1. виртуальные методы

Методы описываются как виртуальные, если у предка и потомка имена методов и списки параметров совпадают, а тела методов различны.

Если в описании типа объекта есть виртуальные методы, то обязательно должен быть конструктор.

И конструктор должен быть запущен перед запуском виртуального метода, поскольку без этого действия не определён путь к виртуальному методу объекта.

Если метод не является виртуальным, связь его с объектом осуществляется компилятором на основании описания объекта, такая связь называется “ранним связыванием” метода с объектом.

Виртуальные методы связываются на этапе RUN при запуске конструктора. Это называется “позднее связывание” метода с объектом

1. полиморфизм. Простое переопределение методов

Полиморфизм позволяет по-разному использовать методы с одинаковыми именами в иерархии типов объектов.

Многообразие способов исполнять методы с одинаковыми именами проявляется в возможности переопределения статических методов в иерархии типов, в использовании виртуальных методов и в возможности вызова метода предка из потомка.

Простое переопределение методов используется, когда у методов меняется набор параметров.

1. полиморфизм. Вызов метода предка из потомка

Полиморфизм позволяет по-разному использовать методы с одинаковыми именами в иерархии типов объектов.

Многообразие способов исполнять методы с одинаковыми именами проявляется в возможности переопределения статических методов в иерархии типов, в использовании виртуальных методов и в возможности вызова метода предка из потомка.

Чтобы вызвать метод предка из потомка надо использовать inherited.

Пример: inherited Write(…);

1. наследование типов

С простейшим наследованием типа переменных программисты стал-киваются уже при формировании производных типов .

type

t=string[10];

tt=array of t ;

ранее описанный тип данного t наследуется типом tt.

Факт наследования полей данных и методов от предка фиксируется ссылкой в скобках после слова object на имя типа потомка.

Type

o1=object

A1,b1:real;

Procedure c1(…….) ;

Function ff (…..):**char**

End;

o2=object(o1)

A2:Boolean;

Procedure g2(……..);

End;

Var

ob1:o1;ob2:o2;

Имена полей данных у предка и потомка должны быть различными и описываться перед объявлением методов. Имена методов у предка и потомка могут совпадать. Существует несколько типов наследования полей методов:

1. Простое наследование, когда имена методов у предка и потомка не совпадают.
2. Наследование с переопределением методов, когда потомок использует метод с именем метода предка, то есть, переопределяет метод, чтобы иметь возможность работать именно со своим методом, не меняя описание методов предка.
3. Наследование при использовании виртуальных методов.
4. пирамидальная сортировка (+примеры)

Сложность: O(n log n)

Алгоритм пирамидальной сортировки примечателен тем, что независимо от набора данных у него одна и та же сложность по времени – O(n log n).

Пирамидальная сортировка использует бинарное сортирующее дерево. Сортирующее дерево — это такое дерево, у которого выполнены условия: Каждый лист имеет глубину либо d, либо d-1, где d — максимальная глубина дерева. Значение в любой вершине не меньше (другой вариант — не больше) значения её потомков.

**Достоинства**

Имеет доказанную оценку худшего случая O(n log n).

Сортирует на месте, то есть требует всего O(1) дополнительной памяти (если дерево организовывать так, как показано выше).

**Недостатки**

Сложен в реализации.

Неустойчив — для обеспечения устойчивости нужно расширять ключ.

На почти отсортированных массивах работает столь же долго, как и на хаотических данных.

На одном шаге выборку приходится делать хаотично по всей длине массива — поэтому алгоритм плохо сочетается с кэшированием и подкачкой памяти.

Методу требуется «мгновенный» прямой доступ; не работает на связанных списках и других структурах памяти последовательного доступа. Из-за сложности алгоритма выигрыш получается только на больших n. На небольших n (до нескольких тысяч) сортировка Шелла быстрее.

1. сортировка Шелла ( плюс пример)

Сложность: O(n)

Усовершенствованный вариант сортировки вставками. Сравниваются не только элементы, стоящие рядом, но и элементы, находящиеся на определённом расстоянии друг от друга.

1. Весь массив разбивается на несколько подмассивов, которые составляются из элементов основного массива, выбираемых с заданным шагом по индексу.
2. Все подмассивы сортируются независимо друг от друга одним из методов.
3. Частично отсортированный исходный массив снова разбивается на подмассивы уже с меньшим шагом разбиения по индексу и снова все массивы сортируются независимо.

Пример:

Возьмем массив A= \{ *56, 43, 12, 78, 42, 93, 16, 55* \} и смещения предложенные Шеллом.

|  |  |  |
| --- | --- | --- |
| **До** | **После** | **Описание шага** |
| *Шаг 1* i = t = 2 |  |  |
| *56, 43, 12, 78, 42, 93, 16, 55* | \{ *56, 42* \} \{ *43, 93* \} \{ *12, 16* \} \{ *78, 55* \} | Разбили массив на 4 списка. |
| *Шаг 2* |  |  |
| \{ *56, 42* \} \{ *43, 93* \} \{ *12, 16* \} \{ *78, 55* \} | \{ *42, 56* \} \{ *43, 93* \} \{ *12, 16* \} \{ *55, 78* \} | Отсортировали элементы списков сортировкой вставками. Количество обменов 2. |
| *Шаг 3* |  |  |
| \{ *42, 56* \} \{ *43, 93* \} \{ *12, 16* \} \{ *55, 78* \} | *42, 43, 12, 55, 56, 93, 16, 78* | Объединили списки в массив. Уменьшаем i на 1. i \geqslant 0, перейдем к шагу 1. |
| *Шаг 1* i = 1 |  |  |
| *42, 43, 12, 55, 56, 93, 16, 78* | \{ *42, 12, 56, 16* \} \{ *43, 55, 93, 78* \} | Разбили массив на 2 списка. |
| *Шаг 2* |  |  |
| \{ *42, 12, 56, 16* \} \{ *43, 55, 93, 78* \} | \{ *12, 16, 42, 56* \} \{ *43, 55, 78, 93* \} | Отсортировали элементы списков сортировкой вставками. Количество обменов 4. |
| *Шаг 3* |  |  |
| \{ *12, 16, 42, 56* \} \{ *43, 55, 78, 93* \} | *12, 43, 16, 55, 42, 78, 56, 93* | Объединили списки в массив. Уменьшаем i на 1. i \geqslant 0, перейдем к шагу 1. |
| *Шаг 1* i = 0 |  |  |
| *42, 43, 12, 55, 56, 93, 16, 78* | \{ *42, 43, 12, 55, 56, 93, 16, 78* \} | Разбили массив на 1 список. |
| *Шаг 2* |  |  |
| \{ *42, 43, 12, 55, 56, 93, 16, 78* \} | \{ *12, 16, 42, 43, 55, 56, 78, 93* \} | Отсортировали элементы списков сортировкой вставками. Количество обменов 7. |
| *Шаг 3* |  |  |
| \{ *12, 16, 42, 43, 55, 56, 78, 93* \} | *12, 16, 42, 43, 55, 56, 78, 93* | Объединили списки в массив. Уменьшаем i на 1. i<0. |

1. динамические объекты (описание, инициализация)

Существуют динамические и статические объекты. У статических объектов переменные сразу после её описания резервируется область памяти, и эта область закрепляется за переменной на все время работы программы. Такого рода переменные называют статическими переменными. Часто бывает так, что какая-то переменная нужна не на всё время выполнение программы, а только на какую-то часть этого времени. Такие временные программные объекты могут занимать значительный объем памяти и одновременное существование всех таких объектов может потребовать столь большого объема машинной памяти, что соответствующая программа просто не сможет разместиться в ограниченной оперативной памяти машины. Переменные, которые могут резервировать область памяти, а затем снова освобождать во время выполнения программы стали называть динамическими. Динамические объекты будут возникать в процессе выполнения программы, а действия над ними необходимо задавать уже до выполнения. В Паскале для работы с динамическими объектами предусматривается специальный тип переменных, так называемый ссылочный тип. Каждому динамическому объекту в программе сопоставляется статическая переменная ссылочного типа. В терминах этих ссылочных переменных обосновываются действия над соответствующими динамическими объектами.

Динамические объекты и ссылки:<имя ссылочного типа> = ^ <имя типа>, где ^ - называемая «домик», признак ссылочного типа, <имя типа> - это имя либо статического, либо ранее описанного типа значения.

1. процедурный тип данных (плюс пример)

Процедурный тип данных позволяет по-новому взглянуть на действия, осуществляемые в процедурах и функциях. В процедурном программировании, как правило, действия отделены от данных и не имеют типа. Так было во всех предыдущих примерах процедур. Однако часто необходимо использовать в качестве формальных и фактических параметров не имена или значения каких-либо данных, а имена функций или процедур. В этом случае необходимо, чтобы тип этих параметров – процедур был описан так же, как описываются обычные данные. Такая возможность появляется, благодаря наличию процедурного типа.

Этот тип данного позволяет задать и описать тип процедуры (функции), которая будет использоваться в качестве формального и фактического параметра.

1. механизм работы конструктора

Помимо действий, заложенных в него программистом, конструктор выполняет подготовку механизма позднего связывания виртуальных методов. Это означает, что еще до вызова любого виртуального метода должен быть выполнен какой-нибудь конструктор. Конструктор – это специальный метод, который инициализирует объект, содержащий виртуальные методы. Заголовок конструктора выглядит так:   
**constructor** **Method** (список параметров);

Основное и особенное назначение конструктора – установление связей с таблицей виртуальных методов ( ТВМ ) – структурой, содержащей ссылки на виртуальные методы. Таким образом, конструктор инициализирует объект установкой связи между объектом и ТВМ с адресами кодов виртуальных методов. При инициализации и происходит позднее связывание. У каждого объекта своя таблица виртуальных методов ТВМ. Именно это и позволяет одноименному методу вызывать различные процедуры. Если при создании какого-либо объекта инициализировать его не нужно, а объект тем не менее содержит виртуальные методы, конструктор все-таки должен быть. Однако в этом случае он представляет собой пустую подпрограмму, не выполняющую явно никаких действий,но заносящую информацию в таблицу виртуальных методов.

Упомянув о конструкторе, следует сказать и о деструкторе. Его роль противоположна: выполнить действия, завершающие работу с объектом, закрыть все файлы, очистить динамическую память, очистить экран и т.д.

Заголовок деструктора выглядит таким образом:

**destructor** **Done** ;

Основное назначение деструкторов – уничтожение ТВМ данного объекта. Часто деструктор не выполняет других действий и представляет собой пустую процедуру.

**destructor** **Done**;

**begin** **end**;

1. связь объектов и процедурного типа и record

В основе ООП лежит понятие объекта (object), сочетающего в себе данные и действия над ними.

Объект в некотором роде похож на стандартный тип-запись (record), но включает в себя не

только поля данных, но также и подпрограммы для обработки этих данных, называемые методами.

Таким образом, в объекте сосредоточены его свойства и поведение. Идеи создания нового

типа-объект были уже заложены при введении процедурного типа, отождествляющего между собой

данные и действия над ними. Фактически тип-объект включает в себя помимо данных элементы процедурных типов.

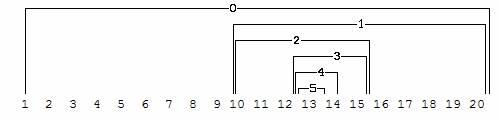
1. принципы поиска в упорядоченном массиве

1. Алгоритм бинарного поиска

Сложность: O(log n)

Рассмотрим упорядоченный по возрастанию набор данных, для которого выполняется условие: ***Массив>[i* - 1*]***? ***Массив[i]***для всех ***i***из интервала **(0, *n*)**.

* Обозначим ***Массив[k]***средний (по номеру) элемент набора данных, сравним его с искомым элементом ***X***.
* Переменные ***A***и ***B***содержат левую и правую границу отрезка набора данных, в котором находится искомый элемент. Если ***X***= ***Массив[k]***, то задача решена, ***X***> ***Массив[ k ]***, то это значит, что искомый элемент расположен ниже ***Массив[k]***(между элементами с номерами ***A = n*/2 + 1**и ***B = n***).
* Если ***X***< ***Массив[k]***, то это значит, что искомый элемент расположен выше среднего элемента (между элементами с номерами ***A* = 0**и ***B = n* /2 - 1**).
* После того как определена часть набора данных, в которой может находиться искомый элемент, по формуле ***(A + B)* /2**вычисляется новое значение ***k***, и поиск продолжается.
* Алгоритм бинарного поиска, заканчивает свою работу, если искомый элемент найден или если перед выполнением очередного цикла поиска обнаруживается, что значение ***A > B***.

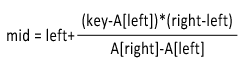


2. Алгоритм интерполяционного поиска

Сложность: O(loglog n) но может ухуджиться до O(n)

Алгоритм интерполяционного поска отличается от бинарного поиска выбором элемента, с которым сравнивается искомое значение. Чаще всего алгоритм интерполяционного поиска сравнивают с поиском слова в словаре. Представьте, что мы ищем слово «интерполяционный», мы откроем словарь не посередине, а ближе к началу. Таким образом, если мы примерно знаем закон возрастания элементов набора данных, то можем использовать эту информацию для поиска.

Формула, определяющая алгоритм интерполяционного поиска выглядит следующим образом:



mid:=left+((key-A[left])\*(right-left)) **div** (A[right]-A[left]);

Здесь mid – номер элемента, с которым сравнивается значение ключа, key – ключ (искомый элемент), A – массив упорядоченных элементов, left и right – номера крайних элементов области поиска. Важно отметить, операция деления в формуле строго целочисленная, т. е. дробная часть, какая бы она ни была, отбрасывается.



1. сортировка файлов

1. hash таблицы и смысл их применения

Хеш-таблица — это [структура данных](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%B0_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85), реализующая интерфейс [ассоциативного массива](https://ru.wikipedia.org/wiki/%D0%90%D1%81%D1%81%D0%BE%D1%86%D0%B8%D0%B0%D1%82%D0%B8%D0%B2%D0%BD%D1%8B%D0%B9_%D0%BC%D0%B0%D1%81%D1%81%D0%B8%D0%B2), а именно, она позволяет хранить пары (ключ, значение) и выполнять три операции: операцию добавления новой пары, операцию поиска и операцию удаления пары по ключу.

В общем случае хеш-таблица позволяет организовать массив, специфика которого проявляется в связанности индексов по отношению к хеш-функции; индексы могут быть не только целого типа данных (как это было в простых массивах), но и любого другого, для которого вычислимы хеш-коды. Данные, хранящиеся в виде такой структуры, удобны в обработке: хеш-таблица позволяет за минимальное время *O*(1) выполнять операции поиска, вставки и удаления элементов.

1. коллизии и их устранения при работе с хэш таблицам

Коллизий следует избегать, выбирая «хорошую» хеш-функцию, и используя один из методов разрешения конфликтов: открытое (метод цепочек) или закрытое (открытая адресация) хеширование.

**Открытое хеширование (метод цепочек)**

Принцип организации хеш-таблицы методом открытого хеширования заключается в реализации логически связанных цепочек, начинающихся в ячейках хеш-таблицы. Под цепочками подразумеваются связанные списки, указатели на которые хранятся в ячейках хеш-таблицы. Каждый элемент связанного списка – блок данных, и если с некоторым указателем, хранящимся по адресу i, связаны n таких блоков (n>1), то это свидетельствует о том, что n ключей получили один и тот же хеш-код i, т. е. имеет место коллизия. Но метод открытого хеширования устраняет конфликт, поскольку данные хранятся не в самой таблице, а в связных списках, которые увеличиваются при возникновении конфликта.

**Закрытое хеширование (метод с открытой адресацией)**

Первый метод назывался открытым, потому что он позволял хранить сколь угодно много элементов, а при закрытом хешировании их количество ограниченно размером хеш-таблицы. В отличие от открытого хеширования закрытое не требует каких-либо дополнительных структур данных. В ячейках таблицы хранятся не указатели, а элементы исходного массива, доступ к каждому из которых осуществляется по хеш-коду ключа, при этом одна ячейка может содержать только один элемент.

Сам процесс заполнения хеш-таблицы с использованием алгоритма закрытого хеширования осуществляется следующим образом:

имеется изначально пустая хеш-таблица T размера M, массив A размера N (M≥N) и хеш-функция h(), пригодная для обработки ключей массива A;

элемент xi, ключ которого keyi, помещается в одну из ячеек хеш-таблицы, руководствуясь следующим правилом:

если h(keyi) – номер свободной ячейки таблицы T, то в последнюю записывается xi;

если h(keyi) – номер уже занятой ячейки таблицы T, то на занятость проверяется другая ячейка, если она свободна то xi заноситься в нее, иначе вновь проверяется другая ячейка, и так до тех пор, пока не найдется свободная или окажется, что все M ячеек таблицы заполнены.

1. виды хэш функций
2. метод деления. Некоторый целый ключ делится на размер таблицы и остаток от деления берется в качестве значения хеш-функции. Эта хеш-функция обозначается h (key) := key mod m.
3. метод середины квадрата. Ключ умножается сам на себя и в качестве индекса используется несколько средних цифр этого квадрата.
4. Аддитивный метод для строк (размер таблицы равен 256). Для строк вполне разумные результаты дает сложение всех символов и возврат остатка от деления на 256.
5. Исключающее ИЛИ для строк (размер таблицы равен 256). Этот метод аналогичен аддитивному, но успешно различает схожие слова и анаграммы (аддитивный метод даст одно значение для XY и YX). Метод заключается в том, что к элементам строки последовательно применяется операция "исключающее или". В алгоритме добавляется случайная компонента, чтобы еще улучшить результат.
6. варианты проявления полиморфизма

Полиморфизм позволяет по-разному использовать методы с одинаковыми именами в иерархии типов объектов.

Многообразие способов исполнять методы с одинаковыми именами проявляется в возможности переопределения статических методов в иерархии типов, в использовании виртуальных методов и в возможности вызова метода предка из потомка.

1. удаление, вставка и поиск в хэш таблице (вариант без списков)

1. Поиск ключа в хеш-таблице с цепочками.

1.1. Вычисляем хеш ключа, находим соответствующую ячейку. Для открытого хеширования ячейка это указатель на (односвязный) список пар "ключ — значение". Ячеек в массиве столько, сколько значений выдаёт хеш-функция.

1.2. Проходим по списку в поисках элемента с совпадающим ключом.

1.3. Нашли — элемент есть. Прошли весь список и не нашли — такого ключа в хеше нет.

2. Вставка ключа в хеш-таблицу с цепочками.

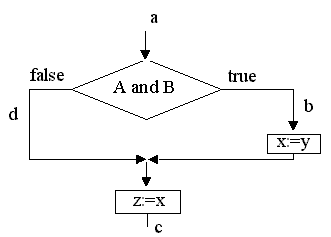
2.1. Проверяем наличие ключа в таблице как в пункте 1.

2.2. Если ключа нет, то приписываем новый элемент в начало списка соответствующей ячейки. Если ключ есть, то варианты: 1) ругаемся; 2) записываем в элемент новое значение.

1. критерии полноты тестирования в случае белого ящика

Критерии, по которым проводится классификация всех возможных вариантов выполнения программы с точки зрения проверки правильности программы, называются критериями полноты тестирования.

При подходе к тестированию программы как “белого ящика” могут использоваться следующие критерии полноты тестирования.



**1.Критерий покрытия операторов**, т.е. набор тестов должен обеспечивать исполнение всех операторов . С этой точки зрения дос-таточно проверить ситуацию A=true и B=true, при которой исполняется и логический оператор IF (символ “Решение”), и оператор при-своения x:=y. Ясно, что при этом даже не все пути проверяются.

**2. Критерий покрытия решений (или переходов).**

При этом необходимо пройти обе ветви всех символов “Решение” . С этой точки зрения достаточной является проверка ситуации A=true, B=true и A=true, B=false. При этом возможны и другие альтернативные, но тоже достаточные варианты значений A и B. В данном примере вариант A=true,B=false и A=false,B=true также решает задачу проверки всех пере-ходов.

**3.Критерий покрытия условий.**

При этом логические условия A и B должны принять различные воз-можные значения (в наборе тестов и A и B должны принимать значения и true и false). Нетрудно видеть, что тест A=true, B=false и A=false, B=true удовлетворяют этому требованию, однако при этом даже не все “решения” покрыты, т.е. этот критерий может оказаться слабее предыдущего крите-рия покрытия переходов.

**4.Покрытие решений и условий.** Это более сильный критерий , чем 2 и 3. Например, ситуация A=true,B=true и A=false ,B=false вместе покры-вают все условия и все переходы.

Однако все предыдущие критерии не способны обеспечить гарантии прохождения всех путей алгоритма. Поэтому самым сильным критерием является так называемое комбинаторное покрытие или покрытие всех путей.

**5.Комбинаторное покрытие условий.** Оно обеспечивает выполнение требований всех предыдущих критериев. Чтобы убедиться в этом, развернем более детально предыдущий алгоритм в виде, приведенным ниже.

Этот алгоритм учитывает всевозможные значения переменных A и B и их различные сочетания. Всего таких сочетаний 4 (A может быть true и false , и B может также принимать два возможных значения). Поэтому, не-смотря на идентичность путей abdh и abeh , все же число условий будет равно 4 , что эквивалентно четырем различным путям : abdh, abeh, acfh, acgh. Задавая различные комбинации значений A и B , получим возмож-ность проверки всех путей алгоритма , тем самым проходя все операторы

1. достоинства и недостатки подходов к тестированию программ (белый и чёрный ящик)

Белый ящик:

**Достоинства** тестирования «белого ящика» связаны с тем, что принцип «белого ящика» позволяет учесть особенности программных ошибок:

1. Количество ошибок минимально в «центре» и максимально на «периферии» программы.

2. Предварительные предположения о вероятности потока управления или данных в программе часто бывают некорректны. В результате типовым может стать маршрут, модель вычислений по которому проработана слабо.

3. При записи алгоритма ПО в виде текста на языке программирования возможно внесение типовых ошибок трансляции (синтаксических и семантических).

4. Некоторые результаты в программе зависят не от исходных данных, а от внутренних состояний программы.

Каждая из этих причин является аргументом для проведения тестирования по принципу «белого ящика». Тесты «черного ящика» не смогут реагировать на ошибки таких типов.

**Недостатки** тестирования «белого ящика»:

1. Количество независимых маршрутов может быть очень велико.

2. Исчерпывающее тестирование маршрутов не гарантирует соответствия программы исходным требованиям к ней.

3. В программе могут быть пропущены некоторые маршруты.

4. Нельзя обнаружить ошибки, появление которых зависит от обрабатываемых данных (это ошибки, обусловленные выражениями типа if abs (a-b) < eps..., if(a+b+c)/3=a...).

**Преимущества** тестирования “чёрного ящика”:

1. Тестирование с точки зрения пользователя

2. Не требует специальных знаний (например, конкретного языка программирования)

3. Позволяет найти проблемы в спецификациях

4. Можно создавать тесты параллельно с кодом

5. Тестировщик может быть отделен от разработчиков

**Недостатки** тестирования “чёрного ящика”:

1. Эффективность зависит от выбора конкретных тестовых значений

2. Необходимость наличия четких и полных спецификаций

3. Невозможность сконцентрироваться на особо сложных частях кода

4. Трудность локализации причины дефекта

5. Возможность не протестировать часть кода

1. типы тестирования

Тест- это совокупность входных и заранее известных выходных данных.

Существует два подхода к формулированию критериев полноты тестирования: критерии «черного ящика» и критерии «белого ящика».

Подход к программе как к белому ящику: Внутренняя структура программы считается известной , и тестируется логика программы. При этом тестируются маршруты или пути программы. Путь – это последовательность операторов от самого начала до самого конца программы.

Подход к программе как к чёрному ящику: Внутренняя структура считается неизвестной, и проверяется правильность ее функционирования на различных наборах входных данных.

1. инициализация полей данных в объектах

Инициализация данных при работе с объектами:

1. Непосредственное присвоение полю данного значения

2. С помощью внешней процедуры инициализации полей

3. С помощью собственной процедуры инициализации, являющейся методом объекта (например constructor)