1 && Colors[j]!=0)
        {
          Set[pos]=Colors[j];
          pos++;
        }
     Colors[i]=Find(pos,Set);  // Окрашивает вершину в
                               // минимально возможный цвет
```

```
    }
  }
  // Выводит результат
  for (i=0; i<N; i++)
    cout<<Colors[i]<<" ";
  getch();   // Задерживает экран
  return 0;  // Завершает работу программы
}
int Find(int pos, int Set[])  // Описание функции,
              // выполняющей поиск минимального номера цвета,
              // в который можно окрасить вершину i
{
  int Min_color, i, p;
  Min_color=0;
  do
  {
    Min_color++;
    p=0;
    for (i=0; i<pos; i++)
      if (Set[i]==Min_color)
      {
        p=1;
        break;
      }
  } while (p!=0);
  return Min_color;  // Возвращает минимальный цвет,
                     // в который можно окрасить вершину
 }
```

Отличие данного алгоритма от поиска в ширину состоит лишь в том, что после того, как вершина `i`, смежная с `cur`, заносится в очередь, строится такое множество `Set`, элементы которого равны цветам, в которые раскрашены смежные с `i` вершины:

```
for (j=0; j<N; j++)
  if (Graf[FIFO[k-1]][j]==1 && Colors[j]!=0)
```

```
  {
    Set[pos]=Colors[j];
    pos++;
  }
```

После того как множество будет построено, вызывается функция `Find(int)`, возвращающая минимальный номер цвета, в который можно окрасить вершину `i`:

```
int Find(int pos, int Set)
{
  int Min_color, i, p;
  Min_color=0;
  do
  {
    Min_color++;
    p=0;
    for (i=0; i<pos; i++)
      if (Set[i]==Min_color)
      {
        p=1;
        break;
      }
  } while (p!=0);
  return Min_color;
}
```

# 9.7. Примеры решения задач при помощи графов

## Пример 1

Существует $K > 3$ городов. Некоторые из них соединены дорогами. Рассмотрим программу, которая определит номера таких городов, до которых можно добраться из любого города за одинаковое время. Причем это время должно быть наименьшим.

**Примечание**

Длину дороги между городами `i` и `j` считать равной единице.

```
#include <iostream.h>
#include <conio.h>
#include <fstream.h>
void OVS(int,int[],int[]);  // Прототип функции
int i, j, k, p, cur;
int Start, N, M;
int **Graf;            // Объявляет матрицу смежности графа
main()
{
  int *Label;          // Объявляет массив меток
  int *FIFO;           // Объявляет структуру данных "очередь"
  int *All_Label;      // Объявляет массив для хранения меток
                       // вершин, которые были получены после
                       // каждого выполненного обхода в ширину
  ifstream input ("input.txt");  // Открывает файл
                                 // для чтения данных
  // Читает из файла N — количество вершин в графе,
  //                  M — количество ребер в графе
  input>>N>>M;
  Label=new int [N];   // Выделяет динамическую память для
                       // массива меток
  FIFO=new int [N];    // Выделяет динамическую память для
                       // структуры данных "очередь".
  All_Label=new int [N];  // Выделяет динамическую память
                          // для массива, который служит
                          // для хранения меток вершин,
                          // полученных после каждого
                          // выполненного обхода в ширину
  // Обнуляет значения элементов массивов, память для которых
  // была выделена динамически
  for (i=0; i<N; i++)
  {
    Label[i]=0;
```

```
  All_Label[i]=0;
  FIFO[i]=0;
}
// Выделяет динамическую память для матрицы N*N
Graf=new int *[N];
for (i=0; i<N; i++)
  Graf[i]=new int [N];
// Обнуляет все элементы матрицы
for (i=0; i<N; i++)
  for (j=0; j<N; j++)
    Graf[i][j]=0;
for (k=0; k<M; k++)
{
  input>>i>>j;        // Читает номера смежных вершин
  Graf[i][j]=1;       // Задает матрицу смежности
}
OVS(0,FIFO,Label);    // Запускает обход в ширину от вершины 0
for (i=0; i<N; i++)
  All_Label[i]=Label[i]; // Снимает копию с массива меток Label
for (i=1; i<N; i++)
{
  OVS(i,FIFO,Label); // Запускает поиск в ширину от вершины i
  // Каждый элемент в позиции j массива меток сравнивается
  // с элементом в такой же позиции массива All_Label.
  // Если некоторые элементы не равны, и значение элемента
  // в данной позиции массива All_Label не равно нулю,
  // то их значение становится равным -1. Если же элемент
  // массива All_Label в позиции j равен нулю,
  // то его значение становится равным Label[j]
  for (j=0; j<N; j++)
    if (All_Label[j]!=Label[j] && Label[j]!=0)
      if (All_Label[j]==0) All_Label[j]=Label[j];
      else All_Label[j]=-1;
}
int Temp=0;
// Выводит ответ
```

```
  for (i=0; i<N; i++)
    if (All_Label[i]!=-1)
    {
      cout<<i<<" ";
      Temp=1;
    }
  if (Temp==0) cout<<"Таких городов нет!";
  getch();        // Задерживает экран
  return 0;       // Завершает программу
}
// Описание функции
void OVS(int Start, int FIFO[], int Label[])
{
  int Z;
  for (Z=0; Z<N; Z++)
  {
    FIFO[Z]=0;
    Label[Z]=32767;
  }
  p=0;
  k=1;
  FIFO[p]=Start;
  Label[Start]=0;
  while (p!=k)
  {
    cur=FIFO[p];
    p++;
    for (Z=0; Z<N; Z++)
      if (Graf[cur][Z]==1 && Label[Z]>Label[cur]+1)
      {
        FIFO[k]=Z;
        k++;
        Label[Z]=Label[cur]+1;
      }
  }
}
```

Решение данной задачи состоит в том, чтобы найти расстояние от любой вершины `i` до любой вершины `j`. Очевидно, что эти действия можно выполнить при помощи поиска в ширину. В данном случае функция `OVS(int,int[],int[])` выполняет поиск от некоторой стартовой вершины `Start`. Рассмотрим подробнее код данной функции:

```
void OVS(int Start, int FIFO[], int Label[])
{
  int Z;
  for (Z=0; Z<N; Z++)
  {
    FIFO[Z]=0;
    Label[Z]=32767;
  }
  p=0;
  k=1;
  FIFO[p]=Start;
  Label[Start]=0;
  while (p!=k)
  {
    cur=FIFO[p];
    p++;
    for (Z=0; Z<N; Z++)
      if (Graf[cur][Z]==1 && Label[Z]>Label[cur]+1)
      {
        FIFO[k]=Z;
        k++;
        Label[Z]=Label[cur]+1;
      }
  }
}
```

На первом шаге выполним поиск из вершины `0`. Таким образом, будет получен массив меток `Label`, `i`-ый элемент которого равен

расстоянию от вершины `0` до вершины `i`. Эти данные мы и сохраним для последующего сравнения:

```
OVS(0,FIFO,Label);
for (i=0; i<N; i++)
  All_Label[i]=Label[i];
```

Теперь на каждом шаге будем выполнять поиск в ширину для каждой вершины от `1` до `N`, вызывая функцию `OVS(int,int[],int[])`. Всякий раз, как функция завершит свою работу, будет получен массив `Label` с новыми значениями. Затем каждый элемент в позиции `j` данного массива необходимо сравнить с элементом в такой же позиции массива `All_Label`. Если окажется, что элементы некоторой пары не равны и `All_Label[j]!=0` (т. е. расстояние до этой вершины еще не было найдено), то `All_Label[j]=-1`. В противном случае `All_Label[j]=Label[j]` (т. е. подсчитано первое расстояние, и его необходимо запомнить в результирующем массиве `All_Label`):

```
for (j=0; j<N; j++)
  if (All_Label[j]!=Label[j] && Label[j]!=0)
    if (All_Label[j]==0) All_Label[j]=Label[j];
    else All_Label[j]=-1;
```

Таким образом, когда поиск в ширину будет выполнен от каждой вершины, мы получим обработанный массив `All_Label`, причем `All_Label[i]!=-1` тогда и только тогда, когда расстояние от любой вершины до вершины `i` одинаково. В соответствии с данным утверждением можно вывести ответ:

```
int Temp=0;
for (i=0; i<N; i++)
  if (All_Label[i]!=-1)
  {
    cout<<i<<" ";
    Temp=1;
  }
if (Temp==0) cout<<"Таких городов нет !";
```

## Пример 2

Дан взвешенный граф G, в котором насчитывает N вершин и M ребер. Рассмотрим программу, которая определяет номера вершин, по которым пересекаются кратчайшие пути из вершины A в B и из C в D.

Вот код решения данной задачи:

```
#include <iostream.h>
#include <conio.h>
#include <fstream.h>
int min(int[]);          // Прототип функции
void Shorter_Way(int);   // Прототип функции
void Way(int,int);       // Прототип функции
int **Graf, *Label, *Active, *pred, i, j;
int A, B, C, D, N, M, V, Last, k, *Temp;
main()
{
  ifstream input ("input.txt");  // Открывает файл для чтения
  // Читает из файла N — количество вершин в графе,
  //                 M — количество ребер в графе,
  //             A и B — первая пара вершин, между которыми
  //                     необходимо найти длину
  //                     кратчайшего пути,
  //             C и D — вторая пара вершин,
  //                     между которыми необходимо
  //                     найти длину кратчайшего пути
  input>>N>>M>>A>>B>>C>>D;
  // Выделяет динамическую память для матрицы смежности графа
  Graf=new int *[N];
  for (i=0; i<N; i++)
    Graf[i]=new int [N];
  // Обнуляет все элементы матрицы
  for (i=0; i<N; i++)
    for (j=0; j<N; j++)
      Graf[i][j]=0;
  Active=new int [N];  // Выделяет динамическую память для
```

```
                        // массива, в котором мы будем
                        // помечать, что некоторая вершина
                        // графа пройдена на данном шаге
  pred=new int [N];     // Выделяет динамическую память для
                        // массива предков
  Label=new int [N];    // Выделяет динамическую память для
                        // массива меток
  Temp=new int [N];
  for (k=0; k<M; k++)
  {
    input>>i>>j>>V;     // Читает номера смежных вершин и вес
                        // ребра, которое их соединяет
    Graf[i][j]=V;       // Заполняет матрицу смежности
                        // взвешенного графа
  }
  for (i=0; i<N; i++)
    Temp[i]=-1;
  Shorter_Way(A);       // Запускает функцию, которая найдет
                        // кратчайший путь из A в B
  Way(A,B);             // Запускает функцию, которая запомнит
                        // найденный путь и проверит наличие
                        // пересечения вершин
  Shorter_Way(C);       // Запускает функцию, которая найдет
                        // кратчайший путь из C в D
  Way(C,D);             // Запускает функцию, которая запомнит
                        // найденный путь и проверит
                        // пересечения вершин
  getch();   // Задерживает экран
  return 0;  // Завершает программу
}
void Way(int Start, int Last)  // Описание функции
{
  j=Last;
  i=1;
  Temp[Start]=0;
```

```
  while (j!=Start)
  {
    if (Temp[j]!=-1)   // Если вершина уже пройдена ранее,
                       // значит, в ней пересекаются пути
    {
      Temp[j]=pred[j];
      cout<<j<<" ";    // Выводит ее на экран
    } else Temp[j]=pred[j];
    j=pred[j];
  }
}
int min(int array[])   // Описание функции
{
  int min, k, min_pos=-1;
  min=32767;
  // Находит минимальный элемент в массиве и его позицию
  for (k=0; k<M; k++)
    if (array[k]<min && Active[k]==1)
    {
      min=array[k];
      min_pos=k;
    }
  return min_pos;      // Возвращает позицию
                       // минимального элемента массива
}
// Описание функции, которая находит кратчайший путь из
// вершины Start в вершину Last
void Shorter_Way(int Start)
{
  for (i=0; i<M; i++)
    Label[i]=32767;
  Active[Start]=1;  // Помечает стартовую вершину как пройденную
  Label[Start]=0;
  i=Start;
```

```
  do
  {
    for (j=0; j<N; j++)
      if (Graf[i][j]!=0 && Label[j]>Label[i]+Graf[i][j])
      {
        Active[j]=1;   // Помечает вершину как пройденную.
        Label[j]=Label[i]+Graf[i][j];
        pred[j]=i;     // Запоминает номер вершины,
                       // из которой "пришли" в j
      }
    Active[i]=0;
    i=min(Label);      // Возвращает номер вершины
                       // с минимальной меткой
  } while (i!=-1);
}
```

# 9.8. Упражнения

1. Некоторые школы связаны компьютерной сетью. Каждая школа имеет список школ-получателей, которым она рассылает программное обеспечение всякий раз, получив новые бесплатные программы. При этом, если школа `B` есть в списке получателей школы `A`, то школы `A` может не быть в списке получателей школы `B`. Напишите программу, определяющую минимальное количество школ, которым надо передать по одному экземпляру нового программного обеспечения, чтобы распространить его по всем школам сети. Кроме того, надо обеспечить возможность рассылки нового программного обеспечения из любой школы по всем остальным школам. Для этого можно расширять списки получателей некоторых школ, добавляя в них новые школы. Найдите минимальное суммарное количество расширений списков, при которых программное обеспечение из любой школы достигло бы всех остальных школ. Одно расширение означает добавление одной школы-получателя в список получателей какой-либо из школ.

2. Задан ориентированный граф с $N$ вершинами, пронумерованными целыми числами от `1` до $N$. Напишите программу, которая подсчитывает количество различных путей между всеми парами вершин графа.
3. Даны два числа $N$ и $M$. Напишите программу, которая построит граф из $N$ вершин и $M$ ребер. Каждой вершине ставится в соответствие число ребер, входящих в нее. Граф должен быть таким, чтобы сумма квадратов этих чисел была минимальна.
4. Задан ориентированный граф с $N$ вершинами, каждому ребру которого сопоставлен неотрицательный вес. Напишите программу, которая сможет найти и вывести на экран простой цикл, для которого среднее геометрическое весов его ребер было бы минимально. (Следует напомнить, что цикл называется простым, если вершины, из которых он построен, не повторяются, возможно, за исключением двух крайних.)
5. Задан неориентированный граф с $N$ вершинами, пронумерованными целыми числами от 1 до $N$. Напишите программу, которая ориентирует максимальное количество ребер, чтобы получившийся граф оказался сильно связным.
6. Ребенок нарисовал кружки, и некоторые из них соединил отрезками. Кружки он пометил целыми числами от 1 до $N$, а на каждом отрезке поставил стрелочку. Затем он приписал каждому кружочку вес в виде некоторого целого числа и определил начальный и конечный кружочки. Из первого он должен выйти, а во второй попасть. Ребенок решил для себя следующее:
   - набрать максимально возможное суммарное количество очков;
   - по каждому отрезку пройти ровно один раз;
   - если в кружок он попадает при движении по направлению стрелки, то к суммарному количеству очков вес этого кружка прибавляется;
   - если в кружок он попадает при движении против направления стрелки, то из суммарного количества очков вес этого кружка вычитается.

Напишите программу, которая бы помогла ребенку построить путь, удовлетворяющий всем этим требованиям.

7. В некотором городе есть метро, состоящее из $N$ станций и $M$ линий, соединяющих их. Каждая линия обеспечивает проезд между какими-то двумя станциями в обе стороны. Между любой парой станций проведено не более одной линии. Сеть метро построена таким образом, чтобы от каждой станции можно было проехать на каждую (возможно, через промежуточные станции). Назовем это свойство связностью метро.

   Из-за изобретения принципиально нового вида транспорта метро стало убыточным, и его работу решили прекратить. На заседании мэрии города было постановлено закрывать каждый год по одной станции, но так, чтобы связность метро каждый раз сохранялась. При закрытии какой-либо станции линии, ведущие из этой станции в другие, естественно, тоже перестают функционировать.

   Напишите программу, которая по введенной информации о сети метро разработает порядок закрытия станций, при котором метро всегда будет оставаться связным.

8. Задан связный ориентированный граф. Напишите программу, которая определит, какое максимальное количество ребер можно исключить из графа, чтобы он по-прежнему оставался связным.

9. Задан несвязный неориентированный граф. Напишите программу, которая определит, какое минимальное количество ребер необходимо добавить в граф, чтобы он стал связным.

10. Задан связный граф с четными степенями вершин. Напишите программу, которая определит, можно ли из каждой вершины удалить по $K$ выходящих ребер таким образом, чтобы получившийся граф остался связным.

**Примечание**

Во всех задачах *главы 9* входные данные считать корректными.

# Глава 10

# Основы шифрования

В последнее время возникает потребность в передаче данных между предприятиями через Internet, локальную сеть и другими способами. В процессе передачи данных есть риск перехвата и злоупотребления полученной информацией, поэтому в целях защиты используют шифрование информации.

Некоторые алгоритмы шифрования основаны на том, что сам метод шифрования (алгоритм) является секретным. В настоящее время такие методы представляют лишь исторический интерес и не имеют практического значения. Все современные алгоритмы используют ключ (набор секретных данных, при помощи которых можно расшифровать сообщение) для управления шифрованием и дешифрованием; сообщение может быть успешно дешифровано, только если известен ключ. Он может не совпадать с ключом, используемым для шифрования, однако в большинстве алгоритмов ключи совпадают.

## 10.1. Классы алгоритмов шифрования

Алгоритмы с использованием ключа делятся на два класса:

- ❒ симметричные или алгоритмы секретным ключом;
- ❒ асимметричные или алгоритмы с открытым ключом.

Разница между ними в том, что *симметричные* алгоритмы используют один и тот же ключ для шифрования и для дешифрования

(или же ключ для дешифрования вычисляется на основе ключа шифрования), в то время как *асимметричные* алгоритмы используют разные ключи, и ключ для дешифрования не может быть вычислен по ключу шифрования.

Симметричные алгоритмы подразделяют на потоковые и блочные шифры. Потоковые позволяют шифровать информацию побитно, в то время как блочные работают с некоторым набором битов данных (обычно размер блока составляет 64, 128, 256 битов) и шифруют этот набор как единое целое.

Ассиметричные шифры допускают, чтобы открытый ключ был доступен всем, скажем, опубликован в газете. Это позволяет любому зашифровать сообщение. Однако расшифровать это сообщение сможет только нужный человек (тот, кто владеет ключом дешифрования). Ключ для шифрования называют *открытым ключом*, а ключ для дешифрования — *закрытым ключом* или *секретным ключом*.

Современные алгоритмы шифрования/дешифрования достаточно сложны, и их весьма трудно выполнять вручную. Настоящие криптографические алгоритмы разработаны для использования персональными компьютерами или специальными вычислительными устройствами. В большинстве приложений шифрование производится программным обеспечением, и на рынке имеется множество доступных криптографических пакетов.

Вообще говоря, симметричные алгоритмы работают быстрее, чем ассиметричные. На практике оба типа алгоритмов часто используются вместе: алгоритм с открытым ключом предназначен для того, чтобы передать случайным образом сгенерированный секретный ключ, который затем используется для дешифрования сообщения.

## 10.2. Требования к криптографическим системам

Процесс криптографического закрытия данных может осуществляться как программно, так и аппаратно. Аппаратная реализация отличается существенно большей стоимостью, однако она имеет

свои преимущества: высокую производительность, простоту, защищенность и т. д. Программная реализация более практична, допускает известную гибкость в использовании. Для современных криптографических систем защиты информации сформулированы следующие общепринятые требования:

- ❒ зашифрованное сообщение должно поддаваться чтению только при наличии ключа;
- ❒ число операций, необходимых для определения использованного ключа шифрования по фрагменту шифрованного сообщения и соответствующего ему открытого текста должно быть не меньше, чем число операций при последовательном переборе общего числа возможных ключей;
- ❒ число операций, необходимых для расшифровки информации путем перебора всевозможных ключей, должно иметь строгую нижнюю оценку и выходить за пределы возможностей современных компьютеров (с учетом возможности использования сетевых вычислений);
- ❒ знание алгоритма шифрования не должно влиять на надежность защиты;
- ❒ незначительное изменение ключа должно приводить к существенному изменению вида зашифрованного сообщения даже при шифровании одного и того же исходного текста;
- ❒ незначительное изменение исходного текста должно приводить к существенному изменению вида зашифрованного сообщения даже при использовании одного и того же ключа;
- ❒ структурные элементы алгоритма шифрования должны быть неизменными;
- ❒ дополнительные биты, вводимые в сообщение в процессе шифрования, должны быть полностью и надежно скрыты в шифрованном тексте;
- ❒ длина шифрованного текста должна быть равной длине исходного текста;
- ❒ не должно быть простых и легкоустанавливаемых зависимостей между ключами, последовательно используемыми в процессе шифрования;

- любой ключ из множества возможных ключей должен обеспечивать надежную защиту информации;
- алгоритм должен допускать как программную, так и аппаратную реализацию, при этом изменение длины ключа не должно вести к качественному ухудшению алгоритма шифрования.

## Криптоанализ

Попытка вскрытия алгоритма шифрования называется *криптоанализом*. Основное положение криптоанализа, впервые сформулированное в XIX веке Датчманом А. Кирхгофом (Dutchman A. Kerckhoffs), состоит в том, что безопасность полностью определяется ключом. Кирхгоф предполагает, что у криптоаналитика есть полное описание алгоритма и его реализации. Хотя в реальном мире криптоаналитики не всегда обладают подробной информацией, такое предположение является хорошей рабочей гипотезой. Если противник не сможет взломать алгоритм, даже зная, как он работает, то тем более не сможет вскрыть алгоритм без этого знания. Далее перечислены основные типы криптоаналитического вскрытия. Для каждого из них, конечно, предполагается, что криптоаналитик обладает всей полнотой знания об используемом алгоритме шифрования.

- *Вскрытие с использованием только шифротекста.* У криптоаналитика есть шифротексты нескольких сообщений, зашифрованных одним и тем же алгоритмом шифрования. Задача криптоаналитика состоит в получении открытого текста как можно большего числа сообщений, или в получении ключа (ключей), использованного для шифрования сообщений, для дешифрования других сообщений, зашифрованных теми же ключами.
- *Вскрытие с использованием открытого текста.* У криптоаналитика есть доступ не только к шифротекстам нескольких сообщений, но и к открытому тексту этих сообщений. Его задача состоит в получении ключа (или ключей), использованного для шифрования сообщений, с целью дешифрования других сообщений, зашифрованных тем же ключом (ключами).

- *Вскрытие с использованием выбранного открытого текста.* У криптоаналитика не только есть доступ к шифротекстам и открытым текстам нескольких сообщений, но и возможность выбирать открытый текст для шифрования. Это предоставляет больше вариантов, чем вскрытие с использованием случайного открытого текста, т. к. криптоаналитик может выбирать шифруемые блоки открытого текста, что может дать больше информации о ключе. Его задача состоит в получении ключа (или ключей), использованного для шифрования сообщений, или алгоритма, позволяющего дешифровать новые сообщения, зашифрованные тем же ключом (или ключами).
- *Адаптивное вскрытие с использованием открытого текста.* Это частный случай вскрытия с использованием выбранного открытого текста. Криптоаналитик не только может выбирать шифруемый текст, но также может строить свой последующий выбор на базе полученных результатов шифрования. При вскрытии с использованием выбранного открытого текста криптоаналитик мог выбрать для шифрования только один большой блок открытого текста, при адаптивном вскрытии с использованием выбранного открытого текста он может выбрать меньший блок открытого текста, затем выбрать следующий блок, используя результаты первого выбора, и т. д.
- *Вскрытие с использованием выбранного шифротекста.* Криптоаналитик может выбирать различные шифротексты для дешифрования и имеет доступ к дешифрованным открытым текстам. Например, у криптоаналитика есть доступ к устройству, которое выполняет автоматическое дешифрирование. Его задача состоит в получении ключа.
- *Вскрытие с использованием выбранного ключа.* Такой тип вскрытия означает, что у криптоаналитика есть некоторая информация о связи между различными ключами.
- *Вскрытие с использованием человеческого фактора.* Криптоаналитик угрожает, шантажирует или пытает кого-нибудь, пока не получит ключ. Взяточничество иногда называется вскрытием с покупкой ключа. Это очень эффективные и дешевые способы вскрытия, часто являющиеся наилучшими.

Вскрытия с известным открытым текстом и с использованием выбранного открытого текста встречаются чаще всего. Существуют также возможности для криптоаналитика добыть открытый текст шифрованного сообщения или подкупить кого-нибудь, кто зашифрует выбранное сообщение. Многие сообщения имеют стандартные начало и окончание, что может быть известно криптоаналитику. Особенно уязвим шифрованный исходный код из-за частого использования ключевых слов `#define`, `struct`, `else`, `return` и т. п. Те же проблемы и у шифрованного исполняемого кода — функции, циклические структуры и т. д. Вскрытия с известным открытым текстом (и вскрытия с выбранным шифротекстом) успешно использовались в борьбе с фашистами в ходе Второй мировой войны.

## Безопасность алгоритмов

Различные алгоритмы предоставляют разные степени безопасности в зависимости от того, насколько трудно взломать алгоритм. Если стоимость взлома алгоритма выше, чем стоимость зашифрованных данных, алгоритм можно считать надежным. Если время взлома алгоритма больше, чем время, в течение которого зашифрованные данные должны сохраняться в секрете, то алгоритм также можно считать надежным. Если объем данных, зашифрованных одним ключом, меньше, чем объем данных, необходимый для взлома алгоритма, тогда тоже можно считать алгоритм надежным. Датский ученый Ларс Кнудсен (Lars Knudsen) разбил результаты вскрытия алгоритмов по категориям, приведенным далее в порядке убывания значимости.

1. Полное вскрытие. Криптоаналитик получил ключ.
2. Глобальная дедукция. Криптоаналитик получил альтернативный алгоритм, эквивалентный вскрываемому, без знания ключа.
3. Местная (или локальная) дедукция. Криптоаналитик получил открытый текст для перехваченного шифротекста.
4. Информационная дедукция. Криптоаналитик получил некоторую информацию о ключе или открытом тексте. Такой информацией могут быть несколько битов ключа, сведения о форме открытого текста и т. д.

Алгоритм является *безусловно безопасным*, если, независимо от объема шифротекстов у криптоаналитика, информации для получения открытого текста недостаточно. По сути, только шифрование невозможно вскрыть при бесконечных ресурсах. Все остальные криптосистемы подвержены вскрытию с использованием только шифротекста простым перебором возможных ключей и проверкой осмысленности полученного открытого текста. Это называется вскрытием грубой силой. Криптография больше интересуется криптосистемами, которые тяжело взломать вычислительным способом. Алгоритм считается *вычислительно безопасным* или, как иногда называют, *сильным*, если он не может быть взломан с использованием доступных ресурсов в обозримом будущем. Термин "доступные ресурсы" является достаточно расплывчатым. Сложность вскрытия можно измерить по одному из следующих параметров:

- сложность данных — это объем данных, используемых на входе операции вскрытия;
- сложность обработки — время, нужное для проведения вскрытия (часто называется коэффициентом работы);
- требования к памяти — объем памяти, необходимый для вскрытия.

В качестве эмпирического метода сложность вскрытия определяется по максимальному из этих трех коэффициентов. Ряд операций вскрытия предполагают взаимосвязь коэффициентов: более быстрое вскрытие возможно за счет увеличения требований к памяти, например. Если шифрование сообщения было основано на сложных логических и математических операциях, то на его расшифровывание могут уйти миллионы лет и более. Вычислительная сложность вскрытия остается постоянной (пока не будет найден более эффективный алгоритм), а производительность компьютеров растет. Многие криптографические алгоритмы могут быть реализованы на параллельных компьютерах: задача разбивается на множество подзадач, решение которых не требует межпроцессорного взаимодействия. Если криптосистему нелегко взломать, используя современную технику, то это еще не дает

гарантии ее надежности в будущем. Хорошие криптосистемы проектируются устойчивыми к взлому с учетом развития вычислительных средств на много лет вперед.

# 10.3. Перестановочные алгоритмы

## Простой столбцевой перестановочный шифр

В данном виде шифра текст пишется горизонтально на листе бумаги фиксированной ширины, а шифротекст считывается по вертикали. Дешифрование заключается в записи шифротекста вертикально, а затем считывании открытого текста горизонтально, т. е. если лист представить в виде матрицы размером $N \times K$, то исходный текст необходимо выписывать с элемента `[0][0]` матрицы до элемента `[N][0]`, затем переходить в следующий столбец и продолжать выписывание.

Пусть задан для шифрования следующий текст:

```
Пример использования простого столбцевого перестановочного
шифра
```

Зашифрованный текст получается следующим образом:

- подсчитаем количество символов в исходной строке, чтобы потом определить матрицу нужного размера;
- определим матрицу размером 10×7, т. к. в исходном сообщении 64 символа;
- запишем исходный текст по одному символу в каждую ячейку матрицы, причем пробелу будет эквивалента пустая ячейка матрицы.

Таким образом, для нашего примера матрица будет иметь вид, представленный на рис. 10.1.

Тогда зашифрованный текст будет иметь вид:

```
Пиопоцпнгарсвр ееоо ипаосврв  монстоеош елитогсчи рьяолотнф
з гб аор
```

| | | | | | | |
|---|---|---|---|---|---|---|
| П | р | и | м | е | р | |
| и | с | п | о | л | ь | з |
| о | в | а | н | и | я | |
| п | р | о | с | т | о | г |
| о | | с | т | о | л | б |
| ц | е | в | о | г | о | |
| п | е | р | е | с | т | а |
| н | о | в | о | ч | н | о |
| г | о | | ш | и | ф | р |
| а | | | | | | |

**Рис. 10.1.** Матрица столбцевого перестановочного шифра

Далее приведен код программы, в которой реализован данный алгоритм. Условимся, что в матрице будет всегда 6 столбцов. Количество строк будет зависеть от количества символов.

```
#include <iostream.h>
#include <conio.h>
#include <fstream.h>
void Coder(int, int, char *[]);  // Прототип функции
main()
{
  int k, i, j;
  char **array, S, Z;
  ifstream Temp("input.txt");
  // Подсчитывает количество символов в файле
  while (!Temp.eof())
  {
    Temp>>Z;
    Kol++;
  }
  // Выделяет динамическую память для матрицы
  array=new char *[Kol];
  for (int i=0; i<Kol; i++)
```

```
    array[i]=new char[6];
  // Задает начальное значение ячеек
  for (j=0; j<Kol; j++)
    for (i=0; i<=6; i++)
      array[j][i]='\0';  // Значение этого элемента матрицы
                         // становится равным пустому символу
  ifstream input("input.txt");  // Открывает файл
                                // для чтения данных
  // Запускает цикл повторений, который продолжает работу до тех пор,
  // пока не будут считаны и обработаны все символы из файла
  i=0;
  j=0;
  while (!input.eof())
  {
    input>>S;       // Читает из файла очередной символ
    array[i][j]=S;  // Заносит символ в матрицу
    j++;
    if (j==6)
    {
      i++;
      j=0;
    }
  }
  array[i][j-1]='\0';  // Удаляет последний ненужный символ
  Coder(0,i+1,array);  // Запускает функцию, которая
                       // шифрует исходное сообщение
  getch();             // Задерживает экран
  return 0;            // Завершает программу
}
void Coder(int j, int i, char *array[]) // Описание функции
{
  int k;
  for (k=0; k<i; k++)
    if (array[k][j]!='\0') cout<<array[k][j];
```

```
  if (j!=6)
  {
    j++;
    Coder(j,i,array);
  }
}
```

## Перестановочный шифр с ключевым словом

Для использования данного алгоритма необходимо буквы открытого текста записать в клетки прямоугольной таблицы по строкам так же, как в предыдущем алгоритме. Буквы ключевого слова пишутся над столбцами и указывают порядок этих столбцов (по возрастанию номеров букв в алфавите). Чтобы получить зашифрованный текст, надо выписывать буквы по столбцам с учетом их нумерации, т. е. сначала выписываются буквы, находящиеся в том столбце, в котором находится буква ключевого слова, расположенная в алфавите раньше других.

Проанализируем работу такого алгоритма на следующем примере открытого текста:

`Перестановочный шифр с ключевым словом`

В качестве ключевого возьмем слово `алгоритм`.

Таким образом, заполненная по вышеуказанным правилам матрица будет иметь вид, представленный на рис. 10.2.

| а | л | г | о | р | и | т | м |
|---|---|---|---|---|---|---|---|
| 1 | 4 | 2 | 6 | 7 | 3 | 8 | 5 |
| П | е | р | е | с | т | а | н |
| о | в | о | ч | н | ы | й | |
| ш | и | ф | р | | с | | к |
| л | ю | ч | е | в | ы | м | |
| с | л | о | в | о | м | | |

**Рис. 10.2.** Матрица перестановочного шифра с ключевым словом

Тогда зашифрованный текст примет вид:

```
Пошлсрофчотысымевиюлн к ечревсн воай м
```

Так как символы криптотекста те же, что и в открытом тексте, то частотный анализ покажет, что каждая буква встречается приблизительно с той же частотой, что и обычно. Это означает, что шифр является перестановочным. Применение к криптотексту второго перестановочного фильтра значительно повысит безопасность. Существуют и еще более сложные перестановочные шифры, но с применением компьютера можно раскрыть почти все из них.

Хотя многие современные алгоритмы используют перестановку, с этим связана проблема использования большого объема памяти, а также иногда требуется работа с сообщениями определенного размера. Поэтому чаще используют подстановочные шифры (они будут рассмотрены в следующем разделе).

Далее приведен код программы, в которой реализован перестановочный алгоритм с ключевым словом:

```
#include <iostream.h>
#include <conio.h>
#include <fstream.h>
void Coder(int, char *[]);  // Прототип функции
main()
{
  int k, i, j;
  char **array, S, *Key_word;
  // Для простоты выделим память для матрицы 100*100,
  // но работать будем только с количеством столбцов,
  // соответствующим длине кодового слова
  array=new char *[100];
  for (int i=0; i<100; i++)
    array[i]=new char[100];
  // Задает начальное значение ячеек
  for (j=0; j<100; j++)
    for (i=0; i<100; i++)
```

```
      array[j][i]=' ';
  ifstream input("input.txt");     // Открывает файл для
                                   // чтения данных
  Key_word=new char [100];         // Выделяет память
                                   // под массив типа char
  input>>Key_word;       // Читает из файла ключевое слово
  // Заносит каждый символ ключевого слова в матрицу
  for (i=0; i<strlen(Key_word); i++)
    array[0][i]=Key_word[i];
  i=1;
  j=0;
  while (!input.eof())
  {
    input>>S;        // Читает из файла очередной символ
    array[i][j]=S;  // Заносит символ в матрицу
    j++;
    if (j==strlen(Key_word))
    {
      i++;
      j=0;
    }
  }
  array[i][j-1]=' ';
  // Запускает функцию, которая зашифрует исходное сообщение
  Coder(strlen(Key_word),array);
  getch();           // Задерживает экран
  return 0;          // Завершает программу
}
// Описание функции, находящей номер столбца матрицы,
// в котором есть символ с наименьшим ASCII-кодом
int min(int Len_key, char *array[])
{
  int i, min_A=256, min_pos;
  for (i=0; i<Len_key; i++)
    if (int(array[0][i])<min_A && array[0][i]!=' ')
```

```
    {
      min_A=int(array[0][i]);
      min_pos=i;
    }
  array[0][min_pos]=' ';     // Определяет символ
                             // как просмотренный
  return min_pos;            // Возвращает номер столбца
}
void Coder(int Len_key, char *array[]) // Описание функции
{
  int k, j;
  j=min(Len_key,array);  // Возвращает номер столбца матрицы,
       // в котором находится символ с наименьшим ASCII-кодом
  for (k=1; k<Len_key; k++)
    if (array[k][j]!=' ') cout<<array[k][j];
  if (j!=Len_key) Coder(Len_key,array);
}
```

# 10.4. Подстановочные шифры

В подстановочных шифрах буквы исходного сообщения заменяются на некоторые другие символы, причем последние могут быть получены по определенному алгоритму (далее мы рассмотрим несколько способов). Замены в криптотексте расположены в том же порядке, что и в оригинале. Если использование замен постоянно на протяжении всего текста, то криптосистема называется *одноалфавитной* (моноалфавитной). В *многоалфавитных* системах использование подстановок меняется в различных частях текста.

## Шифр Полибия

Возможно, что наиболее древним из известных подстановочных шифров является шифр греческого историка Полибия.

Рассмотрим прямоугольник, часто называемый доской Полибия (рис. 10.3).

| | а | б | в | г | д | е |
|---|---|---|---|---|---|---|
| а | а | б | в | г | д | е |
| б | ж | з | и | й | к | л |
| в | м | н | о | п | р | с |
| г | т | у | ф | х | ц | ч |
| д | ш | щ | ъ | ы | ь | э |
| е | ю | я | . | , | – | |

**Рис. 10.3.** Доска Полибия

Каждая буква может быть представлена парой букв, указывающих строку и столбец, в которых расположена данная буква. То есть буква `р` находится на пересечении столбца, которому соответствует буква `д`, и строки, которой соответствует буква `в`. Таким образом, `р=вд`.

После того как вы зашифруете текст

```
шифр полибия
```

получите следующий результат:

```
дабвгввдеевгбебваббвеб
```

**Примечание**

На доске Полибия в первой строке и в первом столбце могут располагаться любые символы. В качестве алфавита можно также использовать любые символы.

При реализации алгоритма Полибия в качестве алфавита возьмем все символы, ASCII-код которых попадает в промежуток [44; 124], т. к. они являются наиболее часто используемыми. Вот эти символы:

```
, – . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ? @ A B C D E F G H I J
K L M N O P Q R S T V U V W X Y Z [ \ ] ^ _ ` a b c d e f g h
I j k l m n o p q r s t v u w x y z
```

Чтобы на каждом шаге не искать местоположение символа из исходной строки в матрице, заведем дополнительный массив `ASCII_Code`, состоящий из двух строк. В первой будут распола-

гаться номера строк матрицы, в которых находятся те или иные символы, а во второй строке, соответственно, номера столбцов.

Таким образом, если матрица имеет вид, как показано на рис. 10.4, то в массиве `ASCII_Code` ячейки заполняются следующим образом:

- в ячейке [0][ASCII код символа `x`] устанавливаем единицу, при этом ячейка [1][ASCII код символа `x`] также содержит единицу;
- в ячейке [0][ASCII код символа `y`] устанавливаем единицу, при этом ячейка [1][ASCII код символа `y`] содержит двойку;
- в ячейке [0][ASCII код символа `z`] устанавливаем двойку, при этом ячейка [1][ASCII код символа `z`] содержит единицу;
- в ячейке [0][ASCII код символа `c`] устанавливаем двойку, при этом ячейка [1][ASCII код символа `c`] также содержит двойку.

| | a | b |
|---|---|---|
| a | x | y |
| b | z | c |

**Рис. 10.4.** Пример образования массива `ASCII_Code`

Далее приведен код программы, в которой реализован данный алгоритм:

```
#include <iostream.h>
#include <conio.h>
#include <fstream.h>
void Coder(char *[]);     // Прототип функции
int ASCII_Code[2][125];   // Так как значение ASCII-кода
                          // не превышает 124
ifstream input("input.txt");
main()
{
  int k, i, j, Symbol;
  char **array;
  int ASCII;
```

```
  // Выделяет динамическую память для матрицы размером 10*10.
  // Так как это достаточный размер матрицы, в которую
  // может вместиться 124-44=80 элементов (границы выбранного
  // отрезка ASCII-значений) плюс нулевой столбец и строка.
  // Итого ровно 100
  array=new char *[10];
  for (int i=0; i<100; i++)
    array[i]=new char[10];
  // Задает начальное значение ячеек
  for (j=0; j<10; j++)
    for (i=0; i<10; i++)
      array[j][i]=' ';
  ASCII=43;
  // Заполняет матрицу символами
  for (i=1; i<10; i++)
    for (j=1; j<10; j++)
    {
      array[i][j]=char(ASCII++);
      ASCII_Code[0][ASCII-1]=i;
      ASCII_Code[1][ASCII-1]=j;
    }
  Symbol=97; // 97 — ASCII-код символа "a", с него мы и начнем
        // заполнять нулевую строку и нулевой столбец матрицы
  // Заполняет нулевую строку и нулевой столбец матрицы
  for (j=1; j<10; j++)
  {
    array[0][j]=char(Symbol);
    array[j][0]=char(Symbol);
    Symbol++;
  }
  Coder(array);  // Запускает функцию, которая шифрует
                 // исходное сообщение
  getch();       // Задерживает экран
  return 0;      // Завершает программу
}
```

```
void Coder(char *array[])  // Описание функции
{
  char S;
  int i, j;
  while (!input.eof())
  {
    input>>S;    // Считывает из файла очередной символ
    cout<<array[ASCII_Code[0][S]][0];
    cout<<array[ASCII_Code[1][S]][0];
  }
}
```

# 10.5. Шифрование сдвигом ASCII-значений

Примером шифрования может служить сдвигание каждой буквы сообщения на *K* позиций.

Рассмотрим код данной программы при *K* = 3.

```
#include <fstream.h>
#include <iostream.h>
#include <conio.h>
char array[80];  // Размер массива равен 80, т. к. из
                 // файла мы будем считывать по 80 символов
int length;
void code();     // Прототип функции
main()
{
  ifstream input;  // Объявление файла
  int i;
  input.open("test.txt");  // Открытие файла для чтения данных
  while (!input.eof())     // Цикл продолжает работу, пока не
                           // конец файла
  {
    for (i=0; i<80 && !input.eof(); i++) // Считывает из
                                         // файла 80 символов
```

```
      input.get(array[i]);
    length=i-1;
    code();       // Кодирует символы
  }
  input.close();  // Закрывает файл
  getch();        // Задерживает экран
  return 0;       // Завершает программу
}
void code()      // Описание функции
{
  int i;
  char temp;
  for (i=0; i<length; i++)
  {
    temp=array[i];  // Берет очередной элемент массива
    temp+=3;        // Прибавляет 3 к ASCII-значению символа
    cout<<temp;     // Выводит закодированный символ
  }
}
```

Рассмотрим функцию `code()`, которая кодирует символы:

```
void code()
{
  int i;
  char temp;
  for (i=0; i<length; i++)
  {
    temp=array[i];
    temp+=3;
    cout<<temp;
  }
}
```

Цикл `for` перебирает все элементы массива, которые были получены при считывании из файла, и к ASCII-значению каждого из них прибавляет 3.

Функция `main()` открывает файл, считывает из него символы и передает их для шифрования в функцию `code()`. В конце работы функции файл закрывается.

```
main()
{
  ifstream input;
  int i;
  input.open("test.txt");
  while (!input.eof())
  {
    for (i=0; i<80 && !input.eof(); i++)
      input.get(array[i]);
    length=i-1;
    code();
  }
  input.close();
  getch();
  return 0;
}
```

В переменную `length` записывает количество записанных в массив символов. Это на случай, если в файле оставалось не 80 символов, а, к примеру, 35. Тогда надо шифровать не все элементы массива, а лишь 35.

# 10.6. Криптосистема RSA

Криптосистема RSA была разработана в 1977 году и получила название в честь ее создателей — Рона Райвеста (Ron Rivest), Ади Шамира (Adi Shamir) и Леонарда Эдлмана (Leonard Adleman). Они воспользовались тем фактом, что нахождение больших простых чисел в вычислительном отношении осуществляется легко, но разложение на множители произведения двух таких чисел практически невыполнимо. Доказано, что раскрытие шифра RSA эквивалентно такому разложению. Поэтому для любой длины ключа можно дать нижнюю оценку числа операций

для раскрытия шифра, а с учетом производительности современных компьютеров оценить и необходимое на это время. Возможность гарантированно оценить защищенность алгоритма RSA стала одной из причин популярности этой криптосистемы на фоне десятков других схем. Поэтому алгоритм RSA используется в банковских компьютерных сетях, особенно для работы с удаленными клиентами.

Криптосистема RSA является криптосистемой с открытым ключом. Криптосистемы с открытым ключом позволяют обмениваться секретными сообщениями по открытому каналу, не договариваясь заранее о ключе шифра. Даже перехватив весь разговор от начала до конца, злоумышленник не узнает секретного сообщения. Понятно, что такие методы находят широкое применение в банках, при подписывании контрактов, при денежных переводах.

## Математические идеи алгоритма

Чтобы применить алгоритм RSA, необходимо выполнить следующие математические преобразования:

- получить два больших простых числа, $Num_1$ и $Num_2$, например, состоящих не менее чем из 300 цифр;
- вычислить значение $N$ (открытый модуль) по формуле: $N = Num_1 \times Num_2$ (для решения данной задачи используется алгоритм умножения арифметики больших чисел);
- выбрать значение $E$ (открытая степень), такое, что $E < N$ и является взаимно простым с $(Num_1 - 1) \times (Num_2 - 1)$;
- вычислить значение $D$ (закрытая степень), такое, что

$$E \times D = \mathrm{mod}(Num_1 - 1) \times (Num_2 - 1).$$

  Это можно сделать с помощью алгоритма Евклида.

Пара чисел ($E$, $N$) публикуется как открытый ключ, а число $D$ хранится в секрете — это и есть закрытый ключ, который позволит читать все сообщения, зашифрованные с помощью пары чисел ($E$, $N$).

## Алгоритм шифрования

Алгоритм шифрования при помощи криптосистемы RSA состоит из нескольких этапов.

1. Отправитель разбивает свое сообщение на блоки длиной $K = [\log_2(N)]$ битов, где квадратные скобки обозначают целую часть числа. Подобный блок может быть интерпретирован как число из диапазона $(0; 2^K - 1)$.
2. Для каждого такого числа (назовем его $L_i$) вычисляется выражение $C_i = (L_i^E) \bmod N$. Блоки $C_i$ представляют собой зашифрованное сообщение. Его можно передавать по открытому каналу, поскольку операция возведения в степень по модулю простого числа является необратимой математической задачей. Обратная ей задача носит название "логарифмирование в конечном поле" и является намного более сложной задачей. То есть даже если злоумышленник знает числа $E$ и $N$, то по известному $C_i$ прочесть $L_i$ он не сможет никак, кроме как последовательным перебором по всему множеству исходных сообщений или закрытых ключей.

## Алгоритм дешифрования

В свое время Эйлером была доказана теорема, частный случай которой утверждает, что если число $N$ представимо в виде произведения двух простых чисел $Num_1$ и $Num_2$, то для любого $x$ имеет место равенство:

$$\left(x^{(Num_1-1)(Num_2-1)}\right) \bmod N = 1.$$

Для дешифрования RSA-сообщений воспользуемся этой формулой. Возведем обе ее части в степень $Y$:

$$\left(x^{Y(Num_1-1)(Num_2-1)}\right) \bmod N = 1^Y \ .$$

Теперь умножим обе ее части на $x$:

$$\left(x^{Y(Num_1-1)(Num_2-1)+1}\right) \bmod N = x$$

Поскольку $E \times D = \bmod(Num_1 - 1) \times (Num_2 - 1)$, это означает, что

$$E \times D = Y \times (Num_1 - 1) \times (Num_2 - 1) + 1.$$

Следовательно, в выражении $\left(\right.^{Y(Num_1-1)(Num_2-1)+1}\left.\right) \bmod N = x$ мы можем заменить показатель степени на число $E \times D$. Получаем:

$$\left(\right.^{E\times D}\left.\right) \bmod N = x.$$

То есть чтобы прочесть сообщение $C_i = (L_i^E) \bmod N$, достаточно возвести его в степень $D$ по модулю $N$: $\left(\right.C_i^D\left.\right) \bmod N = \left(\right.^{E\times D}_i\left.\right) \bmod N = L_i$.

Операции возведения в степень больших чисел достаточно трудоемки для современных процессоров, даже если они производятся по оптимизированным по времени алгоритмам. Поэтому обычно весь текст сообщения кодируется обычным блочным шифром, но с использованием цифровой подписи, которая шифруется асимметричным алгоритмом с помощью открытого ключа получателя и помещается в начало файла.

## Пример использования алгоритма RSA

В реальных приложениях при шифровании с помощью RSA используют большие целые числа, порядка 200—400 десятичных цифр. Рассмотрим следующий пример с небольшими числами, чтобы только проиллюстрировать схему алгоритма:

1. Пусть $Num_1 = 11$, а $Num_2 = 13$, тогда $N = 11 \times 13 = 143$.
2. Выберем число $E$ (открытая степень). Оно должно быть взаимно простым с $(Num_1 - 1) \times (Num_2 - 1) = (11 - 1) \times (13 - 1) = 10 \times 12 = 120$. Пусть $E = 113$.
3. Вычислим закрытую степень $D$, воспользовавшись равенством

$$E \times D = \bmod (Num_1 - 1) \times (Num_2 - 1).$$

4. При помощи алгоритма Евклида найдем, что $D = 17$.

Пара (143, 113) составляет открытый ключ, число 17 — закрытый (секретный) ключ. Отображение $E$ (шифрование) состоит в возведении исходного сообщения в степень 113 по модулю 143, отображение $D$ (дешифрование) заключается в возведении зашифрованного сообщения в степень 17 по модулю 143.

Рассмотрим произвольное сообщение `Text=123`. Тогда

$$E(123) = 123^{113} \bmod 143 = 41.$$

Таким образом, 41 — это зашифрованное сообщение. Применим к нему процедуру дешифрования:

$$D(41) = 41^{17} \bmod 143 = 123.$$

В итоге получено исходное сообщение.

Число $123^{113}$ имеет чуть более 200 знаков (т. к. $100^{100} = (10^2)^{100} = 10^{200}$), поэтому вполне может быть вычислено с помощью рассмотренной ранее арифметики больших чисел, хотя на практике применяются алгоритмы, очень значительно экономящие вычислительную сложность подобных операций.

# 10.7. Цифровые подписи

Некоторые из асимметричных алгоритмов могут использоваться для генерирования цифровой подписи. *Цифровой подписью* называют блок данных, сгенерированный с использованием некоторого секретного ключа. При этом с помощью открытого ключа можно проверить, что данные были действительно сгенерированы с помощью этого секретного ключа. Алгоритм генерации цифровой подписи должен обеспечить, чтобы было невозможно без секретного ключа создать подпись, которая при проверке окажется правильной.

Цифровые подписи используются для того, чтобы подтвердить, что сообщение пришло действительно от данного отправителя (в предположении, что лишь отправитель обладает секретным ключом, соответствующим его открытому ключу). Кроме того, подписи используются для проставления штампа времени (Time Stamp) на документах: сторона, которой мы доверяем, подписывает документ со штампом времени с помощью своего секретного ключа и, таким образом, подтверждает, что документ уже существовал в момент, объявленный в штампе времени.

Цифровые подписи также можно применять для удостоверения того, что документ принадлежит определенному лицу. Это дела-

ется следующим образом: открытый ключ и информация о том, кому он принадлежит, подписываются стороной, которой доверяют. При этом доверять подписывающей стороне можно на основании того, что ее ключ был подписан третьей стороной. Таким образом, возникает иерархия доверия. Очевидно, что некоторый ключ должен быть корнем иерархии. В распределенной инфраструктуре нет необходимости иметь универсальные для всех корневые ключи, и каждая из сторон может доверять своему набору корневых ключей (скажем, своему собственному ключу и ключам, ею подписанным). Эта концепция носит название Сети доверия (Web Of Trust) и реализована, например, в PGP (программе для шифрования данных).

Цифровая подпись документа обычно создается следующим образом: из документа генерируется так называемый дайджест (Message Digest), и к нему добавляется информация о том, кто подписывает документ, штамп времени и пр. Получившаяся строка далее зашифровывается секретным ключом подписывающего с использованием того или иного алгоритма. Получившийся зашифрованный набор битов представляет собой подпись. К подписи обычно прикладывается открытый ключ подписывающего. Получатель сначала решает для себя, верит ли он, что открытый ключ принадлежит именно тому, кому должен принадлежать (с помощью Сети доверия или априорного знания), затем дешифрует подпись с помощью открытого ключа. Если подпись успешно дешифровалась, и ее содержимое соответствует документу (дайджесту и др.), то сообщение считается подтвержденным.

# Предметный указатель