# Оглавление

# 1.Основные свойства и конструкции языка программирования Ruby. Базовые встроенные классы и методы. Библиотека Test::Unit и тестирование программ.

## *1.1. Свойства*

Интерпретируемый язык:

- Возможность прямых системных вызовов.
- Мощная поддержка операций со строками и правилами (регулярными выражениями).
- Мгновенное проявление изменений во время разработки.
- Отсутствие стадии компиляции.

Простое и быстрое программирование:

- Не надо объявлять переменные.
- Переменные динамически типизированы.
- Простой и последовательный синтаксис.
- Автоматическое управление оперативной памятью.

Объектно-ориентированное программирование:

- Всё есть объект. Даже имя класса есть экземпляр класса `Class`.
- Классы, методы, наследование, полиморфизм, инкапсуляция и так далее.
- Методы-одиночки.
- Примеси при помощи модулей (возможность расширить класс без наследования);
- Итераторы и замыкания.
- Широкие возможности метапрограммирования.

Удобства:

- Неограниченный диапазон значений целых чисел.
- Модель обработки исключений.
- Все операторы возвращают значения, даже управляющие структуры.
- Динамическая загрузка.
- Механизм перехвата исключений.
- Поддержка потоков; как собственных, так и систем семейства UNIX.

Недостатки:

- Неуправляемость некоторых процессов (таких, как выделение памяти), невозможность задания низкоуровневых структур данных или подпрограмм;
- Невозможность компиляции и сопутствующей ей оптимизации программы;
- Открытость исходного кода даже в готовой программе (есть средство упаковки исходного кода в .exe-файл под Windows);
- Следствие двух первых недостатков — весьма низкая скорость запуска и выполнения программ.

## *1.2. Конструкции*

## Контейнеры

Работа с массивами — одна из сильных сторон Руби. Они автоматически изменяют размер, могут содержать любые элементы и язык предоставляет мощные средства для их обработки.

```
# создаём массив
a = [1, 'hi', 3.14, 1, 2, [4, 5] * 3]
                    # => [1, "hi", 3.14, 1, 2, [4, 5, 4, 5, 4, 5]]
                    # обращение по индексу
a[2]                  # => 3.14
                    # «разворачиваем» все внутренние массивы, удаляем одинаковые элементы
a.flatten.uniq          # => [1, 'hi', 3.14, 2, 4, 5]
                    # пытаемся найти индекс элемента со значением 4
a.index(4)            # => nil
                    # предыдущая попытка найти элемент неудачна - все предыдущие функции
                    # возвращают копии, но Руби почти для всех функций предоставляется аналог
                    # с тем же названием, но заканчивающийся на «!»,
                    # который модифицирует контейнер
a.flatten!          # => [1, "hi", 3.14, 1, 2, 4, 5, 4, 5, 4, 5]
a.index(4)            # => 5
```

## Процедурные объекты и итераторы

В языке есть 2 эквивалентных способа записи блоков кода:

```
{ puts "Hello, World!" }

do puts "Hello, World!" end
```

Сопрограммы применяются с большинством встроенных методов:

```
File.open('file.txt', 'w') { |file| # открытие файла «file.txt» для записи («w» - write)
  file.puts 'Wrote some text.'
} # Конструкция устраняет неопределённость с закрытием файла: закрывается здесь при любом исходе
```

Следующий пример показывает использование сопрограмм и итераторов для работы с массивами, который показывает краткость записи на Руби многих достаточно сложных действий (случайно выбираем из последовательности квадратов чисел от «0» до «10» и распечатываем вместе с индексами):

```
(0..10).collect{|v| v ** 2 }.select{ rand(2) == 1 }.each_with_index{|v,i| printf "%2d\t%2d\n", i, v }
```

## Классы, примеси, методы, перегрузка операторов

Следующий пример определяет класс с именем `Person`, предназначенный для хранения информации о имени и возрасте некоторой персоны.

```
class Person                # объявление класса начинается с ключевого слова class, за которым
                            # следует имя
  include Comparable        # [[b:Ruby/Справочник/Comparable|Comparable]] подмешивается к классу и добавляет
                            # методы <, <=, ==, >=, > и between?
                            # с использованием нижеопределённого
                            # в классе <=>
                            #
  @@count_obj = 0           # переменная класса для подсчёта числа созданных объектов
                            #
                            # конструктор для создания объектов с помощью new
  def initialize(name, age) # name, age - параметры метода
                            # название переменных объекта начинается с @
    @name, @age = name, age # создаём объекты и увеличиваем счётчик на 1
    @@count_obj += 1
  end

  def <=>(person)           # переопределение оператора <=>
                            # (это даёт возможность использовать метод sort
    @age <=> person.age     # из метода возвращается последнее вычисленное выражение,
  end

  def to_s                  # для форматированного вывода информации puts
    "#{@name} (#{@age})"    # конструкция #{x} в 2-х кавычках замещается в Руби значением x
  end

  def inspect               # похож на to_s, но используется для диагностического вывода
    "<#{@@count_obj}:#{to_s}>"
  end
                            # пример метапрограммирования: добавляет методы для доступа к
                            # переменным объекта
  attr_reader :name, :age
end
                            # создаём массив объектов
group = [ Person.new("John", 20),
          Person.new("Markus", 63),
          Person.new("Ash", 16) ]
                  # => [<3:John (20)>, <3:Markus (63)>, <3:Ash (16)>]
                  # здесь при работе с irb автоматически вызывается метод inspect
```

```
                # вызываем методы массива сортировка и разворачивание его в обратном порядке
puts group.sort.reverse # Печатает:
                # Markus (63)
                # John (20)
                # Ash (16)
                        # обращаемся к функции, которая была добавлена
                        # автоматически(используя <=>) при включении Comparable
group[0].between?(group[2], group[1]) # => true
```

### *1.3.Test::Unit*

**Test::Unit** является классическим средством тестирования в стиле xUnit. В Rails-приложениях именно он и используется по умолчанию. Набор тестов с его использованием описывается в виде класса, методы которого представляют различные тесты. В коде методов в необходимых точках добавляются проверки, представленные вызовами `assert_*`.

/media/CORSAIR/Gos/методы классы и т.п.pdf

# 2. Понятия объекта, класса, метода. Наследование и полиморфизм. Использование объектно-ориентированного подхода при проектировании и разработке программ на языке Ruby.

/media/CORSAIR/Gos/методы классы и т.п.pdf

# 3. Использование предикатов для проектирования и документирования программ. Итерация и рекурсия. Проектирование цикла при помощи инварианта. Схема вычисления инвариантной функции.

Все знают, что в программах бывают *ошибки (bugs)*. Существуют специальные теории, посвященные тому, как лучше их находить и исправлять (*debugging* дословно означает <<выведение клопов>>). Зачастую нахождение ошибки -- очень нетривиальная задача, так как ее последствия могут сказываться совершенно в другом месте программы и быть весьма неожиданными.

При этом часто забывается тот очевидный факт, что ошибку гораздо легче *предотвратить* при написании программы, нежели найти и исправить потом. Существуют методы проектирования программ (по крайней мере, небольших), позволяющие не только написать правильную программу, но и получить одновременно с этим совершенно строгое доказательство ее правильности. Изучению таких методов и будет посвящена в основном вторая глава данной книги.

Однако для того, чтобы изучить какую-либо теорию, необходимо выучить язык, на котором теория может быть изложена. *Язык предикатов* -- это именно тот язык, на котором можно строго сформулировать постановку задачи и доказать правильность конкретной программы.

Так как нам предикаты нужны прежде всего для описания программ, введем следующее определение.

**Определение** *Высказывание* или *предикат* -- это функция, действующая из некоторого множества значений переменных программы (идентификаторов) в множество из двух значений { $T$ , $F$ } (*Да* и *Нет*).

В соответствии с ним предикатами будут следующие фразы:

- значение переменной `i` равно двум;
- переменная `k` положительна, а значение переменной `m` при этом не превосходит 100;
- значения всех целочисленных переменных программы являются нулевыми.
- неправда, что значение переменной `i` неположительно.

Чуть менее понятно, можно ли считать предикатами такие фразы:

- если предположить, что значение переменной `i` равно двум, то значения всех остальных целочисленных переменных программы будут неотрицательными;
- данная программа является правильной;

- данное высказывание ложно.

Особенно показательным является последний пример. Если мы согласимся с тем, что он представляет из себя предикат, то возникает естественный вопрос о его истинности. Предположение о том, что он истинен, заставляет считать его ложным, и наоборот. Получаемое противоречие говорит о том, что определение 3.1 не является достаточно корректным.

Ситуация с предикатами напоминает уже обсуждавшуюся нами ситуацию с языками для записи алгоритмов -- необходим какой-то четкий критерий, позволяющий однозначно определить, что является предикатом, а что нет.

В теории формальных языков, более детальное знакомство с которой состоится позже, принято задавать язык с помощью грамматики. Грамматику же достаточно часто определяют с помощью так называемой нормальной формы Бэкуса-Наура (НФБН). Вот все необходимые общие определения.

**Определение** *Алфавит* $\Sigma$ -- произвольное непустое множество.

Мы будем иметь дело только с конечными алфавитами, примерами которых являются алфавит русского языка, английский алфавит, множество цифр, алфавит всех символов, имеющихся на клавиатуре компьютера.

**Определение** *Символом алфавита* $\Sigma$ называют любой его элемент, а *цепочкой над алфавитом* -- произвольную последовательность символов $\omega$.

Цепочки часто называют также словами, фразами и предложениями. Пустая цепочка обозначается специальным символом $\varepsilon$, а множество всех цепочек над алфавитом $\Sigma$ принято обозначать $\Sigma^*$. Если в качестве $\Sigma$ взять множество букв русского алфавита, дополненное символом пробела и знаками пунктуации, то в $\Sigma^*$ будут содержаться все фразы русского языка.

**Определение** *Длиной* $|\omega|$ цепочки $\omega \in \Sigma^*$ называется количество входящих в нее символов.

Длина пустой цепочки равна нулю, а длины всех остальных цепочек над любым алфавитом положительны.

**Определение** Операция $\circ\colon \Sigma^* \times \Sigma^* \to \Sigma^*$ *конкатенации* двух цепочек определена следующем образом. Пусть $\omega_1 = a_1 a_2 \ldots a_n$, $\omega_2 = b_1 b_2 \ldots b_m$, тогда $\omega_1 \circ w_2 = a_1 a_2 \ldots a_n b_1 b_2 \ldots b_m$.

Операция конкатенации, называемая также операцией сцепления или дописывания, обладает следующими свойствами.

**Предложение** Для любых цепочек $\omega$, $\omega_1$, $\omega_2$ и $\omega_3$ справедливы следующие равенства:
1) $\varepsilon \circ \omega = \omega \circ \varepsilon = \omega$,
2) $(\omega_1 \circ \omega_2) \circ \omega_3 = \omega_1 \circ (\omega_2 \circ \omega_3)$.

Имея все эти определения, уже можно дать формальное определения *языка над алфавитом* $\Sigma^*$.

**Определение** *Язык* $L$ -- это произвольное подмножество множества цепочек $\Sigma^*$.

Если язык конечен, то есть состоит из конечного множества входящих в него цепочек, то его можно задать, просто перечислив все его элементы. Для бесконечных языков, которые чаще всего и представляют наибольший интерес, такой способ не годится. Достаточно часто для задания языка используют грамматику, несколько упрощенное определение которой мы сейчас рассмотрим.

**Определение** Пусть $\Sigma$ -- некоторый алфавит, $N$ -- метаалфавит, т.е. какой-то другой алфавит, не пересекающийся с $\Sigma$ ($\Sigma \cap N = \varnothing$). Элементы метаалфавита $N$ называются *метасимволами*. *Грамматикой* $G$ называется набор ($\Sigma, N, P, S$), где $\Sigma$ -- множество

символов, $N$ -- множество метасимволов, $P$ -- множество правил вывода вида: $\alpha \to \beta$, где $\alpha \in N$ -- какой-то метасимвол, $\beta \in (\Sigma \cup N)^*$ -- произвольная цепочка над объединением двух алфавитов, и для каждого $\alpha \in N$ встречается хотя бы одно правило с $\alpha$ в левой части (до стрелочки), а $S \in N$ -- так называемый *стартовый метасимвол*.

Содержательно каждое правило грамматики имеет смысл подстановки. Например, строка $\alpha \to \alpha\gamma\alpha$ означает возможность замены метасимвола $\alpha$ на цепочку $\alpha\gamma\alpha$. Начав со стартового символа и пользуясь различными правилами грамматики, мы можем получать различные цепочки из символов, которые называются *выводимыми цепочками*.

Заметим, что если в цепочке встречается метасимвол, то ее можно преобразовать дальше, применив одно из правил грамматики с этим метасимволом в левой части. Если же метасимволов в в цепочке не осталось, то процесс ее преобразования закончен и больше с цепочкой ничего сделать нельзя. По этой причине обычные символы (из алфавита $\Sigma$) часто называют *терминалами*, а метасимволы (из $N$) -- *нетерминалами*.

**Определение** *Языком* $L(G)$, порожденным грамматикой $G$, называется множество всех терминальных выводимых цепочек.

Для задания грамматики часто используют очень наглядную форму представления, называемую *нормальной формой Бэкуса-Наура (НФБН)*. Набор правил $P$ задают при этом в виде совокупности правил со стрелочками, перечисляющими все возможные цепочки, на которые может быть заменен каждый из метасимволов грамматики в процессе вывода, а стартовым метасимволом считается тот, который присутствует в левой части самого первого правила.

В качестве примера дадим строгое определение языка предикатов, или, как принято еще говорить, зададим *синтаксис* этого языка.

**Определение** *Множество предикатов* -- это язык, порожденный следующей грамматикой:

$$\begin{array}{lll} \overset{e}{\to} & T & \text{(1-е правило: истина)} \\ | & F & \text{(2-е правило: ложь)} \\ | & id & \text{(3-е правило: идентификатор)} \\ | & (!e) & \text{(4-е правило: отрицание)} \\ | & (e \vee e) & \text{(5-е правило: дизъюнкция)} \\ | & (e \| e) & \text{(6-е правило: условное } \textit{Или}\text{)} \\ & & \text{(7-е правило: конъюнкция)} \\ & & \text{(8-е правило: условное } \textit{И}\text{)} \\ & & \text{(9-е правило: импликация)} \\ & & \text{(10-е правило: эквивалентность)} \end{array}$$

## *3.1. Итерация и рекурсия*

*Рекурсия* -- это такой способ организации обработки данных, при котором программа вызывает сама себя непосредственно, либо с помощью других программ.
*Итерация* -- способ организации обработки данных, при котором определенные действия повторяются многократно, не приводя при этом к рекурсивным вызовам программ.
*Математическая модель рекурсии* заключается в вычислении рекурсивно определенной функции на множестве программных переменных. Примерами таких функций могут служить факториал числа и числа Фибоначчи. В каждом из этих случаев значение функции для всех значений аргумента, начиная с некоторого, определяется через предыдущие значения.

*Математическая модель итерации* сводится к повторению некоторого преобразования (отображения) на множестве переменных программы (прямом произведении множеств значений отдельных переменных). Программной реализацией итерации является обычно некоторый цикл, тело которого осуществляет преобразование .

Любой алгоритм, реализованный в рекурсивной форме, может быть переписан в итерационном виде, и наоборот.

## *3.2. Схема вычисления инвариантной функции*

Общая схема итерации значительно упрощается для случая вычисления значений инвариантных функций.

Пусть -- некоторое множество, -- заданная на нем функция, а -- предикат такой, что *легко вычислить*). Обозначим через то подмножество множества , где . Если существует преобразование такое, что , то функция называется *-инвариантной* или просто *инвариантной функцией*.

Простейшим примером инвариантной функции является хорошо известная еще из средней школы функция , . Она является -инвариантной относительно преобразования , .

Наибольший общий делитель двух целых чисел (greatest common divisor, или просто НОД(x,y)) инвариантен относительно преобразования , задаваемого формулой

Для вычисления значения -инвариантной функции в точке применяется следующая схема.

Схема вычисления инвариантной функции. *Многократно выполняется преобразование , дающее последовательность точек Если очередная точка попадает в подмножество , то итерации завершаются. По определению инвариантной функции легко вычисляется и совпадает с искомым* .

Рисунок 4: Схема вычисления инвариантной функции

# 4. Индуктивные функции на пространстве последовательностей. Построение индуктивных расширений. Понятие о минимальном индуктивном расширении.

Еще одной ситуацией, в которой общая схема итерации значительно упрощается, является задача вычисления индуктивных функций. Такие функции определены на последовательностях элементов из некоторого алфавита . Напомним важнейшие из определений [3].

*Алфавит* -- произвольное непустое множество.

*Символом алфавита* $X$ называют любой его элемент, а *цепочкой над алфавитом* -- произвольную последовательность символов $\omega$. Цепочки часто называют также словами, фразами и предложениями. Пустая цепочка обозначается специальным символом $\varepsilon$, а множество всех цепочек над алфавитом $X$ принято обозначать $X^*$.

*Длиной* $|\omega|$ цепочки $\omega \in X^*$ называется количество входящих в нее символов.

Множество всех цепочек *длины не менее* $k$ обозначают через $X_k^*$. Справедлива следующая последовательность включений:

$$X^* \supset X_1^* \supset X_2^* \supset X_3^* \supset \dots$$

Операция $\circ\colon X^* \times X^* \to X^*$ *конкатенации* (или *сцепления*) двух цепочек $\omega_1 = a_1 a_2 \dots a_n$, $\omega_2 = b_1 b_2 \dots b_m$ определена следующем образом. Пусть $\omega_1 \circ w_2 = a_1 a_2 \dots a_n b_1 b_2 \dots b_m$ тогда .

Теперь можно дать определение индуктивной функции.

**Определение** Функция $f\colon X^* \to Y$ называется *индуктивной*, если $f(\omega \circ x)$ можно вычислить, зная и , т.е. если такое, что .

Одним из простейших примеров индуктивной функции является функция *длина цепочки* $|\omega|\colon X^* \to \mathbb{Z}^+$. Она индуктивна, так как для нее существует функция $G\colon \mathbb{Z}^+ \times X \to \mathbb{Z}^+$, определенная формулой $G(y, x) = y + 1$, удовлетворяющая предыдущему определению.

Для вычисления значения $f(\omega)$ индуктивной функции $f$ на цепочке $\omega = a_1 a_2 \dots a_n$ применяется следующая схема.

Схема вычисления индуктивной функции. *Рассматривается последовательность цепочек* $\varepsilon$, $a_1$, $a_1 a_2$, $\dots$, $a_1 a_2 \dots a_n = \omega$. *Сначала вычисляется значение функции* $f$ *на пустой цепочке* $\varepsilon$, *а затем используется отображение* $G$, *позволяющее найти значение функции* $f$ *на удлиненной цепочке, что дает возможность последовательно определить все требуемые величины вплоть до* $f(\omega)$.

На рисунке 5 приведена графическая иллюстрация схемы вычисления индуктивной функции.

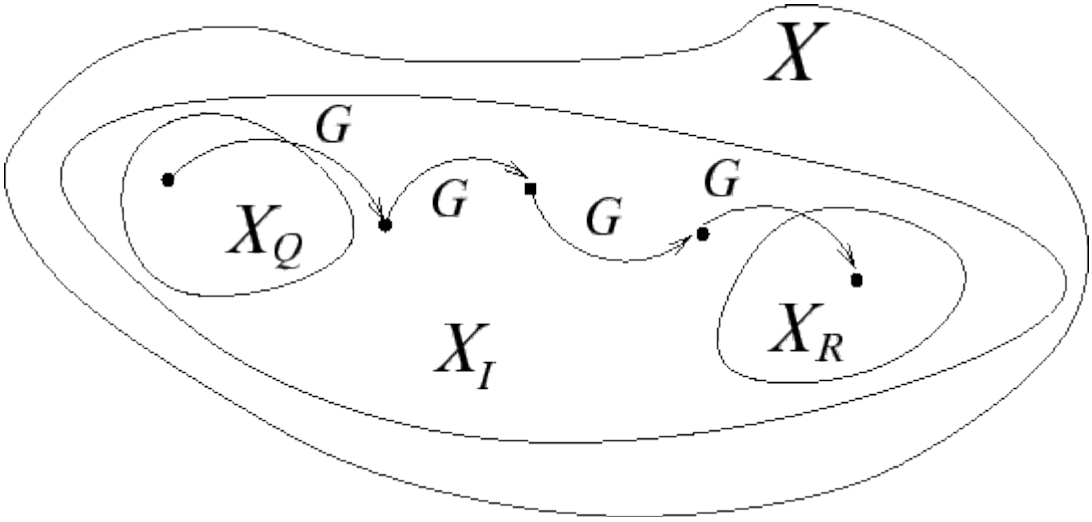



Рисунок 5: Схема вычисления индуктивной функции

Схема вычисления индуктивной функции напоминает метод доказательства по индукции. Аналогом базы индукции является вычисление $f(\varepsilon)$, а индуктивному переходу соответствует вычисление функции $f(\omega \circ x)$ на удлиненной цепочке $\omega \circ x$ с использованием вычисленного на предыдущем шаге значения $f(\omega)$.

### *4.1. Построение индуктивных расширений*

http://www.ctc.msiu.ru/materials/Book/node71.html

# 5. Простейшие структуры данных: вектор, стек, очередь, дек, множество, одно- и дву- связные списки. Непрерывные и ссылочные реализации этих структур данных.

### *Вектор*

Вектор (одномерный массив) - структура данных с фиксированным числом элементов одного и того же типа. Каждый элемент вектора имеет уникальный в рамках заданного вектора номер и имя.

### *Стек*

Стек - такой последовательный список с переменной длиной, включение и исключение элементов из которого выполняются только с одной стороны списка, называемого вершиной стека. Основные операции над стеком - включение нового и исключение элемента из стека. Полезными могут быть также вспомогательные операции: определение текущего числа элементов в стеке и очистка стека.

Стек можно представить, например, в виде стопки книг (элементов), лежащей на столе. Присвоим каждой книге свое название, например A,B,C,D... Тогда в момент времени, когда на столе книг нет, про стек аналогично можно сказать, что он пуст, т.е. не содержит ни одного элемента. Если же мы начнем последовательно класть книги одну на

другую, то получим стопку книг (допустим, из *n* книг), или получим стек, в котором содержится *n* элементов, причем вершиной его будет являться элемент *n+1*. Удаление элементов из стека осуществляется аналогичным образом т. е. удаляется последовательно по одному элементу, начиная с вершины, или по одной книге из стопки.

## *Очереди FIFO*

Очередью FIFO (First In First Out - "первым пришел - первым вышел") называется такой последовательный список с переменной длиной, в котором включение элементов выполняется только с одной стороны списка (конец), а исключение - с другой стороны (начало). Основные операции над очередью - включение, исключение, определение размера, очистка, неразрушающее чтение.

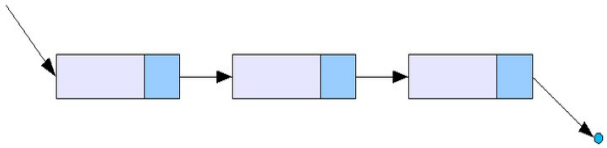
## *Деки*

Дек - особый вид очереди. Дек (от англ. deq - double ended queue, т.е очередь с двумя концами) - это такой последовательный список, в котором как включение, так и исключение элементов может осуществляться с любого из двух концов списка. Операции над деком: включение элемента справа, слева; исключение элемента справа, слева; определение размера; очистка.

## *Списки*

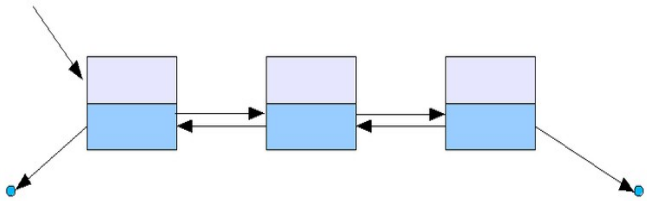
**свя́зный спи́сок** — структура данных, состоящая из узлов, каждый из которых содержит как собственно данные, так и одну или две ссылки («связки») на следующий и/или предыдущий узел списка.[1] Принципиальным преимуществом перед массивом является структурная гибкость: порядок элементов связного списка может не совпадать с порядком расположения элементов данных в памяти компьютера, а порядок обхода списка всегда явно задаётся его внутренними связями.

### Односвязный список (Однонаправленный связный список)



Здесь ссылка в каждом узле указывает на следующий узел в списке. В односвязном списке можно передвигаться только в сторону конца списка. Узнать адрес предыдущего элемента, опираясь на содержимое текущего узла, невозможно.

**Двусвязный список (Двунаправленный связный список)**



Здесь ссылки в каждом узле указывают на предыдущий и на последующий узел в списке. По двусвязному списку можно передвигаться в любом направлении — как к началу, так и к концу. В этом списке проще производить удаление и перестановку элементов, так как всегда известны адреса тех элементов списка, указатели которых направлены на изменяемый элемент.

**Достоинства**

- лёгкость добавления и удаления элементов
- размер ограничен только объёмом памяти компьютера и разрядностью указателей
- динамическое добавление и удаление элементов

**Недостатки**

- сложность определения адреса элемента по его индексу (номеру) в списке
- на поля-указатели (указатели на следующий и предыдущий элемент) расходуется дополнительная память (в массивах, например, указатели не нужны)
- работа со списком медленнее, чем с массивами, так как к любому элементу списка можно обратиться, только пройдя все предшествующие ему элементы
- элементы списка могут быть расположены в памяти разреженно, что окажет негативный эффект на кэширование процессора
- над связными списками гораздо труднее (хотя и в принципе возможно) производить параллельные векторные операции, такие как вычисление суммы
- кэш-промахи при обходе списка

***Ссылочные реализации структур данных***

Большинство структур данных реализуется на базе массива. Все реализации можно разделить на два класса: непрерывные и ссылочные. В непрерывных реализациях элементы структуры данных располагаются последовательно друг за другом в непрерывном отрезке массива, причем порядок их расположения в массиве соответствует их порядку в реализуемой структуре. Рассмотренные выше реализации очереди и стека относятся к непрерывным.

В ссылочных реализациях элементы структуры данных хранятся в произвольном порядке. При этом вместе с каждым элементом хранятся ссылки на один или несколько соседних элементов. В качестве ссылок могут выступать либо индексы ячеек массива, либо адреса памяти. Можно представить себе шпионскую сеть, в которой каждый участник знает лишь координаты одного или двух своих коллег. Контрразведчикам, чтобы обезвредить сеть, нужно пройти последовательно по всей цепочке, начиная с выявленного шпиона.

Ссылочные реализации обладают двумя ярко выраженными недостатками: 1) для хранения ссылок требуется дополнительная память; 2) для доступа к некоторому элементу структуры необходимо сначала добраться до него, проходя последовательно по цепочке других элементов.

Все недостатки ссылочных реализаций компенсируются одним чрезвычайно важным достоинством: в них можно добавлять и удалять элементы в середине структуры данных, не перемещая остальные элементы.

# 6. Внутренние и внешние сортировки. Основные характеристики алгоритмов сортировки. Оценки сложности сортировок выбором, вставками, обменом и слиянием.

## *6.1.* Методы внутренней сортировки

### 6.1.1. Сортировка включением

Одним из наиболее простых и естественных методов внутренней сортировки является сортировка с простыми включениями. Идея алгоритма очень проста. Пусть имеется массив ключей a[1], a[2], ..., a[n]. Для каждого элемента массива, начиная со второго, производится сравнение с элементами с меньшим индексом (элемент a[i] последовательно сравнивается с элементами a[i-1], a[i-2] ...) и до тех пор, пока для очередного элемента a[j] выполняется соотношение a[j] > a[i], a[i] и a[j] меняются местами. Если удается встретить такой элемент a[j], что a[j] <= a[i], или если достигнута нижняя граница массива, производится переход к обработке элемента a[i+1] (пока не будет достигнута верхняя граница массива).

Легко видеть, что в лучшем случае (когда массив уже упорядочен) для выполнения алгоритма с массивом из n элементов потребуется n-1 сравнение и 0 пересылок. В худшем случае (когда массив упорядочен в обратном порядке) потребуется n?(n-1)/2 сравнений и столько же пересылок. Таким образом, можно оценивать сложность метода простых включений как O(n2).

Можно сократить число сравнений, применяемых в методе простых включений, если воспользоваться тем фактом, что при обработке элемента a[i] массива элементы a[1], a[2], ..., a[i-1] уже упорядочены, и воспользоваться для поиска элемента, с которым должна быть произведена перестановка, методом двоичного деления. В этом случае оценка числа требуемых сравнений становится O(n?log n). Заметим, что поскольку при выполнении перестановки требуется сдвижка на один элемент нескольких элементов, то оценка числа пересылок остается O(n2).

### 6.1.2. Обменная сортировка (пузырьковая)

Простая обменная сортировка (в просторечии называемая "методом пузырька") для массива a[1], a[2], ..., a[n] работает следующим образом. Начиная с конца массива сравниваются два соседних элемента (a[n] и a[n-1]). Если выполняется условие a[n-1] > a[n], то значения элементов меняются местами. Процесс продолжается для a[n-1] и a[n-2] и т.д., пока не будет произведено сравнение a[2] и a[1]. Понятно, что после этого на месте a[1] окажется элемент массива с наименьшим значением. На втором шаге процесс повторяется, но последними сравниваются a[3] и a[2]. И так далее. На последнем шаге будут сравниваться только текущие значения a[n] и a[n-1]. Понятна аналогия с пузырьком, поскольку наименьшие элементы (самые "легкие") постепенно "всплывают" к верхней границе массива.

### 6.1.3. Сортировка выбором

При сортировке массива a[1], a[2], ..., a[n] методом простого выбора среди всех элементов находится элемент с наименьшим значением a[i], и a[1] и a[i] обмениваются значениями. Затем этот процесс повторяется для получаемых подмассивов a[2], a[3], ..., a[n], ... a[j], a[j+1], ..., a[n] до тех пор, пока мы не дойдем до подмассива a[n], содержащего к этому моменту наибольшее значение.

### 6.1.4. Сортировка разделением (Quicksort)

Основная идея алгоритма состоит в том, что случайным образом выбирается некоторый элемент массива x, после чего массив просматривается слева, пока не встретится элемент a[i] такой, что a[i] > x, а затем массив просматривается справа, пока не встретится элемент a[j] такой, что a[j] < x. Эти два элемента меняются местами, и процесс просмотра, сравнения и обмена продолжается, пока мы не дойдем до элемента x. В результате массив окажется разбитым на две части - левую, в которой значения ключей будут меньше x, и правую со значениями ключей, большими x. Далее процесс рекурсивно продолжается для левой и правой частей массива до тех пор, пока каждая часть не будет содержать в точности один элемент. Понятно, что как обычно, рекурсию можно заменить итерациями, если запоминать соответствующие индексы массива.

### 6.1.5. Сортировка с помощью дерева (Heapsort)

Начнем с простого метода сортировки с помощью дерева, при использовании которого явно строится двоичное дерево сравнения ключей. Построение дерева начинается с листьев, которые содержат все элементы массива. Из каждой соседней пары выбирается наименьший элемент, и эти элементы образуют следующий (ближе к корню уровень дерева). Из каждой соседней пары выбирается наименьший элемент и т.д., пока не будет построен корень, содержащий наименьший элемент массива. Итак, мы уже имеем наименьшее значение элементов массива. Для того, чтобы получить следующий по величине элемент, спустимся от корня по пути, ведущему к листу с наименьшим значением. В этой листовой вершине проставляется фиктивный ключ с "бесконечно большим" значением, а во все промежуточные узлы, занимавшиеся наименьшим элементом, заносится наименьшее значение из узлов - непосредственных потомков. Процесс продолжается до тех пор, пока все узлы дерева не будут заполнены фиктивными ключами

### 6.1.6. Сортировка со слиянием

Сортировки со слиянием, как правило, применяются в тех случаях, когда требуется отсортировать последовательный файл, не помещающийся целиком в основной памяти. Методам внешней сортировки посвящается следующая часть книги, в которой основное внимание будет уделяться методам минимизации числа обменов с внешней памятью. Однако существуют и эффективные методы внутренней сортировки, основанные на разбиениях и слияниях.

Один из популярных алгоритмов внутренней сортировки со слияниями основан на следующих идеях (для простоты будем считать, что число элементов в массиве, как и в нашем примере, является степенью числа 2). Сначала поясним, что такое слияние. Пусть имеются два отсортированных в порядке возрастания массива p[1], p[2], ..., p[n] и q[1], q[2], ..., q[n] и имеется пустой массив r[1], r[2], ..., r[2?n], который мы хотим заполнить значениями массивов p и q в порядке возрастания. Для слияния выполняются следующие действия: сравниваются p[1] и q[1], и меньшее из значений записывается в r[1]. Предположим, что это значение p[1]. Тогда p[2] сравнивается с q[1] и меньшее из значений заносится в r[2]. Предположим, что это значение q[1]. Тогда на следующем шаге

сравниваются значения p[2] и q[2] и т.д., пока мы не достигнем границ одного из массивов. Тогда остаток другого массива просто дописывается в "хвост" массива r.

## 6.2. Методы внешней сортировки

### *6.2.1. Прямое слияние*

Начнем с того, как можно использовать в качестве метода внешней сортировки алгоритм простого слияния, обсуждавшийся в конце предыдущей части. Предположим, что имеется последовательный файл A, состоящий из записей a1, a2, ..., an (снова для простоты предположим, что n представляет собой степень числа 2). Будем считать, что каждая запись состоит ровно из одного элемента, представляющего собой ключ сортировки. Для сортировки используются два вспомогательных файла B и C (размер каждого из них будет n/2).

Сортировка состоит из последовательности шагов, в каждом из которых выполняется распределение состояния файла A в файлы B и C, а затем слияние файлов B и C в файл A. (Заметим, что процедура слияния для файлов полностью иллюстрируется рисунком 2.14.) На первом шаге для распределения последовательно читается файл A, и записи a1, a3, ..., a(n-1) пишутся в файл B, а записи a2, a4, ..., an - в файл C (начальное распределение). Начальное слияние производится над парами (a1, a2), (a3, a4), ..., (a(n-1), an), и результат записывается в файл A. На втором шаге снова последовательно читается файл A, и в файл B записываются последовательные пары с нечетными номерами, а в файл C - с четными. При слиянии образуются и пишутся в файл A упорядоченные четверки записей. И так далее. Перед выполнением последнего шага файл A будет содержать две упорядоченные подпоследовательности размером n/2 каждая. При распределении первая из них попадет в файл B, а вторая - в файл C. После слияния файл A будет содержать полностью упорядоченную последовательность записей.

### *6.2.2. Естественное слияние*

При использовании метода прямого слияния не принимается во внимание то, что исходный файл может быть частично отсортированным, т.е. содержать упорядоченные подпоследовательности записей. Серией называется подпоследовательность записей ai, a(i+1), ..., aj такая, что ak <= a(k+1) для всех i <= k < j, ai < a(i-1) и aj > a(j+1). Метод естественного слияния основывается на распознавании серий при распределении и их использовании при последующем слиянии.

Как и в случае прямого слияния, сортировка выполняется за несколько шагов, в каждом из которых сначала выполняется распределение файла A по файлам B и C, а потом слияние B и C в файл A. При распределении распознается первая серия записей и переписывается в файл B, вторая - в файл C и т.д. При слиянии первая серия записей файла B сливается с первой серией файла C, вторая серия B со второй серией C и т.д. Если просмотр одного файла заканчивается раньше, чем просмотр другого (по причине разного числа серий), то остаток недопросмотренного файла целиком копируется в конец файла A. Процесс завершается, когда в файле A остается только одна серия.

### *6.2.3. Сбалансированное многопутевое слияние*

В основе метода внешней сортировки сбалансированным многопутевым слиянием является распределение серий исходного файла по m вспомогательным файлам B1, B2, ..., Bm и их слияние в m вспомогательных файлов C1, C2, ..., Cm. На следующем шаге производится слияние файлов C1, C2, ..., Cm в файлы B1, B2, ..., Bm и т.д., пока в B1 или C1 не образуется одна серия.

При использовании рассмотренного выше метода сбалансированной многопутевой внешней сортировки на каждом шаге примерно половина вспомогательных файлов используется для ввода данных и примерно столько же для вывода сливаемых серий. Идея многофазной сортировки состоит в том, что из имеющихся m вспомогательных файлов (m-1) файл служит для ввода сливаемых последовательностей, а один - для вывода образуемых серий. Как только один из файлов ввода становится пустым, его начинают использовать для вывода серий, получаемых при слиянии серий нового набора (m-1) файлов. Таким образом, имеется первый шаг, при котором серии исходного файла распределяются по m-1 вспомогательному файлу, а затем выполняется многопутевое слияние серий из (m-1) файла, пока в одном из них не образуется одна серия.

### *6.2.4. Многофазная сортировка*

При использовании рассмотренного выше метода сбалансированной многопутевой внешней сортировки на каждом шаге примерно половина вспомогательных файлов используется для ввода данных и примерно столько же для вывода сливаемых серий. Идея многофазной сортировки состоит в том, что из имеющихся m вспомогательных файлов (m-1) файл служит для ввода сливаемых последовательностей, а один - для вывода образуемых серий. Как только один из файлов ввода становится пустым, его начинают использовать для вывода серий, получаемых при слиянии серий нового набора (m-1) файлов. Таким образом, имеется первый шаг, при котором серии исходного файла распределяются по m-1 вспомогательному файлу, а затем выполняется многопутевое слияние серий из (m-1) файла, пока в одном из них не образуется одна серия.

## 6.3. Основные характеристики алгоритмов сортировки. Оценки сложности сортировок выбором, вставками, обменом и слиянием.

Для метода с**ортировки простым выбором** требуемое число сравнений - n**x(n-1)/2. Порядок требуемого числа пересылок (включая те, которые требуются для выбора минимального элемента) в худшем случае составляет O(n2). Однако порядок среднего числа пересылок есть O(n?ln n), что в ряде случаев делает этот метод предпочтительным.**

Для **сортировки включением**, в лучшем случае (когда массив уже упорядочен) для выполнения алгоритма с массивом из n элементов потребуется n-1 сравнение и 0 пересылок. В худшем случае (когда массив упорядочен в обратном порядке) потребуется n?(n-1)/2 сравнений и столько же пересылок. Таким образом, можно оценивать сложность метода простых включений как O(n2).

Для метода **простой обменной сортировки (пузырек)** требуется число сравнений n**x(n-1)/2, минимальное число пересылок 0, а среднее и максимальное число пересылок - O(n2).**

При применении **сортировки со слиянием** число сравнений ключей и число пересылок оценивается как O(n?log n). Но следует учитывать, что для выполнения алгоритма для сортировки массива размера n требуется 2?n элементов памяти.

# 8. Теорема о нижней оценке эффективности алгоритмов сортировок, основанных на сравнении элементов. Примеры альтернативных более быстрых сортировок.

## 8.1. Теорема о нижней оценке эффективности алгоритмов сортировок, основанных на сравнении элементов.

Теорема: В худшем случае любой алгоритм сортировки сравнениями выполняет сравнений, где n — число сортируемых элементов.

Доказательство: Любому алгоритму сортировки сравнениями можно сопоставить дерево. В нем узлам соответствуют операции сравнения элементов, ребрам — переходы между состояниями алгоритма, а листьям — конечные перестановки элементов (соответствующие завершению алгоритма сортировки). Необходимо доказать, что высота такого дерева для любого алгоритма сортировки сравнениями не меньше чем nlog n , где n — количество элементов.

При сравнении двух элементов, существует два возможных исхода (ai < aj u ai >= aj), значит, каждый узел дерева имеет не более двух сыновей. Всего существует n! различных перестановок n элементов, значит, число листьев нашего дерева не менее n! (в противном случае некоторые перестановки были бы не достижимы из корня, а, значит, алгоритм не правильно работал бы на некоторых исходных данных).

## 8.2. Примеры альтернативных более быстрых сортировок

**Алгоритмы устойчивой сортировки**

1. Сортировка пузырьком (англ. Bubble sort ) — сложность алгоритма: O(n2); для каждой пары индексов производится обмен, если элементы расположены не по порядку.
2. Сортировка перемешиванием (Шейкерная, Cocktail sort, bidirectional bubble sort) — Сложность алгоритма: O(n2)
3. Гномья сортировка — имеет общее с сортировкой пузырьком и сортировкой вставками. Сложность алгоритма — O(n2).
4. Сортировка вставками (Insertion sort) — Сложность алгоритма: O(n2); определяем где текущий элемент должен находиться в упорядоченном списке и вставляем его туда
5. Блочная сортировка (Корзинная сортировка, Bucket sort) — Сложность алгоритма: O(n); требуется O(k) дополнительной памяти и знание о природе сортируемых данных, выходящее за рамки функций "переставить" и "сравнить".
6. Сортировка подсчётом (Counting sort) — Сложность алгоритма: O(n+k); требуется O(n+k) дополнительной памяти (рассмотрено 3 варианта)
7. Сортировка слиянием (Merge sort) — Сложность алгоритма: O(n log n); требуется O(n) дополнительной памяти; выстраиваем первую и вторую половину списка отдельно, а затем — сливаем упорядоченные списки
8. Сортировка с помощью двоичного дерева (англ. Tree sort) — Сложность алгоритма: O(n log n); требуется O(n) дополнительной памяти

**Алгоритмы неустойчивой сортировки**

Сортировка выбором (Selection sort) — Сложность алгоритма: O(n2); поиск наименьшего или наибольшего элемента и помещения его в начало или конец упорядоченного списка
Сортировка Шелла (Shell sort) — Сложность алгоритма: O(n log2 n); попытка улучшить сортировку вставками

Сортировка расчёской (Comb sort) — Сложность алгоритма: O(n log n)
Пирамидальная сортировка (Сортировка кучи, Heapsort) — Сложность алгоритма: O(n log n); превращаем список в кучу, берём наибольший элемент и добавляем его в конец списка
Плавная сортировка (Smoothsort) — Сложность алгоритма: O(n log n)
Быстрая сортировка (Quicksort), в варианте с минимальными затратами памяти — Сложность алгоритма: O(n log n) — среднее время, O(n2) — худший случай; широко известен как быстрейший из известных для упорядочения больших случайных списков; с разбиением исходного набора данных на две половины так, что любой элемент первой половины упорядочен относительно любого элемента второй половины; затем алгоритм применяется рекурсивно к каждой половине. При использовании O(n) дополнительной памяти, можно сделать сортировку устойчивой.
Introsort — Сложность алгоритма: O(n log n), сочетание быстрой и пирамидальной сортировки. Пирамидальная сортировка применяется в случае, если глубина рекурсии превышает log(n).
Patience sorting — Сложность алгоритма: O(n log n + k) — наихудший случай, требует дополнительно O(n + k) памяти, также находит самую длинную увеличивающуюся подпоследовательность
Stooge sort — рекурсивный алгоритм сортировки с временной сложностью .
Поразрядная сортировка — Сложность алгоритма: O(n·k); требуется O(k) дополнительной памяти.
Цифровая сортировка — то же, что и Поразрядная сортировка.

**Непрактичные алгоритмы сортировки**
Bogosort — O(n·n!) в среднем. Произвольно перемешать массив, проверить порядок.
Сортировка перестановкой — O(n·n!) — худшее время. Для каждой пары осуществляется проверка верного порядка и генерируются всевозможные перестановки исходного массива.
Глупая сортировка (Stupid sort) — O(n3); рекурсивная версия требует дополнительно O(n2) памяти
Bead Sort — O(n) or O(√n), требуется специализированное аппаратное обеспечение
Блинная сортировка (Pancake sorting) — O(n), требуется специализированное аппаратное обеспечение

**Алгоритмы, не основанные на сравнениях:**
Блочная сортировка (Корзинная сортировка, Bucket sort)
Лексикографическая или поразрядная сортировка (Radix sort)
Сортировка подсчётом (Counting sort)
Прочие алгоритмы сортировки
Топологическая сортировка
Внешняя сортировка.

**9. Архитектура ЭВМ: многоуровневое представление. Понятия интерпретации и транс-ляции (компиляции). Процессор: назначение, основные узлы и их взаимодействие.**

**Оперативная память: организация, порядок следования байтов, адресация, контроль**

**ошибок, алгоритм Ричарда Хэмминга.**

### *9.1. Архитектура ЭВМ: многоуровневое представление.*

Архитектура - это множество ресурсов ЭВМ, доступных пользователю на логическом уровне, без детализации способов взаимодействия процессоров, устройств памяти, внешних устройств и программных средств.
Организация - это способы распределения функций, установления связи и взаимодействия процессоров, устройств памяти и внешних устройств, используемые для реализации возможностей, заложенных в архитектуре. При изучении организации рассматривают:

- представление и формат данных;
- уровни памяти и их взаимодействие;
- состав и формат машинных команд;
- систему прерываний;
- способы обмена данными.

Реализация – способы технического исполнения конкретных устройств, линий или шин связи и протоколов взаимодействия между ними.

Обычно на уровнях организации и реализации происходит перераспределение функций между аппаратными и программными средствами. Это порождает семейство машин одной архитектуры, но разной производительности.

**Многоуровневая организация ЭВМ.**

В общем случае обработку информации на ЭВМ можно рассматривать в виде иерархической системы уровней, представленных следующей таблицей.

| Пользователь данного уровня | УРОВЕНЬ | СОДЕРЖАНИЕ |
|---|---|---|
| Постановщик задач,<br><br>Системный аналитик | **Концептуальный** | Пользователь задает режимы и виды обработки данных, необхо-димые для решения задачи |
| Пользователь функционального ПО, специалист в конкретной предметной области | **Уровень проблеммно-ориентированных ПС** | Решение предметных задач готовыми программными средствами |
| Разработчик функциональных программных комплексов, | **Уровень промежуточного ПО** (Middleware) | Технологии разработки программных систем COM, DCOM, CORBA, |

| | | |
|---|---|---|
| системный архитектор | | RMI |
| Разработчик функциональных программ | **Уровень интегрированных сред и языков высокого уровня (ЯВУ)** | Паскаль, СИ, Delphi, C++ Builder, Visual C |
| Системный программист<br><br>Прикладной программист | **Уровень ассемблера** | Программирование фрагментов программ высокой эффективности |
| Системный программист | **Уровень ОС** | Обеспечение выполнения привилегированных команд |
| Программист/Электронщик | **Уровень машинных команд** | Цифровое кодирование и представление команд |
| Программист/Электронщик | **Уровень микрокоманд** | Описание набора элементарных операций, реализующих машинные команды |
| Электронщик | **Уровень межрегистровых передач** | Реализация операций на уровне пересылок между регистрами |
| Электронщик-технолог | **Уровень вентилей** | Технологический уровень, устройства машины представля-ются в виде интегральных схем |

***Достоинства такого представления ЭВМ:***

- Каждый верхний уровень интерпретируется одним или несколькими нижними уровнями;
- Каждый из уровней можно проектировать независимо;
- Чем ниже уровень, на котором реализуется программа, тем более высокая производительность достижима;
- Модификация нижних уровней не влияет на реализацию верхних.

### *9.2. Понятия интерпретации и трансляции (компиляции)*

**Трансляция** программы — преобразование программы, представленной на одном из языков программирования, в программу на другом языке и, в определённом смысле, равносильную первой. При трансляции выполняется перевод программы, понятной человеку, на язык, понятный компьютеру. Выполняется специальными программными средствами (транслятором).

**Трансляторы** реализуются в виде компиляторов или интерпретаторов. С точки зрения выполнения работы компилятор и интерпретатор существенно различаются. Если цель трансляции – преобразование всего исходного текста на внутренний язык компьютера (т.е. получение некоторого нового кода) и только, то такая трансляция называется также **компиляцией**. Исходный текст называется также исходной программой или исходным модулем, а результат компиляции – объектным кодом или объектным модулем. Если же трансляции подвергаются отдельные операторы исходных текстов и при этом полученные коды сразу выполняются, такая трансляция называется **интерпретацией**. Поскольку трансляция выполняется специальными программными средствами (трансляторами), последние носят название компилятора или интерпретатора, соответственно.

Цель трансляции — преобразовать текст с одного языка на другой, который понятен адресату текста. В случае программ-трансляторов, адресатом является техническое устройство (процессор) или программа-интерпретатор.

**Компиляция** — преобразование программой-компилятором исходного текста программы, написанного на языке высокого уровня в машинный язык, в язык, близкий к машинному, или в объектный модуль. Результатом компиляции является объектный файл с необходимыми внешними ссылками для компоновщика.

Компилятор читает всю программу целиком, делает ее перевод и создает законченный вариант программы на машинном языке, который затем и выполняется.

**Интерпретация** — процесс непосредственного покомандного выполнения программы без предварительной компиляции, «на лету»; в большинстве случаев интерпретация намного медленнее работы уже скомпилированной программы, но не требует затрат на компиляцию, что в случае небольших программ может повышать общую производительность.

Типы интерпретаторов

*Простой интерпретатор* анализирует и тут же выполняет (собственно интерпретация) программу покомандно (или построчно), по мере поступления её исходного кода на вход интерпретатора. Его достоинство - мгновенная реакция. Недостаток — такой интерпретатор обнаруживает ошибки в тексте программы только при попытке выполнения команды (или строки) с ошибкой.

*Интерпретатор компилирующего типа* — это система из компилятора, переводящего исходный код программы в промежуточное представление, например, в байт-код или p-код, и собственно интерпретатора, который выполняет полученный промежуточный код (так называемая виртуальная машина). Его достоинство – большее

быстродействие выполнения программ (за счёт выноса анализа исходного кода в отдельный, разовый проход, и минимизации этого анализа в интерпретаторе). Недостатки — большее требование к ресурсам и требование на корректность исходного кода.

### *9.3.1. Процессор: назначение*

Процессор – это главная часть цифровой ЭВМ, осуществляющая сложную переработку информации. В него входит также устройство управления ЭВМ. Процессор не только обрабатывает информацию и управляет данным процессом, но и обеспечивает при этом взаимодействие с устройствами памяти, ввода и вывода.

В ЭВМ первых поколений, построенных на дискретных (т. е. отдельных) элементах (электронных лампах, полупроводниковых триодах), процессор представлял собой большое устройство, состоявшее из нескольких электронных плат с размещенными на них навесными компонентами радиоэлектроники. Кроме ламп и транзисторов на этих платах находились диоды, резисторы, конденсаторы. Все соединения между отдельными компонентами осуществлялись с помощью пайки и проводов, а позднее – печатным монтажом. Успехи микроэлектронных технологий позволили в одном элементе объединять несколько транзисторов, диодов, резисторов и соединений между ними. Таким образом, появились так называемые интегральные схемы (ИС). С годами степень интеграции (т.е. число элементов в одной ИС) возрастала, появились большие интегральные схемы (БИС), а затем и сверхбольшие интегральные схемы (СБИС). Основой ИС является кристалл полупроводника, на котором формируются полупроводниковые переходы, выполняющие роль транзисторов и диодов. На том же кристалле создаются микрообласти с добавлением примесей, осуществляющие функции резисторов и конденсаторов; выполняются также электрические соединения между ними. Если в первых ИС на одном кристалле размещалось до десятка транзисторов, то в современных — сотни миллионов элементов. Использование СБИС позволяет значительно повысить эффективность цифровых систем: увеличить их производительность и надежность, уменьшить габаритные размеры, массу и потребляемую мощность. Современные технологии изготовления СБИС очень сложны и требуют дорогостоящего оборудования. Создание завода по производству СБИС обходится в миллиарды долларов. Но стоимость цифровой техники, построенной на СБИС, неуклонно снижается. Объясняется это следующим обстоятельством. Интегральная схема, содержащая большое число элементов, является универсальной, т.е. находит применение в самых разных устройствах.

Следовательно, ее можно выпускать огромными тиражами — миллионами штук, а при массовом выпуске экономически оправдано использование высокопроизводительных автоматических и робототехнических линий и участков производства.

Применение СБИС оказало большое влияние на принципы построения цифровых систем, их архитектуру, логическую структуру, математическое обеспечение. Появился новый подход к проектированию таких систем – на основе программируемой логики. Этот подход предполагает использование при построении систем одной (или очень малого количества) стандартной универсальной СБИС, управляемой программно. Специализация системы осуществляется программой, которая управляет стандартной универсальной СБИС. В 1970-х годах появилась СБИС, которая в значительной степени была способна выполнять функции процессора. Такая интегральная схема получила название микропроцессор (МП).

Уже треть века истории развития микропроцессорной техники на ведущую позицию в этой области претендует американская фирма Intel.

Если к микропроцессору добавляется память (запоминающее устройство) и устройство ввода-вывода, то такая система может выполнять функции ЭВМ. Созданные на

основе микропроцессора вычислительные машины стали называться микроЭВМ. Именно благодаря появлению микропроцессоров удалось сделать доступные для многих ЭВМ, получившие название «персональный компьютер».

Итак, микропроцессор — это выполненное по интегральной технологии цифровое устройство, обрабатывающее информацию в соответствии с программой и управляющее вводом и выводом информации. Наибольшее распространение получили микропроцессоры, выполненные на одном кристалле, или однокристальные МП

Микропроцессор представляет собой неразъемный конструктивный элемент, подсоединяемый к другим элементам вычислительной машины с помощью выводов. Корпус МП сделан обычно из пластмассы или керамики. Число выводов может быть разным: 28, 40, 64 и больше. Первые МП имели выводы с двух сторон корпуса, по одному ряду с каждой стороны. Современные МП имеют выводы на нижней плоскости с четырех сторон, по несколько рядов с каждой стороны. С ростом числа компонентов в одном МП (счет, как уже отмечалось, идет на миллионы) увеличивается и число выводов. В современных МП число выводов более тысячи. По соображениям удобства на число выводов стараются наложить ограничения.

Непрерывное совершенствование интегральных технологий приводит к изменениям в структуре микропроцессора.

### *9.3.2. Процессор: основные узлы и их взаимодействие*

Процессор - основная часть компьютера, осуществляющая управление (эти возможности реализуются при помощи логических операций) и обработку данных. Переход от первых процессоров, имевших простую архитектуру и работавших на частотах 2,5 - 4 МГц к современным процессорам, выполненным на СБИС, включающих в себя десятки миллионов транзисторов (, работающих на частотах 200 - 500 МГц, сопровождается переходом к более совершенным и мощным компьютерам. Процессор предназначен для выполнения последовательности команд, записанных в оперативной памяти компьютера. Структура процессора (рис.4.1.), позволяющая реализовать его функции, включает в себя:
·устройство управления (УУ), дешифрирующее команды и вырабатывающее сигналы управления для блоков, выполняющих эти команды;

·арифметико - логическое устройство (АЛУ), выполняющее арифметические и логические операции;

·блок регистров общего назначения (РОН), позволяющий выполнять операции с предельно высокой скоростью;

·блоки сверхоперативной памяти (Кэш 1-го уровня) для хранения команд и данных. Введение Кэш позволяет уменьшить количество обращений к оперативному запоминающему устройству компьютера для чтения последовательности команд и данных;

·блоки, осуществляющие интерфейс с памятью компьютера. Они обеспечивают связь с внешним оперативным запоминающим устройством или блоком быстрой памяти (Кэш 2-го уровня), устанавливаемым между процессором и оперативной памятью;

·системный интерфейс, который обеспечивает связь процессора с системными блоками компьютера и внешними устройствами (ВУ).

Команда, считанная из ОЗУ, определяет вид действий над операндами, адреса операндов и адрес результата операции. Дешифрация команды в процессоре может быть сразу полной или частичной. При частичной дешифрации часть кода команды передаётся в АЛУ и дешифрируется при выполнении операции.

**Алгоритм работы**:

Процессор обеспечивает выборку команды из памяти и её выполнение. Алгоритм работы процессора включает следующие шаги:
1. Определение адреса команды.

Адрес команды хранится в регистре счетчике команд и, в случае линейного выполнения программы, после выполнения каждой команды счетчик команд увеличивает содержимое на количество слов команды. В случае безусловного перехода в счетчик команд записывается адрес перехода. В современных процессорах выборка команд производится целым блоком, который записывается во внутренний КЭШ команд. Команды выполняются конвейером команд, таким образом, что одновременно может выполняться несколько команд на разных ступенях конвейера. Специальное устройство предсказывает последовательность выполнения команд, и производится опережающее выполнение тех команд, операнды которых на данный момент определены. Если действительный порядок выполнения команд отличается от предсказанного, последовательность команд выгружается из конвейера и производится загрузка конвейера новой последовательностью команд.

2. Выборка адреса команды.

Для чтения блока команд из оперативной памяти процессор устанавливает адрес блока команд на шине адреса и производит выборку.

3. Выборка команды.

Блок сопряжения выполняет ввод блока команд через интерфейс с памятью. Блок команд запоминается в КЭШ команд.

4. Дешифрация команды.

Если команда состоит из нескольких слов, то в дешифратор кода команды передается только первое слово команды, которое содержит код операции и признаки адресации. В этом случае по первому слову определяется длина команды и выбор следующих слов происходит по мере необходимости. Процесс дешифрации может быть разделен на первичную и вторичную. Первичная дешифрация определяет тип команды , ее цель - определяет к какой группе команд относится данная команда. Первичная дешифрация позволяет уменьшить объем алгоритма обработки программ за счет одинаковой обработки команд одного типа. Вторичная дешифрация выполняется на более поздних этапах, обычно после вычисления адресов операндов. Для команд арифметико-логической группы вторичная дешифрация может выполняться непосредственно в АЛУ;

5. Вычисление адресов операндов. Если команда адресная, то на следующем этапе вычисляются адреса операндов. Вычисление адреса и выборка для каждого операнда чередуются. Адрес операнда, если он является адресом ячейки ОЗУ, помещается в регистр адреса памяти;

6. Выборка операндов.

Выборка операндов производится для большинства адресных команд арифметико-логической группы. Содержимое ячейки памяти вводится в процессор для выполнения операции в АЛУ процессора. Если операнды размещаются во внутренних регистрах процессора, то операция выполняется значительно быстрее, чем при извлечении данных из памяти. Данные из оперативной памяти извлекаются блоками, которые помещаются в КЭШ данных внутри процессора.

7. Исполнение операции.

На этой стадии, если это необходимо, производится вторичная дешифрация команды непосредственно в АЛУ, где и выполняется операция над подготовленными заранее операндами. Кроме арифметических или логических операций могут выполняться операции по пересылке операндов, в этом случае операнд извлекается из соответствующего регистра и пересылается на место операнда- приемника. Если выполняется команда безусловного перехода, то вычисленный адрес перехода записывается в регистр - счетчик команд. Команды вызова подпрограмм требуют запоминания состояния вычислительного процесса. Для этого используется сохранение данных в стеке (области памяти, предназначенной для записи данных в определенной последовательности и их последовательного извлечения). Для записи данных в стек и их извлечения из стека используется специальный адресный регистр, автоинкрементно изменяющий адрес. Указатель стека всегда указывает на следующую после последнего обращения к стеку ячейку памяти.

8. Запись результата.

После выполнения команды результат операции обычно помещается в регистр аккумулятор. Затем он должен быть записан в оперативную память и , если это необходимо, выведен на внешнее запоминающее устройство, на дисплей монитора или передано другому внешнему устройству. Ввод и вывод информации для освобождения центрального процессора производят специальные каналы ввода/вывода. При этом канал управляется процессором ввода/вывода, который анализирует ситуацию и осуществляет обмен.

Способ реализации команд зависит от архитектуры процессора. Традиционно, архитектура процессора развивалась в двух конкурирующих направлениях: с сокращенным набором команд - RISC (Reduced Instruction Set Computer) и с полным набором команд - CISC (Complex Instruction Set Computer).

Идея архитектуры RISC возникла в связи с развитием интеграции полупроводниковых схем и стремлением реализовать минимальными аппаратными средствами достаточно высокие вычислительные возможности. Задачи оптимизации структуры и технических возможностей реализовались в компьютерах с архитектурой фон - Неймана в идее Мини-ЭВМ, а в архитектуре процессора в идее RISC - архитектуры.

### 9.4.1. Оперативная память: организация

Оперативная память – это основная память, на работу с которой ориентирован процессор, точнее программа, выполняемая процессором. Это адресная память.

В большинстве ЭВМ оперативная память – энергозависимая. Это означает, что при выключении питания информация в оперативной памяти не сохраняется. Для сохранения информации при выключении питания содержимое оперативной памяти переписывают в энергонезависимую память, например на основе записи на магнитную поверхность. Это вторичная (внешняя) память на магнитных дисках (память прямого доступа) или на магнитных лентах (архивная память).

Множество адресов, которые могут использоваться в командах процессора, составляют его адресное пространство**.**

Здесь применяются различные термины – адресное пространство процессора, адресное пространство математической памяти, адресное пространство программы.

Адресное пространство процессора определяется разрядностью ЭВМ по заданию адреса. Процессоры, использующие 32-х разрядные адреса, имеют адресное пространство равное $2^{32}$ (4 Гб).

Современные ЭВМ ориентированы на работу с "наращиваемым" объемом физической памяти. Это означает, что:

- адресное пространство процессора и физической памяти могут не совпадать,
- размещение программы и данных в физической памяти могут не совпадать с их размещением по адресам в адресном пространстве процессора,
- прикладные программы вместо прямой адресации физической памяти используют обращение к некоторой модели (отображению) памяти,
- обращение к физической памяти производится при помощи диспетчера памяти, согласующего модель математической памяти с динамикой распределения программ и данных в физической памяти.

Диспетчер памяти может быть реализован программно или схемно-программно. В последнем случае говорят о реализации виртуальной памяти.

Размер адресного пространство процессора определяется разрядностью адресных шин, которая ограничена разрядностью процессора. Это случай плоской модели математической памяти.

Для снятия этого ограничения некоторые процессоры, например МП Intel, допускают использование множества адресных пространств. В этих случаях говорят о структурированной (сегментированной) математической памяти.

Использование сегментированной памяти увеличивает адресное пространство процессора, но усложняет адресацию. В сегментированной памяти адреса операндов и команд задаются вектором: указанием используемого сегмента (например, через базовый адрес сегмента в линейной памяти) и адреса в сегменте. Но, так как оперативная память остается не сегментированной (линейной), требуется пересчет сегментированного адреса в линейный адрес. Эта процедура называется трансляцией сегментов.

Многие процессоры могут оперировать с несколькими плоскими или сегментированными пространствами адресов:

- пространством адресов оперативной памяти,
- пространством адресов регистров устройств ввода/вывода (портов).

Физически – это различные системы памяти, но во многих архитектурах для доступа к ним используется единая система адресации ячеек памяти.

В этих структурах обращение к портам по записи и чтению производится обычными командами обращения к памяти. При этом портам обычно приписываются адреса в старшем диапазоне.

В архитектуре МП Intel используются два независимых адресных пространства: портов и ячеек памяти.

### 9.4.2. Оперативная память: порядок следования байтов

Порядок от старшего к младшему
Порядок от старшего к младшему или (англ. big-endian, дословно: «тупоконечный»):A1...An, запись начинается со старшего и заканчивается младшим. Этот порядок является стандартным для протоколов TCP/IP, он используется в заголовках пакетов данных и во многих протоколах более высокого уровня, разработанных для использования поверх TCP/IP. Поэтому, порядок байтов от старшего к младшему часто называют сетевым порядком байтов (англ. network byte order). Этот порядок байтов используется процессорами IBM 360/370/390, Motorola 68000, SPARC (отсюда третье название — порядок байтов Motorola, Motorola byte order).

В этом же виде (используя представление в десятичной системе счисления) записываются числа индийско-арабскими цифрами в письменностях с порядком знаков слева направо (латиница, кириллица). Для письменностей с обратным порядком (арабская) та же запись числа воспринимается как «от младшего к старшему».

Порядок байтов от старшего к младшему применяется во многих форматах файлов — например, PNG, FLV, EBML.

Порядок от младшего к старшему

Порядок от младшего к старшему или (англ. little-endian, дословно: «остроконечный», о происхождении термина ниже): An … A0, запись начинается с младшего и заканчивается старшим. Этот порядок записи принят в памяти персональных компьютеров с x86-процессорами, в связи с чем иногда его называют интеловский порядок байтов (по названию фирмы-создателя архитектуры x86). В некоторых кругах используется название англ. VAX order, например, в документации Perl[1].

В противоположность «тупоконечному» порядку, меньше[источник не указан 352 дня] кросс-платформенных протоколов и форматов данных с «остроконечным» порядком байт; заметные исключения: USB, конфигурация PCI, таблица разделов GUID, рекомендации FidoNet.

Переключаемый порядок

Многие процессоры могут работать и в порядке от младшего к старшему, и в обратном, например, ARM, PowerPC (но не PowerPC 970), DEC Alpha, MIPS, PA-RISC и IA-64. Обычно порядок байтов выбирается программно во время инициализации операционной системы, но может быть выбран и аппаратно перемычками на материнской плате. В этом случае правильнее говорить о порядке байтов операционной системы. Переключаемый порядок байтов иногда называют англ. bi-endian.

Смешанный порядок

Смешанный порядок байтов (англ. middle-endian) иногда используется при работе с числами, длина которых превышает машинное слово. Число представляется последовательностью машинных слов, которые записываются в формате, естественном для данной архитектуры, но сами слова следуют в обратном порядке.

Классический пример middle-endian — представление 4-байтных целых чисел на 16-битных процессорах семейства PDP-11 (известен как PDP-endian). Для представления двухбайтных значений (слов) использовался порядок little-endian, но 4-хбайтное двойное слово записывалось от старшего слова к младшему.

В процессорах VAX и ARM используется смешанное представление для длинных вещественных чисел.

### 9.4.3. Оперативная память: адресация

Кратко говоря, память программы может рассматриваться как один очень-очень длинный ряд байтов. Байт - это единица измерения количества информации, в мире Delphi и Windows он равен восьми битам и может хранить одно из `256` различных значений (от `0` до `255`). На память можно смотреть как на массив байт. Что именно содержат эти байты - зависит от того, как интерпретировать их содержимое, т.е. от того, как их используют. Значение 97 может означать число `97`, или же ANSI букву `'a'`. Если вы рассматриваете вместе несколько байт, то вы можете хранить и большие значения. Например, в 2-х байтах вы можете хранить одно из `256*256 = 65536` различных значений, две ANSI буквы `'ab'` или Unicode букву `'a'` - и т.д.

Чтобы обратиться к конкретному байту в памяти (адресовать его), можно присвоить каждому байту номер, пронумеровав их целыми положительными числами, включив ноль за начало отсчёта. Индекс байта в этом огромном массиве и называется его адресом, а весь массив целиком - памятью программы. Диапазон адресов от `0` до максимума называется адресным пространством программы. А максимум (длина) массива называется размером адресного пространства.

(примечание: ну, на самом деле, есть тысяча и один способ адресовать память, но в рамках современного мира и этой статьи мы ограничимся только этим способом).

Адресное пространство (вернее, его размер) определяет способность программы работать с данными. Чем оно больше - тем с большим количеством данных программа сможет работать (в один момент времени). Если у программы заканчивается свободное место в адресном пространстве (т.е. все адреса в нём выделены под какие-то объекты в программе) - то у программы заканчивается память (out of memory).

### 9.4.4. Оперативная память: контроль ошибок

#### 9.4.4.1. Контроль четности и коды коррекции ошибок (ECC)

Ошибки при хранении информации в оперативной памяти неизбежны. Они обычно классифицируются как аппаратные отказы и нерегулярные ошибки (сбои).

Если нормально функционирующая микросхема вследствие, например, физического повреждения начинает работать неправильно, то это называется аппаратным отказом. Чтобы устранить данный тип отказа, обычно требуется заменить некоторую часть аппаратных средств памяти, например неисправную микросхему, модуль SIMM или DIMM.

Другой, более коварный тип отказа — нерегулярная ошибка (сбой). Это непостоянный отказ, который не происходит при повторении условий функционирования или через регулярные интервалы. (Такие отказы обычно лечатся” выключением питания компьютера и последующим его включением.)

#### 9.4.4.2. Контроль четности

Это один из введенных IBM стандартов, в соответствии с которым информация в банках памяти хранится фрагментами по 9 бит, причем восемь из них (составляющих один байт) предназначены собственно для данных, а девятый является битом четности. Использование девятого бита позволяет схемам управления памятью на аппаратном уровне контролировать целостность каждого байта данных. Если обнаруживается ошибка, работа компьютера останавливается, а на экран выводится сообщение о неисправности. Если вы работаете на компьютере под управлением Windows или OS/2, то при возникновении ошибки контроля четности сообщение, возможно, не появится, а просто произойдет блокировка системы. После перезагрузки система BIOS должна идентифицировать ошибку и выдать соответствующее сообщение.

Модули SIMM и DIMM поставляются как с поддержкой битов четности, так и без нее. Первые ПК использовали память с контролем четности для регулировки точности

осуществляемых операций. Начиная с 1994 года на рынке ПК стала развиваться тревожная тенденция. Большинство компаний начали предлагать компьютеры с памятью без контроля четности и вообще без каких бы то ни было средств определения или исправления ошибок. Применение модулей SIMM без контроля четности сокращало стоимость памяти на 10-15%. В свою очередь, память с контролем четности обходилась дороже за счет применения дополнительных битов четности. Технология контроля четности не позволяет исправлять системные ошибки, однако предоставляет пользователю компьютера возможность их обнаружить, что имеет следующие преимущества:

- контроль четности защищает от последствий неверных вычислений на базе некорректных данных;

- контроль четности точно указывает на источник возникновения ошибок, помогая разобраться в проблеме и улучшая степень эксплуатационной надежности компьютера.

### 9.4.4.3. Код коррекции ошибок

Коды коррекции ошибок (Error Correcting Code — ECC) позволяют не только обнаружить ошибку, но и исправить ее в одном разряде. Поэтому компьютер, в котором используются подобные коды, в случае ошибки в одном разряде может работать без прерывания, причем данные не будут искажены. Коды коррекции ошибок в большинстве ПК позволяют только обнаруживать, но не исправлять ошибки в двух разрядах. В то же время приблизительно 98% сбоев памяти вызвано именно ошибкой в одном разряде, т.е. она успешно исправляется с помощью данного типа кодов. Данный тип ECC получил название SEC-DED (эта аббревиатура расшифровывается как одноразрядная коррекция, двухразрядное обнаружение ошибок"). В кодах коррекции ошибок этого типа для каждых 32 бит требуется дополнительно семь контрольных разрядов при 4-байтовой и восемь — при 8-байтовой организации (64-разрядные процессоры Athlon/Pentium). Реализация кода коррекции ошибок при 4-байтовой организации, естественно, дороже обычной проверки четности, но при 8-байтовой организации их стоимости равны, поскольку требуют одного и того же количества дополнительных разрядов.

По этой причине можно купить для 32-разрядных систем модули SIMM (36 бит), DIMM (72 бит) или RIMM (18 бит) и использовать их в режиме ECC, если коды коррекции ошибок поддерживаются набором микросхем системной логики. Если в системе используются модули SIMM, можно сформировать банк памяти (72 бит) из двух 36-разрядных модулей и ECC использовать на уровне банка. Если в системе используются модули DIMM, то в качестве банка может выступать один 72-разрядный модуль, обеспечивая необходимое количество дополнительных битов памяти. В случае использования модулей RIMM для организации проверки четности следует отдать предпочтение их 28-разрядным версиям.

Для использования кодов коррекции ошибок необходим контроллер памяти, вычисляющий контрольные разряды при операции записи в память. При чтении из памяти такой контроллер сравнивает прочитанные и вычисленные значения контрольных разрядов и при необходимости исправляет испорченный бит (или биты). Стоимость дополнительных логических схем для реализации кода коррекции ошибок в контроллере памяти не очень высока, но это может значительно снизить быстродействие памяти при операциях записи. Это происходит потому, что при операциях записи и чтения необходимо ожидать завершения вычисления контрольных разрядов. При записи части слова вначале следует прочитать полное слово, затем перезаписать изменяемые байты и только после этого —

новые вычисленные контрольные разряды.

В большинстве случаев сбой памяти происходит в одном разряде, и потому такие ошибки успешно исправляются с помощью кода коррекции ошибок. Использование отказоустойчивой памяти обеспечивает высокую надежность компьютера. Память с кодом ECC предназначена для серверов, рабочих станций или приложений, в которых потенциальная стоимость ошибки вычислений значительно превышает дополнительные средства, вкладываемые в оборудование, а также временные затраты системы. Если данные имеют особое значение, и компьютеры применяются для решения важных задач, без памяти ECC не обойтись. По сути, ни один уважающий себя системный инженер не будет использовать сервер, даже самый неприхотливый, без памяти ECC.

### 9.4.5. Алгоритм Ричарда Хэмминга

Алгоритм, который позволяет закодировать какое-либо информационное сообщение определённым образом и после передачи (например по сети) определить появилась ли какая-то ошибка в этом сообщении (к примеру из-за помех) и, при возможности, восстановить это сообщение.
Код Хэмминга состоит из двух частей. Первая часть кодирует исходное сообщение, вставляя в него в определённых местах контрольные биты (вычисленные особым образом). Вторая часть получает входящее сообщение и заново вычисляет контрольные биты (по тому же алгоритму, что и первая часть). Если все вновь вычисленные контрольные биты совпадают с полученными, то сообщение получено без ошибок. В противном случае, выводится сообщение об ошибке и при возможности ошибка исправляется.
**Кодирование**
Прежде всего, необходимо вставить контрольные биты. Они вставляются в строго определённых местах — это позиции с номерами, равными степеням двойки. В нашем случае (при длине информационного слова в 16 бит) это будут позиции 1, 2, 4, 8, 16. Соответственно, у нас получилось 5 контрольных бит (выделены красным цветом):
было — 1011101110111110, стало — xx1x011x1011101x11110
Таким образом, длина всего сообщения увеличилась на 5 бит. До вычисления самих контрольных бит, мы присвоили им значение «x».
**Вычисление контрольных бит.**
Теперь необходимо вычислить значение каждого контрольного бита. Значение каждого контрольного бита зависит от значений информационных бит (как неожиданно), но не от всех, а только от тех, которые этот контрольных бит контролирует. Для того, чтобы понять, за какие биты отвечает каждых контрольный бит необходимо понять очень простую закономерность: контрольный бит с номером N контролирует все последующие N бит через каждые N бит, начиная с позиции N. Не очень понятно, но по картинке, думаю, станет яснее:
Здесь знаком «X» обозначены те биты, которые контролирует контрольный бит, номер которого справа. То есть, к примеру, бит номер 12 контролируется битами с номерами 4 и 8. Ясно, что чтобы узнать какими битами контролируется бит с номером N надо просто разложить N по степеням двойки.
Но как же вычислить значение каждого контрольного бита? Делается это очень просто: берём каждый контрольный бит и смотрим сколько среди контролируемых им битов единиц, получаем некоторое целое число и, если оно чётное, то ставим ноль, в противном случае ставим единицу. Вот и всё! Можно конечно и наоборот, если число чётное, то ставим единицу, в противном случае, ставим 0.
**Декодирование и исправление ошибок.**

Вся вторая часть алгоритма заключается в том, что необходимо заново вычислить все контрольные биты (так же как и в первой части) и сравнить их с контрольными битами, которые мы получили. Итак заново считаем контрольные биты.
После этого находим контрольные биты которые не совпадают с такими же контрольными битами, которые мы получили. Теперь просто сложив номера позиций неправильных контрольных бит мы получаем позицию ошибочного бита. Теперь просто инвертировав его и отбросив контрольные биты, мы получим исходное сообщение в первозданном виде!

**Альтернативный вариант**

Код хэмминга предназначен для исправления ошибок разрядностью N и исправлению ошибок разрядностью N+1.

Пусть количество сообщений, которые мы хотим передавать равняется 16. Для безызбыточного кодирования сообщений достаточно 4 двоичных разрядов. Для обнаружения и исправления одиночной ошибки мы можем добавить 8 дополнительных разрядов к передаваемому слову (передавая сообщение трижды), что, однако не такой способ кодирования не является эффективным.

Код хэмминга предлагает добавление всего 4 дополнительных разрядов к кодируемому слову. Итак, пусть сообщение записано 4 разрядами a1,a2,a3,a4. Добавим к нему три дополнительных вычисляемых разряда:
a5=a2 ^ a3 ^ a4
a6=a1 ^ a3 ^ a4
a6=a1 ^ a2 ^ a4
Для проверки наличия ошибки при передаче слова a1a2a3a4a5a6a7 при передаче символов a4,a5,a6,a7 следует получить сумму
s1=a4 ^ a5 ^ a6 ^ a7
Если данная сумма равна 1, то ошибка была допущена. В случае "да" проверим не допущена ли ошибка в символах a6, a7, в случае нет - в символах a2, a3. В обоих случаях ответ дает следующая сумма:
s2=a2 ^ a3 ^ a6 ^ a7
Ниже приведена таблица диагностики ситуации, исходя из значений сумм s1 и s2.

s1 s2
0 0 нет ошибки или a1
0 1 a2 или a3
1 0 a4 или a5
1 1 a6 или a7

В каждом из этих случаев осталось выбрать только один вариант. Это можно сделать при помощи вычисления суммы:
s3=a1 ^ a3 ^ a5 ^ a7
Таким образом мы имеем 3 проверочных отношения:
s1=a4 ^ a5 ^ a6 ^ a7
s2=a2 ^ a3 ^ a6 ^ a7
s3=a1 ^ a3 ^ a5 ^ a7
которые позволяет убедится в наличии ошибки и точно указать ее место, причем в случае возникнования одиночной ошибки двоичное число s1s2s3 указывает разряд в котором произошла ошибка. Для обнаружения двойной ошибки следует только добавить еще один проверочный разряд a0 и условие, указывающее на наличие двойной ошибки:
s0=a0 ^ a1 ^ a2 ^ a3 ^ a4 ^ a5 ^ a6 ^ a7

# 14. Взаимодействие параллельных процессов. Взаимное исключение. Синхронизация. Буфер сообщений. Критический участок. Семафор. Порт. Очереди событий. Проблема тупиков. (ЖЕСТЬ !!! Кому-то повезет!)

## 14.1. Взаимодействие параллельных процессов

Практически любая более или менее сложная система имеет в своем составе компоненты, работающие одновременно, т.е. параллельно. Параллельно работающие подсистемы могут взаимодействовать самым различным образом, либо вообще работать независимо друг от друга. Способ взаимодействия подсистем определяет вид параллельных процессов, протекающих в системе. Также, вид моделируемых процессов **влияет на выбор метода их** имитации**.**

Рассмотрим виды параллельных процессов.

Асинхронный параллельный процесс — это такой процесс, состояние которого не зависит от состояния другого параллельного процесса (ПП).

Пример асинхронных ПП, протекающих в рамках одной системы, — это подготовка и проведение рекламной кампании фирмой и работа сборочного конвейера. Или например, из области вычислительной техники — выполнение вычислений процессором и вывод информации на печать.

Синхронный ПП — это такой процесс, состояние которого зависит от состояния взаимодействующих с ним ПП. Пример синхронного ПП — работа торговой организации и доставка товара со склада (нет товара — нет торговли).

Один и тот же процесс может быть синхронным по отношению к одному из активных ПП и асинхронным по отношению к другому.

Подчиненный ПП создается и управляется другим процессом (более высокого уровня). Примером таких процессов является ведение боевых действий подчиненными подразделениями.

Независимый ПП — процесс, который не является подчиненным ни для одного из процессов. Например, после запуска неуправляемой зенитной ракеты ее полет можно рассматривать как независимый процесс, одновременно с которым самолет ведет боевые действия другими средствами.

Способ организации параллельных процессов в системе зависит от физической сущности этой системы.

Разработка и использование любой ИМ предполагает ее программную реализацию и исследование с применением вычислительных систем (ВС). Реализация параллельных процессов в ВС имеет следующие особенности:

- на уровне задач вычислительные процессы могут быть истинно параллельными только в многопроцессорных ВС или вычислительных сетях;

- многие ПП используют одни и те же ресурсы, поэтому даже асинхронные ПП в пределах одной ВС вынуждены согласовывать свои действия при обращении к общим ресурсам;

- в ВС используется еще два вида ПП: родительский и дочерний ПП; особенность их состоит в том, что процесс-родитель не может быть завершен, пока не завершатся все его дочерние процессы.

Для организации взаимодействия параллельных процессов в ВС используются три основных подхода:

- на основе «взаимного исключения» - предполагает запрет доступа к общим ресурсам (общим данным) для всех ПП, кроме одного, на время его работы с этими ресурсами (данными);

- на основе синхронизации посредством сигналов - подразумевает обмен сигналами между двумя или более процессами по установленному протоколу. Такой «сигнал» рассматривается как некоторое событие, вызывающее у получившего его процесса соответствующие действия;

- на основе обмена информацией (сообщениями) – когда необходимо передавать от одного ПП другому более подробную информацию, чем просто «сигнал-событие».

Эти механизмы реализуются в ВС на двух уровнях — системном и прикладном.

Механизм взаимодействия между ПП на системном уровне определяется на этапе разработки ВС и реализуется в средствами операционной системы (частично — с использованием аппаратных средств).

На прикладном уровне взаимодействие между ПП реализуется программистом средствами языка, на котором разрабатывается программное обеспечение.

## 14.2. Взаимное исключение

объект типа взаимное исключение (mutex) позволяет только одному потоку в данное время владеть им. если продолжать аналогии, то этот объект можно сравнить с эстафетной палочкой.

класс, инкапсулирующий взаимное исключение, — tmutex — находится в модуле ipcthrd.pas (пример ipcdemos). конструктор:

```
constructor create (const name: string);
```

задает имя создаваемого объекта. первоначально он не принадлежит никому. (но функция api createmutex, вызываемая в нем, позволяет передать созданный объект тому потоку, в котором это произошло.) далее метод

```
function get(timeout: integer): boolean;
```

производит попытку в течение timeout миллисекунд завладеть объектом (в этом случае результат равен true). если объект более не нужен, следует вызвать метод

```
function release: boolean;
```

программист может использовать взаимное исключение, чтобы избежать считывания и записи общей памяти несколькими потоками одновременно.

## 14.3. Синхронизация

подразумевает обмен сигналами между двумя или более процессами по установленному протоколу. Такой «сигнал» рассматривается как некоторое событие, вызывающее у получившего его процесса соответствующие действия.

Часто возникает необходимость передавать от одного ПП другому более подробную информацию, чем просто «сигнал-событие». В этом случае процессы согласуют свою работу на основе обмена сообщениями.

Перечисленные механизмы реализуются в ВС на двух уровнях - системном и прикладном.

Механизм взаимодействия между ПП на системном уровне определяется еще на этапе разработки ВС и реализуется в основном средствами операционной системы (частично — с использованием аппаратных средств).

На прикладном уровне взаимодействие между ПП реализуется программистом средствами языка, на котором разрабатывается программное обеспечение.

Наибольшими возможностями в этом отношении обладают так называемые языки реального времени и языки моделирования.

Языки моделирования по сравнению с языками реального времени требуют от разработчика значительно менее высокого уровня подготовки в области программирования, что обусловлено двумя обстоятельствами:

• во-первых, средства моделирования изначально ориентированы на квазипараллельную обработку параллельных процессов;

• во-вторых, механизмы реализации ПП относятся, как правило, к внутренней организации системы (языка) моделирования и их работа скрыта от программиста.

**Объекты синхронизации**

- **Мьютекс** (mutual exclusive) — двоичная переменная, находящаяся в двух состояниях — свободном и занятом. Операция «проверить» ждет освобождения мьютекса и переводит его в занятое состояние, операция «освободить» переводит в свободное состояние.
- **Семафор** — целая переменная с неотрицательным значением. Операция «проверить» вычитает значение семафора, если он положителен, и ожидает увеличения если равен нулю. Операция «освободить» увеличивает значение семафора на 1.

**Синхронизация в ОС Windows**

- Функция Sleep
- Объекты синхронизации
- Функции ожидания
- Критические секции

**Функция Sleep**

- VOID Sleep(DWORD dwMilliseconds);
- Переводит поток в состояние блокировки на указанное количество милисекунд.
- В качестве значения параметра можно использовать макрос INFINITE.
- Поток не продолжит работу в течении не менее чем указанное количество времени.

**Объекты синхронизации ОС Windows**

- Объекты синхронизации ОС Windows — это объекты, которые имеют дескрипторы и могут находится в двух состояниях: сигналированом и несигналированом.
- Объекты синхронизации используются в функциях ожидания.
- Функция ожидания позволяет процессу дождаться сигналирования одного или нескольких объектов.

- Все операции над объектами синхронизации (в т.ч. производимые функциями ожидания) атомарны.

**Мьютекс**

- Мьютекс — бинарный объект синхронизации. В каждый момент времени он свободен (сигналирован) или принадлежит какому-либо потоку (несигналирован).
- Успешно завершенная функция ожидания захватывает мьютекс. Для его освобождения поток использует функцию releaseMutex
- Если поток начал ожидание на мьютексе, которым уже владеет, то оно немедленно заканчивается. Функцию ReleaseMutex для освобождения необходимо вызвать столько раз, сколько была вызвана функция ожидания.

**Операции над мьютексами**

● Создание мьютекса
● Освобождение мьютекса

**Семафор**

- Семафор — объект, состояние которого описывается неотрицательным целым числом (с пределом значения).
- Семафор сигналирован, когда его значение положительно, несигналирован — когда оно равно нулю.
- Функция ожидания (после завершения ожидания при необходимости) уменьшает значение семафора на 1.
- Функция ReleaseSemaphore увеличивает значение семафора.
- Если же поток организует несколько ожиданий на одном семафоре не отпуская его, то каждое ожидание уменьшает семафор.

**Операции над семафорами**

● Создание семафора

● Увеличение значения семафора

**Таймер**

Таймер сигналируется по истечении определенного времени.
Таймер может быть

1. сбрасываемым вручную — остается сигналированным до установки нового значения времени (перезапуска);
2. с автоматическим сбросом — переходит в несигналированное состояние при завершении операции ожидания.

Периодический таймер автоматически перезапускается через определенный период времени.

**Операции над таймерами**

● Создание таймера

● Запуск таймера

## 14.4. Буфер сообщений

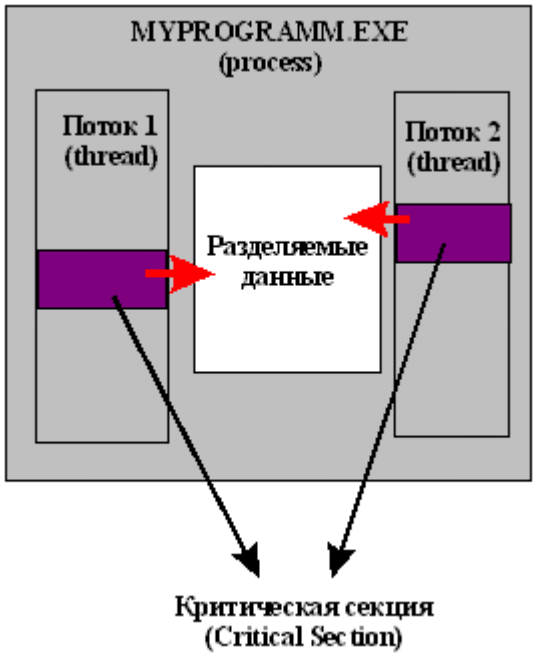
Каждый процесс способен создать любое количество структур называемых очередями: каждая структура может содержать любое количество сообщений разных типов, которые имеют разную природу и содержат любую информацию; каждый процесс способен послать сообщение в очередь, зная ее идентификатор. Также процесс способен получить доступ к очереди и прочитать сообщения в хронологическом порядке ( начиная с самого первого, последнего, самого недавнего и последнего, поступившего в очередь ), но выборочно, т.е. сообщения только определенного типа, что обеспечивает контроль за возможностью прочитать эти сообщения.

Использование очереди легко понять представив ее в виде почтовой системы между процессами: каждый процесс обладает адресом для взаимодействия с другими процессами. Процесс читает сообщения, предназначенные ему и дальнейшая его работа зависит от этих сообщений.

Итак, синхронизация двух процессов достигается использованием сообщений: ресурсы по-прежнему будут обладать семафорами, чтобы процессы знали их статус, а разделение времени работы происходит при помощи сообщений. Теперь вы понимаете, что использование сообщений не такая большая сложность как это могло показаться сначала.

## 14.5. Критический участок

Критическая секция (**Critical Section) это участок кода, в котором поток (thread) получает доступ к ресурсу (например переменная), который доступен из других потоков.**




Объект критическая секция обеспечивает синхронизацию. Этим объектом может владеть только один поток, что и обеспечивает синхронизацию. Для работы с критическими секциями есть ряд функций API и тип данных CRITICAL_SECTION. Для использования критической секции нужно создать переменную данного типа, и проинициализировать ее перед использованием с помощью функции InitializeCriticalSection(). Для того, чтобы войти в секцию нужно вызвать функцию EnterCriticalSection(), а после завершения работы LeaveCriticalSection(). Что будет, если поток обратится к секции, в которой сейчас другой поток ? Тот поток, который обратится будет блокирован пока критическая секция не будет освобождена. Саму критическую секцию можно удалить функцией DeleteCriticalSection(). Для того, чтобы обойти блокировку потока при обращении к занятой секции есть функция TryEnterCriticalSection(), которая позволяет проверить критическую секцию на занятость.

Здесь есть один маленький и тонкий момент, который относится к синхронизации как таковой. Этот момент в том, что физически данные не защищены. Никто не помещает потоку обратиться к разделяемым данным. Синхронизация это подсказка как должен себя вести объект при доступе к данным.

## 14.6. Семафор

Семафор (**semaphore) подобен взаимному исключению. Разница между ними в том, что семафор может управлять количеством потоков, которые имеют к нему доступ. Семафор устанавливается на предельное число потоков, которым доступ разрешен. Когда это число достигнуто, последующие потоки будут приостановлены, пока один или более потоков не отсоединятся от семафора и не освободят доступ.**

В качестве примера использования семафора рассмотрим случай, когда каждый из группы потоков работает с фрагментом совместно используемого пула памяти. Так как совместно используемая память допускает обращение к ней только определенного числа потоков, все прочие должны быть блокированы вплоть до момента, когда один или несколько пользователей пула откажутся от его совместного использования.

## 14.7. Порт

Параллельный адаптер - неотъемлемая составляющая ПК, имеющаяся как в самых первых реализациях IBM PC, так и в современных решениях. В состав компьютера входит, как правило, всего один параллельный адаптер. В самых новых моделях он соответствует стандарту IEEE 1284 - "Стандартный метод передачи сигналов для периферийного двунаправленного параллельного интерфейса для персональных компьютеров". Данный стандарт обеспечивает высокую скорость двунаправленной связи между PC и внешней периферией, которая может быть в 50-100 раз больше, чем у первых параллельных адаптеров. При этом сохраняется полная обратная совместимость со всеми существующими периферийными устройствами параллельного адаптера и принтерами.

Стандарт 1284 определяет 5 режимов передачи данных, каждый из которых обеспечивает способы передачи в прямом направлении (от PC к периферии), и обратном направлении (от периферии к PC) или двунаправленную передачу данных (полу дуплекс).
Существуют следующие режимы работы:
1) режим совместимости (Compatibility Mode) - Centronics, или стандартный режим.
2) режим тетрады (Nibble Mode) - 4 бита одновременно использующие линии состояния для чтения данных.
3) режим байта (Byte Mode) - 8 битов, одновременно использующие линии данных. Иногда рассматривается как "двунаправленный" порт.
4) EPP - режим (Enhanced Parallel Port) - расширенный параллельный порт. Предназначен преимущественно для работы не с принтерами, а CD-ROM'ами, ленточными накопителями, жесткими дисками, сетевыми адаптерами и т. д.
5) ECP-режим (Extended Capabilities Mode) - порт с расширенными способностями, прежде всего для нового поколения принтеров и сканеров.

Режим совместимости, или Centronics может посылать данные только в одном направлений с типичной скоростью 50 Кб в секунду.
ECP - и EPP - порты используют дополнительные аппаратные средства, чтобы управлять подтверждением связи (занято устройство или нет), так как программная проверка готовности устройства (принтера) ограничивает скорость обмена данными. Это предусматривает только одну инструкцию ввода – вывода, что позволяет увеличить скорость. Порты могут работать со скоростью 1 - 2 Мб в секунду. ECP - порт, также позволяет работать с каналом прямого доступа к памяти (DMA channel) и FIFO - буфера, что дает возможность перемещать данные не используя прямые операции ввода-вывода на порт.

## 14.8. Очереди событий

## 14.9. Проблема тупиков

Тупик – Такая ситуация, когда два или более процесса бесконечно долго ждут завершения, какого либо события, которое никогда не произойдёт.

Необх. Усл возникновения:
1) Условие неразделяемости ресурсов. Ресурс устроен таким образом, что в определённый момент им может владеть монопольно один процесс.
2)Условие ожидания. При запросе процессом уже занятого ресурса процесс блокируется до тех пор, пока ресурс не освободится.
3) условие неперераспределяемости ресурсов. При запросе процессом нового ресурса он не освобождает ранние им захваченные ресурсы.
4) условие кругового ожидания. Существует замкнутая цепь процессов, каждый из которых требует ресурса занятого процессом следующего по цепи.

Методы борьбы:
1) Предотвращение тупиков.
2) обход тупиков.
3) распознавание тупиков. Восстановление памяти.

Стратегии предотвращения тупиков.
Предотвращение тупиков. Заключается в нарушении одного из пунктов необходимого условия для возникновения тупиков:
1) условие неразделяемости ресурсов (условие невозможно нарушить программным способом, так как неразделяемость ресурсов – аппаратное свойство ресурсов).
2) условие ожидания (нарушается тем, что процессу все необходимые ему работы ресурсов выдаются перед его стартом, таким образом, процесс во время выполнения не требует никаких ресурсов и ничего не ожидает).
3) условие неперераспределяемости (нарушение заключается в том, что при запросе процессом уже занятого ресурса у него отнимаются все захваченные им ресурсы, и он блокируется до тех пор, пока не освободится требуемый ему ресурс плюс все ранние им захваченные). Недостатки: 1.1) после блокирования процесса он будет блокирован длительное время; 1.2) сложный алгоритм перераспределения ресурсов. 2.1) непредсказуемое время старта процесса; 2.2) дискриминация задач требующих большого количества ресурсов.
4) условие кругового ожидания (все ресурсы номеруются по типу. Если процесс требует ресурс типа с номером N, то он не должен занимать ресурсы с типами больше чем N. Если же он занимает ресурсы с типами больше чем N, то он должен их сначала освободить. Контроль за этим может лежать как по ОС, так и на самих пользователях). Недостатки: сложный алгоритм.