МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Кузбасский государственный технический университет имени Т. Ф. Горбачева»

Кафедра информационной безопасности

Составители
Е. В. Прокопенко
И. В. Чичерин

ТЕХНОЛОГИИ И МЕТОДЫ ПРОГРАММИРОВАНИЯ

Методические материалы

Рекомендованы учебно-методической комиссией специальности 10.05.03 Информационная безопасность автоматизированных систем в качестве электронного издания для использования в образовательном процессе

Кемерово 2018

Рецензенты

Стенин Д. В. – кандидат технических наук, доцент директор ИИТМА
Сыркин И. С. – кандидат технических наук, доцент кафедры информационных и автоматизированных производственных систем

Прокопенко Евгения Викторовна
Чичерин Иван Владимирович

Технологии и методы программирования: методические материалы [Электронный ресурс] для обучающихся специальности 10.05.03 Информационная безопасность автоматизированных систем очной формы обучения / сост. Е. В. Прокопенко, И. В. Чичерин; КузГТУ. – Электрон. издан. – Кемерово, 2018.

1. Введение. Основные управляющие конструкции языков высокого уровня

Алгоритм, записанный на языке программирования, называется программой. В алгоритме обязательно должны быть предусмотрены все ситуации, которые могут возникнуть в процессе решения комплекса задач.

Программирование – теоретическая и практическая деятельность, связанная с созданием программы.

Программа – результат интеллектуального труда, для которого характерно творчество, индивидуальность разработчиков. Вместе с тем программирование предполагает и рутинные работы, которые могут и должны иметь строгий регламент выполнения и соответствовать стандартам. Программирование базируется на комплексе научных дисциплин, направленных на исследование, разработку и применение методов и средств разработки программ (специального инструментария создания программы).

Эволюция языков программирования

20-е годы XIX века. Предварительная запись порядка действий машины на перфокарте для последующей автоматической реализации вычислений – программе (предложена Ч.Бэббиджем). Ада Лавлейс теоретически разработала некоторые приемы управления последовательностью вычислений, которые используются до сих пор.

40-е годы XX века. Создание программ на основе кодирования *машинных* команд (Грейс Мюрей Хоппер).

50–60-е годы. Роль программирования в машинных кодах уменьшается, появляются *процедурные* языки программирования высокого уровня (FORTRAN, ALGOL). Для преобразования команд в машинные коды используются *трансляторы*.

Середина 60-х годов. Создание специализированного языка программирования, состоящего из простых слов английского языка (BASIC), попытки создать универсальный язык (PL/1, АЛГОЛ68).

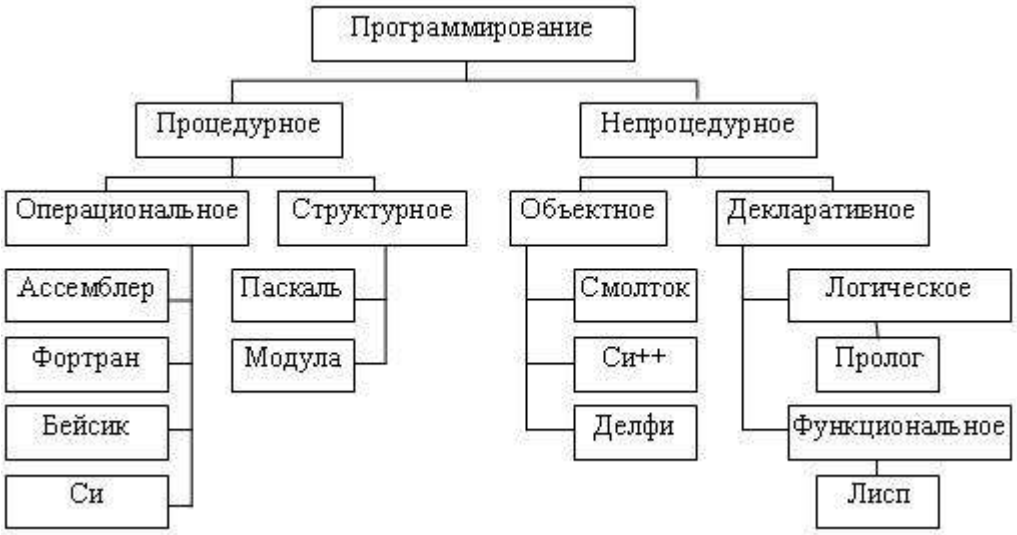
Начало 70-х годов. Развитие идеи АЛГОЛА о структуризации разработки алгоритмов, создание Н. Виртом языка Паскаль. Создание языка АДА, предназначенного для создания и длительного сопровождения больших программных систем, допускающего возможность параллельной обработки, управления процессами в реальном времени и др.

1972 г. (Первая версия языка Си). Появление языка сочетающего черты языка высокого уровня с *машинно-ориентированным* языком, который допускает программиста ко всем машинным ресурсам.

В течение многих лет программное обеспечение строилось на основе *операциональных* и *процедурных* языков (Ассемблеры, Фортран, Бейсик, Паскаль, Ада, Си). По мере эволюции языков программирования широкое распространение получили и другие принципиально новые подходы к созданию программ *непроцедурное* программирование: *объектно-ориентированное* программирование (языки Си++, Delphi, Visual Basic) и *декларативное* программирование. Декларативные языки делятся на *логические* (Пролог) и *функциональные*

(Лисп). В настоящее время разработаны языки работающие в управляемом окружении, обеспечивающие высокую надежность и защищенность создаваемых программ (Java, C#, VB.net).

Классификация языков программирования



Понятие о языках программирования высокого уровня

Языки программирования – это формальные языки специально созданные для общения человека с компьютером. Каждый язык программирования, равно как и «естественный язык» (русский, английский и т. д.) имеет:

- **Алфавит** – фиксированный для данного языка набор основных символов, допускаемых для составления текста программы на этом языке.
- **Синтаксис** – система правил, определяющих допустимые конструкции языка программирования.
- **Семантика** – система правил однозначного толкования отдельных языковых конструкций, позволяющих воспроизвести процесс обработки данных.

При описании языка и его применении используют **понятия языка.**

Понятие – некоторая синтаксическая конструкция и определяемые ею свойства программных объектов или процесса обработки данных.

Взаимодействие синтаксических и семантических правил определяет те или иные понятия языка, например, *операторы, идентификаторы, переменные, функции и процедуры, модули* и т. д. В отличие от естественных языков правила грамматики и семантики для языков программирования, как и для всех формальных языков, должны быть явно, однозначно и четко сформулированы.

Языки программирования, имитирующие естественные языки, обладающие укрупненными командами, ориентированными на решение прикладных содержательных задач, называются **языками высокого уровня.**

Язык программирования высокого уровня (highlevel programming language).

Язык программирования, в который введены элементы, допускающие описание задачи в наглядном, легко воспринимаемом виде, упрощающие и автомати-

зирующие процесс программирования, управляющие конструкции и структуры данных.

ЯПВУ отражают естественные для человека понятия, а не архитектуру вычислительной системы. Поэтому программа, составленная на ЯПВУ, сначала *транслируется* самой ЭВМ на машинный язык (низкого уровня), а затем выполняется.

В алфавит ЯПВУ могут входить буквы, цифры, математические символы и даже так называемые ключевые слова, например:

- if (если);
- then (тогда);
- else (иначе) и т. п.

Из исходных символов по правилам синтаксиса строятся предложения, обычно называемые операторами, например: if x *1 следует воспользоваться формулой y = x – 1*.

Достоинства ЯПВУ

- Алфавит языка значительно шире машинного, что делает его гораздо более выразительным и существенно повышает наглядность и понятность текста.
- Набор операций, допустимых для использования, не зависит от набора машинных операций, а выбирается из соображений удобства формулирования алгоритмов решения задач определенного класса.
- Конструкции команд (операторов) отражают содержательные виды обработки данных и задаются в удобном для человека виде.
- Используется аппарат переменных и действий с ними.
- Поддерживается широкий набор типов данных.

Таким образом, языки программирования высокого уровня являются **машинно-независимыми** и требуют использования соответствующих программ переводчиков (трансляторов) для представления программы на языке машины, на которой она будет исполняться.

Примеры языков высокого уровня

Fortran. Первый компилируемый язык созданный Джимом Бэкусом в 50-е годы. Для этого языка было создано огромное количество библиотек, начиная от статических комплексов и кончая пакетами управления спутниками, поэтому Fortran продолжает активно использоваться во многих организациях, а сейчас ведутся работы над очередным стандартом Фортрана F2k, который появился в 2000 году. Имеется стандартная версия Фортрана HPF (High Perfomance Fortran) для параллельных супер компьютеров со множеством процессоров.

Cobol. Это компилируемый язык для применения в экономической области и решения бизнес-задач, разработанный в начале 60-х годов. Он отличается большой «многословностью» – его операторы выглядят как обычные английские фразы. В Коболе были реализованы очень мощные средства работы с большими объемами данных, хранящимися на различных внешних носителях. На этом языке создано много различных приложений, которые активно эксплуатируются и сегодня. Достаточно сказать, что наибольшую зарплату в США получают программисты на Коболе.

Algol. Компилируемый язык, созданный в 1960 году. Он был призван заменить Фортран, но из-за более сложной структуры не получил широкого распространения. В 1968 году была создана версия Алгол68,по своим возможностям опережающая и сегодня многие языки программирования, однако из-за отсутствия достаточно эффективных компьютеров для нее не удалось своевременно создать хорошие компиляторы.

Pascal. Язык Паскаль, созданный в конце 70-х годов основоположником множества идей современного программирования Николаусом Виртом, во многом напоминает Алгол, но в нем ужесточен ряд требований к структуре программы и имеются возможности, позволяющие успешно применять его при создании крупных проектов.

Basic. Для этого языка имеются и компиляторы, и интерпретаторы, а по популярности он занимает первое место в мире. Он создавался в конце 60-х годов в качестве учебного пособия и очень прост в изучении.

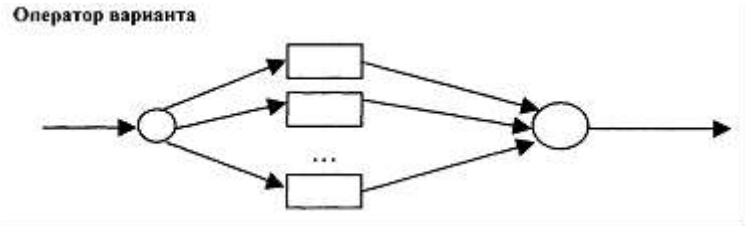
C. Данный язык был создан в лаборатории Bell и первоначально не рассматривался как массовый. Он планировался для замены ассемблера, чтобы иметь возможность создавать столь же эффективные и компактные программы, и в то же время не зависеть от конкретного вида процессора.

C++. Это объектно-ориентированное расширения языка Си, созданное Бьярном Страуструпом в 1980 году. Множество новых мощных возможностей, позволивших резко увеличить производительность программистов, наложилось на унаследованную от языка Си определенную низкоуровневость, в результате чего создание сложных и надежных программ потребовало от разработчиков высокого уровня профессиональной подготовки.

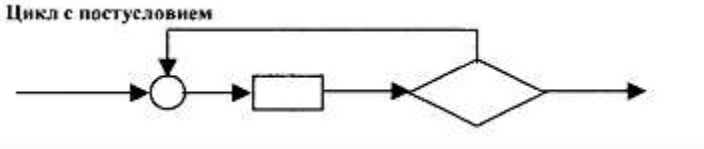
Java. Этот язык был создан компанией Sun в начале 60-х годов на основе Си++. Он призван упростить разработку приложений на основе Си++ путем исключения из него всех низкоуровневых возможностей. Но главная особенность этого языка – компиляция не в машинный код, а в платформо-независимый байт-код. Этот байт-код может выполняться с помощью интерпретатора виртуальной машины Java-машины JVM (Java Virtyal Machine), версии которой созданы сегодня для любых платформ. Благодаря наличию Java-машин программы на Java можно переносить не только на уровне исходных текстов, но и на уровне обычного байт-кода, поэтому по популярности язык Ява сегодня занимает второе место в мире после Бейсика.

Структурность управляющих конструкций

Второй составляющей структурного программирования является структурность применяемых управляющих конструкций языка (управляющих операторов). Это вытекает из требования, чтобы процесс исполнения управляющих конструкций был легко понимаем по их текстуальному представлению. Свойство структурности управляющих конструкций состоит в том, что каждая из них должна иметь в тексте одну входную точку (с которой начинается исполнение конструкции) и одну выходную точку (в которой завершается исполнение конструкции).



Большинство управляющих конструкций языков программирования построено именно таким образом. Такими конструкциями являются составной оператор, условный оператор, оператор варианта, операторы циклов. Приведем диаграммы некоторых управляющих конструкций языка, демонстрирующих требование к структурности этих конструкций.



Оператор варианта

Структурированная программа не должна содержать операторы перехода в произвольную точку текста, композиционно не связанную с текущей точкой. В версии языков программирования (например, Турбо Паскаль 7.0) включаются дополнительные операторы (например, Exit, Continue и Break), способствующие созданию структурных управляющих конструкций без применения операторов перехода.

Надежное программирование

Стиль надежного программирования, по существу, впитал в себя все идеи модульного и структурного стилей, в то же время акцентируя внимание на некоторых специфических аспектах. Являясь преемником структурного подхода, надежное программирование ужесточает требования и регламенты, которые должны выполняться на всех этапах разработки.

Одной из концепций надежного стиля разработки является использование языков высокого уровня, не зависящих от реализации. Богатство и разнообразие средств языка высокого уровня позволяет кодировать программу разными способами. Специфика надежного программирования состоит в выборе таких способов программирования, которые повышают надежность программы. В понятие "надежность" входят две составляющие корректность и устойчивость.

Понятия и средства надежного программирования

Надежность является частью более общего понятия «качество». Качественная программа не только надежна, но и компактна, совместима с другими программами, эффективна, удобна в сопровождении и вполне понятна.

Создано множество технологических средств и методов, являющихся обязательными при разработке надежных программ. Укажем только некоторые из них.

1. Надежный стиль программирования обеспечивается применением способов надежного программирования. Эти способы представляют собой совокупность приемов программирования. Каждый из них состоит в применении опреде-

ленных языковых средств и композиций в конкретных ситуациях, алгоритмах, схемах вычислений и представлениях структур данных. Основная цель наших лекций изложение таких приемов с иллюстрацией их применения для конкретных алгоритмов, схем вычислений и представлений данных.

2. Разработаны (и применяются в крупных программных продуктах) методы защиты от ошибок. Они позволяют создавать программы, работающие при наличии ошибок (ошибок пользователя, программной среды, сбоев аппаратуры).

Их сущность сводится к следующему.

Ограничение последствий ошибки: программа строится так, чтобы ошибка не искажала ее работу вне того участка, где эта ошибка возникла.

Локализация ошибки: программа содержит процедуры для возобновления ее правильной работы после устранения ошибки.

Дуальное программирование: программа сдержит избыточные ветви, позволяющие заменить хотя бы частично неверные ветви в момент возникновения ошибки.

3. Большое внимание в надежном программировании уделяется вопросам тестирования программ. Отметим здесь лишь некоторые специальные программные инструментальные средства, которые рекомендуется применять при тестировании.

Генератор данных для тестирования, создающий тесты, удовлетворяющие заданным требованиям.

Диспетчер тестирования. При нисходящем методе разработки и тестирования программ частями такого диспетчера являются заглушки модулей. При восходящем методе диспетчер тестирования разрабатывается как совокупность программ, каждая из которых тестирует один или несколько разрабатываемых модулей проекта.

Имитатор внешней среды (в противоположность диспетчеру тестирования) используется как средство тестирования программы в целом. Имитаторы, как правило, очень сложны. Они особенно полезны при разработке программ, работающих в реальном масштабе времени проверяется взаимодействие с внешними устройствами (космическими аппаратами, самолетами, датчиками от производственных процессов).

Разнообразие внешних воздействий и взаимодействие с внешней средой описывается с помощью сценариев имитации окружающей среды.

Требования к надежным программам

Понятие «надежность» имеет две составляющие: корректность и устойчивость. Корректность – свойство программы удовлетворять поставленным требованиям, т. е. получать результаты, точно соответствующие решению задачи и требованиям к ее интерфейсу. Устойчивость способность программы отслеживать ошибки при вводе и вычислении данных и сообщать об этих ситуациях, вместо выдачи неправильных результатов.

Другими словами, надежная программа гарантирует правильность получаемых результатов и безотказность программы. Последнее означает, что программа должна быть такова, что ее исполнение на каждом из допустимых наборов исходных данных приводил к ожидаемым результатам ее выполнения, не должно воз-

никать непредвиденных ситуаций, например переполнения, нехватки памяти, зацикливаний и т. д.

При построении устойчивой (безотказной) программы возникают два вопроса:

как определить возможность и место возникновения непредвиденных ситуаций,

как построить программу, чтобы она не допускала возникновения таких ситуаций.

Ответ на первый вопрос заключается в том, что при создании программы нужно не просто программировать вычисления, а представлять, как они будут выполняться с разными значениями операндов. Очень важно тщательно протестировать программу для граничных значений аргументов операций.

Отсюда следует, что один из важнейших принципов надежного программирования определение (подтверждение) области допустимых значений данных на основе анализа и тестирования программы.

Ответ на второй вопрос заключается в применении приемов надежного программирования.

Вся совокупность таких приемов естественным образом разбивается на две группы:

как нужно представлять данные,

какие применять средства и способы обработки данных.

Программа, построенная с применением приемов надежного программирования, должна:

1) сообщать пользователю об области допустимых значений исходных данных при формулировке задачи или при вводе данных;

2) контролировать значения исходных данных при их вводе, сообщать о невозможности выполнения вычислений для недопустимых значений;

3) обеспечивать для каждой из подобластей допустимых значений соответствующие ей вычисления, которые могут отличаться:

типами данных, участвующих в вычислениях,

алгоритмами, схемами вычислений;

4) контролировать промежуточные результаты вычислений, прекращать вычисления или изменять их порядок при обнаружении недопустимых ситуаций.

2. Работа с памятью. Структуры данных

Физическая (оперативная) память может быть представлена как массив байт. Процессор имеет возможность обращаться к данным из этого массива по индексу ячейки памяти (физическому адресу). В старых процессорах (например, i8086) каждый процесс использовал команды процессора для физической адресации к оперативной памяти, что, конечно, приводило к многочисленным ошибкам при неправильной (или злонамеренной) работе с памятью. Попытки разделить доступ разным процессам к физической памяти привели к появлению в i80286 процессорах защищенного режима (*protected mode*)[1]. В i80386 процессоре защищен-

ный режим был расширен механизмом страничной адресации, которая по сей день является основным механизмом изоляции памяти процессов.

Работал этот механизм приблизительно так: каждый процесс мог обращаться к любой ячейке памяти из диапазона [0, 2^32 1] (такой диапазон называется виртуальным адресным пространством). Адресное пространство (виртуальное и физическое) условно делилось на блоки (страницы) по 4Кб, таким образом адрес ячейки (32-битное число) естественным образом можно было интерпретировать как пару (индекс страницы (20 бит), смещение в странице (12 бит)).

Получив запрос на обращение к ячейке памяти (p, o), процессор обращался к уникальной для каждого процесса таблице (в первом приближении ее можно считать массивом из 2^20 32битных чисел) по индексу страницы p. В этой таблице для всех страниц виртуального адресного пространства процесса прописывались индексы страниц физической памяти (20-битное число) и некоторая служебная информация (12-битное число: флаг доступности страницы в физической памяти (present flag), флаг возможности записи (write flag), флаг изменения страницы (dirty flag) и т. д.). В случае, если страница обнаруживалась в физической памяти, процессор вычислял физический адрес искомой ячейки, взяв ее смещение относительно начала страницы. В случае же если искомой странице не соответствовала страница в физической памяти, процессор бросал исключение page fault, которое перехватывала операционная система. Конечно, 4Мб на каждый процесс непозволительная трата ресурсов, поэтому вместо массива использовалось двухуровневое дерево. Массив из 2^20 элементов условно делился на 2^10 блока по 2^10 записей. Если блок полностью состоял из отсутствующих в физической памяти страниц, страницы, содержащей его, не было. Список из 2^10 блоков содержался в специальной странице.

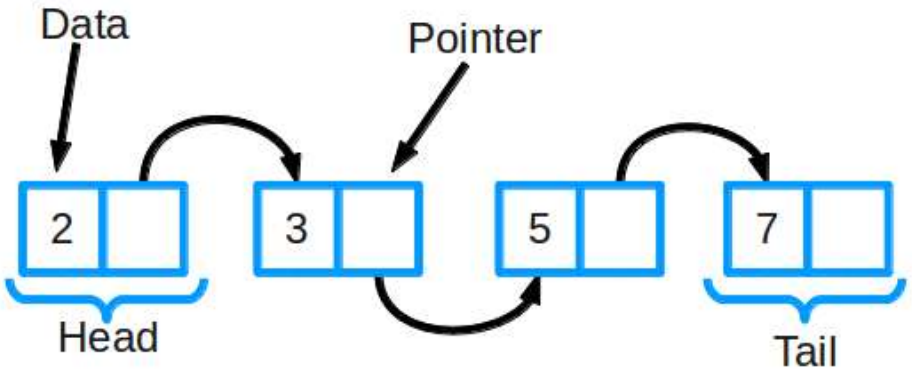
Примерно так работает страничная адресация и в современных процессорах. Как же заполняется таблица виртуального адресного пространства? Операционная система позволяет с помощью своего API резервировать и освобождать страницы в виртуальном адресном пространстве и сопоставлять этим страницам физическую память. Конечно, если все процессы начнут забирать физическую память, рано или поздно система не сможет найти свободную страницу в физической памяти. В этом случае она начнет использовать файл подкачки (или раздел жесткого диска, как в linux). В первом приближении этот механизм работает так: ОС выбирает страницу, которую давно не использовали и, если она была модифицирована (установлен флаг dirty) или если ее образа нет в файле подкачки, сохраняет страницу в файле подкачки. Далее ОС модифицирует записи в таблицах виртуальных адресных пространств процессов, использовавших эту страницу, сбрасывая флаг present. На место этой страницы помещается страница из файла подкачки (если процесс хотел обратиться к странице, которую уже когдато использовал) или она просто заполняется нулями. Страницы могут подгружаться не только из файла подкачки в адресное пространство загружается код процесса и код всех его зависимостей. В адресное пространство может быть спроецирован файл с жесткого диска. В этих случаях страницы могут загружаться из соответствующих файлов.

Структуры данных играют важную роль в процессе разработки ПО, а еще по ним часто задают вопросы на собеседованиях для разработчиков. Хорошая новость в том, что по сути они представляют собой всего лишь специальные форматы для организации и хранения данных

Ниже представлены десять самых распространенных структур данных.

1. Связные списки

Связный список – одна из базовых структур данных. Ее часто сравнивают с массивом, так как многие другие структуры можно реализовать с помощью либо массива, либо связного списка. У этих двух типов есть преимущества и недостатки.



Устройство связного список

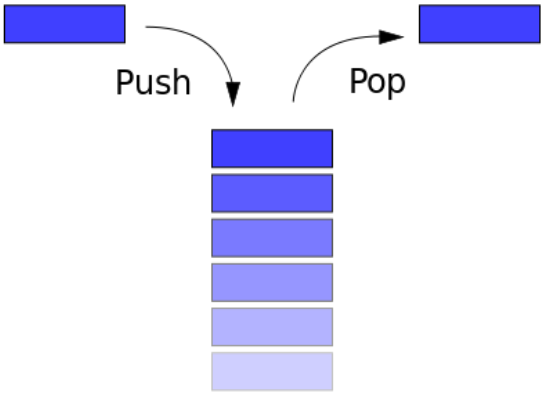
Связный список состоит из группы узлов, которые вместе образуют последовательность. Каждый узел содержит две вещи: фактические данные, которые в нем хранятся (это могут быть данные любого типа) и указатель (или ссылку) на следующий узел в последовательности. Также существуют двусвязные списки: в них у каждого узла есть указатель и на следующий, и на предыдущий элемент в списке.

Основные операции в связном списке включают добавление, удаление и поиск элемента в списке.

2. Стеки

Стек – это базовая структура данных, которая позволяет добавлять или удалять элементы только в её начале. Она похожа на стопку книг: если вы хотите взглянуть на книгу в середине стека, сперва придется убрать лежащие сверху.

Стек организован по принципу LIFO (Last In First Out, «последним пришёл – первым вышел»). Это значит, что последний элемент, который вы добавили в стек, первым выйдет из него.

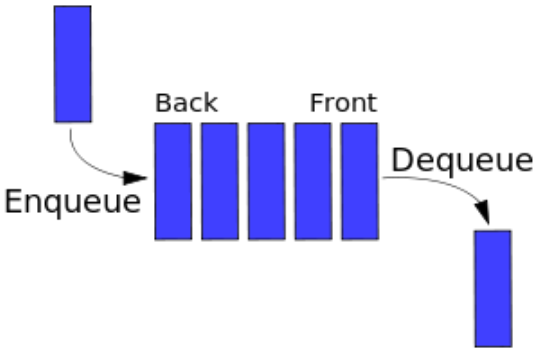


Устройство стека

В стеках можно выполнять три операции: добавление элемента (push), удаление элемента (pop) и отображение содержимого стека (pip).

3. Очереди

Эту структуру можно представить как очередь в продуктовом магазине. Первым обслуживают того, кто пришёл в самом начале — всё как в жизни.

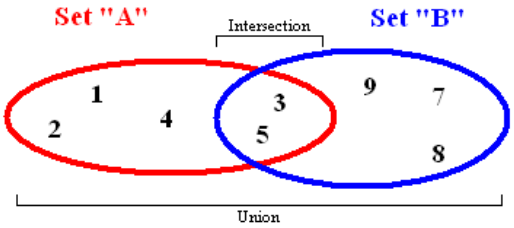


Устройство очереди

Очередь устроена по принципу FIFO (First In First Out, «первый пришёл — первый вышел»). Это значит, что удалить элемент можно только после того, как были убраны все ранее добавленные элементы.

Очередь позволяет выполнять две основных операции: добавлять элементы в конец очереди (*enqueue*) и удалять первый элемент (*dequeue*).

4. Множества



Устройство множества

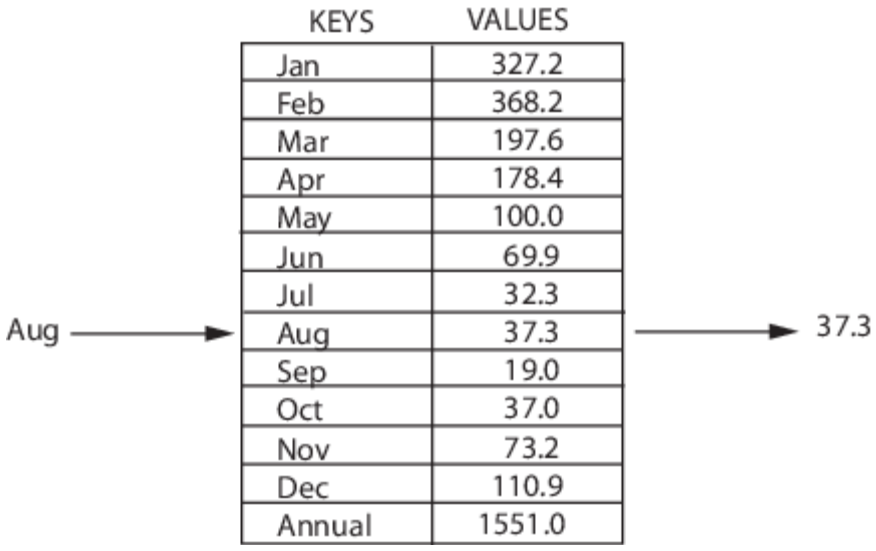
Множество хранит значения данных без определенного порядка, не повторяя их. Оно позволяет не только добавлять и удалять элементы: есть ещё несколько важных функций, которые можно применять к двум множествам сразу.

- Объединение комбинирует все элементы из двух разных множеств, превращая их в одно (без дубликатов).
- Пересечение анализирует два множества и создает еще одно из тех элементов, которые присутствуют в обоих изначальных множествах.
- Разность выводит список элементов, которые есть в одном множестве, но отсутствуют в другом.
- Подмножество выдает булево значение, которое показывает, включает ли одно множество все элементы другого множества.

5. Map

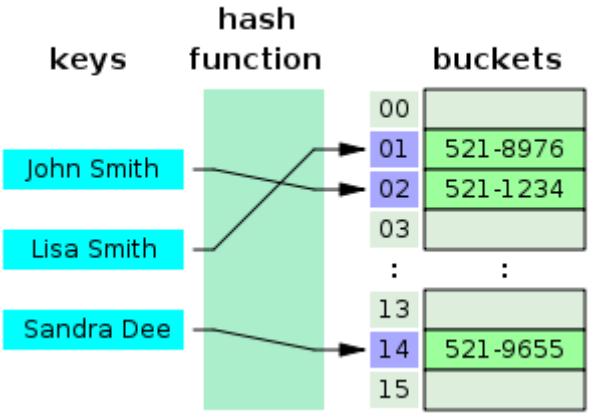
Map – это структура, которая хранит данные в парах ключ/значение, где каждый ключ уникален. Иногда её также называют ассоциативным массивом или словарём. Map часто используют для быстрого поиска данных. Она позволяет делать следующие вещи:

- добавлять пары в коллекцию;
- удалять пары из коллекции;
- изменять существующей пары;
- искать значение, связанное с определенным ключом.



Устройство структуры map

6. *Хэш-таблицы*



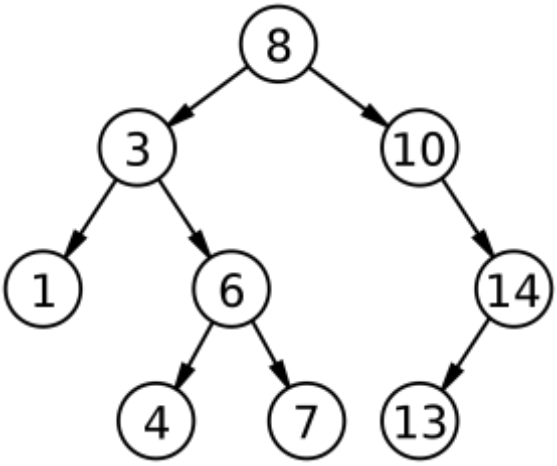
Работа хэш-таблиц и хэш-функции

Хэш-таблица – это похожая на Map структура, которая содержит пары ключ/значение. Она использует хэш-функцию для вычисления индекса в массиве из блоков данных, чтобы найти желаемое значение.

Обычно хэш-функция принимает строку символов в качестве вводных данных и выводит числовое значение. Для одного и того же ввода хэш-функция должна возвращать одинаковое число. Если два разных ввода хэшируются с одним и тем же итогом, возникает коллизия. Цель в том, чтобы таких случаев было как можно меньше.

Таким образом, когда вы вводите пару ключ/значение в хэш-таблицу, ключ проходит через хэш-функцию и превращается в число. В дальнейшем это число используется как фактический ключ, который соответствует определенному значению. Когда вы снова введёте тот же ключ, хэш-функция обработает его и вернет такой же числовой результат. Затем этот результат будет использован для поиска связанного значения. Такой подход заметно сокращает среднее время поиска.

7. *Двоичное дерево поиска*



Двоичное дерево поиска

Дерево – это структура данных, состоящая из узлов. Ей присущи следующие свойства:

- Каждое дерево имеет корневой узел (вверху).

- Корневой узел имеет ноль или более дочерних узлов.
- Каждый дочерний узел имеет ноль или более дочерних узлов, и так далее.

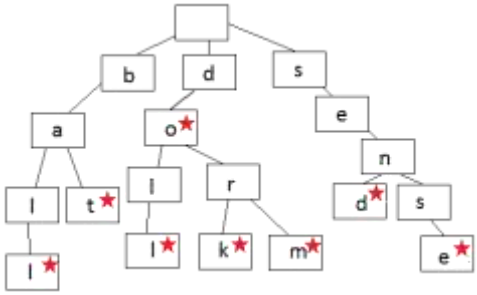
У двоичного дерева поиска есть два дополнительных свойства:

- Каждый узел имеет до двух дочерних узлов (потомков).
- Каждый узел меньше своих потомков справа, а его потомки слева меньше его самого.

Двоичные деревья поиска позволяют быстро находить, добавлять и удалять элементы. Они устроены так, что время каждой операции пропорционально логарифму общего числа элементов в дереве.

8. Префиксное дерево

Префиксное (нагруженное) дерево – это разновидность дерева поиска. Оно хранит данные в метках, каждая из которых представляет собой узел на дереве. Такие структуры часто используют, чтобы хранить слова и выполнять быстрый поиск по ним – например, для функции автозаполнения.

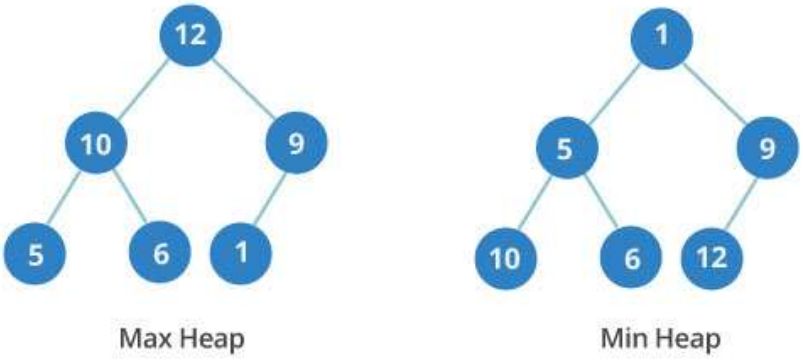


Устройство префиксного дерева

Каждый узел в языковом префиксном дереве содержит одну букву слова. Чтобы составить слово, нужно следовать по ветвям дерева, проходя по одной букве за раз. Дерево начинает ветвиться, когда порядок букв отличается от других имеющихся в нем слов или когда слово заканчивается. Каждый узел содержит букву (данные) и булево значение, которое указывает, является ли он последним в слове.

9. Двоичная куча

Двоичная куча – ещё одна древовидная структура данных. В ней у каждого узла не более двух потомков. Также она является совершенным деревом: это значит, что в ней полностью заняты данными все уровни, а последний заполнен слева направо.



Устройство минимальной и максимальной кучи

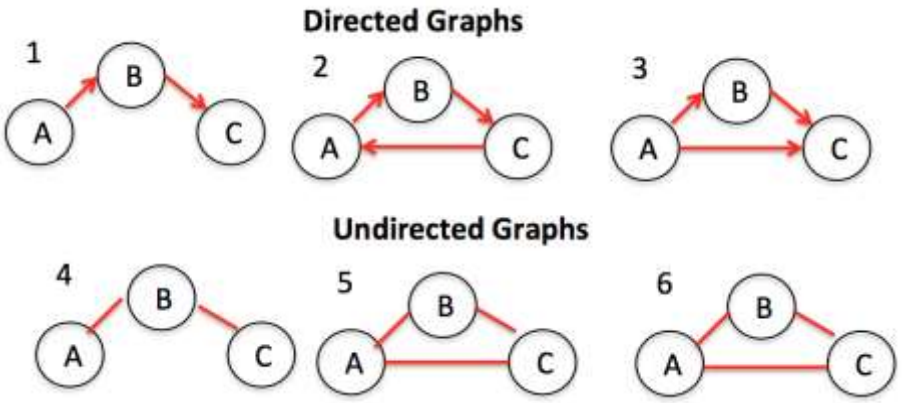
Двоичная куча может быть минимальной или максимальной. В максимальной куче ключ любого узла всегда больше ключей его потомков или равен им. В минимальной куче всё устроено наоборот: ключ любого узла меньше ключей его потомков или равен им.

Порядок уровней в двоичной куче важен, в отличие от порядка узлов на одном и том же уровне. На иллюстрации видно, что в минимальной куче на третьем уровне значения идут не по порядку: 10, 6 и 12.

10. Граф

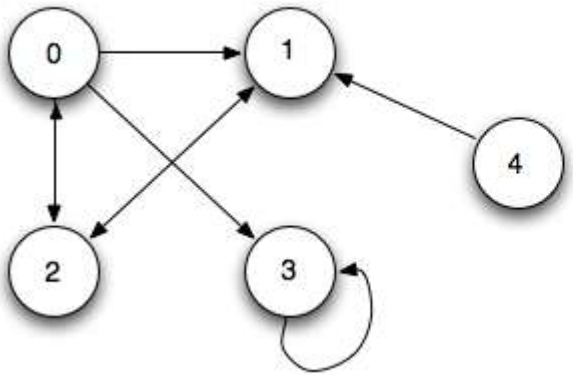
Графы – это совокупности узлов (вершин) и связей между ними (рёбер). Также их называют сетями.

По такому принципу устроены социальные сети: узлы – это люди, а рёбра – их отношения.



Графы делятся на два основных типа: ориентированные и неориентированные. У неориентированных графов рёбра между узлами не имеют какого-либо направления, тогда как у рёбер в ориентированных графах оно есть.

Чаще всего граф изображают в каком-либо из двух видов: это может быть список смежности или матрица смежности.



	0	1	2	3	4
0	0	1	1	1	0
1	0	0	1	0	0
2	1	1	0	0	0
3	0	0	0	1	0
4	0	1	0	0	0

Граф в виде матрицы смежности

Список смежности можно представить как перечень элементов, где слева находится один узел, а справа – все остальные узлы, с которыми он соединяется.

Матрица смежности – это сетка с числами, где каждый ряд или колонка соответствуют отдельному узлу в графе. На пересечении ряда и колонки находится число, которое указывает на наличие связи. Нули означают, что она отсутствует; единицы – что связь есть. Чтобы обозначить вес каждой связи, используют числа больше единицы.

Существуют специальные алгоритмы для просмотра рёбер и вершин в графах – так называемые алгоритмы обхода. К их основным типам относят поиск в ширину (*breadthfirst search*) и в глубину (*depthfirst search*). Как вариант, с их помощью можно определить, насколько близко к корневому узлу находятся те или иные вершины графа. В видео ниже показано, как на JavaScript выполнить поиск в ширину.

3. Введение в объектно-ориентированное программирование на языке C++ и абстрактные типы данных

Общие замечания

Объектно-ориентированная парадигма программирования не нова. Её истоки восходят к Симуле-67, хотя впервые она была полностью реализована в Smalltalk80. ООП (Объектно-ориентированное программирование) приобрело популярность во второй половине 80х вместе с такими языками, как C++, Objective C (другое расширение C), Object Pascal и Turbo Pascal, CLOS (объектно-ориентированное расширение Lisp'a), Eiffel, Ada (в её последних воплощениях) и недавно – в Java. В этой статье внимание сосредоточено на C++, Object Pascal и Java, иногда упоминаются и другие языки.

Ключевые черты ООП хорошо известны:

1. Первая – **инкапсуляция** – это определение классов – пользовательских типов данных, объединяющих своё содержимое в единый тип и реализующих некоторые операции или методы над ним. Классы обычно являются основой модульности, инкапсуляции и абстракции данных в языках ООП.

2. Вторая ключевая черта, – **наследование** – способ определения нового типа, когда новый тип наследует элементы (свойства и методы) существующего,

модифицируя или расширяя их. Это способствует выражению специализации и генерализации.

3. Третья черта, известная как **полиморфизм**, позволяет единообразно ссылаться на объекты различных классов (обычно внутри некоторой иерархии). Это делает классы ещё удобнее и облегчает расширение и поддержку программ, основанных на них.

Инкапсуляция, наследование и полиморфизм – фундаментальные свойства, которыми должен обладать язык, претендующий называться объектно-ориентированным (языки, не имеющие наследования и полиморфизма, но имеющие только классы, обычно называются основанными на классах). Различные ОО языки используют совершенно разные подходы. Мы можем различать ОО языки, сравнивая механизм контроля типов, способность поддерживать различные программные модели и то, какие объектные модели они поддерживают.

Алан Кей в свое время вывел пять основных черт языка Smalltalk – первого удачного ОО языка:

1. **Все является объектом.** Объект как хранит информацию, так и способен ее преобразовывать. В принципе любой элемент решаемой задачи (дом, собака, услуга, химическая реакция, город, космический корабль и т. д.) может представлять собой объект. Объект можно представить себе как швейцарский нож: он является набором различных ножей и «открывашек» (хранение), но в то же самое время им мы можем резать или открывать что-либо (преобразование).

2. **Программа – совокупность объектов, указывающих друг другу что делать.** Для обращения к одному объекту другой объект «посылает ему сообщение». Как вариант возможно и «ответное сообщение». Программу можно представить себе как совокупность к примеру 3 объектов: писателя, ручки и листа бумаги. Писатель «посылает сообщение» ручке, которая в свою очередь «посылает сообщение» листу бумаги – в результате мы видим текст (посыл сообщения от листа к писателю).

3. **Каждый объект имеет свою собственную «память» состоящую из других объектов.** Таким образом программист может скрыть сложность программы за довольно простыми объектами. К примеру, дом (достаточно сложный объект) состоит из дверей, комнат, окон, проводки и отопления. Дверь, в свою очередь, может состоять из собственно полотна, ручки, замка и петель. Проводка тоже состоит из проводов, розеток и, к примеру, щитка.

4. **У каждого объекта есть тип.** Иногда тип называют еще и классом. Класс (тип) определяет какие сообщения объекты могут посылать друг другу. Например, аккумуляторная батарея может передавать электролампе ток, а вот импульс или физическое усилие нет.

5. **Все объекты одного типа могут получать одинаковые сообщения.** К примеру у нас есть 2 объекта: синяя и красная кружки. Обе разные по форме и материалу. Но из обеих мы можем пить (или не пить, если они пустые). В данном случае кружка – это тип объекта.

Самое лаконичное описание объекта предложил *Буч*: «Объект обладает *состоянием*, *поведением* и *индивидуальностью*».

Контроль во время компиляции и во время выполнения

Языки программирования можно оценить по тому, насколько они строги к типам. Контроль типов включает проверку существования вызываемых методов, видов их параметров, проверку границ массивов и подобное.

C++, Java, и Object Pascal предпочитают более или менее тщательный контроль типов во время компиляции. C++, возможно, наименее точен в этом отношении (на что указывает, к примеру, возможность присвоения double к float), тогда как Java использует проверку типов наиболее широко. Это оттого, что C++ обеспечивает совместимость с Си, который не очень строго проверяет типы во время компиляции. Например, C и C++ считают, что все арифметические типы совместимы (хотя присвоение float целой переменной вызовет предупреждение компилятора). В Object Pascal и Java логическое значение не целое, а символ еще один отличный и несовместимый тип.

Тот факт, что виртуальная машина Java интерпретирует байтовый код во время выполнения, не означает, что этот язык отказывается от проверки типов во время компиляции. Наоборот, в этом языке проверка наиболее тщательна. Другие ОО языки, такие как Smalltalk и CLOS, наоборот, склонны большинство проверок типов (если не все) осуществлять во время исполнения.

Чисто объектно-ориентированные и гибридные языки

Различаются чистые и гибридные объектно-ориентированные языки. Чистые – языки, которые позволяют использовать только одну модель программирования – объектно-ориентированную. Можно объявлять классы и методы, но не можете завести глобальные переменные и обычные функции и процедуры старого типа.

Среди наших четырех языков только Java и C# являются чистыми ОО языками (как Eiffel и Smalltalk). На первый взгляд это кажется положительной идеей. Однако она ведет к тому, что вы используете кучу статических методов и статических данных, что не так уж отличается от использования глобальных функций и данных, за исключением более сложного синтаксиса. Чистые ОО языки дают преимущество новичкам в ООП, потому что программист вынужден использовать (и учить) модель ООП. C++ и Object Pascal, наоборот, типичные примеры гибридных языков, которые позволяют программистам использовать при необходимости традиционный подход C или Pascal.

Smalltalk расширяет эту идею до уровня «объектирования» таких предопределенных типов данных, как целые и символы, а также языковых конструкций (таких как циклы). Это теоретически интересно, но сильно уменьшает эффективность. Java и C# останавливаются намного раньше, допуская присутствие простых не ОО типов данных (хотя имеются необязательные классы обертки и для простых типов).

Простая объектная модель и ссылочно-объектная модель

Свойство: Третий элемент, по которому различаются языки ООП их **объектная модель**. Некоторые традиционные языки ООП позволяют программистам создавать объекты в стеке, в куче (в хипе heap) или в статической памяти. В этих языках переменная типа класс соответствует объекту в памяти. Так работает C++.

В последнее время появилась тенденция использовать другую модель, часто называемую ссылочно-объектной моделью. В этой модели каждый объект динамически размещается в куче, а переменная типа класс фактически является ссылкой или хэндлом объекта в памяти (технически это нечто вроде указателя). Java и Object Pascal оба используют эту ссылочную модель. В C# используется преимущественно ссылочно-объектная модель, однако имеется возможность создавать т. н. структуры (по сути дела, структура здесь специальная разновидность класса), объекты которых будут располагаться в стеке и статической памяти. Как мы увидим, вкратце это значит, что вам необходимо не забыть выделить память для объекта.

Классы, объекты и ссылки

Свойство: Так как мы обсуждаем языки ООП, то после этого введения, начнём обсуждать *классы* и *объекты*. Я надеюсь, что каждый ясно понимает разницу между этими двумя терминами. В двух словах, *класс* это тип данных, а *объект* экземпляр типа класс. «Кружка» это класс (тип). А уж которая, синяя или красная, это два разных объекта (экземпляра), типа «кружка». Как нам теперь использовать объекты в языках, использующих различные объектные модели?

C++: если у нас есть класс *MyClass* с методом *MyMethod*, мы можем написать:

```
MyClass Obj;
Obj.MyMethod();
```

и получить объект класса *MyClass* с именем *Obj*. Память для этого объекта обычно выделяется в стеке, и вы можете сразу начать использовать объект, как это сделано во второй строке.

Также возможно выделить память для объекта в куче и оперировать указателем на объект:

```
MyClass *obj = new MyClass();
obj>MyMethod();
delete obj;//освобождаем память
```

Java: подобная инструкция выделяет место только для хэндла объекта, а не для самого объекта:

```
MyClass obj;
obj = new MyClass();
obj.myMethod();
```

Прежде чем использовать объект, вы должны вызвать "new" для выделения под него памяти. Конечно, вы можете объявить и проинициализировать объект в одном предложении, избегая использования неинициализированных объектных хэндлов:

```
MyClass obj = new MyClass();
```

```
obj.myMethod();
```

Object Pascal: использует подобный подход, но требует отдельных предложений для объявления и инициализации:

```
var
 Obj: MyClass;
begin
 Obj := MyClass.Create;
 Obj.MyMethod;
```

В **C#** работа с объектами классов и структур внешне выглядит похоже:

```
MyClass obj1 = new MyClass();
obj1.Method1();
MyStruct1 obj2 = new MyStruct("Hello!");
obj2.Method2();
MyStruct2 obj3;
obj3.Method3();
```

Но все же есть одно принципиальное отличие память под объект структуры выделяется статически, поэтому вызов конструктора перед вызовом метода не является обязательным объект уже существует, хотя его поля могут содержать «мусорные» значения.

Замечание: Если вам кажется, что ссылочно-объектная модель требует большей работы от программиста, обратите внимание на то, что в C++ вы часто должны использовать указатели и ссылки на объекты. Только используя указатели и ссылки, вы можете добиться полиморфизма. Ссылочно-объектная модель, наоборот, делает использование указателей подразумеваемым, скрывая от программиста сложность этого подхода. В Java, в частности, официально указателей нет, хотя они там повсюду. Только программисты не имеют над ними прямого контроля, и поэтому, из соображений безопасности, не могут попасть в произвольное место памяти.

Мусорная корзина

Свойство: Если вы создали и не использовали объект, вам нужно уничтожить его, чтобы не занимать неиспользуемую память.

C++: В C++ уничтожить объект, расположенный в стеке, довольно просто. С другой стороны, уничтожение объектов, созданных динамически, зачастую является сложной проблемой. Есть много решений, включая подсчет ссылок и «интеллектуальные» указатели, но ни один из них не даёт простого решения. Первое впечатление для C++ программистов, что использование ссылочно-объектной модели сделает ситуацию только хуже.

Java: Это, конечно, не касается Java, так как виртуальная машина запускает алгоритм сборки мусора (в фоновом процессе, согласно теории Java; или начинает этот процесс после того, как ненадолго остановит программу, как в большинстве

реальных JVM). Сбор мусора происходит без участия программиста, но он может неблагоприятно влиять на эффективность выполнения приложения. Отсутствие явно записываемых деструкторов может приводить к ошибкам в завершающем коде (см. также главу о деструкторах и методе finalize() ниже).

OP: В Object Pascal, наоборот, нет механизма сбора мусора. Однако компоненты Delphi поддерживают идею владельца (owner) объекта: владелец становится ответственным за уничтожение всех объектов, которыми он владеет. Это делает управление уничтожением объекта очень простым и прямым. Delphi также использует механизм подсчёта ссылок для строк, динамических массивов и интерфейсов, освобождая объект в памяти, когда на него нет больше ссылок.

C#: В CLR (среде выполнения .NET) имеется сборщик мусора, аналогичный используемому в виртуальной машине Java. Поэтому программисту также не приходится заботиться об удалении созданных объектов. Однако, имеется специальный интерфейс IDisposable, которым «обозначаются» объекты, требующие ручной деинициализации (например, закрытия потоков вводавывода).

```
MyClass obj = new MyClass();//класс MyClass реализует интерфейс IDisposable
obj.Method1();
...
obj.Method2();
//объект obj нам больше не нужен
obj.Dispose();
```

Определение новых классов

Свойство: Теперь, когда мы рассмотрели, как создавать объекты для существующих классов, мы можем обратиться к определению новых классов. Класс – это просто набор методов, работающих с определёнными локальными данными.

C++: Вот C++ синтаксис определения простого класса:

```
class Date {
private:
 int dd;
 int mm;
 int yy;
public:
 void Init (int d, int m, int y);
 int Day ();
 int Month ();
 int Year ();
};
```

А вот определение пользовательского метода инициализатора:

```
void Date::Init (int d, int m, int y)
{
```

```
 dd = d;
 mm = m;
 yy = y;
}
```

Java: Синтаксис Java похож на синтаксис C++:

```
class Date {
 private int dd = 1;
 private int mm = 1;
 private int yy = 1;
 public void Init ( int d, int m, int y ) {
   dd = d;
   mm = m;
   yy = y;
 }
 public int getDay () { return dd; }
 public int getMonth () { return mm; }
 public int getYear () { return yy; }
}
```

Основная разница состоит в том, что код каждого метода пишется там же, где он объявляется (при этом функции не становятся вставными (inline), как в C++), и в том, что вы можете инициализировать элементы данных класса. Фактически, если вы не сделаете этого, то Java проинициализирует все элементы данных за вас, используя значения по умолчанию.

OP: В Object Pascal синтаксис определения класса другой, но похожий скорее на C++, чем на Java:

```
type
 Date = class
 private
   dd, mm, yy: Integer;
 public
   procedure Init (d, m, y: Integer);
   function GetMonth: Integer;
   function GetDay: Integer;
   function GetYear: Integer;
 end;

procedure Date.Init (d, m, y: Integer);
begin
 dd := d;
 mm := m;
 yy := y;
```

```
end;

function Date.GetDay: Integer;
begin
 Result := dd;
end;

...
```

Как видите, здесь есть синтаксические отличия: методы определяются с ключевыми словами **function** и **procedure**, методы без параметров не используют скобок, методы просто объявляются внутри определения класса, тогда как определяются позже, как это обычно делается в C++. Однако Pascal использует нотацию с точкой, а C++ – оператор : (недоступный в Object Pascal и Java).

C# Синтаксис C# очень похож на Java (в данном примере идентичен):

```
class Date
{
  private int dd;
  private int mm;
  private int yy;

  void init(int d, int m, int y)
  {
            dd = d;
            mm = m;
            yy = y;
  }

  int Day()
  {
            return dd;
  }

  ...
}
```

Примечание: Доступ к текущему объекту. В ОО языках методы отличаются от глобальных функций тем, что у них присутствует скрытый параметр, ссылка или указатель на объект, с которым мы работаем. Просто эта ссылка на текущий объект поразному называется. Это this в C++, Java и C#, Self в Object Pascal.

Создание и уничтожение объектов

Конструкторы

Свойство: Вышеупомянутый класс ужасно прост. Первое, что мы могли бы к нему добавить, это конструктор, что является хорошей техникой для решения проблемы инициализации объекта.

C++: В C++, так же, как и в Java, имя конструктора совпадает с именем класса. Если вы не определили никакого конструктора, компилятор синтезирует конструктор по умолчанию, добавляя его к классу. В обоих этих языках вы можете завести несколько конструкторов благодаря перегрузке функций.

```
class MyClass
{
private:
  int i;
public:
  MyClass(int i);
}

MyClass::MyClass(int i)
{
  this>i = i;
}
```

Java: Всё работает как в C++, хотя конструкторы называются также инициализаторами. Это подчеркивает тот факт, что объект создаёт виртуальная машина Java, тогда как код, который вы пишете в конструкторе, просто инициализирует свежесозданный объект. (То же самое фактически происходит и в Object Pascal.)

```
class MyClass {
  private int i;

  MyClass(int i) {
          this.i = i;
  }
}
```

OP: В Object Pascal вы используете специальное ключевое слово constructor и можете дать конструктору любое имя. Хотя Borland в Delphi 4 добавила поддержку перегрузки методов, программисты всё ещё дают разным конструкторам разные имена. В Object Pascal у каждого класса по умолчанию есть конструктор Create (наследуемый от TObject), если вы не перегрузите его конструктором с тем же именем и, возможно, другими параметрами. Этот конструктор, как мы увидим позднее, просто наследуется от общего базового класса.

```
type
  TMyClass = class
  private
          I: Integer;
  public
          constructor Create(AI: Integer);
  end;
...
constructor TMyClass(AI:Integer);
begin
  I := AI;
end;
```

C#: Синтаксис объявления конструктора очень похож на Java, а в нашем примере будет идентичен.

Деструкторы и финализация

Свойство: Деструктор играет роль противоположную конструктору и обычно вызывается при уничтожении объекта. Если конструктор нужен большинству классов, то только некоторые из них нуждаются в деструкторе. Деструктор в основном используется для освобождения ресурсов, зарезервированных конструктором (или другими методами во время жизни объекта). Эти ресурсы включают память, файлы, базы данных, ресурсы ОС и т. д.

C++: деструкторы вызываются автоматически, когда объект выходит из области определения или когда вы удаляете объект, заведенный динамически. У каждого класса есть только один деструктор, который объявляется как ~Class(), где Class имя класса. Если объект создан в куче, то он не может быть автоматически удален и если не объявить деструктор явно в программе, то происходит утечка памяти (в Java данная проблема решена сборщиком мусора).

OP: деструкторы похожи на деструкторы C++. (Для деструкторов используется ключевое слово destructor [мое примечание – В. К.]) Object Pascal использует стандартный виртуальный деструктор, называемый *Destroy*. Этот деструктор вызывается стандартным методом *Free*. Все объекты динамические, поэтому предполагается, что вы вызовете *Free* для каждого объекта, созданного вами, если у того нет владельца, отвечающего за его уничтожение. Теоретически вы можете объявить несколько деструкторов, что имеет смысл, если вы можете вызывать деструкторы в своем коде (это не делается автоматически).

Java: В Java нет деструкторов. Объекты, на которые нет ссылок, уничтожаются сборщиком мусора, который работает в виде фоновой задачи и делает это автоматически (как описывалось ранее). Прежде чем уничтожать объект, сборщик мусора должен вызвать метод *finalize()*. Однако нет никакой гарантии, что этот метод вызывается в каждой JVM. По этой причине, если вам нужно освободить ресурсы, вы должны добавить какой-нибудь метод, и убедиться, что он вызывается (эти дополнительные усилия не нужны в других ОО языках).

C#: Как и в виртуальной машине Java, в CLR используется автоматическая сборка мусора. Как было сказано выше, существует специальный интерфейс для объектов, требующих ручного освобождения ресурсов. В C# также можно создать метод вида ~*имя_класса*(), который полностью аналогичен методу *finalize()* в Java.

Инкапсуляция (Private и Public)

Свойство: Общим элементом всех трех языков является присутствие трех спецификаторов доступа, указывающих на различные уровни инкапсуляции класса: public, protected, и private. Public означает: видимый любым другим классом, protected означает: видимый производными классами, private означает: отсутствие видимости извне. В деталях, однако, есть различия.

C++: В C++ вы можете использовать ключевое слово friend для обхода инкапсуляции. Видимость по умолчанию для класса – private, для структур – public.

OP: В Object Pascal private и protected относятся только к классам других юнитов. В терминах C++, класс является дружественным для любого другого класса, определенного в том же юните (или файле исходного кода). В Delphi есть еще один модификатор доступа – published, который генерирует информацию времени выполнения (RTTI) об элементах.

Java: В Java отличие синтаксиса в том, что модификатор доступа повторяется для каждого элемента класса. А конкретнее, по умолчанию в Java используется friendly, это значит, что элемент видим для других классов этого же пакета (или файла исходного кода, как в OP). Подобным образом, protected означает видимость для подклассов.

C#: В CLR используется пять уровней доступа к членам класса, которые в C# обозначаются как private, protected, public, internal, protected internal. Первые три аналогичны соответствующим модификаторам в C++, Object Pascal и Java. Модификатор internal ограничивает видимость текущей сборкой (т. е. исполняемым файлом или файлом динамической библиотеки, к которому относится данный класс), protected internal разрешат также доступ в унаследованных классах из других сборок.

Файлы, модули и пакеты

Свойство: Важное различие между тремя языками заключается в организации исходного кода в файлах. Все три языка используют файлы в качестве стандартного механизма для запоминания исходного кода классов (в отличие от других ОО языков, таких как Smalltalk), но компилятор C++, в отличие от OP или Java, не понимает файлов. Эти же два языка работают с идеей модулей, хотя называют их поразному.

C++: В C++ программист обычно помещает определение класса в файл объявлений, а определение методов – в отдельный файл кода. Обычно у этих двух файлов одинаковые имена и различные расширения. Компилируемый блок, как правило, ссылается (включает в себя) на свой файл объявлений и на файлы объявлений тех классов (или функций), на которые ссылается код. Все эти соглашения не утруждают компилятор. Это значит, что линкеру предстоит большая работа, потому что компилятор не может знать, в каком другом модуле может быть определен нужный метод. Однако также существуют пространства имен, позволяю-

щие во многих случаях избежать неоднозначностей. Например, классы и прочие элементы библиотеки STL находятся в пространстве имен std.

OP: В Object Pascal каждый файл исходного кода называется unit, и он делится на две части: интерфейс и реализация, отмечаемые соответственно ключевыми словами interface и implementation. Секция интерфейса включает в себя определения классов (с объявлениями методов), а секция реализации должна включать в себя определения методов, объявленных в интерфейсе. Писать фактический код в секции интерфейса нельзя. Вы можете сослаться на объявления другого файла, используя предложение uses. Этим включается в компиляцию интерфейс того файла:

```
uses
Windows, Form, MyFile;
```

Java: В Java каждый файл исходного кода или единица компиляции компилируется отдельно. Затем вы можете отметить группу единиц компиляции как части одного пакета. В отличие от двух других языков, вы пишете весь код методов тут же при объявлении класса. При включении какого-либо файла предложением import, компилятор читает только public объявления, а не весь код:

```
import where.Myclass;
import where.* // все классы
```

C#: На платформе CLR (.NET) код разделен на сборки, которые могут представлять собой динамически подключаемую библиотеку либо исполняемый файл. Кроме того, для группировки типов с целью сведения к минимуму вероятности конфликта имен, существуют пространства имен, которые, однако, никак не связаны со сборками или файлами исходного кода.

Примечание: Модули как пространство имён. Другим важным отличием Java и OP является их способность читать откомпилированные файлы и извлекать из них определения, как бы извлекая заголовки из скомпилированного кода. С другой стороны, для преодоления отсутствия модулей C++ включает пространство имен (namespace). В Java и OP, когда два имени конфликтуют, вы можете просто использовать имя модуля в качестве префикса. Это не требует дополнительной работы по определению пространств имен, а просто включено в языки.

Методы/данные класса и объекта класса

Свойство: ОО языки обычно разрешают заводить методы и данные, относящиеся к классу целиком, а не к отдельным объектам. Метод класса обычно может быть вызван как для объекта класса, так и применён к классу в целом. Данные класса не повторяются для каждого объекта, а разделяются между всеми объектами данного типа.

C++: В C++ методы и данные класса отмечаются ключевым словом static. Данные класса должны быть проинициализированы специальным объявлением, ещё одной уступкой отсутствию модулей.

OP: В OP допустимы только методы класса, которые отмечаются словом class. Данные класса можно заменить, так сказать, приватными глобальными переменными в секции исполнения юнита, описывающего класс.

Java: Java использует то же слово, что и C++, static. Статические методы используются очень часто (и даже слишком) изза отсутствия глобальных функций. Статические данные можно инициализировать прямо в объявлении класса.

C#: Также, как и в C++ и Java, используется ключевое слово static. Кроме того, можно отметить как статический целый класс, что запрещает создание его объектов и определение нестатических членов.

Классы и наследование

Свойство: Наследование у классов – одно из оснований ООП. Оно может быть использовано для выражения генерализации или специализации. Основная идея в том, что вы определяете новый тип, расширяя или модифицируя существующий, другими словами, производный класс обладает всеми данными и методами базового класса, новыми данными и методами и, возможно, модифицирует некоторые из существующих методов. Различные ОО языки используют различные жаргоны для описания этого механизма (derivation, inheritance, subclassing), для класса, от которого вы наследуете (базовый класс, родительский класс, суперкласс) и для нового класса (производный класс, дочерний класс, подкласс).

C++: C++ использует слова public, protected, и private для определения типа наследования и чтобы спрятать наследуемые методы или данные, делая их приватными или защищёнными. Хотя публичное наследование наиболее часто используется, по умолчанию берётся приватное. Как мы увидим далее, C++ – единственный из этих четырех языков, поддерживающий множественное наследование. Вот пример синтаксиса наследования:

```
class Dog: public Animal {
 ...
};
```

OP: Object Pascal при наследовании использует не ключевые слова, а специальный синтаксис, добавляя в скобках имя базового класса. Этот язык поддерживает только один тип наследования, который в C++ называется публичным. Как мы увидим позднее, классы OP происходят от одного общего базового класса.

```
type
Dog = class (Animal)
 ...
end;
```

Java: Java использует слово extends для выражения единственного типа наследования, соответствующего публичному наследованию в C++. Java не поддерживает множественное наследование. Классы Java тоже происходят от общего базового класса.

```
class Dog extends Animal {
 ...
}
```

C#: C#, как и Java, не поддерживает множественное наследование. Синтаксис наследования несколько отличается от остальных рассматриваемых языков:

```
class Dog : Animal
{
...
}
```

Примечание: Конструкторы и инициализация базового класса. В C++, C# и Java у конструкторов наследующих классов сложная структура. В Object Pascal за инициализацию базового объекта отвечает программист. Это довольно сложный раздел, поэтому я пропустил его в этой статье. Вместо этого я сосредоточусь на общем базовом классе, множественном наследовании, интерфейсах, позднем связывании и других родственных предметах.

Предок всех классов

Свойство: В некоторых OO языках каждый класс происходит по крайней мере от некоторого базового класса по умолчанию. Этот класс, часто называемый Object, или подобно этому, обладает некоторыми основными способностями, доступными всем классам. Фактически, все другие классы в обязательном порядке его наследуют. Этот подход является общим ещё и потому, что так первоначально делалось в Smalltalk.

C++: Хотя язык C++ и не поддерживает такое свойство, многие структуры приложений базируются на нём, вводя идею общего базового класса. Пример тому – MFC с его классом CObject. Фактически это имело большой смысл вначале, когда языку не хватало шаблонов.

OP: Каждый класс автоматически наследует класс TObject. Так как язык не поддерживает множественное наследование, все классы формируют гигантское иерархическое дерево. Класс TObject поддерживает RTTI и обладает некоторыми другими возможностями. Общей практикой является использование этого класса, когда вам нужно передать объект неизвестного типа.

Java: Как и в OP, все классы безоговорочно наследуют класс Object. И в этом языке у общего класса тоже есть некоторые ограниченные свойства и небольшая поддержка RTTI.

C# (CLR): Также имеется класс System.Object (в C# также существует ключевое слово *object* для его обозначения), являющийся предком не только всех классов, но и любых других типов, даже примитивных, которые также включены в структуру наследования и имеют методы.

Доступ к методам базового класса

Свойство: Когда вы пишете метод класса или перекрываете метод базового класса, вам нередко надо сослаться на методы базового класса. Если этот метод

переопределен в производном классе, то, используя его имя, вы получите новую версию. В ОО языках есть некоторые приёмы или ключевые слова, позволяющие решить эту проблему.

C++: В C++ для указания нужного класса можно использовать оператор (::). Вы можете получить доступ не только к методам базового класса, но к классам выше по иерархии. Это очень мощная техника, но она создаёт проблемы, когда вы добавляете в иерархию промежуточный класс.

OP: В Object Pascal для этой цели есть специальное слово inherited. После этого слова вы можете написать имя метода базового класса или (в некоторых случаях) просто использовать это ключевое слово для доступа к соответствующему методу базового класса.

Java: Java для этого использует ключевое слово super. В этом языке, так же, как и в OP, нет возможности сослаться на другой предшествующий класс. На первый взгляд, это может показаться ограничением, но оно позволяет расширять иерархию, вводя промежуточные классы. К тому же, если вы не нуждаетесь в функциях базового класса, вам, наверное, не следует его наследовать.

Совместимость подтипов

Свойство: Как я указывал в начале, не все ОО языки строго типизированы, но все три языка, на которых мы сконцентрировались, обладают этим свойством. В основном это означает, что объекты различных классов несовместимы по типу. Из этого правила есть исключение: объекты производных классов совместимы с типом их базового класса. (Примечание: обратное обычно неверно).

C++: В C++ правило совместимости подтипов справедливо только для указателей и ссылок, но не для обычных объектов. Фактически, у различных объектов разный размер, поэтому их нельзя расположить на том же месте в памяти.

OP: Благодаря ссылочно-объектной модели, совместимость подтипов возможна для каждого объекта. Более того, все объекты совместимы по типу с TObject.

Java: Java использует ту же модель, что и Object Pascal.

Примечание: Полиморфизм. Совместимость подтипов особенно важна для позднего связывания и полиморфизма, как это показано в следующей секции.

Позднее связывание (и полиморфизм)

Свойство: Когда различные классы в иерархии переопределяют некоторый метод, очень полезна возможность ссылаться на общий объект этих классов (благодаря совместимости подклассов) и вызывать этот метод, результатом чего будет вызов метода надлежащего класса. Для этого компилятор должен поддерживать позднее связывание, то есть не генерировать вызов специфической функции, а ждать, пока во время выполнения не определятся фактический тип объекта и функция, которую нужно вызвать.

C++: В C++ позднее связывание доступно только для виртуальных методов (вызов которых становится немного медленнее). Метод, объявленный в базовом классе как виртуальный (virtual), поддерживает это свойство (но только если описания методов совпадают). Обычные, не виртуальные методы не позволяют позднее связывание, как и OP.

OP: В Object Pascal позднее связывание вводится с помощью ключевых слов virtual и dynamic (разница между ними только в оптимизации). В производных классах переопределённые методы должны быть отмечены словом override (это заставляет компилятор проверять описание метода). Рациональное объяснение этой особенности ОР состоит в том, что разрешается больше изменений в базовом классе и предоставляет некоторый дополнительный контроль во время компиляции.

Java: В Java все методы используют позднее связывание, если вы не отметите их явно как final. Финальные методы не могут быть переопределены и вызываются быстрее. В Java написание методов с нужной сигнатурой жизненно важно для обеспечения полиморфизма. Тот факт, что в Java по умолчанию используется позднее связывание, тогда как в C++ стандартом является раннее связывание, – явный признак разного подхода этих двух языков: C++ временами жертвует ОО моделью в пользу эффективности, тогда как Java – наоборот

Примечание: Позднее связывание для конструкторов и деструкторов. Object Pascal, в отличие от других двух языков, позволяет определять виртуальные конструкторы. Все три языка поддерживают виртуальные деструкторы.

Абстрактные методы и классы

Свойство: При построении сложной иерархии, для обеспечения полиморфизма программисты часто вынуждены вводить методы в классы верхнего уровня, даже если эти методы ещё не определены для этой специфической абстракции. Здесь можно было бы оставить пустые методы, но многие ОО языки предлагают такой специфический механизм, как определение абстрактных методов, то есть методов без реализации. Классы, имеющие хотя бы один абстрактный метод, часто называются абстрактными классами.

C++: В C++ абстрактные методы или чисто виртуальные функции получаются добавлением так называемого чистого описателя (=0) в определение метода. Абстрактные классы являются просто классами с одним или более абстрактным методом (или наследующие их). Вы не можете создать объект абстрактного класса.

OP: Object Pascal для выделения этих методов использует ключевое слово abstract. Кроме того, абстрактными классами являются классы, имеющие или наследующие абстрактные методы. Вы можете создать объект абстрактного класса (хотя компилятор выдаст предупреждающее сообщение). Это подвергает программу риску вызвать абстрактный метод, что приведёт к генерации ошибки времени выполнения и завершению программы.

Java: В Java и абстрактные методы, и абстрактные классы отмечаются ключевым словом abstract (действительно, в Java обязательно определять как абстрактный, класс, имеющий абстрактные методы, – хотя это кажется некоторым излишеством). Производные классы, которые не переопределяют все абстрактные методы, должны быть отмечены как абстрактные. Как и в C++, нельзя создавать объекты абстрактных классов.

Множественное наследование и интерфейсы

Свойство: Некоторые ОО языки допускают наследование более чем одному базовому классу. Другие языки позволяют вам наследовать только от одного

класса, но дополнительно позволят вам наследовать также от многочисленных интерфейсов или чисто абстрактных классов, то есть классов, состоящих только из виртуальных функций.

C++: C++ – единственный из трех языков, поддерживающий множественное наследование. Некоторые программисты считают положительным фактом, другие – отрицательным, и я не буду вмешиваться сейчас в эту дискуссию. Определенно, что множественное и повторяющееся наследование влечет за собой такие понятия, как виртуальные базовые классы, которые не легко освоить, хотя они предоставляют мощную технику. C++ не поддерживает понятия интерфейсов, хотя их можно заменить множественным наследованием чисто абстрактным классам (интерфейсы можно рассматривать как подмножество множественного наследования).

Java: Java, как и Object Pascal, не поддерживает множественное наследование, но полностью поддерживает интерфейсы. Методы интерфейсов допускают полиморфизм, и Вы можете использовать объект, осуществляющий интерфейс, когда ожидается интерфейсный объект. Класс может наследовать или расширить всего лишь один базовый класс, но может осуществить (это – ключевое слово) многочисленные интерфейсы. Хотя это не было спланировано заранее, интерфейсы Java очень хорошо укладываются в модель COM. Вот пример интерфейса:

```
public interface CanFly {
 public void Fly();
}
public class Bat extends Animal implements CanFly {
 public void fly() { /* the bat flies... */ }
}
```

OP: Delphi 3 ввел в Object Pascal понятие, подобное интерфейсам Java, но эти интерфейсы строго соответствуют COM (хотя технически возможно использовать их в обычных неCOM программах). Интерфейсы формируют иерархию, отдельную от классов, но, как и в Java, класс может наследовать одному базовому классу и осуществлять различные интерфейсы. Отображение методов класса на методы интерфейсов, осуществляемых классом, является одной из наиболее сложных частей языка Object Pascal. Delphi 4 добавляет к этой структуре возможность передать реализацию интерфейса подобъекту, делая эту технику почти такой же эффективной, как и множественное наследование.

Другие свойства

Помимо обеспечения объектно-ориентированного программирования, эти языки предлагают другие интересные и мощные характеристики, которые дополняют поддержку ООП. В следующих разделах я просто представлю некоторые из них.

RTTI

Свойство: В строго типизованных ОО языках компилятор осуществляет весь контроль типов, так что нет особой необходимости хранить информацию о классах и типах в работающей программе. Тем не менее, есть случаи (как, например, динамическое преобразование типов), которые требуют информацию о типе.

По этой причине все три ОО языка, рассматриваемые здесь, более или менее поддерживают Идентификацию/Информацию о Типе Времени Выполнения (RTTI).

C++: первоначально не поддерживал RTTI. Это было добавлено позже для динамического преобразования типа (dynamic_cast) и сделало доступной некоторую информацию о типе для классов. Вы можете запросить идентификацию типа для объекта, и проверить, принадлежат ли два объекта одному классу.

OP: поддерживает и требует много RTTI. Доступен не только контроль соответствия и динамическое преобразование типов (с помощью операторов *is* и *as*). Классы генерируют расширенную RTTI для своих *published*свойств, методов и полей. Фактически это ключевое слово управляет частью генерации RTTI. Вся идея свойств, механизм потоков (файлы форм – DFM), и среда Delphi, начиная с Инспектора Объектов, сильно опирается на RTTI классов. У класса *TObject* есть (кроме прочих) методы *ClassName* и *ClassType*. *ClassType* возвращает переменную типа класса, объект специального типа ссылки на класс (который не является самим классом).

Java: как и в Object Pascal, в Java тоже есть единый базовый класс, помогающий следить за информацией о классе. Безопасное преобразование типов (typesafe downcast) встроено в этот язык. Метод *getClass()* возвращает своего рода метакласс (объект класса, описывающего классы), и Вы можете применить функцию *getName()* для того, чтобы получить строку с именем класса. Вы можете также использовать оператор *instanceof*. Java включает в себя расширенную RTTI для классов или интроспекцию, которая была введена для поддержки компонентной модели JavaBeans. В Java существует возможность создавать классы во время исполнения программы.

Пример: Вот синтаксис безопасного преобразования типов на всех трех языках. В случае ошибки в Delphi и Java происходит исключение, а в C++ возвращается нулевой указатель:

```
// C++
 Dog* MyDog = dynamic_cast <Dog*> (myAnimal);
// Java
 Dog MyDog = (Dog) myAnimal;
// Object Pascal
 myDog := myAnimal as Dog;
```

Обработка исключений

Свойство: Основная идея обработки исключений – упростить код обработки ошибок в программе, предоставив стандартный встроенный механизм, с целью сделать программы более устойчивыми. Обработка исключений – это тема, требующая отдельного рассмотрения, поэтому я только очерчу некоторые ключевые элементы и различия.

C++: C++ использует ключевое слово throw для генерации исключения, try для отметки охраняемого блока и catch для записи кода обработки исключения. Исключения – объекты специального класса, которые могут образовывать некоторую иерархию во всех трёх языках. При возникновении исключения C++ вы-

полняет очистку стека до точки перехвата исключения. Перед удалением каждого объекта в стеке вызывается соответствующий деструктор.

OP: Object Pascal использует подобные ключевые слова: raise, try, и except и обладает подобными свойствами. Единственное существенное отличие состоит в том, что опустошение стека не производится, просто потому, что в стеке нет объектов. Кроме того, вы можете добавить в конце блока try слово finally, отмечая блок, который должен выполняться всегда, независимо от того, было или нет вызвано исключение. В Delphi классы исключений – производные Exception.

Java: Использует ключевые слова C++, но ведёт себя как Object Pascal, включая дополнительное ключевое слово finally. (Это общее свойство всех языков со ссылочно-объектной моделью, оно включено Borland также и в C++Builder 3.) Присутствие алгоритма сборки мусора ограничивает использование finally в классе, который распределяет другие ресурсы, кроме памяти. Также Java строже требует, чтобы все функции, которые могут вызвать исключение, описывали в соответствующем блоке, какие исключения могут быть вызваны функцией. Эти описания исключений проверяются компилятором, что является хорошим свойством, даже если оно подразумевает некоторую дополнительную работу для программиста. В классах Java объекты исключения должны наследовать класс Throwable.

Шаблоны (обобщенное программирование)

Свойство: Обобщенное программирование – это техника написания функций и классов, оставляя некоторые типы данных неопределёнными. Спецификация типа осуществляется, когда эта функция или класс используется в исходном коде. Всё делается под строгим контролем компилятора, и ничего не остаётся без определения во время выполнения. Наиболее типичный пример шаблона класса – это контейнерные классы.

C++: есть шаблонные классы и функции, отмечаемые ключевым словом *template*. Стандартный C++ включает обширную библиотеку шаблонов, называемую STL (Standart Template Library ,Стандартная библиотека шаблонов), которая поддерживает специфический и мощный стиль программирования: обобщенное программирование. C++ – единственный из рассматриваемых трех языков, который основывается на поддержке обобщенного программирования, помимо ООП.

OP: нет шаблонов. Контейнерные классы обычно строятся как контейнеры объектов класса *TObject*, а затем уточняются для необходимых объектов.

Java: реализуются в рамках Generics (введенного в JDK 1.5 «Tiger»). Концептуально они не отличаются от шаблонов в C++, но имеют некоторые особенности, которые диктуются свойствами самого языка. В отличие от C++, в Java невозможно во время выполнения получить информацию о конкретном типе шаблона. Предусмотрены контейнеры на все случаи жизни: List (хранение последовательностей элементов), Map или ассоциативные массивы (связывание одних объектов с другими), Set (уникальность значений для каждого типа).

Другие специфические свойства

Свойство: Есть еще другие свойства, не упомянутые мной, хотя они важны, только из-за того, что они специфичны только для одного из трёх языков.

C++: Я уже упомянул множественное наследование, виртуальные базовые классы и шаблоны. Эти свойства отсутствуют в двух других ОО языках. В C++ есть ещё перегрузка операторов, тогда как перегрузка методов присутствует также в Java и была недавно добавлена в Object Pascal. C++ позволяет программистам перегружать и глобальные функции. Вы можете перегрузить операторы преобразования типов, написав конвертирующие методы, которые будут вызываться «за кулисами». Объектная модель C++ требует копировать конструкторы и перегружать операторы присваивания, в чем не нуждаются остальные два языка, поскольку базируются на ссылочнообъектной модели.

Java: Только Java поддерживает многопоточность непосредственно в языке. Объекты и методы поддерживают механизм синхронизации (с ключевым словом synchronized): два синхронизированных метода одного класса не могут выполняться одновременно. Для создания нового потока вы просто наследуете от класса Thread, перегружая метод run(). Как альтернативу вы можете осуществить интерфейс Runnable (что вы обычно делаете в апплетах, поддерживающих многопоточность). Мы уже обсуждали сборщик мусора. Ещё одно ключевое свойство Java, конечно, идея переносимого байтового кода, но это не относится непосредственно к языку. Другое примечательное свойство – это поддержка основанных на языке компонентов, известных как JavaBeans и многие другие свойства, недавно добавленные в этот язык.

OP: Вот некоторые специфические черты Object Pascal: ссылки на классы, легкие для использования указатели на методы (основа модели обработки событий) и, в частности, свойства (property). Свойство – это просто имя, скрывающее путь, которым вы получаете доступ к данным или методу. Свойство может проецироваться на прямое чтение или запись данных, а может ссылаться на метод, обеспечивающий доступ. Даже если вы меняете способ доступа к данным, вам не нужно менять вызывающий код (хотя вам нужно будет его перекомпилировать): это делает свойства очень мощным средством инкапсуляции.

Стандарты

Свойство: Для каждого языка требуется, чтобы кто-то установил его стандарт и проверял все реализации на соответствие ему.

C++: Стандарт ANSI/ISO C++ явился завершением многотрудных усилий соответствующего комитета. Большинство авторов компиляторов, кажется, пытаются подчиняться стандарту, хотя есть ещё много странностей. Теоретически развитие языка должно на этом закончиться. На практике, инициативы вроде компилятора Borland C++Builder, конечно, не способствуют улучшению ситуации, но многие чувствуют, что C++ очень нуждается в визуальном окружении программирования. В то же время, популярный Visual C++ тянет C++ в другом направлении, например, с явным злоупотреблением макросов. (По моему личному мнению, у каждого языка есть собственная модель развития, и поэтому нет большого смысла в попытках использовать язык для того, для чего он не был предназначен.) Много новых возможностей будут введены новым стандартом C++ 0x.

OP: Object Pascal – язык-собственность, поэтому у него нет стандарта. Borland лицензировал язык для пары продавцов небольших компиляторов на

OS/2, но это не оказало большого влияния. Borland расширяет язык с каждым новым выпуском Delphi.

Java: Компания-создатель Sun обладает торговой маркой Java. Однако Sun лицензирует его для продавцов других компиляторов, и убедило ISO создать стандарт Java, не создавая специальный комитет, а просто приняв предложения Sun как есть. Кроме формального стандарта, однако, Java требует высокосовместимых JVM. С недавней поры Sun выдвинула инициативу открыть исходные коды Java (OpenJDK) и сделать ее доступной для всех разработчиков в рамках лицензии GPL 2.

4. Сложность вычислений на примере алгоритмов сортировки

- **Сортировка выбором (Selection sort)**
- **Пузырьковая сортировка (Bubble sort)**
- **Сортировка вставками (Insertion sort)**
- **Сортировка слиянием (Merge sort)**
- **Быстрая сортировка (Quick sort)**

Было подсчитано, что до четверти времени централизованных компьютеров уделяется сортировке данных. Это потому, что намного легче найти значение в массиве, который был заранее отсортирован. В противном случае поиск немного походит на поиск иголки в стоге сена.

Есть программисты, которые всё рабочее время проводят в изучении и внедрении алгоритмов сортировки. Это потому, что подавляющее большинство программ в бизнесе включает в себя управление базами данных. Люди ищут информацию в базах данных всё время. Это означает, что поисковые алгоритмы очень востребованы.

Но есть одно «но». Поисковые алгоритмы работают намного быстрее с базами данных, которые уже отсортированы. В этом случае требуется только линейный поиск.

В то время как компьютеры находятся без пользователей в некоторые моменты времени, алгоритмы сортировки продолжают работать с базами данных. Снова приходят пользователи, осуществляющие поиск, а база данных уже отсортирована, исходя из той или иной цели поиска.

В этой статье приведены примеры реализации стандартных алгоритмов сортировки.

Сортировка выбором (Selection sort)

Для того чтобы отсортировать массив в порядке возрастания, следует на каждой итерации найти элемент с наибольшим значением. С ним нужно поменять местами последний элемент. Следующий элемент с наибольшим значением становится уже на предпоследнее место. Так должно происходить, пока элементы, находящиеся на первых местах в массиве, не окажутся в надлежащем порядке.

Код C++

```
void SortAlgo::selectionSort(int data[], int lenD)
```

```
{
 int j = 0;
 int tmp = 0;
 for(int i=0; i<lenD; i++){
  j = i;
  for(int k = i; k<lenD; k++){
   if(data[j]>data[k]){
    j = k;
   }
  }
  tmp = data[i];
  data[i] = data[j];
  data[j] = tmp;
 }
}
```

Пузырьковая сортировка (Bubble sort)

При пузырьковой сортировке сравниваются соседние элементы и меняются местами, если следующий элемент меньше предыдущего. Требуется несколько проходов по данным. Во время первого прохода сравниваются первые два элемента в массиве. Если они не в порядке, они меняются местами и затем сравнивается элементы в следующей паре. При том же условии они так же меняются местами. Таким образом сортировка происходит в каждом цикле пока не будет достигнут конец массива.

Код C++

```
void SortAlgo::bubbleSort(int data[], int lenD)
{
 int tmp = 0;
 for(int i = 0;i<lenD;i++){
  for(int j = (lenD-1);j>=(i+1);j--){
   if(data[j]<data[j-1]){
    tmp = data[j];
    data[j]=data[j-1];
    data[j-1]=tmp;
   }
  }
 }
}
```

Сортировка вставками (Insertion sort)

При сортировке вставками массив разбивается на две области: упорядоченную и и неупорядоченную. Изначально весь массив является неупорядоченной областью. При первом проходе первый элемент из неупорядоченной области изымается и помещается в правильном положении в упорядоченной области.

На каждом проходе размер упорядоченной области возрастает на 1, а размер неупорядоченной области сокращается на 1.

Основной цикл работает в интервале от 1 до N-1. На j-й итерации элемент [i] вставлен в правильное положение в упорядоченной области. Это сделано путем сдвига всех элементов упорядоченной области, которые больше, чем [i], на одну позицию вправо. [i] вставляется в интервал между теми элементами, которые меньше [i], и теми, которые больше [i].

Код C++

```
void SortAlgo::insertionSort(int data[], int lenD)
{
  int key = 0;
  int i = 0;
  for(int j = 1;j<lenD;j++){
    key = data[j];
    i = j-1;
    while(i>=0 && data[i]>key){
      data[i+1] = data[i];
      i = i-1;
      data[i+1]=key;
    }
  }
}
```

Сортировка слиянием (Merge sort)

При рекурсивной сортировке слиянием массив сначала разбивается на мелкие кусочки – на первом этапе – на состоящие из одного элемента. Затем эти кусочки объединяются в более крупные кусочки – по два элемента и элементы при этом сравниваются, а в результате в новом кусочке меньший элемент занимает место слева, а больший – справа. Далее происходит слияние в ещё более крупные кусочки и так до конца алгоритма, когда все кусочки будут объединены в один, уже отсортированный массив. Если есть интерес, есть статья о рекурсивных функциях.

Код C++

```
void SortAlgo::mergeSort(int data[], int lenD)
{
  if(lenD>1){
    int middle = lenD/2;
    int rem = lenD-middle;
    int* L = new int[middle];
    int* R = new int[rem];
    for(int i=0;i<lenD;i++){
      if(i<middle){
```

```
      L[i] = data[i];
     }
     else{
      R[i-middle] = data[i];
     }
    }
    mergeSort(L,middle);
    mergeSort(R,rem);
    merge(data, lenD, L, middle, R, rem);
  }
}

void SortAlgo::merge(int merged[], int lenD, int L[], int lenL, int R[], int lenR){
  int i = 0;
  int j = 0;
  while(i<lenL||j<lenR){
    if (i<lenL & j<lenR){
      if(L[i]<=R[j]){
        merged[i+j] = L[i];
        i++;
      }
      else{
        merged[i+j] = R[j];
        j++;
      }
    }
    else if(i<lenL){
      merged[i+j] = L[i];
      i++;
    }
    else if(j<lenR){
      merged[i+j] = R[j];
      j++;
    }
  }
}
```

Быстрая сортировка (Quick sort)

Быстрая сортировка использует алгоритм «разделяй и властвуй». Она начинается с разбиения исходного массива на две области. Эти части находятся слева и справа от отмеченного элемента, называемого опорным. В конце процесса одна часть будет содержать элементы меньшие, чем опорный, а другая часть будет содержать элементы больше опорного.

Код C++

```
void SortAlgo::quickSort(int* data, int const len)
{
  int const lenD = len;
  int pivot = 0;
  int ind = lenD/2;
  int i,j = 0,k = 0;
  if(lenD>1){
    int* L = new int[lenD];
    int* R = new int[lenD];
    pivot = data[ind];
    for(i=0;i<lenD;i++){
      if(i!=ind){
        if(data[i]<pivot){
          L[j] = data[i];
          j++;
        }
        else{
          R[k] = data[i];
          k++;
        }
      }
    }
    quickSort(L,j);
    quickSort(R,k);
    for(int cnt=0;cnt<lenD;cnt++){
      if(cnt<j){
        data[cnt] = L[cnt];;
      }
      else if(cnt==j){
        data[cnt] = pivot;
      }
      else{
        data[cnt] = R[cnt-(j+1)];
      }
    }
  }
}
```

5. Динамическое программирование и жадные алгоритмы

Итак, что же представляют собой жадные алгоритмы? Перед тем, как ответить на этот вопрос, ответим на другой: а что же есть *алгоритм?* Кажется, интуи-

тивно мы это понимаем, однако интуиция имеет свойство подводить в самый не подходящий момент, причем очень незаметно.

Просто *алгоритмом* будем называть механизм, который должен гарантировать не только то, что решение будет найдено, но и то, что будет найдено именно оптимальное решение. Кроме того, алгоритм должен обладать следующими качествами:

- Ограниченность по времени (работа алгоритма должна прекратиться за разумное время);
- Правильность (алгоритм должен находить правильное решение);
- Детерминистичность (сколько бы раз не исполнялся алгоритм с одинаковыми входными данными, результат должен быть один и тот же);
- Конечность (описание работы алгоритма должно содержать конечное число шагов);
- Однозначность (каждый шаг алгоритма должен интерпретироваться однозначно).

Эвристика – это прямая противоположность алгоритму, так как она не дает ни каких гарантий ни того, что решение будет найдено, ни того, что оно будет оптимальным. Между алгоритмом и эвристикой лежат два понятия, которые называют *приблизительный алгоритм* (решение гарантировано, но его оптимальность – нет) и *вероятностный алгоритм* (решение не гарантировано, но если оно будет найдено, то будет обязательно оптимальным).

А теперь можно перейти к рассмотрению жадных алгоритмов. Этот класс алгоритмов намного проще и быстрее, чем динамическое программирование. Жадный алгоритм делает на каждом шаге локально оптимальный вы бор, надеясь, что итоговое решение окажется оптимальным. Однако это не всегда так, и потому иногда бывает, что жадный алгоритм в терминах выше упомяну того определения – не полноценный алгоритм, а приблизительный. Но для большого числа алгоритмических задач жадные алгоритмы действительно дают оптимум. Общую схему работы жадных алгоритмов дадим в конце, после рассмотрения конкретных примеров.

Задача о выборе заявок

Пусть подано N заявок на проведение занятий в одной и той же аудитории. Для каждого занятия известно время его проведения $[b_i,e_i]$, где b_i – время начала занятия, e_i – время окончания. Два разных занятия не могут перекрываться по времени, но условимся, что начало одного может совпадать с окончанием другого. Задача состоит в том, чтобы выбрать максимальное число совместимых по времени занятий.

Жадный алгоритм поступает так: упорядочим заявки в порядке возрастания времени окончания: $e_1 \le e_1 \le \ldots e_n$. На сортировку понадобится времени $\underline{O}(N \cdot \log(N))$ в худшем случае. Так как в этой статье проблема сортировки не рассматривается, положим, что массив уже отсортирован по возрастанию. Далее все предельно просто:

var

```
b,e: array[1..100] of integer; {входные данные} A: set of byte; {искомое множество}
i,j,n: integer;
begin
read(n);
for i:=1 to n do read(b[i],e[i]);
A := [1]; {начальное значение} j := 1; {самый правый отрезок множества A – первый}
for i:=2 to n do
if b[i]>=e[j] then begin
(если iй отрезок лежит правее jго) A:=A+[i];
(то включаем jй )
j := i; (и запоминаем его как самый правый в множестве A)
end;
for i:=1 to n do (выводим результат)
if (i in A) then writeln(i);
end.
```

Листинг – Задача о выборе заявок, решаемая «жадным» алгоритмом

Вначале A содержит заявку j=1. Далее в цикле по i ищется заявка, начинающаяся не раньше, чем оканчивается заявка j. Если таковая найдена, она включается в множество A**,** и перемен ной j присваивается ее номер. Алгоритм требует всего лишь $\underline{O}(n)$ шагов (без учета предварительной сортировки). Жадность этого алгоритма состоит в том, что на каждом шаге он делает выбор так, чтобы остающееся свободным время было максимальным (это и есть локально-оптимальный выбор).

Теперь докажем, что этот алгоритм дает оптимальное решение. Прежде всего, докажем, что существует оптимальное решение за дачи о выборке заявок, содержащее заявку 1 (с самым ранним временем окончания). В самом деле, если в каком-то оптимальном множестве заявок заявка 1 не содержится, то можно заменить в нем заявку с самым ранним временем окончания на заявку 1, что не повредит совместности заявок, (ибо заявка 1 кончается еще раньше, чем прежняя, и ни с чем пере сечься не может) и не изменит их общего количества. Стало быть, можно искать оптимальное множество заявок A среди содержащих заявку 1. После того, как мы договорились рассматривать только наборы, содержащие заявку 1, все несовместные с ней за явки можно выкинуть, и задача сводится к выбору оптимального набора заявок из множества оставшихся заявок (совместных с заявкой 1). Другими словами, мы свели задачу к аналогичной задаче с меньшим числом заявок. Рассуждая по индукции, получаем, что, делая на каждом шаге жадный выбор, мы придем к оптимальному решению.

Вспомним динамическое программирование и попробуем решить ту же задачу с помощью него. Заведем динамическую таблицу $m[1..n]$, где $m[i]$ означает максимальное число совместных заявок среди набора с номерами $1..i$. Допустим, что подзадачи $m[1]$, $m[2]$,...,$m[k-1]$ уже решены. Установим рекуррентную зависимость для

решения задачи $m[k]$: *k-й* отрезок можно брать, а можно и не брать в искомое множество. Если мы его не берем, то $m[k] = m[k–1]$, а если мы его берем, то

$m[k] = 1 + m[$ max(*i* такое, что $e[i] \le s[k]$)]

Последнее выражение означает, что мы отбросили все заявки, не совместные с *k*-й (левее нее), взяли оптимум для оставшегося множества из динамической таблицы плюс заявку *k*. Таким образом, рекуррентная зависимость такова

$m[k] = \max\{ m[k–1], 1 + m[\max(i$ та кое, что $e[i] \le b[k])$])

Для того, чтобы найти оптимальное множество (а не только количество $m[n]$ элементов в нем), надо завести дополнительный массив $prev[1..n]$, где $prev[k]$ будет означать предыдущий элемент, (если мы *k*-й брали), или –1 (если не брали *k*-й). Покажем, как выглядит реализация ДП-алгоритма, а потом сделаем его оценку.

```
var
b,e,m,prev: array[0..100] of integer;
i,j,k,n: integer;
b egin
read(n); for i:=1 to n do read(b[i],e[i]);
fillchar(prev, sizeof(prev), $FF);(заполняем 1)
m [0] := 0;
for k:=2 to n do begin
i:= k1; (ищем i такое, что e[i]=b[k])
while (i>0) and (e[i] >b[k] ) do dec(i);
if m[k1] >= 1 + m[i] then
(если элемент лучше не брать)
m[k] := m[k1] (то не берем его)
else begin
m[k]:= 1 + m[i];
(иначе берем и)
prev[k] := i; (запоминаем, что перед ним идет элемент i)
end;
end;
i:=n; (пробежимся с конца до начала и выведем все элементы)
repeat
if prev[i] =1 then dec(i)
(если iЙ не брали, движемся дальше)
else begin (в противном случае)
writeln(i); (выводим этот и)
i:= prev[i];
(перепрыгиваем через несовместимые слева от него)
end;
until i=0;
end.
```

Листинг – Задача о выборе заявок, решаемая ДП-алгоритмом

Даже не делая оценок сложности, легко видеть, что ДП-алгоритм сложнее жадного. Ну, а если быть более точным, то его сложность равняется $\underline{O}(n^2)$, так как существует два вложенных цикла (*for* по *k* и внутренний *while*, который делает в среднем порядка $\underline{O}(n)$ сравнений и уменьшений *i*). К тому же он требуется порядка $\underline{O}(n)$ дополнительной памяти. Таким образом, в этом случае жадный алгоритм намного эффективнее динамического программирования.

Дискретная задача о рюкзаке

На складе есть *N* предметов, для которых известны их веса *w*[1..*N*] и их стоимости *v*[1..*M*]. На склад пробрался вор, который хочет украсть предметов на максимальную сумму денег. Однако вес, который вор может вынести, ограничен и равняется *TotalW*. Какие предметы должен взять вор, чтобы их суммарная стоимость была наибольшей, а вес был ограничен величиной *TotalW* ?

Эта задача встречается во многих источниках, но часто ее решают простым backtracking'ом, где на каждом шаге пробуют взять или не взять предмет. Однако такой подход не всегда эффективен. Существует как жадный, так и ДП-алгоритм ее решения.

Жадный алгоритм поступает так: вычисляется цена единицы веса каждого предмета, то есть *price*[*i*] := *v*[*i*] / *w*[*i*]. Потом предметы сортируются в порядке убывания *price*[*i*], и вор начинает помещать в свой рюкзак предметы по порядку (*i* = 1,2,...*N*) из отсортированного списка. Если предмет *i* не помещается (по ограничению оставшегося свободного веса в рюкзаке) – вор рассматривает следующий предмет 1+1 (*price*[*i* + 1] < *price*[*i*]), и так до конца. Жадность состоит в том, что вор на каждом шаге пытается взять предмет с наибольшей ценой.

Оценим сложность описанного алгоритма. Для сортировки надо $\underline{O}(N \cdot log(N))$ операций. Далее надо пробежать циклом по *i* от 1 до *N* и пробовать впихнуть предмет *i*, таким образом, это займет еще $\underline{O}(n)$ операций. Итого, общая сложность есть $\underline{O}(N \cdot \log(N) + N)$, что в общем случае эквивалентно $\underline{O}(N \cdot \log(N))$.

Но если хорошо присмотреться, то такой алгоритм не всегда дает оптимум. Вот вам пример: 3 предмета, 1 – ($60, 10 кг), 2 – ($100, 20 кг), 3 – ($120, 30 кг) с ценами соответственно 6 $/кг, 5 $/кг, 4 $/кг. Предметы уже отсортированы по убыванию цены. Допустим, максимальный вес рюкзака вора – 50 кг. Следуя жадному алгоритму, вор берет первый предмет с ценой 6 $/кг, который весит 10 кг, затем следующий предмет с ценой 5 $/кг (и весом 20 кг), после чего в его рюкзаке остается место только на 20 кг, и оставшийся предмет уже не влезает, так как весит 30 кг. Итого, действуя по жадному алгоритму, вор украл два предмета на сумму $160 и весом 30 кг. Если бы вор украл второй и третий предметы (суммарным весом 50 кг), он бы вынес $220. Как видите, жадный алгоритм не дает оптимального решения, а значит, он является приблизительным алгоритмом.

Если решать эту же задачу с помощью динамического программирования, то надо поступить следующим образом. Допустим, надо найти максимальную сумму *val*[*W*, *i*], которую может вынести вор, если допустить, что объем его рюкзака не больше *W* и можно брать предметы от 1 до *i* (предметы не отсортирова-

ны!) Допустим, мы уже нашли все *val*[1..*W*, 1..*i*–1] (для веса не больше *W* и с возможностью брать предметы от 1 до *i*–1). Рассматривается предмет *i*. Если его вес *w*[*i*] меньше *W*, рассмотрим, стоит ли его брать:

- Если его взять, то доступный вес в рюкзаке станет *W*–*w*[*i*] и мы сможем забрать предметов на сумму *val*[*W*,*i*] = *val*[*W*–*w*[*i*], *i*–1] + *v*[*i*] (задача *val*[*W*–*w*[*i*], *i*–1] уже решена, плюс стоимость *v*[*i*] этого предмета);
- Если его не брать, то доступный вес остается тем же, и тогда *val*[*W*, *i*] = *val*[*W*, *i* – 1].

Из этих двух вариантов выбирается тот, что дает большее значение *val*[*W*, *i*]. Реализация ДП-алгоритма будет выглядеть так:

```
const MAXW = 500; MAXN = 25;
var
val: array[0..MAXW,0..MAXN]of integer; (динамические массивы}
take: array[0.. MAXW,0..MAXN] of boolean;
v,w: array[1..MAXN] of integer;(ценности и веса предметов)
л, TotalW: integer; (кол. предметов, максимальный вес)
weight, i: integer;(переменные циклов}
begin
{читаем входные данные} read(n, TotalW);
for i:=1 to n do read(w(i] v[i]);
(делаем начальную инициализацию для веса 0}
for i:=0 to n do begin
val[0,i]:= 0;
take[0,i]:= false;
end;
for weight:=1 to Totalw do val[weight 0] := 0;
for weight:=1 to TotalW do
for i := 1 to N do
if (w[i]>weight)
{если вещь не влезет} {или лучше ее не брать} or ( val[weight, i1] >= val [weightw[i],il] i v[i))
then begin (то не берем ее)
val[weight,i] :=val[weight,i1]; take[weight,i] := false;
(отмечаем, что вещь i не взята)
end
else {иначе) begin
{оптимум := оптимум для веса weightw[i] и использования
вещей 1..il + цена вещи i, которую мы берем}
val [weight, i]:= val(weightw[i],i1] + v[i];
take[weight,i]:= true;
(отмечаем, что вещь i взята)
end;
(вывод результатов) writeln('Best value:', val[TotalW, N]);
write('Taken things; '); weight := TotalW; for i := N downto 1 do
```

```
if take[weight, i] then begin
write(i,' ');
weight := weight  w[i];
end;
end.
```

Листинг – Дискретная задача о рюкзаке, решаемая ДП-алгоритмом

Этот алгоритм будет выдавать оптимальное решение, но какой ценой? Ценой лишней памяти: надо порядка $O(TotalW \cdot N)$ памяти под динамические таблицы стоимости и запоминания того, брать и каждый предмет или нет. Сложность алгоритма та же: $O(TotalW \cdot N)$, что следует из циклов по *weight* и *i*.

Таким образом, ДП-алгоритм хоть и работает безупречно правильно, но требует дополнительных затрат. Есть, правда, еще одна проблема с ДП-алгоритмом: если *TotalW* слишком велико, или же оно является действительным (а не целым) числом, то ДП-алгоритм вообще не применим.

Непрерывная задача о рюкзаке

Условия те же самые, с той лишь разницей, что предметы можно де лить на части, то есть вор может украсть часть одного предмета. В такой задаче применим жадный алгоритм, описанный выше (с той лишь разницей, что если очередной предмет из отсортированного списка предметов не помещается в рюкзак целиком, то о т него вор берет ту часть, что помещается в рюкзак). Таким образом, если бы мы рассматривал и тот же пример из трех предметов 1 – ($60, 10 кг), 2 – ($100, 20 кг), 3 – ($120, 30 кг), вор бы взял целиком первый и второй из них; осталось свободным 20 кг в рюкзаке; вор отделил бы 2/3 части от третьего предмета (что составит 20 кг) и забрал в сумме $60+$100 + 2/3*$120 = $240. Для непрерывной задачи о рюкзаке жадный алгоритм будет давать оптимальное решение.

Числа Фибоначчи

Вычислить *N* чисел в последовательности Фибоначчи: 1, 1, 2, 3, 5, 8, … , где первые два члена равны единице, а все остальные представляют собой сумму двух предыдущих, *N* меньше 100.

Самый очевидный способ «решения» задачи состоит в написании рекурсивной функции примерно следующего вида:

```
function F(X: integer): longint;
begin
if (X = 1) or (X = 2) then F := 1
else F := F(X1) + F(X2)
end;
```

Листинг – Рекурсивная функция вычисления чисел Фибоначчи

При этом на шестом-седьмом десятке вызовов программе перестанет хватать временных ресурсов самой современной вычислительной машины. Это происходит по следующим причинам.

Для вычисления F(40) вначале вычисляется F(39) и F(38). Причем F(38) вычисляется заново, не используя значение, которое было вычислено, когда считалось F(39).

То есть значение функции при одном и том же значении аргумента считается много раз. Если исключить повторный счет, то функция станет заметно эффективней. Для этого приходится завести массив, в котором хранятся значения нашей функции:

```
var D: array[1..100] of longint;
```

Сперва массив заполняется значениями, которые заведомо не могут быть значениями функции (чаще всего, это «−1», но в вполне пригоден 0). При попытке вычислить какое-то значение, программа смотрит, не вычислялось ли оно ранее, и если да, то берет готовый результат.

Функция принимает следующий вид:

```
function F(X: integer): longint;
begin
if D[X] = 0 then
if (X=1) or (X=2) then D[X] := 1
else D[X] := F(X1) + F(X2);
F := D[X];
end;
```

Листинг – ДП-функция вычисления чисел Фибоначчи

Этот подход динамического программирования называется подходом «сверху вниз». Он запоминает решенные задачи, но очередность решения задач все равно контролирует рекурсия.

Можно еще более упростить решение, убрав рекурсию вообще. Для этого необходимо сменить нисходящую логику рассуждения (от того, что надо найти, к тривиальному) на восходящую (соответственно наоборот). В этой задаче такой переход очевиден и описывается простым циклом:

```
D[1] := 1; D[2] := 1;
For i := 3 to N do
D[i] := D[i1] + D[i2];
```

Листинг 5.26 – Итерационное вычисления чисел Фибоначчи

Здесь использован подход «снизу вверх». Чаще всего, такой способ раза в три быстрее. Однако в ряде случаев такой метод приводит к необходимости решать большее количество подзадач, нежели при рекурсии.

Очень часто для его написания приходится использовать как промежуточный результат нисходящую форму, а иногда безрекурсивная (итеративная) форма оказывается чрезвычайно сложной и малопонятной.

Таким образом, если алгоритм превышает отведенное ему время на тестах большого объема, то необходимо осуществлять доработку этого алгоритма.

Динамическое программирование применимо именно в случаях, когда подзадачи не являются независимыми или же их зависимость установить очень трудно. Алгоритм, основанный на динамическом программировании, решает каждую из подзадач всего один раз и запоминает ответы для подзадач в специальной таблице. Если подзадача встретится еще раз, решение можно взять из таблицы.

Как правило, динамическое программирование применяется к *задачам оптимизации***,** то есть к таким, для которых существует большое количество решений, но их качество определяется каким-либо фактором или параметром. Задача состоит в том, чтобы найти из всех возможных решений то, которое обеспечивает экстремум (максимум или минимум) этого параметра.

Алгоритм, основанный на динамическом программировании, строится следующим образом:

1. Описывается строение оптимального решения;
2. Строится рекуррентное соотношение, связывающее задачу с ее подзадачами;
3. Делается обход дерева подзадач и вычисляем оптимальное значение искомого параметра для каждой из них, запоминая эти решения в таблице;
4. Строится оптимальное решение с использованием полученной информации.

Задача триангуляции многоугольника

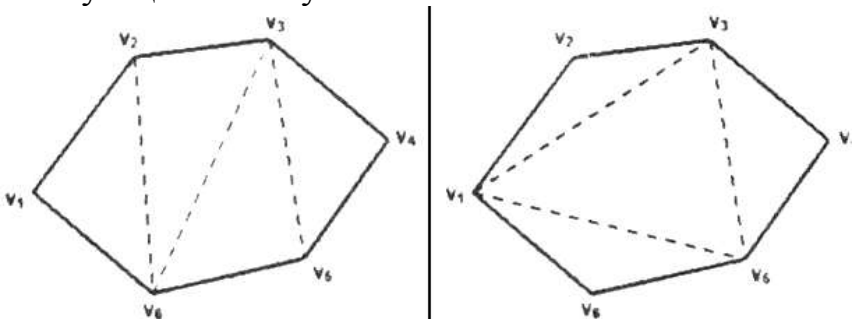


Иллюстрация к задаче триангуляции

Имеется выпуклый n-угольник $P_n = (v_1, v_2 ... v_n)$. Надо разбить его на непересекающиеся треугольники так, чтобы сумма длин отрезков разбиения была наименьшей. На рисунке показано два возможных варианта триангуляции шестиугольника. Будем решать данную задачу в соответствии с вышеизложенными пунктами.

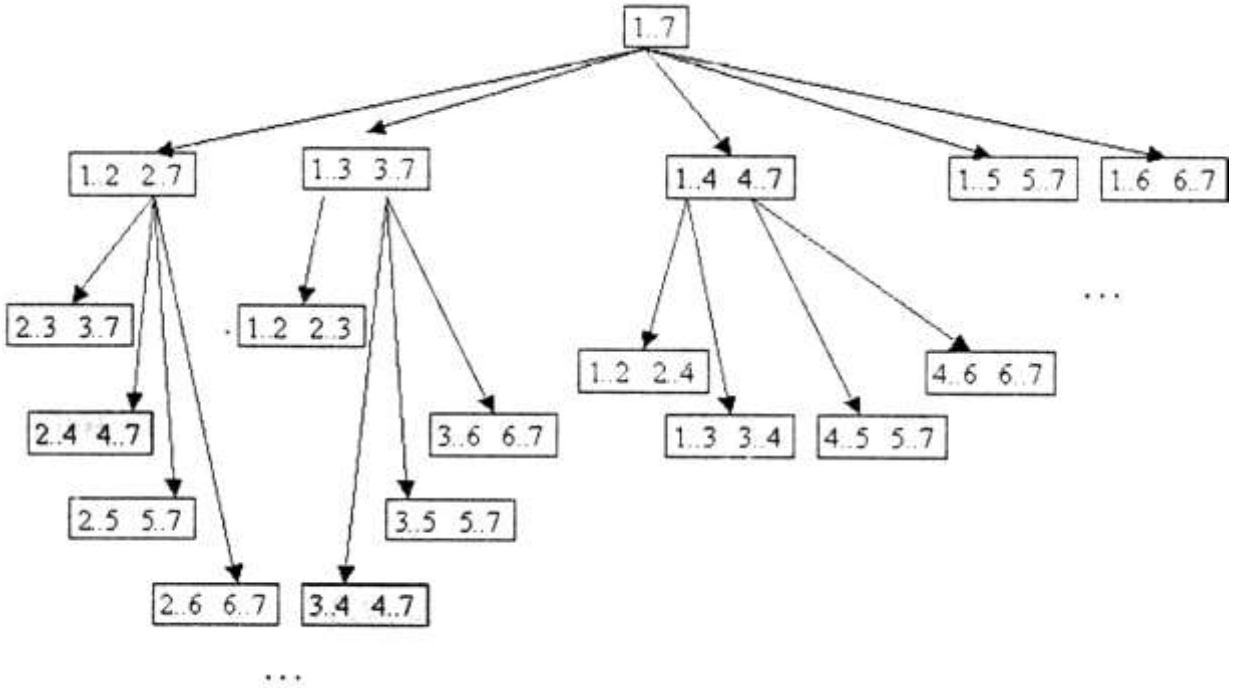
Договоримся обозначать символом $P_{i..j} = (v_i ... v_i)$ многоугольник, образованный вершинами от v_i до v_j ($i < j$), а символом Δ_{klm} – треугольник с вершинами v_k, v_l, v_m. Таким образом, $P_{1..n}$ – это исходный много угольник. Пусть $d(v_i, v_j)$ равняется длине отрезка между вершинами v_i и v_j), если $v_i.v_j$ – диагональ многоугольника и $d(v_i, v_j) = 0$, если v_iv_j – его сторона. Пусть $f(i..j)$ – сумма длин отрезков

оптимального раз биения многоугольника $P_{i..j}$ на треугольники. Заметим следующее: из условия непересекаемости треугольников следует, что каждое ребро $P_{1..n}$ входит ровно в один из треугольников. В частности это касается и ребра v_1v_n которое с некоей вершиной $k(1 < k < n)$ образует Δ_{1kn}. Тогда рассмотрим два оставшиеся многоугольника $P_{1..k}$ и $P_{k..n}$, которые в нашем случае и будут ни чем иным, как подзадачами. Если для всех таких подзадач (при разных k) уже найдено оптимальное решение, то остается лишь вы брать оптимальное значение k.

В задаче требуется найти $f(1..n)$. Построим формулу для нахождения $f(i..j)$ – суммы длин отрезков триангуляции для многоугольника $P_{i..j}$. Имеет место следующая зависимость:

$$f(i..j) = \begin{cases} 0, \text{если } i = j \text{ или } i = j-1 \\ \max\limits_{k=i+1..j-1} \{f(i..k) + f(k..j) + d(v_i, v_k) + d(v_k, v_j)\} \end{cases}$$

Формула следует из того, что вы брав вершину k, мы получаем два меньших многоугольника $P_{i..k}$ и $P_{k..j}$, которые надо триангулировать далее и два отрезка (v_iv_k и v_kv_j), длины которых входят в нашу триангуляцию (если один из них – сторона многоугольника, то согласно выбору d его длина не будет учитываться). Очевидно, что эта формула соответствует рекурсивной процедуре с циклом по k внутри и входными параметрами i и j. Чем не принцип «разделяй и властвуй»? Почти, но не совсем.



Разбиение на подзадачи при триангуляции

Покажем с помощью дерева, как будем собирать целостное решение из решений подзадач для семиугольника. Задача найти $f(1..7)$ распалась на 5 пар задач ($f(1..k)$, $f(k..7)$) для $k = 2..5$. Далее каждая из подзадач рекурсивно разбивается на

более мелкие, пока последние не станут тривиальными (вида *f*(*i*..*i*+1) или *f*(*i*..*i*). Если присмотреться к этому дереву, видно, что одни и те же подзадачи на разных ветках дерева повторяются. Более того, повторяются не только три вальные подзадачи (например, *f*(2..3) на голубых узлах), но и не тривиальные (задача *f*(4..7) на зеленых узлах), которые порождают целые деревья рекурсивных вызовов.

Динамическое программирование заключается как раз в том, что под задачи нужно решать только один раз, а при каждом последующем вызове рекурсивной процедуры для решения той же подзадачи нужно просто прочитать данные из таблицы. Так мы и поступим при написании кода.

```
const
InFileName = 'in.txt'; MaxN = 50; NotCalculated = 1; MaxReal = 1e30;
var
N: integer; (количество вершин) d: array[1..MaxN, 1..MaxN] of real;
(расстояния между вершинами)
f: array[1..MaxN, 1..MaxN] of real;
(динамическая таблица решений)
Cut: аггау[1..иахМ, 1..MaxN] of integer;
(таблица для запоминания вершины разреза)
(функция нахождения расстояния между двумя точками}
function distance(x1,y1,x2,y2: real):real;
begin
distance := sqrt(sqr(xlx2)+sqr(y1y2));
end;
(процедура считывания входящих данных и нахождения расстояний
между вершинами)
procedure ReadData; var
ft: text; x,y: array[1..MaxN] of real;
i,j: integer;
begin
assign(ft,InFileName); reset(ft); read(ft,N); for i:= 1 to N do
readln(ft, x[1],y[i]);
close(ft);
for i:= 1 to N do
for j:= 1 to д do
begin
if abs(ij) <= 1 then
(если это сторона, то) d[i,j]:=0 (в разрез она не войдет,
значит пусть ее длина=0)
else d[i,j]:=Distance(x[i],y[1],x[2],y[2]);
f [i,j]:= NotCalculated;
(ставим отметку, что подзадача f(i,.j) не решена)
end;
end;
(рекурсивная процедура поиска оптимального решения)
```

```
procedure Divide(i,j:integer); var k, minK: integer;
L, minL: real;
begin
if f[i,j] <> NotCalculated then
(если подзадача уже решалась, выходим) exit;
if i+2 >= j then begin
(если задача тривиальна (треугольник, отрезок или точка))
Cut[i,j] := 0; (то разбиений нет)
f [1,2] := 0; (и длина такого несуществующего разбиения
равна нулю)
exit;
end;
minL := MaxReal;
(ищем минимум no k (рекуррентная формула))
for k := 1<1 to j1 do begin
Divide(i,k); (решаем подзадачу f(i..k))
Divide(k,j); (решаем подзадачу f(k..2})
L := P[i,k] + F[k,j] + D[i,k] + D[k,j];
if L < MinL then begin
MinL := L; (запоминаем минимальную длину}
MinK := k; (и точку разбиения)
end;
end;
F[i,j] MinL; (запоминаем оптимум в динамической таблице)
Cut[i,j] :=MinK;
end;
(процедура рекурсивно выводит разбиение)
procedure PrintCut(i,j integer) begin
if Cut[i,j] <> 0 then begin
if i+1 < Cut[i,j] then
WriteLn(i,",Cut[i,j]);
if Cut[i,j]+l < j then
WriteLn(Cut[i,j],",3);
PrintCut(i,Cut[1 3));
PrintCut(Cut[i,j],j);
end;
end;
begin
ReadData; (читаем входные данные) Divide(1,N); (выполняем разбиение)
(выводим результаты) wreteln('Min(mal cut length:f[l,n]);
writeLn('Cut the polygon along these diagonals:');
PrintCut(1,N);
end.
```

Листинг – ДП-алгоритм триангуляции многоугольника

Оптимальное решение будет находиться в нашей таблице после вы зова рекурсивной процедуры с параметрами (1, *n*).

Дп, жадный алгоритм или что-то другое?

Между ДП и жадными алгоритмам и есть нечто общее. Решаемые с помощью жадных алгоритмов задачи, как и задачи, решаемые с помощью ДП, обладают свойством оптимальности для подзадач − когда оптимальное решение всей за дачи содержит в себе оптимальные решения подзадач. Если A – оптимальный набор заявок, содержащий заявку 1, то $A' = A \setminus \{1\}$ – оптимальный набор за явок для подмножества заявок, для которых $b[i] >= e[1]$. Из-за этого общего свойства иногда может появиться желание применить ДП в ситуации, где хватило бы жадного алгоритма, или наоборот, приме нить жадный алгоритм к задаче, в которой он не выдаст оптимального решения (как в дискретной задаче о рюкзаке).

Алгоритм надо выбирать не только в зависимости от возможностей, но и от ваших нужд (например, если нет четкой необходимости получить самое оптимальное решение). Яркий пример – та же задача коммивояжера. Алгоритм ее решения NP-полон, что означает, что кроме как с помощью полного перебора задачу ре шить невозможно (этот факт математически доказан). Да, для нахождения самого краткого пути требуется экспоненциальное число шагов. А что, если задачу надо решить в краткий срок, причем хоть как-то минимизировать путь для обхода всех пунктов? На помощь приходит жадный алгоритм, который будет всего лишь приблизительным алгоритмом, но он найдет один из кратчайших (хотя и не обязательно самый короткий) вариантов обхода. Алгоритм работает таким образом. Допустим, мы находимся в вершине i. Рассмотрим расстояния от i ко всем вершинам, в которых мы еще не были, и выберем среди них вершину j с минимальным расстоянием к i. Далее, двигаемся в вершину j и проделываем для нее то же самое, что и для вершины i, и т. д. Остановимся тогда, когда во всех вершинах мы уже побывали, и соединим последнюю с первой. Общая сложность алгоритма – $\underline{O}(n^2)$, и часто такой алгоритм дает решение, близкое к оптимальному.

6. Алгоритмы на графах

Теория графов в последнее время широко используется в различных отраслях науки и техники. Быстрое развитие данная теория получила с созданием электронно-вычислительной техники, которая позволяла решить многие задачи *алгоритмизации*.

Граф – это совокупность двух конечных множеств: *множества* точек и *множества* линий, попарно соединяющих некоторые из этих точек. Множество точек называется *вершинами (узлами) графа*. Множество линий, соединяющих *вершины графа*, называются *ребрами (дугами) графа*.

Ориентированный граф (орграф) – граф, у которого все *ребра* ориентированы, т. е. *ребрам* которого присвоено направление.

Неориентированный граф (неорграф) – граф, у которого все *ребра* неориентированы, т. е. *ребрам* которого не задано направление.

Смешанный граф – граф, содержащий как ориентированные, так и неориентированные *ребра*.

Петлей называется *ребро*, соединяющее вершину саму с собой. Две вершины называются *смежными*, если существует соединяющее их *ребро*. *Ребра*, соединяющие одну и ту же пару вершин, называются *кратными*.

Простой граф – это *граф*, в котором нет ни петель, ни кратных ребер.

Мультиграф – это *граф*, у которого любые две вершины соединены более чем одним *ребром*.

Маршрутом в графе называется конечная чередующаяся последовательность смежных вершин и ребер, соединяющих эти вершины.

Маршрут называется *открытым*, если его начальная и конечная вершины различны, в противном случае он называется замкнутым.

Маршрут называется **цепью**, если все его *ребра* различны. Открытая цепь называется **путем**, если все ее вершины различны.

Замкнутая цепь называется **циклом**, если различны все ее вершины, за исключением концевых.

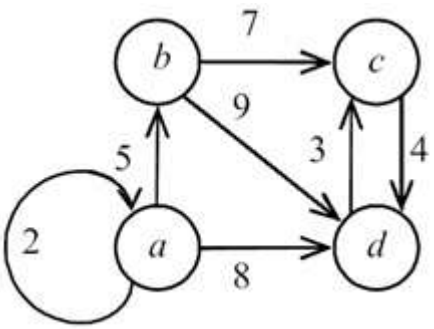
Граф называется *связным*, если для любой пары вершин существует соединяющий их *путь*.

Вес вершины – число (действительное, целое или рациональное), поставленное в соответствие данной вершине (интерпретируется как *стоимость*, *пропускная способность* и т. д.). *Вес (длина) ребра* – число или несколько чисел, которые интерпретируются по отношению к ребру как *длина*, *пропускная способность* и т. д.

Взвешенный граф – граф, каждому ребру которого поставлено в соответствие некое *значение* (*вес ребра*).

Выбор структуры данных для хранения графа в памяти компьютера имеет принципиальное *значение* при разработке *эффективных алгоритмов*. Рассмотрим несколько **способов представления графа**.

Пусть задан *граф*, у которого количество вершин равно n, а количество ребер – m. Каждое *ребро* и каждая *вершина* имеют *вес* – целое положительное число. Если *граф* не является помеченным, то считается, что *вес* равен единице.



Граф

1. *Список ребер* – это множество, образованное парами смежных вершин. Для его хранения обычно используют *одномерный массив* размером m, содержащий список пар вершин, смежных с одним *ребром* графа. *Список ребер* более удобен для реализации различных алгоритмов на графах по сравнению с другими способами.

a	*a*	*a*	*b*	*b*	*c*	*d*
a	*b*	*d*	*c*	*d*	*d*	*c*
2	5	8	7	9	4	3

Список ребер графа

2. *Матрица смежности* – это *двумерный массив* размерности n x n, значения элементов которого характеризуются *смежностью вершин графа*. При этом значению элемента матрицы присваивается количество ребер, которые соединяют соответствующие вершины. Данный способ действенен, когда надо проверять смежность или находить вес *ребра* по двум заданным вершинам.

	a	*b*	*c*	*d*
a	2	5	0	8
b	0	0	7	9
c	0	0	0	4
d	0	0	3	0

Матрица смежности графа

3. *Матрица инцидентности* – это *двумерный массив* размерности n x m, в котором указываются связи между *инцидентными* элементами графа (*ребро* и вершина). Столбцы матрицы соответствуют *ребрам*, строки – вершинам. Ненулевое значение в ячейке матрицы указывает связь между вершиной и *ребром*. Данный способ является самым емким для хранения, но облегчает нахождение циклов в графе.

	a	*b*	*c*	*d*
(*a*, *a*)	2	0	0	0
(*a*, *b*)	0	5	0	0
(*a*, *d*)	0	0	0	8
(*b*, *c*)	0	0	7	0
(*b*, *d*)	0	0	0	9
(*c*, *d*)	0	0	0	4
(*d*, *c*)	0	0	3	0

Матрица инцидентности графа

Существует много алгоритмов на графах, в основе которых лежит *систематический* перебор *вершин графа*, такой, что каждая *вершина* просматривается (посещается) в точности один раз. Поэтому важной задачей является нахождение хороших методов поиска в графе.

Под **обходом графов (поиском на графах)** понимается процесс *систематического* просмотра всех ребер или *вершин графа* с целью отыскания ребер или вершин, удовлетворяющих некоторому условию.

При решении многих задач, использующих графы, необходимы эффективные методы регулярного обхода вершин и ребер графов. К стандартным и наиболее распространенным методам относятся:

- *поиск в глубину* (Depth First Search, *DFS*);
- *поиск в ширину* (Breadth First Search, *BFS*).

Эти методы чаще всего рассматриваются на *ориентированных графах*, но они применимы и для неориентированных, *ребра* которых считаются *двунаправленными*. Алгоритмы *обхода в глубину* и в ширину лежат в основе решения различных задач обработки графов, например, построения *остовного леса*, проверки *связности*, ацикличности, вычисления расстояний между вершинами и других.

Поиск в глубину

При *поиске в глубину* посещается первая *вершина*, затем необходимо идти вдоль ребер графа, до попадания в *тупик*. *Вершина* графа является *тупиком*, если все смежные с ней вершины уже посещены. После попадания в *тупик* нужно возвращаться назад вдоль пройденного пути, пока не будет обнаружена *вершина*, у которой есть еще не посещенная *вершина*, а затем необходимо двигаться в этом новом направлении. Процесс оказывается завершенным при возвращении в начальную вершину, причем все смежные с ней вершины уже должны быть посещены.

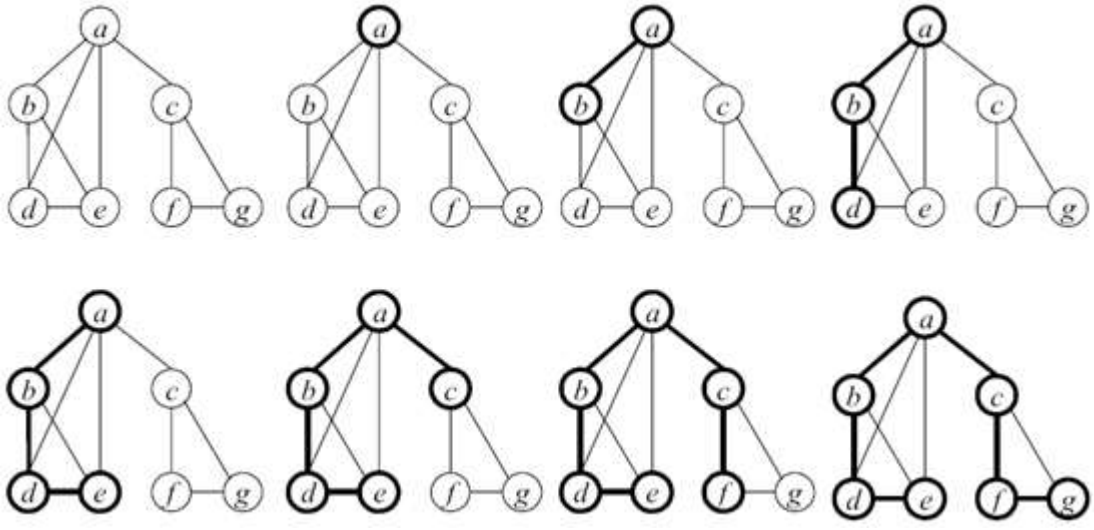
Таким образом, основная идея поиска в глубину – когда возможные пути по *ребрам*, выходящим из вершин, разветвляются, нужно сначала полностью исследовать одну ветку и только потом переходить к другим веткам (если они останутся нерассмотренными).

Алгоритм поиска в глубину

Шаг 1. Всем вершинам графа присваивается *значение* не посещенная. Выбирается первая *вершина* и помечается как посещенная.

Шаг 2. Для последней помеченной как посещенная вершины выбирается смежная *вершина*, являющаяся первой помеченной как не посещенная, и ей присваивается *значение* посещенная. Если таких вершин нет, то берется предыдущая помеченная *вершина*.

Шаг 3. Повторить шаг 2 до тех пор, пока все вершины не будут помечены как посещенные.



Демонстрация алгоритма поиска в глубину

```
//Описание функции алгоритма поиска в глубину
void Depth_First_Search(int n, int **Graph, bool *Visited,
                        int Node){
  Visited[Node] = true;
  cout << Node + 1 << endl;
  for (int i = 0 ; i < n ; i++)
    if (Graph[Node][i] && !Visited[i])
      Depth_First_Search(n,Graph,Visited,i);
}
```

Также часто используется нерекурсивный *алгоритм* поиска в глубину. В этом случае *рекурсия* заменяется на *стек*. Как только *вершина* просмотрена, она помещается в *стек*, а использованной она становится, когда больше нет новых вершин, смежных с ней.

Временная сложность зависит от представления графа. Если применена *матрица смежности*, то временная сложность равна $O(n^2)$, а если нематричное *представление* – O(n+m): рассматриваются все вершины и все *ребра*.

Поиск в ширину

При *поиске в ширину*, после посещения первой вершины, посещаются все соседние с ней вершины. Потом посещаются все вершины, находящиеся на расстоянии двух ребер от начальной. При каждом новом шаге посещаются вершины, *расстояние* от которых до начальной на единицу больше предыдущего. Чтобы предотвратить повторное посещение вершин, необходимо вести *список* посещенных вершин. Для хранения временных данных, необходимых для работы алгоритма, используется *очередь* – упорядоченная последовательность элементов, в которой новые элементы добавляются в конец, а старые удаляются из начала.

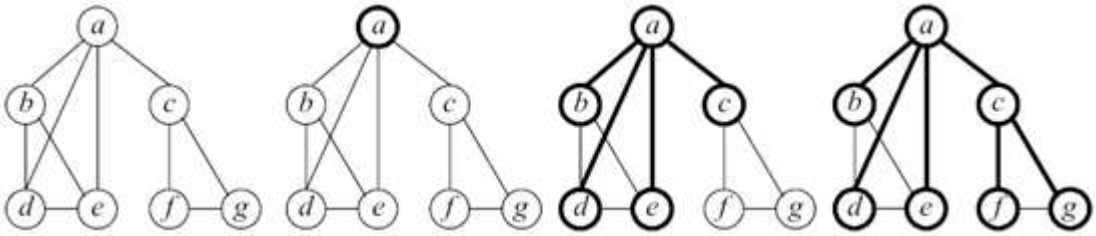
Таким образом, основная идея *поиска в ширину* заключается в том, что сначала исследуются все вершины, смежные с начальной вершиной (*вершина* с которой начинается обход). Эти вершины находятся на расстоянии 1 от начальной. Затем исследуются все вершины на расстоянии 2 от начальной, затем все на расстоянии 3 и т. д. Обратим внимание, что при этом для каждой вершины сразу находятся *длина* кратчайшего маршрута от начальной вершины.

Алгоритм поиска в ширину

Шаг 1. Всем вершинам графа присваивается *значение* не посещенная. Выбирается первая *вершина* и помечается как посещенная (и заносится в *очередь*).

Шаг 2. Посещается первая *вершина* из очереди (если она не помечена как посещенная). Все ее соседние вершины заносятся в *очередь*. После этого она удаляется из очереди.

Шаг 3. Повторяется шаг 2 до тех пор, пока *очередь* не пуста.



Демонстрация алгоритма поиска в ширину

```
//Описание функции алгоритма поиска в ширину
void Breadth_First_Search(int n, int **Graph,
                          bool *Visited, int Node){
  int *List = new int[n]; //очередь
  int Count, Head;        // указатели очереди
  int i;
  // начальная инициализация
  for (i = 0; i < n ; i++)
    List[i] = 0;
  Count = Head = 0;
  // помещение в очередь вершины Node
  List[Count++] = Node;
  Visited[Node] = true;
  while ( Head < Count ) {
    //взятие вершины из очереди
    Node = List[Head++];
    cout << Node + 1 << endl;
    // просмотр всех вершин, связанных с вершиной Node
    for (i = 0 ; i < n ; i++)
      // если вершина ранее не просмотрена
      if (Graph[Node][i] && !Visited[i]){
        // заносим ее в очередь
        List[Count++] = i;
        Visited[i] = true;
      }
  }
}
```

Сложность *поиска в ширину* при нематричном представлении графа равна O(n+m), ибо рассматриваются все n вершин и m ребер. Использование *матрицы смежности* приводит к оценке $O(n^2)$

Ключевые термины

Вес (длина) ребра – это число или несколько чисел, которые интерпретируются по отношению к ребру как *длина*, *пропускная способность*.

Вес вершины – это число (действительное, целое или рациональное), поставленное в соответствие данной вершине.

Взвешенный граф – это *граф*, каждому ребру которого поставлен в соответствие его *вес*.

Граф – это совокупность двух конечных множеств: *множества* точек и *множества* линий, попарно соединяющих некоторые из этих точек.

Вершины (узлы) графа – это множество точек, составляющих *граф*.

Замкнутый маршрут – это *маршрут* в графе, у которого начальная и конечная вершины совпадают.

Кратные ребра – это *ребра*, соединяющие одну и ту же пару вершин.

Маршрут в графе – это конечная чередующаяся последовательность смежных вершин и ребер, соединяющих эти вершины.

Матрица инцидентности – это *двумерный массив*, в котором указываются связи между *инцидентными* элементами графа (*ребро* и *вершина*).

Матрица смежности – это *двумерный массив*, значения элементов которого характеризуются *смежностью вершин графа*

Мультиграф – это *граф*, у которого любые две вершины соединены более чем одним *ребром*.

Неориентированный граф (неорграф) – это *граф*, у которого все *ребра* неориентированы, то есть *ребрам* которого не задано направление.

Обход графа (поиск на графе) – это процесс *систематического* просмотра всех ребер или *вершин графа* с целью отыскания ребер или вершин, удовлетворяющих некоторому условию.

Ориентированный граф (орграф) – это *граф*, у которого все *ребра* ориентированы, то есть *ребрам* которого присвоено направление.

Открытый маршрут – это *маршрут* в графе, у которого начальная и конечная вершины различны.

Петля – это *ребро*, соединяющее вершину саму с собой.

Поиск в глубину – это *обход графа* по возможным путям, когда нужно сначала полностью исследовать одну ветку и только потом переходить к другим веткам (если они останутся нерассмотренными).

Поиск в ширину – это *обход графа* по возможным путям, когда после посещения вершины, посещаются все соседние с ней вершины.

Простой граф – это *граф*, в котором нет ни петель, ни кратных ребер.

Путь – это открытая цепь, у которой все вершины различны.

Ребра (дуги) графа – это множество линий, соединяющих *вершины графа*.

Связный граф – это *граф*, у которого для любой пары вершин существует соединяющий их *путь*.

Смежные вершины – это вершины, соединенные общим *ребром*.

Смешанный граф – это *граф*, содержащий как ориентированные, так и неориентированные *ребра*.

Список ребер – это множество, образованное парами смежных вершин

Тупик – это *вершина* графа, для которой все смежные с ней вершины уже посещены

Цепь – это *маршрут* в графе, у которого все *ребра* различны.

Цикл – это замкнутая цепь, у которой различны все ее вершины, за исключением концевых.