Языки программирования высокого уровня. Парадигмы программирования. Объектно-ориентированное программирование.

Язык программирования высокого уровня – это формализованная семантическая система, максимально приближенная к обычному человеческому языку или иным привычным знаковым системам (например, математическим формулам). Язык высокого уровня в минимальной степени привязан к процессору или операционной системе и направлен на то, чтобы программист сосредоточился на решении поставленной задачи, не отвлекаясь на особенности устройства компьютера.

Написание большинства современных компьютерных программ осуществляется при задействовании языков высокого уровня. Благодаря языка высокого уровня программирование стало массовой профессией. Переход к языку высокого уровня дал возможность укрупнять конструкции при подготовке текстов программ.

Программирование как наука, искусство и технология исследует и творчески развивает процессы создания и применения программ, определяет средства и методы конструирования программ, разнообразие которых складывается в практике и экспериментах и фиксируется в форме языка программирования. Сложности классификации быстро расширяющегося множества языков программирования приводит к выделению «парадигмы программирования», число которых меняется не столь стремительно. Это ставит задачу определения принадлежности языка программирования конкретной парадигме программирования или поддержки парадигмы программирования определённым языком.

Парадигма программирования — это совокупность идей и понятий, определяющих стиль написания компьютерных программ. Это способ концептуализации, определяющий организацию вычислений и структурирование работы, выполняемой компьютером.

Парадигма программирования не определяется однозначно языком программирования; практически все современные языки программирования в той или иной мере допускают использование различных парадигм (мультипарадигмальное программирование). Так, на языке Python можно работать в соответствии с принципами структурного и объектно-ориентированного программирования; функциональное программирование можно применять при работе на любом императивном языке, в котором имеются функции, и т. д.

Также существующие парадигмы зачастую пересекаются друг с другом в деталях (например, императивное и объектно-ориентированное программирование). На рисунке 1 представлена некоторая систематизация парадигм программирования.

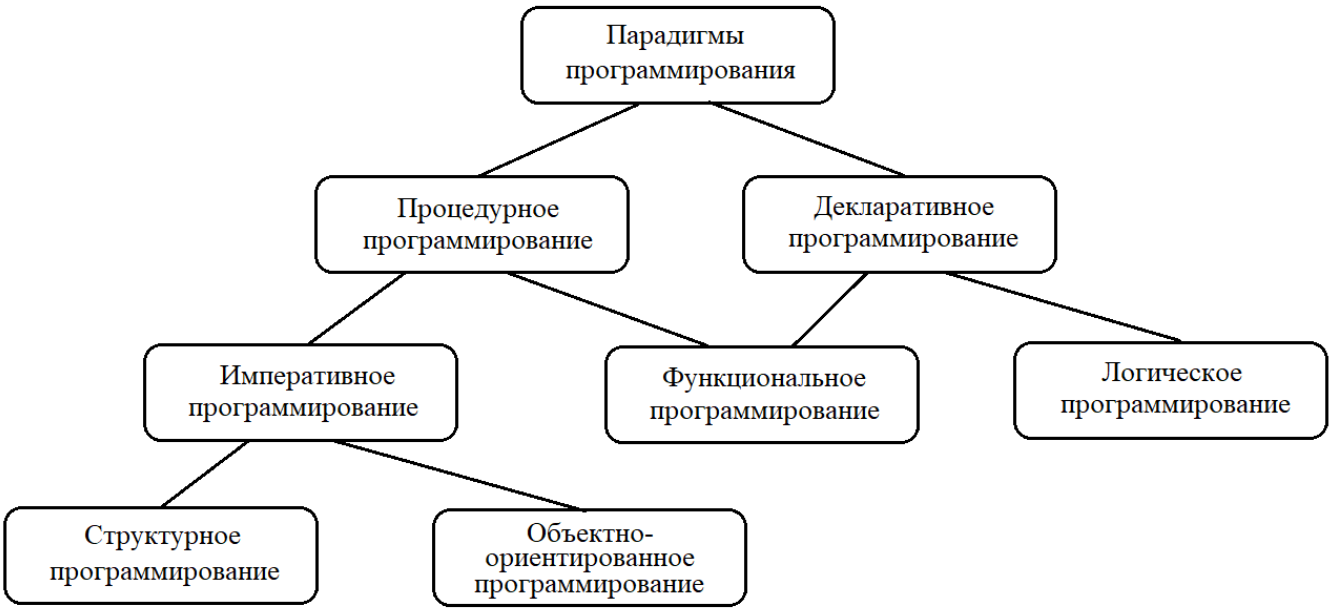

Рисунок 1 — Систематизация парадигм программирования

Процедурное программирование — есть отражение фон Неймановской архитектуры компьютера. Программа, написанная на процедурном языке, представляет собой последовательность команд, определяющих алгоритм решения задачи. Основная идея процедурного программирования - использование памяти для хранения данных. Основная команда — присвоение, с помощью которой определяется и меняется память компьютера. Программа производит преобразование содержимого памяти, изменяя его от исходного состояния к результирующему. Для управления процессом выполнения используются следующие конструкции: последовательность, ветвление, цикл и вызов процедуры.

Эта парадигма является самой старой. Она развивалась по мере появления новых концепций в языках программирования: трансляция (Fortran), типизация (Pascal), модули (Modula) и универсальность (Ada, C).

Преимущества процедурного программирования:

- Любая процедура (функция) может быть вызвана неограниченное количество раз.
- Возможность оперативно решить задачу, в которой отсутствует сложная иерархия.

Недостатки процедурного программирования:

- Риск возникновения множества ошибок при работе над большим проектом. Приходится писать много процедур, и это не может не сказаться на чистоте и работоспособности кода.
- Все данные процедуры доступны только внутри нее. Их нельзя вызвать из другого места программы и при необходимости придется писать аналогичный код.

Декларативное программирование — парадигма программирования, в которой задаётся спецификация решения задачи, то есть описывается, что представляет собой проблема и ожидаемый результат. В общем и целом, в

декларативном программировании описывается что сделать. Как следствие, декларативные программы не используют понятия состояния, в частности, не содержат переменных и операторов присваивания, обеспечивается ссылочная прозрачность.

Наиболее близким к «чисто декларативному» программированию является *написание исполнимых спецификаций*. В этом случае программа представляет собой формальную теорию, а её выполнение является одновременно автоматическим доказательством этой теории.

К подвидам декларативного программирования также зачастую относят функциональное и логическое программирование — несмотря на то, что программы на таких языках нередко содержат алгоритмические составляющие, архитектура в императивном понимании (как нечто отдельное от кодирования) в них также отсутствует: схема программы является непосредственно частью исполняемого кода.

Функциональное программирование — парадигма программирования, в которой процесс вычисления трактуется как вычисление значений функций в математическом понимании последних (в отличие от функций как подпрограмм в процедурном программировании). При необходимости, в функциональном программировании вся совокупность последовательных состояний вычислительного процесса представляется явным образом, например, как список.

Функциональное программирование предполагает обходиться вычислением результатов функций от исходных данных и результатов других функций, и не предполагает явного хранения состояния программы. Соответственно, не предполагает оно и изменяемость этого состояния.

В функциональном языке программирования при вызове функции с одними и теми же аргументами всегда получается одинаковый результат, то есть выходные данные зависят только от входных. Это позволяет средам выполнения программ на функциональных языках кешировать результаты функций и вызывать их в порядке, не определяемом алгоритмом и распараллеливать их без каких-либо дополнительных действий со стороны программиста (что обеспечивают функции без побочных эффектов — чистые функции).

Лямбда-исчисление являются основой для функционального программирования, многие функциональные языки можно рассматривать как «надстройку» над ними.

Наиболее известными языками функционального программирования являются: Lisp, Erlang (функциональный язык с поддержкой процессов), APL (предшественник современных научных вычислительных сред, таких как Matlab), ML, F# (функциональный язык семейства ML для платформы .NET), Haskell (чистый функциональный, названный в честь Хаскелла Карри).

Логическое программирование — парадигма программирования, основанная на автоматическом доказательстве теорем, а также раздел дискретной математики, изучающий принципы логического вывода информации на основе заданных фактов и правил вывода. Логическое программирование основано на теории и аппарате математической логики с использованием математических принципов резолюций.

Самым известным языком логического программирования является Prolog.

Первым языком логического программирования был язык Planner, в котором была заложена возможность автоматического вывода результата из данных и заданных правил перебора вариантов. Planner использовался для того, чтобы понизить требования к вычислительным ресурсам (с помощью поиска с возвратом) и обеспечить возможность вывода фактов, без активного использования стека. Затем был разработан язык Prolog, который не требовал плана перебора вариантов и был, в этом смысле, упрощением языка Planner.

От языка Planner также произошли логические языки программирования Popler, Conniver и QLISP. Языки программирования Mercury, Visual Prolog произошли уже от языка Prolog.

Императивное программирование — это парадигма программирования, для которой характерно следующее:

- в исходном коде программы записываются инструкции (команды);
- инструкции должны выполняться последовательно;
- данные, получаемые при выполнении предыдущих инструкций, могут читаться из памяти последующими инструкциями;
- данные, полученные при выполнении инструкции, могут записываться в память.

Императивная программа похожа на приказы, выражаемые повелительным наклонением в естественных языках, то есть представляют собой последовательность команд, которые должен выполнить компьютер.

Основные черты императивных языков:

- использование именованных переменных;
- использование оператора присваивания;
- использование составных выражений;
- использование подпрограмм.

Структурное программирование — парадигма программирования, в основе которой лежит представление программы в виде иерархической структуры блоков. В соответствии с парадигмой, любая программа, которая строится без использования оператора goto, состоит из трёх базовых управляющих конструкций: последовательность, ветвление, цикл; кроме того, используются подпрограммы. При этом разработка программы ведётся пошагово, методом «сверху вниз».

Методология структурного программирования появилась как следствие возрастания сложности решаемых на компьютерах задач, и соответственно, усложнения программного обеспечения. Поэтому потребовалась систематизация процесса разработки и структуры программ.

Становление и развитие структурного программирования связано с именем Эдсгера Дейкстры, который описал семь принципов структурного программирования:

Принцип 1. Следует отказаться от использования оператора безусловного перехода goto.

Принцип 2. Любая программа строится из трёх базовых управляющих конструкций: последовательность, ветвление, цикл.

Принцип 3. В программе базовые управляющие конструкции могут быть вложены друг в друга произвольным образом. Никаких других средств управления последовательностью выполнения операций не предусматривается.

Принцип 4. Повторяющиеся фрагменты программы можно оформить в виде *подпрограмм*. Таким же образом (в виде подпрограмм) можно оформить логически целостные фрагменты программы, даже если они не повторяются.

Принцип 5. Каждую логически законченную группу инструкций следует оформить как блок. Блоки являются основой структурного программирования.

> Блок — это логически сгруппированная часть исходного кода, например, набор инструкций, записанных подряд в исходном коде программы. Понятие *блок* означает, что к блоку инструкций следует обращаться как к единой инструкции. Блоки служат для ограничения области видимости переменных и функций. Блоки могут быть пустыми или вложенными один в другой. Границы блока строго определены. Например, в if-инструкции блок ограничен фигурными скобками {…} (в языке C) или отступами (в языке Python).

Принцип 6. Все перечисленные конструкции должны иметь один вход и один выход.

> Ограничив управляющими конструкциями с одним входом и одним выходом, получается возможность построения произвольных алгоритмов любой сложности с помощью простых и надежных механизмов.

Принцип 7. Разработка программы ведётся пошагово, методом «сверху вниз».

> В соответствии с методом сначала пишется текст основной программы, в котором, вместо каждого связного логического фрагмента текста, вставляется вызов подпрограммы, которая будет выполнять этот фрагмент. Вместо настоящих, работающих подпрограмм, в программу вставляются фиктивные части — *заглушки*, которые ничего не делают, но удовлетворяют требованиям интерфейса заменяемого фрагмента (модуля).
> Затем заглушки заменяются или дорабатываются до настоящих полнофункциональных фрагментов (модулей) в соответствии с планом программирования. На каждой стадии процесса реализации уже созданная программа должна правильно работать по отношению к более низкому уровню. Полученная программа проверяется и отлаживается.
> После того, как программист убедится, что подпрограммы вызываются в правильной последовательности (то есть общая структура программы верна), подпрограммы-заглушки последовательно заменяются на реально работающие, причём разработка каждой подпрограммы ведётся тем же методом, что и основной программы. Разработка заканчивается тогда, когда не останется ни одной заглушки.

Такая последовательность гарантирует, что на каждом этапе разработки программист одновременно имеет дело с обозримым и понятным ему множеством фрагментов, и может быть уверен, что общая структура всех более высоких уровней программы верна.

Методология структурной разработки программного обеспечения была признана «самой сильной формализацией 70-х годов».

Структурное программирование стало основой всего, что сделано в методологии программирования, включая и объектное программирование.

Объектно-ориентированное программирование (ООП) — методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определённого класса, а классы образуют иерархию наследования.

Идеологически ООП — подход к программированию как к моделированию информационных объектов, решающий на новом уровне основную задачу структурного программирования: структурирование информации с точки зрения управляемости, что существенно улучшает управляемость самим процессом моделирования, что, в свою очередь, особенно важно при реализации крупных проектов.

Основные принципы структурирования в случае ООП связаны с различными аспектами базового понимания предметной задачи, которое требуется для оптимального управления соответствующей моделью:

- Абстракция данных, которая означает выделение значимой информации и исключение из рассмотрения незначимой. В ООП рассматривают лишь абстракцию данных (нередко называя её просто «абстракцией»), подразумевая набор наиболее значимых характеристик объекта, доступных остальной программе.
- инкапсуляция — это свойство системы, позволяющее объединить данные и методы, работающие с ними, в классе;
- наследование — это свойство системы, позволяющее описать новый класс на основе уже существующего с частично или полностью заимствующейся функциональностью. Класс, от которого производится наследование, называется базовым, родительским или суперклассом. Новый класс — потомком, наследником, дочерним или производным классом.
- полиморфизм — это свойство системы, позволяющее использовать объекты с одинаковым интерфейсом без информации о типе и внутренней структуре объекта.

Кроме этого, основными понятиями ООП являются:

Класс — универсальный, комплексный тип данных, состоящий из тематически единого набора «полей» и «методов», то есть он является моделью информационной сущности с внутренним и внешним интерфейсами для оперирования своим содержимым (значениями полей). В частности, в классах широко используются специальные блоки из одного или чаще двух спаренных методов, отвечающих за элементарные операции с определённым полем (интерфейс присваивания и считывания значения), которые имитируют непосредственный доступ к полю. Эти блоки называются

«свойствами» и почти совпадают по конкретному имени со своим полем. Другим проявлением интерфейсной природы класса является то, что при копировании соответствующей переменной через присваивание копируется только интерфейс, но не сами данные, то есть класс — ссылочный тип данных. Переменная-объект, относящаяся к заданному классом типу, называется экземпляром этого класса. При этом в некоторых исполняющих системах класс также может представляться некоторым объектом при выполнении программы посредством динамической идентификации типа данных. Обычно классы разрабатывают таким образом, чтобы обеспечить отвечающие природе объекта и решаемой задаче целостность данных объекта, а также удобный и простой интерфейс. В свою очередь, целостность предметной области объектов и их интерфейсов, а также удобство их проектирования, обеспечивается наследованием.

Объект — сущность в адресном пространстве вычислительной системы, появляющаяся при создании экземпляра класса (например, после запуска результатов компиляции и связывания исходного кода на выполнение).

Появление в ООП отдельного понятия класса закономерно вытекает из желания иметь множество объектов со сходным поведением. Класс в ООП — это в чистом виде абстрактный тип данных, создаваемый программистом. С этой точки зрения объекты являются значениями данного абстрактного типа, а определение класса задаёт внутреннюю структуру значений и набор операций, которые над этими значениями могут быть выполнены. Желательность иерархии классов (а значит, наследования) вытекает из требований к повторному использованию кода — если несколько классов имеют сходное поведение, нет смысла дублировать их описание, лучше выделить общую часть в общий родительский класс, а в описании самих этих классов оставить только различающиеся элементы.

Необходимость совместного использования объектов разных классов, способных обрабатывать однотипные сообщения, требует поддержки полиморфизма — возможности записывать разные объекты в переменные одного и того же типа. В таких условиях объект, отправляя сообщение, может не знать в точности, к какому классу относится адресат, и одни и те же сообщения, отправленные переменным одного типа, содержащим объекты разных классов, вызовут различную реакцию.

Первоначально (например, в Smalltalk) взаимодействие объектов представлялось как «настоящий» обмен сообщениями, то есть пересылка от одного объекта другому специального объекта-сообщения. Такая модель является чрезвычайно общей. Она прекрасно подходит, например, для описания параллельных вычислений с помощью *активных объектов*, каждый из которых имеет собственный поток исполнения и работает одновременно с прочими. Такие объекты могут вести себя как отдельные, абсолютно автономные вычислительные единицы. Посылка сообщений естественным образом решает вопрос обработки сообщений объектами, присвоенными полиморфным переменным — независимо от того, как объявляется переменная, сообщение обрабатывает код класса, к которому относится присвоенный переменной объект. Данный подход реализован в языках программирования Smalltalk, Ruby, Objective-C, Python.

Однако общность механизма обмена сообщениями имеет и другую сторону — «полноценная» передача сообщений требует дополнительных накладных расходов, что не всегда приемлемо. Поэтому во многих современных объектно-ориентированных языках программирования используется концепция «*отправка сообщения как вызов метода*» — объекты имеют доступные извне методы, вызовами которых и обеспечивается взаимодействие объектов. Данный подход реализован в огромном количестве языков программирования, в том числе C++, Object Pascal, Java. Однако, это приводит к тому, что сообщения уже не являются самостоятельными объектами, и, как следствие, не имеют атрибутов, что сужает возможности программирования. Некоторые языки используют гибридное представление, демонстрируя преимущества одновременно обоих подходов — например, Python.

Концепция виртуальных методов, поддерживаемая этими и другими современными языками, появилась как средство обеспечить выполнение нужных методов при использовании полиморфных переменных, то есть, по сути, как попытка расширить возможности вызова методов для реализации части функциональности, обеспечиваемой механизмом обработки сообщений.

Многие современные языки специально созданы для облегчения объектно-ориентированного программирования. Однако можно применять техники ООП и для не-объектно-ориентированного языка и наоборот, применение объектно-ориентированного языка вовсе не означает, что код автоматически становится объектно-ориентированным.

Как правило, объектно-ориентированный язык содержит следующий набор элементов:

- Объявление классов с полями (данными — членами класса) и методами (функциями — членами класса).
- Механизм расширения класса (наследования) — порождение нового класса от существующего с автоматическим включением всех особенностей реализации класса-предка в состав класса-потомка. Большинство объектно-ориентированных языков поддерживают только единичное наследование.
- Полиморфные переменные и параметры функций (методов), позволяющие присваивать одной и той же переменной экземпляры различных классов.
- Полиморфное поведение экземпляров классов за счёт использования виртуальных методов. В некоторых объектно-ориентированных языках все методы классов являются виртуальными.

Некоторые языки добавляют к указанному минимальному набору те или иные дополнительные средства. В их числе:

- Конструкторы, деструкторы;
- Свойства;
- Индексаторы;
- Средства управления видимостью компонентов классов (интерфейсы или модификаторы доступа, такие как public, private, protected).

Одни языки отвечают принципам ООП в полной мере — в них все основные элементы являются объектами, имеющими состояние и связанные методы.

Примеры подобных языков — Smalltalk. Существуют гибридные языки, совмещающие объектную подсистему в целостном виде с подсистемами других парадигм как «два и более языка в одном», позволяющие совмещать в одной программе объектные модели с иными, и размывающие грань между объектно-ориентированной и другими парадигмами за счёт нестандартных возможностей, балансирующих между ООП и другими парадигмами (таких как множественная диспетчеризация, параметрические классы, возможность манипулировать методами классов как самостоятельными объектами, и др.). Примеры таких языков: Python, Ruby, Objective-C. Однако, наиболее распространены языки, включающие средства эмуляции объектной модели поверх более традиционной императивной семантики. Примеры таких языков — Симула, C++, Delphi, Java, C#.

Основные понятия объектно-ориентированного программирования. Классы и объекты

С момента изобретения компьютера методологии программирования резко изменились, в основном из-за возрастающей сложности программ. По мере совершенствования методологии программирования каждый новый метод позволял создавать все более сложные и крупные программы, улучшая предыдущие подходы. До изобретения объектно-ориентированного программирования многие проекты достигали границ, за которыми структурный подход уже не работал. Для преодоления этих препятствий была создана объектно-ориентированная парадигма.

Объектно-ориентированное программирование унаследовало лучшие идеи структурного программирования и объединило их с новыми понятиями. В результате возник новый способ организации программ. В принципе, есть два способа организации программ: положить в основу её или данные. В рамках структурного подхода главным понятием является код. Так, программа, написанная на структурном языке определяется её функциями, каждая из которых может оперировать данными любого типа.

Объектно-ориентированные программы работают иначе. В их основу положены данные, а базовый принцип формулируется так: "данные контролируют доступ к коду". В объектно-ориентированных языках определяются данные и процедуры, осуществляющие к ним доступ. Итак, тип данных точно определяет, какого рода операции к нему можно применять.

Для поддержки объектно-ориентированного программирования язык должен обладать тремя свойствами: инкапсуляцией, полиморфизмом и наследованием.

Инкапсуляция – это механизм, связывающий воедино код и данные, которыми он манипулирует, а также обеспечивающий их защиту от внешнего вмешательства и неправильного использования. В объектно-ориентированном языке код и данные можно погружать в "черный ящик", который называется *объектом*. Иначе говоря, объект – это средство инкапсуляции.

Внутри объекта код и данные могут быть *закрытыми* (private) или *открытыми* (public). Закрытый код или данные объекта доступны только из другой части этого же объекта. Иначе говоря, к закрытой части кода или данных невозможно обратиться извне. Если код или данные являются открытыми, они доступны из любой части программы. Как правило, открытая часть кода обеспечивает управляемое взаимодействие (интерфейс) с закрытыми элементами объекта.

Как с синтаксической, так и с семантической точки зрения объект представляет собой переменную, тип которой определён пользователем. Определяя новый тип объекта, будет определён новый тип данных. Каждый экземпляр этого типа данных является сложной переменной.

Языки объектно-ориентированного программирования поддерживают *полиморфизм*, который характеризуется фразой "один интерфейс, несколько методов". Проще говоря, полиморфизм – это атрибут, позволяющий с помощью одного интерфейса управлять доступом к целому классу методов. Конкретный

выбор определяется возникшей ситуацией. Например, в программе определены три разных типа стека. Один стек состоит из целых чисел, другой – из символов, а третий – из чисел с плавающей точкой. Благодаря полиморфизму программисту достаточно определить функции **push()** и **pop()**, которые можно применять к любому типу стека. В программе необходимо создать три разные версии этих функций для каждой разновидности стека, но имена этих функций должны быть одинаковыми. Компилятор автоматически выберет правильную функцию, основываясь на информации о типе данных, хранящихся в стеке. Таким образом, функции **push()** и **pop()** – интерфейс стека – одинаковы, независимо от типа стека. Конкретные варианты этих функций определяют конкретные реализации (методы) для каждого типа данных.

Полиморфизм позволяет упростить программу, создавая один интерфейс для выполнения разных действий. Ответственность за выбор *конкретного действия* (метода) в возникшей ситуации перекладывается на компилятор. Программисту не обязательно вмешиваться в этот процесс. Нужно лишь помнить правила и правильно применять *общий интерфейс.*

Наследование – это процесс, в ходе которого один объект может приобретать свойства другого. Он имеет большое значение, поскольку поддерживает концепцию *классификации*. Если подумать, все знания организованы по принципу иерархической классификации. Например, "антоновка" относится к классу *"яблоки",* который в свою очередь является частью класса *"фрукты",* входящего в класс *"пища".* Если бы классификации не существовало, сложно было бы точно описать свойства объектов. Однако при этом необходимо указывать только уникальные свойства объекта, позволяющие выделить его среди других объектов данного класса. Именно этот принцип лежит в основе механизма наследования, дающего возможность считать конкретный объект специфическим экземпляром более общей разновидности.

Для того чтобы создать объект, в языке C++ сначала необходимо определить его общий вид, используя ключевое слово **class**. С синтаксической точки зрения класс похож на структуру. Например, тип **stack** с его помощью можно создать реальный стек, можно определить следующим образом.

```
#define SIZE 100
// В этом фрагменте объявляется класс stack.
class stack {
   int stck[SIZE];
   int tos;
public:
   void init();
   void push(int i);
   int pop();
};
```

Класс может содержать открытую и закрытую части. По умолчанию все члены класса считаются закрытыми. Например, переменные **stck** и **tos** являются закрытыми. Это означает, что к ним невозможно обратиться из функций, не являющихся членами класса **stack**. Таким образом достигается инкапсуляция –

доступ к закрытым элементам класса строго контролируется. В классе можно также определить закрытые функции, которые могут вызываться другими членами класса.

Для того чтобы открыть элементы класса (т.е. сделать их доступными для других частей программы), следует объявить их с помощью ключевого слова **public**. Все переменные или функции, размещённые в разделе **public**, доступны для любых функций программы. По существу, внешняя часть программы получает доступ к объекту именно через его открытые функции-члены. Хотя переменные можно объявлять открытыми, этого следует избегать. Наоборот, все данные рекомендуется объявлять закрытыми, контролируя доступ к ним с помощью открытых функций.

Функции **init()**, **push()** и **pop()** называются *функциями-членами,* поскольку они являются частью класса **stack**. Переменные **stck** и **tos** называются *переменными-членами* (или *данными-членами).* Как известно, объект создаёт связь между кодом и данными. Только функции-члены имеют доступ к закрытым членам своего класса. Следовательно, доступ к переменным **stck** и **tos** имеют только функции **init()**, **push()** и **pop()**.

Определив класс, можно создать объект этого типа. В сущности, имя класса становится новым спецификатором типа данных. Например, следующий оператор создает объект с именем **mystack** типа **stack**.

```
stack mystack;
```

При объявлении объекта класса создаётся *экземпляр* (instance) этого класса. В данном случае переменная **mystack** является экземпляром класса **stack**. Кроме того, объекты можно создавать, указывая их имя сразу после определения класса, т.е. после закрывающей фигурной.

В языке C++ с помощью ключевого слова **class** определяется новый тип данных, который можно использовать для создания объектов этого типа. Следовательно, объект – это экземпляр класса. В этом смысле он ничем не отличается от других переменных, например, от переменной, представляющей собой экземпляр типа **int**. Иначе говоря, класс является логической абстракцией, а объект – её реальным воплощением, существующим в памяти компьютера.

Определение простого класса имеет следующий вид.

class *имя_класса*
{
закрытые переменные и функции
public:
открытые переменные и функции
} *список имен объектов* ;

Список имен объектов может быть пустым.

Внутри определения класса **stack** функции-члены идентифицируются с помощью своих прототипов. В языке C++ все функции должны иметь прототипы. Таким образом, прототипы являются неотъемлемой частью класса. Прототип функции-члена, размещённый внутри определения класса, имеет тот же смысл, что и прототип обычной функции.

Кодируя функцию-член класса необходимо сообщить компьютеру, какому именно классу она принадлежит. Для этого перед её именем следует указать имя соответствующего класса. Вот как, например, определяется функция **push()** из класса **stack**.

```
void stack::push(int i)
{
   if<tos==SIZE) {
      cout << "Стек переполнен.\n";
      return;
   }
   stck[tos] = i;
   tos++;
}
```

Оператор "::" называется *оператором разрешения области видимости*. Он сообщает компилятору, что данная версия функции **push()** принадлежит классу **stack**, т.е. функция **push()** пребывает в области видимости класса **stack**. Благодаря оператору разрешения области видимости компилятор всегда знает, какому классу принадлежит каждая из функций.

Ссылаясь на член класса извне, всегда следует указывать конкретный объект, которому этот член принадлежит. Для этого используется имя объекта, за которым следует оператор "." и имя члена. Это правило относится как к данным-членам, так и к функциям-членам. Вот как, например, вызывается функция **init()**, принадлежащая объекту **stack1**.

```
stack stack1, stack2;
stack1.init();
```

В этом фрагменте создаются два объекта – **stack1** и **stack2**, а затем объект **stack1** инициализируется с помощью функции **init()**. Здесь переменные **stack1** и **stack2** представляют собой два разных объекта. Таким образом, инициализация объекта **stack1** *не означает* инициализации объекта **stack2**. Объекты **stack1** и **stack2** связывает лишь то, что они являются экземплярами одного и того же класса.

Внутри класса функции-члены могут вызывать друг друга и обращаться в переменным-членам, не используя оператор ".". Этот оператор необходим, лишь когда обращение к функциям и переменным-членам осуществляется извне класса.

Вот как может выглядеть функция **main()** программы, в которой определён класс **stack**.

```
int main()
{
   stack stack1, stack2; // Создаются два объекта
                         // класса stack
   stack1.init(); stack2.init();
   stack1.push(1); stack2.push(2);
   stack1.push(3); stack2.push(4);
   std::cout << stack1.pop() << " ";
```

```
   std::cout << stack1.pop() << " ";
   std::cout << stack2.pop() << " ";
   std::cout << stack2.pop() << "\n";
   return 0;
}
```

Очень часто некоторая часть объекта перед его первым использованием должна быть инициализирована. Например, перед первым использованием объекта ранее рассмотренного класса **stack** переменной **tos** следует присвоить число 0. Для этого предназначена функция **init()**. Поскольку инициализация объектов – очень распространенное требование, в языке C++ предусмотрен особый механизм, позволяющий инициализировать объекты в момент их создания. Эта автоматическая инициализация осуществляется конструктором.

Конструктор – это особая функция, являющаяся членом класса. Её имя должно совпадать с именем класса. Например, класс **stack** можно модифицировать, предусмотрев в нём конструктор.

```
// Класс stack с конструктором.
class stack {
   int stck[SIZE];
   int tos;
public:
   stack(); // Конструктор
   void push(int i);
   int pop();
};
```

В объявлении конструктора **stack()** не указывается тип возвращаемого значения, поскольку в языке C++ конструкторы в принципе не могут возвращать значения.

Код конструктора **stack()** может выглядеть так.

```
// Конструктор класса stack
stack::stack()
{
   tos = 0;
   std::cout << "Стек инициализирован\n";
}
```

Здесь сообщение "Стек инициализирован" просто иллюстрирует работу конструктора. На практике конструкторы чаще всего ничего не вводят и не выводят: они просто выполняют различные виды инициализации.

Конструктор автоматически вызывается в момент создания объекта, т.е. при его объявлении. Следовательно, объявление объекта в языке C++ – это не пассивная запись, а активный процесс. В языке C++ объявление является выполняемым оператором. Код, с помощью которого конструируется объект, может быть довольно важным. Для глобальных и статических локальных объектов конструкторы вызываются лишь однажды. При объявлении локальных объектов конструкторы вызываются каждый раз при входе в соответствующий блок.

Антиподом конструктора является *деструктор.* Во многих ситуациях объект должен выполнить некоторое действие или действия, которые уничтожат его. Локальные объекты создаются при входе в соответствующий блок, а при выходе из него они уничтожаются. При разрушении объекта автоматически вызывается его деструктор. Для этого существует несколько причин. Например, объект должен освободить занимаемую им память или закрыть файл, открытый им ранее. В языке C++ эти действия выполняет деструктор. Имя деструктора должно совпадать с именем конструктора, но перед ним ставится знак ~ (тильда). Деструктор, содержащийся в классе **`stack`**, может быть реализован следующим образом.

```
// Объявление класса stack.
class stack {
   int stck[SIZE];
   int tos;
public:
   stack(); // Конструктор
   ~stack(); // Деструктор
   void push(int i) ;
   int pop();
};

// Деструктор класса stack
stack::~stack{)
{
   std::cout << "Стек уничтожен\п";
}
```

Как и в случае выше предложенной реализации конструктора сообщение "Стек уничтожен" просто иллюстрирует работу деструктора.

Перегрузка функций и операторов

Перегрузка функций – это использование одного имени для нескольких функций. Секрет перегрузки заключается в том, что каждое переопределение функции должно использовать либо другие типы параметров, либо другое их количество. Только эти различия позволяют компилятору определять, какую функцию следует вызвать в том или ином случае. Например, в следующей программе функция **myfunc()** перегружена для разных типов параметров.

```
#include <iostream>

using namespace std;

int myfunc(int i); // Эти варианты различаются
                   // типами параметров
double myfunc(double i);

int main()
{
   cout<<myfunc(10)<<" "; // Вызов функции myfunc(int i)
   cout<<myfunc(5.4); // Вызов функции myfunc(double i)
   return 0;
}

double myfunc(double i)
{
   return i;
}

int myfunc(int i)
{
   return i;

}
```

В следующей программе перегруженные варианты функции **myfunc()** используют разное количество параметров.

```
#include <iostream>

using namespace std;

int myfunc(int i); /* Эти варианты различаются количеством
                    параметров */
int myfunc(int i, int j);
int main()
{
   cout<<myfunc(10)<< " "; // Вызов функции myfunc(int i)
   cout<<myfunc(4, 5); // Вызов функции
                       // myfunc(int i, int j)
   return 0;
}
int myfunc(int i)
{
   return i;
```

```
}
int myfunc(int i, int j)
{
   return i*j;
}
```

Перегруженные функции должны отличаться типами или количеством параметров. Тип возвращаемого значения не позволяет перегружать функции. Например, следующий вариант перегрузки функции **myfunc**() неверен.

```
int myfunc(int i); // Ошибка: разных типов возвращаемого
float myfunc(int i); // значения недостаточно для
                        //перегрузки
```

Иногда объявления двух функций внешне отличаются, но фактически совпадают, как в следующем примере.

```
void f(int *p) ;
void f(int p[]); // Ошибка, выражения *p и p[] эквивалентны
```

Компилятор не различает выражения ***p** и **p[]**. Следовательно, хотя внешне два прототипа функции **f** различаются, на самом деле они полностью совпадают.

Конструкторы также можно перегружать. Фактически их чаще всего перегружают. Для перегрузки конструктора существуют три причины: гибкость, возможность создания инициализированных (неинициализированных) объектов и конструкторов копирования.

Довольно часто объекты класса можно создать несколькими способами. Для каждого из этих способов можно определить отдельный вариант перегруженного конструктора. Эти варианты исчерпывают все возможности создать объект – при попытке сделать это непредусмотренным способом компилятор не найдет подходящего конструктора и выдаст сообщение об ошибке.

Перегруженные конструкторы намного повышают гибкость класса. Они позволяют пользователю выбирать оптимальный способ создания объекта. Следующий пример программы создаёт класс **date** для хранения календарной даты. В этом примере конструктор перегружен дважды.

```
#include <iostream>
#include <cstdio>

using namespace std;

class date {
     int day, month, year;
public:
     date(char *d);
     date(int m, int d, int y);
     void show_date();
};
// Инициализация строкой.
date::date(char * d)
{
     sscanf(d, "%d%*c%d%*c%d", &month, &day, &year);
```

```
}
// Инициализация целыми числами.
date::date(int m, int d, int y)
{
     day = d;
     month = m;
     year = y;
}
void date::show_date()
{
     cout << month << "/" << day;
     cout << "/" << year << "\n";
}
int main()
{
     date ob1(12, 4, 2001), ob2("10/22/2001");
     ob1.show_date();
     ob2.show_date();
     return 0;
}
```

В этой программе объект класса **date** можно инициализировать двумя способами: задав месяц, день и год в виде трех целых чисел либо в виде строки *mm/dd/yyyy*. Оба способа применяются довольно часто, поэтому имеет смысл предусмотреть два разных конструктора для создания объектов класса **date**.

Этот пример иллюстрирует основную идею, лежащую в основе перегруженных конструкторов: они позволяют выбирать способ создания объектов, лучше всех соответствующий конкретной ситуации. Например, пользователь может ввести дату в виде массива **s**. Эту строку можно сразу использовать для создания объекта класса **date**. Для этого совершенно не требуется преобразовывать ее в другой вид. Однако, если бы конструктор **date()** не был бы перегружен, строку **s** пришлось бы разбить на три целых числа.

```
int main()
{
     char s[80];
     cout << "Введите новую дату: ";
     cin >> s;
     date d(s);
     d.show_date();
     return 0;
}
```

В другой ситуации пользователю удобнее инициализировать объект класса **date** тремя целыми числами. Например, если дата является результатом неких вычислений, более естественно создавать объект класса **date** с помощью конструктора **date(int, int, int)**. Перегруженный конструктор обеспечивает необходимую гибкость класса, особенно необходимую при создании библиотек.

С перегрузкой функций тесно связан механизм перегрузки операторов. В языке C++ можно перегрузить большинство операторов, настроив их на конкретный класс. Перегруженный оператор сохраняет свое первоначальное предназначение. Просто набор типов, к которым его можно применять, расширяется.

Перегрузка операторов осуществляется с помощью *операторных функции*, которые определяют действия перегруженных операторов применительно к соответствующему классу. Операторные функции создаются с помощью ключевого слова **operator**. Операторные функции могут быть как членами класса, так и обычными функциями. Однако обычные операторные функции, как правило, объявляют дружественными по отношению к классу, для которого они перегружают оператор. В каждом из этих случаев операторная функция объявляется по-разному.

Операторная функция-член имеет следующий вид:

тип_возвращаемого_значения имя_класса :: operator# (*список-аргументов)* {
... // Операции
}

Обычно операторная функция возвращает объект класса, с которым она работает, однако тип возвращаемого значения может быть любым. Символ # заменяется перегружаемым оператором. Например, если в классе перегружается оператор деления "**/**", операторная функция-член называется **operator/**. При перегрузке унарного оператора *список аргументов* остается пустым. При перегрузке бинарного оператора *список аргументов* содержит один параметр.

Ниже представлен пример программы, создающую класс **loc**, в котором хранятся географические координаты: широта и долгота. В программе перегружаются операторы "**+**", "**-**", "**=**", "**++**".

```
#include <iostream>
using namespace std;
class loc {
   int longitude, atitudes;
public :
   loc() {} // Этот конструктор необходим для создания
            // временных объектов
   loc(int lg, int lt) {
      longitude = lg;
      latitude = lt;
   }
   void show() {
      cout << longitude << " " << latitude << "\n" ;
   }
   loc operator+(loc op2);
   loc operator-(loc op2);
   loc operator=(loc op2);
   loc operator++();
} ;
```

```
// Перегруженный оператор + для класса loc.
loc loc::operator+(loc op2) {
   loc temp;
   temp.longitude = op2.longitude + longitude;
   temp.latitude = op2.latitude + latitudes;
   return temp;
}
// Перегруженный оператор - для класса loc.
loc loc::operator-(loc op2) {
   loc temp;
   temp.longitude = longitude - op2.longitude;
   temp.latitude = latitude - op2.latitude;
   return temp;
}
// Перегруженный оператор присваивания для класса 1oc.
loc loc::operator=(loc op2) {
   longitude = op2.longitude;
   latitude = op2.latitudes;
   return *this; // Возвращает объект, генерирующий вызов
}
// Перегруженный префиксный оператор инкрементации ++
// для класса loc.
loc loc::operator++() {
   longitude++;
   latitude++;
   return *this;
}
int main() {
   loc ob1(10, 20), ob2( 5, 30), ob3(90, 90);
   ob1.show(); ob2.show();
   ++obl;
   ob1.show(); // Выводит на экран числа 11 21
   ob2 = ++obl;
   ob1.show(); // Выводит на экран числа 12 22
   ob2.show(); // Выводит на экран числа 12 22
   ob1 = ob2 = ob3; // Множественное присваивание
   obl.show(); // Выводит на экран числа 90 90
   ob2.show(); // Выводит на экран числа 90 90
   return 0;
}
```

Функция **operator+()** имеет только один параметр, несмотря на то, что она перегружает бинарный оператор. Причина заключается в том, что операнд, стоящий в левой части оператора, передается операторной функции неявно с помощью указателя **this**. Операнд, стоящий в правой части оператора, передается операторной функции через параметр **op2**. Отсюда следует важный вывод: при перегрузке бинарного оператора вызов операторной функции генерируется объектом, стоящим в левой части оператора.

Как правило, перегруженные операторные функции возвращают объект класса, с которым они работают. Следовательно, перегруженный оператор можно использовать внутри выражений.

При реализации операторной функции **operator-()** изменён порядок её операндов. Учитывая смысл оператора вычитания, операнд, стоящий в его правой части, вычитается из операнда, стоящего слева. Поскольку вызов операторной функции **operator-()** генерируется объектом, стоящим слева от знака "минус", данные объекта **op2** должны вычитаться из данных объекта, на который ссылается указатель **this**.

Если оператор "**=**" не перегружен, для класса автоматически создаётся оператор присваивания по умолчанию, который создаёт побитовые копии объектов. Перегружая оператор "**=**", можно явно определить оператор присваивания для конкретного класса. В данном случае оператор присваивания ничем не отличается от стандартного, однако в других ситуациях он может выполнять иные действия. Функция **operator=()** возвращает указатель ***this** на объект, сгенерировавший её вызов. Это позволяет выполнять множественное присваивание объектов, как показано ниже.

```
ob1 = ob2 = ob3; // Множественное присваивание
```

В определении операторной функции **operator++()** отсутствуют параметры. Поскольку оператор инкрементации "**++**" является унарным, его единственный операнд неявно передается ему с помощью указателя **this**.

Операторные функции **operator=()** и **operator++()** изменяют значения своих операндов. Левому операнду функции **operator=()** присваивается новое значение, оператор инкрементации увеличивает значение своего операнда на единицу.

Стандарт языка C++ позволяет явно создавать отдельные версии префиксного и постфиксного операторов инкрементации и декрементации. Для этого следует определить две версии функции **operator++()** (и функции **operator--()** соответственно). В предыдущей программе была определена префиксная версия оператора, а постфиксная версия определена ниже.

```
loc operator++(int x);
```

Если символы **++** предшествуют операндам, вызывается операторная функция **operator++()**, если символы **++** следуют за операндами, вызывается операторная функция **operator++ (int x)**.

Вот как выглядит общая форма префиксной и постфиксной операторной функции **operator--()**.

```
// Префиксный оператор декрементации
 тип operator--() {
   // Тело префиксного оператора
}
```

```
// Постфиксный оператор декрементации
 тип operator--(int i) {
   // Тело постфиксного оператора
}
```

Сокращенные операторы присваивания (например, "**+=**", "**-=**" и т.п.) также можно перегружать. Ниже представлен вариант операторной функций **operator+=()** для класса **loc**.

```
loc loc::operator+=(loc op2) {
   longitude = op2.longitude + longitude;
   latitude = op2.latitude + latitude;
   return *this;
}
```

На применение перегруженных операторов налагается несколько ограничений. Во-первых, нельзя изменить приоритет оператора. Во-вторых, невозможно изменить количество операндов оператора, однако операнд можно игнорировать. В-третьих, операторную функцию нельзя вызывать с аргументами, значения которых заданы по умолчанию. И нельзя перегружать следующие операторы:

```
.   ::   .*   ?
```

За исключением оператора операторные функции наследуются производными классами. Однако в производном классе каждый из этих операторов снова можно перегрузить.

Операторы можно перегружать с помощью дружественных функций, не являющихся членами класса. Это значит, что дружественные функции не получают неявного указателя **this**. Следовательно, перегруженная операторная функция получает параметры явно. Таким образом, при перегрузке бинарного оператора дружественная функция получает два параметра, а при перегрузке унарного оператора — один. Первым параметром дружественной функции, перегружающей бинарный оператор, является его левый операнд, а вторым — правый операнд.

В следующей программе функция **operator+()** является дружественной по отношению к классу **loc**.

```
#include <iostream>

using namespace std;

class loc {
   int longitude, latitude;
public:
   loc() {} // Этот конструктор необходим для создания
            // временных объектов
   loc(int lg, int lt) {
      longitude = lg;
      latitude = lt;
   }

   void show() {
      cout << longitude << " ";
      cout << latitude << "\n";
```

```
    }
    friend loc operator+(loc op1, loc op2); // Дружественная
                                            // функция
    loc operator-(loc op2);
    loc operator=(loc op2);
    loc operator++();
};

// Оператор + перегружается с помощью
// дружественной функции.
loc operator+(loc op1, loc op2) {
    loc temp;
    temp.longitude = op1.longitude + op2.longitude;
    temp.latitude = op1.latitude + op2.latitude;
    return temp;
}

// Перегруженный оператор - для класса loc.
loc loc::operator-(loc op2) {
    loc temp;
    temp.longitude = longitude - op2.longitude;
    temp.latitude = latitude - op2.latitude;
    return temp;
}

// Перегруженный оператор присваивания для класса 1oc.
loc loc::operator=(loc op2) {
    longitude = op2.longitude;
    latitude = op2.latitudes;
    return *this; // Возвращает объект, генерирующий вызов
}

// Перегруженный префиксный оператор инкрементации ++
// для класса loc.
loc loc::operator++() {
    longitude++;
    latitude++;
    return *this;
}

int main() {
    loc ob1(10, 20), ob2( 5, 30);
    ob1 = ob1 + ob2;
    ob1.show();
    return 0;
}
```

На применение дружественных операторных функций налагаются несколько ограничений. Во-первых, с помощью дружественных функций нельзя перегружать операторы "**=**", "**()**", "**[]**" и "**->**". Во-вторых, при перегрузке операторов инкрементации и декрементации параметр дружественной функции следует передавать по ссылке.

Наследование классов

Наследование позволяет создавать иерархические классификации. Используя наследование, можно создавать общие классы, определяющие свойства, характерные для всей совокупности родственных классов. Эти классы могут наследовать свойства друг у друга, добавляя к ним свои собственные уникальные характеристики.

Согласно стандартной терминологии языка C++ класс, лежащий в основе иерархии, называется *базовым* (base class), а класс, наследующий свойства базового класса, – *производным* (derived class). Производные классы, в свою очередь, могут быть базовыми по отношению к другим классам.

При наследовании члены базового класса становятся членами производного класса. Для наследования используется следующая синтаксическая конструкция.

```
class имя_производиого_класса :уровень_доступа имя-базового-класса{
// тело класса
};
```

Параметр *уровеннь_доступа* определяет статус членов базового класса в производном классе. в качестве этого параметра используются спецификаторы **public**, **private** или **protected**. Если уровень доступа не указан, то для производного класса по умолчанию используется спецификатор **private**, а для производной структуры – **public**. В этих ситуация могут возникать следующие варианты.

Если уровень доступа к членам базового класса задается спецификатором **public**, то все открытые и защищенные члены базового класса становятся открытыми и защищенными членами производного класса. При этом закрытые члены базового класса не меняют своего статуса и остаются недоступными членам производного. Как демонстрирует следующая программа, объекты класса **derived** могут непосредственно ссылаться на открытые члены класса **base**.

```
#include <iostream>
using namespace std;
class base{
     int i, j;
public:
     void set(int a, int b) {i=a; j=b;}
     void show(){cout << i << " " << j << "\n";}
};
class derived : public base{
     int k;
public:
     derived(int x) {k=x;}
     void showk() {cout << k << "\n";}
};
int main()
{
     derived ob(3);
```

```
    ob.set(1, 2); // Обращение к члену класса base
    ob.show(); // Обращение к члену класса base
    ob.showk(); // Обращение к члену класса derived
    return 0;
}
```

Если свойства базового класса наследуются с помощью спецификатора доступа **private**, все открытые и защищенные члены базового класса становятся закрытыми членами производного класса.

Спецификатор **protected** повышает гибкость механизма наследования. Если член класса объявлен защищенным (protected), то вне класса он недоступен. С этой точки зрения защищенный член класса ничем не отличается от закрытого. Единственное исключение из этого правила касается наследования. В этой ситуации защищенный член класса существенно отличается от закрытого.

При открытом наследовании защищенные члены базового класса становятся защищенными членами производного класса и, следовательно, доступны остальным членам производного класса. Иными словами, защищенные члены класса по отношению к своему классу являются закрытыми и в то же время, могут наследоваться производным классом. Это иллюстрирует следующий пример.

```
#include <iostream>

using namespace std;

class base{
protected:
    int i, j ; // Закрыты по отношению к классу base,
               // но доступны классу derived.
public:
    void set(int a, int b) {i=a; j=b;}
    void show() {cout << i << " " << j << "\n";}
};

class derived : public base(
    int k;
public:
// Класс derived имеет доступ к членам i и j из класса
base
    void setk() {k=i*j;}
    void showk() {cout << k << " \n";}
};

int main()
{
    derived ob;
    ob.set(2, 3); // Все в порядке, этот член доступен
                  // классу derived
    ob.show(); // Все в порядке, этот член доступен
               // классу derived
    ob.setk();
    ob.showk();
```

```
    return 0;
}
```

В данном примере, поскольку класс **derived** наследует свойства класса **base** с помощью открытого наследования, а переменные **i** и **j** объявлены защищенными, функция **setk()** из класса **derived** имеет к ним доступ. Если бы переменные **i** и **j** были объявлены в классе `base` закрытыми, то класс **derived** не имел бы к ним доступа, и программу нельзя было скомпилировать.

Если производный класс является базовым по отношению к другому производному классу, то любой защищенный член исходного базового класса, открыто наследуемый первым производным классом, также может наследоваться вторым производным классом как защищенный член. Например, следующая программа вполне корректна, и класс **derived2** действительно имеет доступ к переменным **i** и **j**.

```
#include <iostream>

using namespace std;

class base{
protected:
    int i, j;
public:
    void set(int a, int b) {i=a; j=b;}
    void show() {cout << i << " " << j << " \n";}


};

// Переменные i и j наследуются как защищенные.
class derived1 : public base{
    int k;
public:
    void setk() {k = i*j;} // допустимо
    void showk() {cout << k << "\n";}
};

// i и j наследованы косвенно через класс derivedl.
class derived2 : public derived1{
    int m
public :
    void setm() {m = i-j;} // Допускается
    void showm() {cout << m << "\n";}
};

int main()
{
    Derived1 ob1;
    derived2 ob2;
    ob1.set(2, 3 );
    ob1.show();
    ob1.setk();
    ob1.showk();
```

```
     ob2.set(3, 4);
     ob2.show();
     ob2.setk();
     ob2.setm();
     ob2.showk();
     ob2.showm();
     return 0;
}
```

Однако, если бы к классу **base** применялся механизм закрытого наследования, то все его члены стали бы закрытыми членами класса **derived1** и были недоступны классу **derived2**. В то же время переменные **i** и **j** были бы по-прежнему доступны классу **derived1**.

Производный класс может одновременно наследовать свойства нескольких базовых классов. Например, в программе, приведенной ниже, класс `derived` наследует свойства классов `base1` и `base2`.

```
#include <iostream>

using namespace std;

class base1{
protected:
     int x;
public:
     void showx() {cout << x << "\n";}

};

class base2{
protected:
     int y;
public:
     void showy() {cout << y << "\n";}
};

// Множественное наследование.
class derived: public basel, public base2{
public:
     void set{int i, int j) {x=i; y=j;}
};

int main{)
{
     derived ob;
     ob.set(10, 20); // Эта функция принадлежит
                        // классу derived.
     ob.showx (); // Эта функция принадлежит классу basel.
     ob.showy(); // Эта функция принадлежит классу base2.
     return 0;
}
```

При множественном наследовании имена базовых классов перечисляются в списке и разделяются запятыми, причем перед каждым именем базового класса указывается свой спецификатор доступа.

В связи с наследованием возникают два вопроса, касающиеся конструкторов и деструкторов. Во-первых, когда вызываются конструкторы и деструкторы базового и производного классов? Во-вторых, как передаются параметры конструкторов базового класса?

Базовый и производный класс могут содержать несколько конструкторов и деструктор. Следующий пример программы дает представление о порядке из вызова при создании и уничтожении объектов производного класса.

```
#include <iostream>

using namespace std;

class base{
public:
   base() {cout << "Создается объект класса base\n";}
   ~base() {cout << "Уничтожается объект класса base\n";}
};
class derived: public base{
public:
   derived() {cout << "Создается объект класса derived\n";}
   ~derived() {
      cout << "Уничтожается объект класса derived\n";
   }
};

int main()
{
     derived ob;
// Кроме создания и уничтожения объекта,
//ничего не происходит
     return 0;
}
```

В ходе выполнения программа выводит на экран следующие сообщения.

```
Создание объекта класса base
Создание объекта класса derived
Уничтожение объекта класса derived
Уничтожение объекта класса base
```

Как видно из примера, сначала вызывается конструктор базового класса, а затем – производного. После этого, поскольку объект **ob** немедленно уничтожается, вызывается деструктор класса **derived**, а за ним – деструктор класса **base**.

Результаты этого эксперимента можно обобщить. При создании объекта производного класса сначала вызывается конструктор базового класса, а потом – производного. При уничтожении объекта производного класса сначала вызывается деструктор производного класса, а затем – базового. Иначе говоря, конструкторы вызываются в иерархическом порядке, а деструкторы – в обратном.

Поскольку базовый класс не имеет никакой информации о производных классах, инициализация его объектов должна выполняться до инициализации любого объекта производного класса. Следовательно, конструктор базового класса должен вызываться первым.

Вполне очевидно, что деструкторы должны вызываться в обратном порядке. Поскольку производный класс наследует свойства базового, уничтожение объекта базового класса вызовет уничтожение объекта производного класса. Следовательно, деструктор производного класса должен вызываться до полного уничтожения объекта.

При иерархическом наследовании (когда производный класс становится базовым для своего наследника) применяется следующее правило: конструкторы вызываются в иерархическом порядке, а деструкторы – в обратном.

Если конструктор производного класса должен получать несколько параметров, следует просто использовать стандартную синтаксическую форму конструктора с параметрами. Для передачи аргументов конструктору базового класса применяется расширенная форма объявления конструктора производного класса, которая позволяет передавать аргументы нескольким конструкторам одного или нескольких базовых классов. Общая форма этой синтаксической конструкции такова.

```
конструктор_производного_класса(список_аргументов) : base1 (список_аргументов),
                                                      base2(список_аргументов),
                                                      ...
                                                      baseN(список_аргументов)
{
   // Тело конструктора производного класса
}
```

Здесь параметры *base1-baseN* являются именами базовых классов. Объявление конструктора производного класса должно быть отделено двоеточием от спецификаций базовых классов, которые, в свою очередь, разделяются запятыми. Следующая программа иллюстрирует это способ передачи параметров конструктору базового класса.

```
#include <iostream>
using namespace std;
class base{
protected:
   int i;
public:
   base(int x) {
      i=x;
      cout << "Создание объекта класса base\n";
   }
   ~base() {cout << "Уничтожение объекта класса base\n";}
};

class derived: public base{
   int j;
public:
```

```
// Класс derived использует переменную x;
// переменная y передается базовому классу.
   derived(int x, int y): base(y){
      j=x;
      cout << "Создание объекта класса derived\n";
   }
   ~derived() {
   cout << "Уничтожение объекта класса derived\n";
   }
    void show() {cout << i << " " << j << " \n";}
};
int main()
{
    derived ob(3, 4) ;
    ob.show(); // Выводит на экран числа 4 3
    return 0;
}
```

Здесь конструктор класса **derived** имеет два параметра: **x** и **y**. Однако в самом конструкторе используется лишь переменная **x**, а переменная **y** передается конструктору базового класса. Как правило, в конструкторе производного класса должны объявляться все параметры, необходимые базовому классу. Для этого они указываются после двоеточия в списке аргументов конструктора базового класса. Следующий пример иллюстрирует передачу параметров в случае множественного наследования.

```
#include <iostream>

using namespace std;

class base1{
protected:
   int i;
public:
   base1(int x) {
      i=x;
      cout << "Создание объекта класса base1\n";
   }
   ~base1() {cout << "Уничтожение объекта класса base1\n";}
};

class base2{
protected:
   int k;
public:
   base2(int x) {
      k=x;
      cout << "Создание объекта класса base2\n";
   }
   ~base2() {cout << "Уничтожение объекта класса base2\n";}
};
```

```
class derived: public base1, public base2 {
   int j;
public:
   derived(int x, int y, int z): base1(y), base2(z)
   {
      j=x;
      cout << "Создание объекта класса derived\n";
   }
   ~derived() {
      cout << "Уничтожение объекта класса derived\n";
   }
   void show() {cout << i << " " << j << " " << k << "\n";}
};

int main()
{
   derived ob(3, 4, 5);
   ob.show(); // Выводит на экран числа 4 3 5
   return 0;
}
```

Аргументы конструктора базового класса передаются с помощью аргументов конструктора производного класса. Следовательно, даже если конструктор производного класса не имеет собственных аргументов, его объявление должно содержать аргументы конструкторов базовых классов. В этом случае аргументы, передаваемые конструктору производного класса, просто переправляются конструкторам базовых классов.

Виртуальные функции и полиморфизм

Полиморфизм в объектно-ориентированном программировании – это атрибут, позволяющий с помощью одного интерфейса управлять доступом к целому классу методов. Конкретный выбор определяется возникшей ситуацией. Полиморфизм позволяет упростить программу, создавая один интерфейс для выполнения разных действий. Ответственность за выбор *конкретного действия* (метода) в возникшей ситуации перекладывается на компилятор.

Язык C++ обеспечивает как статический, так и динамический полиморфизм. Статический полиморфизм достигается с помощью перегрузки функций и операторов. Динамический полиморфизм реализуется на основе наследования и виртуальных функций.

Виртуальная функция – это функция-член, объявленная в базовом классе и переопределенная в производном. Чтобы создать виртуальную функцию, следует указать ключевое слово **`virtual`** перед её объявлением в базовом классе. Производный класс переопределяет эту функцию, приспосабливая её для своих нужд. По существу, виртуальная функция реализует принцип "один интерфейс, несколько методов", лежащий в основе полиморфизма. Виртуальная функция в базовом классе определяет *вид интерфейса*, т.е. способ вызова этой функции. Каждое переопределение виртуальной функции в производном классе реализует операции, присущие лишь данному классу. Иначе говоря, переопределение виртуальной функции создает *конкретный метод*.

При обычном вызове виртуальные функции ничем не отличаются от остальных функций-членов. Особые свойства виртуальных функций проявляются при их вызове с помощью указателей. Указатели на объекты базового класса можно использовать для ссылки на объекты производных классов. Если указатель на объект базового класса устанавливается на объект производного класса, содержащий виртуальную функцию, выбор требуемой функции основывается на *типе объекта, на который ссылается указатель*, причем этот выбор осуществляется *в ходе выполнения программы.* Таким образом, если указатель ссылается на объекты разных типов, то будут вызваны разные виртуальные функции. Это относится и к ссылкам на объекты базового класса.

Следующий пример демонстрирует использование виртуальных функций.

```
#include <iostream>

using namespace std;

class base{
public:
     virtual void vfunc()
     {
          cout << "Функция vfunc() из класса base.\n";
     }
};

class derived1 : public base{
public:
```

```
     void vfunc()
     {
          cout << "Функция vfunc() из класса derived1.\n";
     }
};
class derived2 : public base{
public:
     void vfunc()
     {
          cout << "Функция vfunc() из класса derived2.\n";
     };

};

int main()
{
     base *p, b;
     derived1 d1;
     derived2 d2;
// Указатель на объект базового класса.
     p = &b;
     p->vfunc(); // Вызов функции vfunc() из класса base.
// Указатель на объект класса derived1.
     p = &d1;
     p->vfunc(); // Вызов функции vfunc() из класса derived1.
// Указатель на объект класса derived2.
     p = &d2;
     p->vfunc(); // Вызов функции vfunc() из класса derived2.
     return 0;
}
```

Эта программа выводит на экран следующие строки.

```
Функция vfunc() из класса base.
Функция vfunc() из класса derived1.
Функция vfunc() из класса derived2.
```

Как показывает эта программа, внутри класса **base** объявлена виртуальная функция **vfunc()**, поскольку в объявлении функции использовано ключевое слово **virtual**. При переопределении функции **vfunc()** в классах **derived1** и **derived2** ключевое слово **virtual** не требуется. Однако его использование не является ошибкой, просто оно не обязательно.

В данной программе классы **derived1** и **derived2** являются производными от класса **base**. Внутри каждого из этих классов функция **vfunc()** переопределяется заново в соответствии с новым предназначением. В программе **main()** объявлены четыре переменные: **p** – указатель на базовый класс, **b** – объект базового класса, **d1** – объект класса **derived1**, **d2** – объект класса **derived1**.

Кроме того, указателю **p** присваивается адрес объекта **b,** а функция **vfunc()** вызывается с помощью указателя **p**. Поскольку указатель **p** ссылается на объект

класса **base**, выполняется вариант функции **vfunc()** из базового класса. Затем указателю **p** присваивается адрес объекта **d1**, и функция **vfunc()** снова вызывается с его помощью. На этот раз указатель **p** ссылается на объект класса **derived1**. Следовательно, вызывается функция **derived1::vfunc()**. В результате указателю **p** присваивается адрес объекта **d2**, поэтому выражение **p->vfunc()** приводит к вызову функции **vfunc()** из класса **derived2**. Важно, что вариант вызываемой функции определяется типом объекта, на который ссылается указатель **p**. Кроме того, выбор происходит в ходе выполнения программы, что обеспечивает основу динамического полиморфизма.

Виртуальную функцию можно вызывать обычным способом, используя имя объекта и оператор, однако полиморфизм достигается только при обращении к ней через указатель. Например, следующий фрагмент программы является совершенно правильным.

```
d2.vfunc(); // Вызывается функция vfunc()
               // из класса derived2.
```

Несмотря на то, что такой вызов виртуальной функции ошибкой не является, никаких преимуществ он не предоставляет.

На первый взгляд, переопределение виртуальной функции в производном классе мало отличается от обычной перегрузки функций. Однако это не так, и термин *перегрузка* неприменим к переопределению виртуальных функций по нескольким причинам. Наиболее важное отличие заключается в том, что прототип переопределяемой виртуальной функции должен точно совпадать с прототипом, определенным в базовом классе. Этим виртуальные функции отличаются от перегруженных, которые отличаются типами и количеством параметров. Фактически при перегрузке функций типы и количество их параметров *должны* отличаться. Именно эти отличия позволяют компилятору выбирать правильный вариант перегруженной функции. При переопределении виртуальной функции все аспекты их прототипов должны быть одинаковыми. Если не соблюдать это правило, компилятор будет считать эти функции просто перегруженными, а их виртуальная природа будет потеряна. Второе важное ограничение заключается в том, что виртуальные функции не могут быть статическими членами классов. Кроме того, они не могут быть дружественными функциями. И, наконец, конструкторы не могут быть виртуальными, хотя на деструкторы это ограничение не распространяется.

Из-за перечисленных ограничений *для* переопределения виртуальной функции в производном классе используется термин *замещение*.

При наследовании виртуальной функции ее виртуальная природа также наследуется. Это значит, что если производный класс, унаследовавший виртуальную функцию от базового класса, становится базовым по отношению к другому производному классу, виртуальная функция может по-прежнему замещаться. Иначе говоря, не имеет значения, сколько раз наследовалась виртуальная функция, она все равно остается виртуальной.

Функцию, которая объявлена виртуальной в базовом классе, можно заместить в производном классе. Если виртуальная функция не замещается,

вызывается ее предыдущая переопределенная версия. Например, в следующей программе класс **derived2** является наследником класса **derived1**, который, в свою очередь, является производным от класса **base**. Однако функция **vfunc()** в классе **derived2** не замещается. Следовательно, ближайшая к классу **derived2** версия функции **vfunc()** определена в классе **derived1**. Таким образом, вызов функции **vfunc()** с помощью объекта класса **derived2** относится к функции **derivedl::vfunc()**.

```
#include <iostream>

using namespace std;

class base{
public:
   virtual void vfunc()
   {
      cout << "Функция vfunc() из класса base.\n";
   }
}

class derived1 : public base{
public:
   void vfunc(){
      cout << " Функция vfunc() из класса derived1.\n";
   }
};

class derived2 : public derived1{
public:
/* Функция vfunc() не замещается в классе derived2.
Поскольку класс derived2 является наследником класса
derived1, вызывается функция vfunc() из класса derived1.
*/
};

int main()
{
     base *p, b;
     derived1 d1;
     derived2 d2;
// Указатель на объект класса base
     p = &b;
     p->vfunc(); // Вызов функции vfunc() из класса base.
// Указатель на объект класса derived1.
     p = &d1;
     p->vfunc(); // Функция vfunc() из класса derived1.
// Указатель на объект класса derived2
     p = &d2;
     p->vfunc(); // Функция vfunc() из класса derived1.
     return 0;
}
```

Программа выводит на экран следующие сообщения.

```
Функция vfunc() из класса base.
Функция vfunc() из класса derived1.
Функция vfunc() из класса derived1.
```

Если виртуальная функция не замещается в производном классе, вызывается ее версия из базового класса. Однако во многих случаях невозможно создать разумную версию виртуальной функции в базовом классе. Например, базовый класс может не обладать достаточным объемом информации для создания виртуальной функции. Кроме того, в некоторых ситуациях необходимо гарантировать, что виртуальная функция будет замещена во всех производных классах. Для этих ситуаций в языке C++ предусмотрены чисто виртуальные функции.

Чисто виртуальная функция – это виртуальная функция, не имеющая определения в базовом классе. Для объявления чисто виртуальной функции используется следующая синтаксическая конструкция.

`virtual` *тип имя_функции* (*список_параметров*) = 0;

Чисто виртуальные функции должны переопределяться в каждом производном классе, в противном случае возникнет ошибка компиляции.

Следующая программа содержит простой пример чисто виртуальной функции. Базовый класс **`number`** содержит целое число **`val`**, функцию **`setval()`** и чисто виртуальную функцию **`show()`**. Производные классы **`hextype`**, **`dectype`** и **`octtype`** являются наследниками класса **`number`** и переопределяют функцию **`show()`** так, что она выводит значение **`val`** в соответствующей системе счисления (шестнадцатеричной, десятичной или восьмеричной).

```
#include <iostream>

using namespace std;

class number{
protected:
     int val;
public:
     void setval(int i) {val = i;}
// Функция show() является чисто виртуальной.
     virtual void show() = 0;
};

class hextype : public number{
public:
     void show()
     {
          cout << hex << val << "\n";
     }
};
```

```
class dectype : public number{
public:
     void show()
     {
          cout << val << "\n";
     }
};
class octtype : public number{
public:
     void show()
     {
          cout << oct << val << "\n";
     }
};
int main()
{
     dectype d;
     hextype h;
     octtype o;
     d.setval(20);
     d.show(); // Выводит десятичное число 20.
     h.setval(20);
     h.show(); // Выводит шестнадцатеричное число 14.
     o.setval(20);
     o.show(); // Выводит восьмеричное число 24.
     return 0;
}
```

Этот пример иллюстрирует ситуацию, когда в базовом классе невозможно дать осмысленное определение виртуальной функции. В данном случае класс **number** просто обеспечивает единообразный интерфейс для использования производных типов. Функцию **show**() невозможно определить в классе **number**, поскольку в нем не задана основа системы счисления. Однако чисто виртуальная функция **show()** гарантирует, что в каждом производном классе она будет соответствующим образом переопределена.

Все производные классы обязаны переопределять чисто виртуальную функцию. Если этого не сделать, возникнет ошибка компиляции.

Класс, содержащий хотя бы одну чисто виртуальную функцию, называется *абстрактным*. Поскольку абстрактный класс содержит одну или несколько виртуальных функций, его объекты создать невозможно. Следовательно, абстрактные классы можно использовать лишь как основу для производных классов.

Несмотря на то что объекты абстрактного класса не существуют, можно создать указатели и ссылки на абстрактный класс. Это позволяет применять абстрактные классы для поддержки динамического полиморфизма и выбирать соответствующую виртуальную функцию в зависимости от типа указателя или ссылки.

Виртуальные функции, абстрактные классы и динамический полиморфизм представляют собой один из наиболее мощных и гибких механизмов реализации принципа "один интерфейс, несколько методов". Используя этот механизм, можно создавать иерархии классов, организованные по принципу "от общего – к частному" (от базового класса – к производным). По этому методу в базовом классе следует определять все универсальные свойства и интерфейсы. Если некоторую операцию можно реализовать только в производном классе, используется виртуальная функция. По существу, в базовом классе описываются лишь самые общие свойства, а в производных классах они конкретизируются.

Абстрактные классы и виртуальные функции позволяют создавать *библиотеки классов*, носящие обобщенный характер. Любой программист может создать класс, производный от библиотечного, добавив свои собственные функции. При этом будет сохранен единообразный интерфейс, определенный базовым классом Таким образом, библиотечные классы можно адаптировать к новым ситуациям.

Обработка исключительных ситуаций

Исключение – это аномальное поведение во время выполнения, которое программа может обнаружить, например: недостаток свободной памяти. Такие исключительные ситуации нарушают нормальный ход работы программы, и на них нужно немедленно отреагировать.

Обработка исключительных ситуаций – позволяет правильно реагировать па ошибки, возникающие в ходе выполнения программы. Используя механизм исключительных ситуаций, программа может автоматически вызывать процедуру обработки ошибок. Это избавляет программиста от утомительного кодирования.

Механизм обработки исключительных ситуаций в языке C++ основан на трёх ключевых словах: **try, catch** и **throw.** Фрагменты программы, подлежащие контролю, содержат блок **try.** Если в ходе выполнения программы в блоке **try** возникает исключительная ситуация (т.е. ошибка), она генерируется с помощью ключевого слова **throw.** Затем исключительная ситуация перехватывается блоком **catch** и обрабатывается. Следует уточнить это описание.

Код, подлежащий контролю, должен выполняться внутри блока **try.** Функции, вызываемые внутри блока **try,** также могут генерировать исключительные ситуации. Исключительные ситуации перехватываются оператором **catch,** который следует непосредственно за блоком **try,** в котором они возникли. Общий вид операторов **try** и **catch** показан ниже.

```
try {
     // Тело блока try
} catch (тип 1 аргумент) {
     // Тело блока catch
}
catch (тип2 аргумент) {
     // Тело блока catch
}
catch (тип3 аргумент) {
     // Тело блока catch
}
.
.
.
catch (типN аргумент) {
     // Тело блока catch
}
```

Размер блока **try** может варьироваться. Он может содержать как несколько операторов, так и целую программу, в этом случае функция **main**() целиком помещается в блок **try.**

Возникшая исключительная ситуация перехватывается соответствующим оператором **catch,** который выполняет её обработку. С одним блоком **try** может быть связано несколько операторов **catch.** Выбор правильного оператора **catch** определяется типом исключительной ситуации. Из нескольких вариантов выбирается оператор **catch,** тип аргумента которого совпадает с возникшей

исключительной ситуацией (остальные варианты игнорируются). В процессе перехвата исключительной ситуации *аргументу* присваивается некое значение. Аргумент может быть объектом встроенного типа. Если фрагмент не генерирует никаких исключительных ситуаций (т.е. в блоке **try** ошибки не возникают), не выполняется ни один оператор **catch.**

Оператор **throw** имеет следующий вид.

`throw` *исключительная_ситуация;*

Оператор **throw** генерирует указанную исключительную ситуацию. Если в программе предусмотрен её перехват, оператор **throw** должен выполняться либо внутри блока **try**, либо внутри функции, явно или неявно вызываемой внутри блока **try**.

Если генерируется исключительная ситуация, для которой не предусмотрена обработка, программа может прекратить своё выполнение. В этом случае вызывается стандартная функция **terminate()**, которая по умолчанию вызывает функцию **abort**(). Однако, программист может предусмотреть собственную обработку ошибки.

Ниже представлен пример, демонстрирующий обработку исключительной ситуации.

```
// Простой пример обработки исключительной ситуации.
#include <iostream>

using namespace std;

int main() {
   cout << "Начало\n";
   try { // Начало блока try
      cout << "Внутри блока try\n";
      throw 100; // Генерация ошибки.
      cout << "Этот оператор не выполняется.";
   } catch (int i) { // Перехват ошибки.
      cout << "перехват исключительной ситуации ";
      cout << " - значение равно:" << i << "\n";
   }
   cout << "Конец\n";
   return 0;
}
```

Эта программа выводит на экран следующие строки.

```
Начало
Внутри блока try
Перехват исключительной ситуации - значение равно: 100
Конец
```

Блок **try** содержит три оператора. С ним связан оператор **catch** (**int i**), выполняющий обработку целочисленной исключительной ситуации. Внутри блока **try** выполняются только два из трёх операторов: первый оператор **cout** и оператор **throw**. При генерации исключительной ситуации управление передаётся оператору **catch**, а выполнение блока **try** прекращается. Иначе говоря, блок

catch *не вызывается.* Просто программа переходит к его выполнению. Для этого стек программы автоматически обновляется. Таким образом, оператор **cout**, следующий за оператором **throw**, никогда не выполняется.

Обычно оператор **catch** пытается исправить ошибку, предпринимая соответствующие действия. Если это возможно, выполнение программы возобновляется с оператора, следующего за блоком **catch**. Однако часто ошибку исправить невозможно, и блок **catch** прекращает выполнение программы, вызывая функцию **exit()** или **abort()**.

Тип исключительной ситуации должен совпадать с типом, указанным в операторе **catch**. Например, если в предыдущей программе изменить тип аргумента оператора **catch** на **double**, перехват исключительной ситуации не состоится, и программа завершится аварийно. Следующая программа иллюстрирует эту ситуацию.

```
// Эта программа не работает.
#include <iostream>

using namespace std;

int main() {
   cout << "Начало\n";
   try { // Начало блока try.
      cout << "Внутри блока try\n";
      throw 100; // Генерация ошибки.
      cout << "Этот оператор не выполняется";
   } catch (double i) { // Не перехватывает целочисленные
                        // исключительные ситуации.
      cout << "Перехват исключительной ситуации ";
      cout << " - значение равно: " << i << "\n";
   }
   cout << "Конец\n";
   return 0;
}
```

В этой программе оператор **catch (double i)** не перехватывает целочисленные исключительные ситуации. На экран выводятся следующие сообщения.

```
Начало
Внутри блока try
Abnormal program termination
```

Исключение может генерироваться вне блока **try** только в том случае, если оно генерируется функцией, которая вызывается внутри этого блока. Пример, который представлен ниже, иллюстрирует эту ситуацию.

```
#include <iostream>

using namespace std;
```

```
void Xtest(int test) {
   cout << "Внутри функции Xtest, test =:" << test << "\n";
   if(test) throw test;
}

int main() {
   cout << "Начало\n";
   try { // Начало блока try.
      cout << "Внутри блока try\n";
      Xtest(0);
      Xtest(1);
      Xtest(2);
   } catch (int i) { // Перехват ошибки.
      cout << "Перехват исключительной ситуации ";
      cout << " - значение равно: " << i << "\n";
   }
   cout << "Конец\n";
   return 0;
}
```

Эта программа выводит на экран следующие строки.

```
Начало
Внутри блока try
Внутри функции Xtest, test = 0
Внутри функции Xtest, test = 1
Перехват исключительной ситуации - значение равно: 1
Конец
```

Блок **`try`** может находиться внутри функции. В этом случае при каждом входе в функцию обработка исключительной ситуации выполняется заново. В качестве примера можно проанализировать следующую программу.

```
#include <iostream>

using namespace std;

// Блоки try/catch находятся внутри функции.
void Xhandler(int test) {
   try {
      if(test) throw test; } catch(int i) {
      cout << "Перехват исключительной ситуации: " << i;
      cout << endl;
   }
}

int main() {
   cout << "Начало\n";
   Xhandler(1);
   Xhandler(2);
   Xhandler(0);
   Xhandler(3);
   cout << "Конец\n";
```

```
   return 0;
}
```

Эта программа выводит на экран следующие сообщения.

```
Начало
Перехват исключительной ситуации : 1
Перехват исключительной ситуации : 2
Перехват исключительной ситуации : 3
Конец
```

Как видно, в ходе выполнения программы были перехвачены три исключительные ситуации. После каждой обработки функция возвращает управление вызывающему модулю. При повторном вызове функции обработка исключительной ситуации выполняется вновь.

Код, связанный с оператором **`catch`**, выполняется только при перехвате исключительной ситуации. В противном случае оператор **`catch`** просто игнорируется. Например, в следующей программе исключительные ситуации вообще не генерируются, и оператор **`catch`** не выполняется

```
#include <iostream>

using namespace std;

int main() {
   cout << "Начало\n";
   try { // Начало блока try.
      cout << "Внутри блока try\n";
      cout << "Все еще внутри блока try\n";
   } catch (int i) { // Перехват ошибки
      cout << "Перехват исключительной ситуации ";
      cout << " - значение равно: " << i << "\n";
   }
   cout << "Конец\n";
   return 0;
}
```

Эта программа выводит на экран следующие сообщения.

```
Начало
Внутри блока try
Все еще внутри блока try
Конец
```

Как видно, поток управления обошёл оператор **`catch`** стороной.

Система обработки исключительных ситуаций позволяет реагировать на необычные события, возникающие в ходе выполнения программы. Следовательно, обработчики исключительных ситуаций должны выполнять некие разумные действия, позволяющие исправить ошибку или смягчить её последствия. Следующая программа иллюстрирует использование обработки исключительных ситуаций для предотвращения деления на нуль. Она вводит два числа и делит первое из них на второе.

```
#include <iostream>
using namespace std;
void divide(double a, double b) {
   try {
      if(!b) throw b; // Проверка деления на нуль
      cout << "Результат: " << a/b << endl;
   } catch (double b) {
      cout << "Делить на нуль нельзя.\n";
   }
}
int main() {
   double i, j ;
   do {
      cout << "Введите числитель (0 означает выход): ";
      cin >> i;
      cout << "Введите знаменатель: ";
      cin >> j;
      divide(i, j);
   } while(i != 0);
   return 0;
}
```

Этот пример иллюстрирует принцип обработки исключительных ситуаций. Если знаменатель равен нулю, возникает исключительная ситуация. Её обработчик не предусматривает деления (это привело бы к аварийному завершению работы программы), а просто сообщает пользователю о возникшей ошибке. Таким образом, деления на нуль можно избежать, продолжив выполнение программы. Эта схема работает и в более сложных случаях.

Обработка исключительной ситуации особенно полезна при выходе из глубоко вложенных процедур, в которых возникла катастрофическая ошибка.

Понятие о системе программирования

Система программирования представляет собой программы или комплексы программ предназначенные для автоматизации разработки новых программ. Система программирования представлена такими компонентами, как транслятор с соответствующего языка, библиотеки подпрограмм, редакторы, компоновщики и отладчики.

Трансляторы предназначены для преобразования программ, написанных на языках программирования, в программы на машинном языке. Программа, подготовленная на каком-либо языке программирования, называется исходным модулем. В качестве входной информации трансляторы применяют исходные модули и формируют в результате своей работы объектные модули, являющиеся входной информацией для редактора связей. Объектный модуль содержит текст программы на машинном языке и дополнительную информацию, обеспечивающую настройку модуля по месту его загрузки и объединение этого модуля с другими независимо оттранслированными модулями в единую программу.

Библиотеки подпрограмм содержат в себе наиболее часто используемые подпрограммы в виде готовых объектных модулей.

С помощью *редактора* выполняется ввод и модификация текста разрабатываемой программы.

Компоновщик представляет собой системную обрабатывающую программу, редактирующую и объединяющую объектные модули в единые загрузочные, готовые к выполнению программные модули. Загрузочный модуль может быть помещен ОС в основную память и выполнен.

Отладчик позволяет управлять процессом исполнения программы, является инструментом для поиска и исправления ошибок в программе. Базовый набор функций отладчика включает:

- пошаговое выполнение программы (режим трассировки) с отображением результатов,
- остановка в заранее определенных точках,
- возможность остановки в некотором месте программы при выполнении некоторого условия;
- изображение и изменение значений переменных.

Программы, написанные на языках программирования высокого уровня, перед выполнением на ЭВМ должны транслироваться в *эквивалентные программы*, написанные на машинном коде. *Транслятор* – это *программа*, которая переводит программу на исходном (входном) языке в эквивалентную ей программу на результирующем (выходном) языке. Если исходный язык является языком высокого уровня, а транслятор заменяет каждый оператор такого языка эквивалентным набором команд на машинном языке, причём компьютер выполняет новую программу на машинном языке, вместо старой, которая написана на языке высокого уровня, то такой *транслятор* называется компилятором.

Достоинство компилятора заключается в том, что *программа* компилируется один раз, и при каждом выполнении не требуется дополнительных преобразований. Соответственно, не требуется наличие компилятора на целевой машине, для которой компилируется *программа*. Недостаток: отдельный этап компиляции

замедляет написание и отладку и затрудняет *исполнение* небольших, несложных или разовых программ. В том случае, если исходный язык является языком ассемблера (низкоуровневым языком, близким к машинному языку), *компилятор* такого языка называется ассемблером.

Дугой метод реализации программ, написанных на языке высокого уровня, – *интерпретация*. *Интерпретатор* программно моделирует машину, цикл выборки-исполнения которой работает с командами на языках высокого уровня, а не с машинными командами. Такое программное *моделирование* создает виртуальную машину, реализующую язык. Этот подход называется чистой интерпретацией. Чистая *интерпретация* применяется, как правило, для языков с простой структурой (например, Basic). *Интерпретаторы командной строки* обрабатывают команды в скриптах в *UNIX* или в пакетных файлах (.bat) в *MS-DOS*, как правило, также в режиме чистой интерпретации.

Достоинство чистого интерпретатора: отсутствие промежуточных действий для трансляции упрощает реализацию интерпретатора и делает его удобнее в использовании, в том числе в диалоговом режиме. Недостаток – *интерпретатор* должен быть в наличии на целевой машине, где должна исполняться *программа*. А свойство чистого интерпретатора, что ошибки в интерпретируемой программе обнаруживаются только при попытке выполнения команды (или строки) с ошибкой, можно признать как недостатком, так и достоинством.

Существуют компромиссные между компиляцией и чистой интерпретацией варианты реализации языков программирования, когда *интерпретатор* перед исполнением программы транслирует ее на *промежуточный язык* (например, в байт-код), более удобный для интерпретации (т.е. речь идет об интерпретаторе со встроенным транслятором). Такой метод называется смешанной реализацией. Примером смешанной реализации языка может служить Java. Этот подход сочетает как достоинства компилятора и интерпретатора (большая скорость исполнения и *удобство использования*), так и недостатки (для трансляции и хранения программы на промежуточном языке требуются дополнительные ресурсы; для исполнения программы на целевой машине должен быть представлен *интерпретатор*). Так же, как и в случае компилятора, смешанная реализация требует, чтобы перед исполнением исходный код не содержал ошибок (лексических, синтаксических и семантических).

Для компиляции *компилятор* должен выполнить *анализ* исходной программы, а затем синтез объектной программы. Сначала исходная *программа* разбивается на её составные части; затем из них строятся части эквивалентной объектной программы. Для этого на этапе анализа *компилятор* строит несколько таблиц (рисунок 1), которые используются затем как при анализе, так и при синтезе.

При анализе программы из *описаний, заголовков* процедур, заголовков циклов и т.д. извлекается *информация* и сохраняется для последующего применения. Эта *информация* обнаруживается в отдельных точках программы и организуется так, чтобы к ней можно было обратиться из любой части компилятора. Например, при каждом использовании идентификатора необходимо знать, как был описан этот *идентификатор* и как он работал в других местах

программы. В каждом компиляторе в той или иной форме используется *таблица символов* (иногда ее называют списком идентификаторов или таблицей имен). Это *таблица* идентификаторов, встречающихся в исходной программе, вместе с их атрибутами. К атрибутам относятся тип идентификатора, его *адрес* в объектной программе и любая другая *информация* о нем, которая может понадобиться при генерации объектной программы.

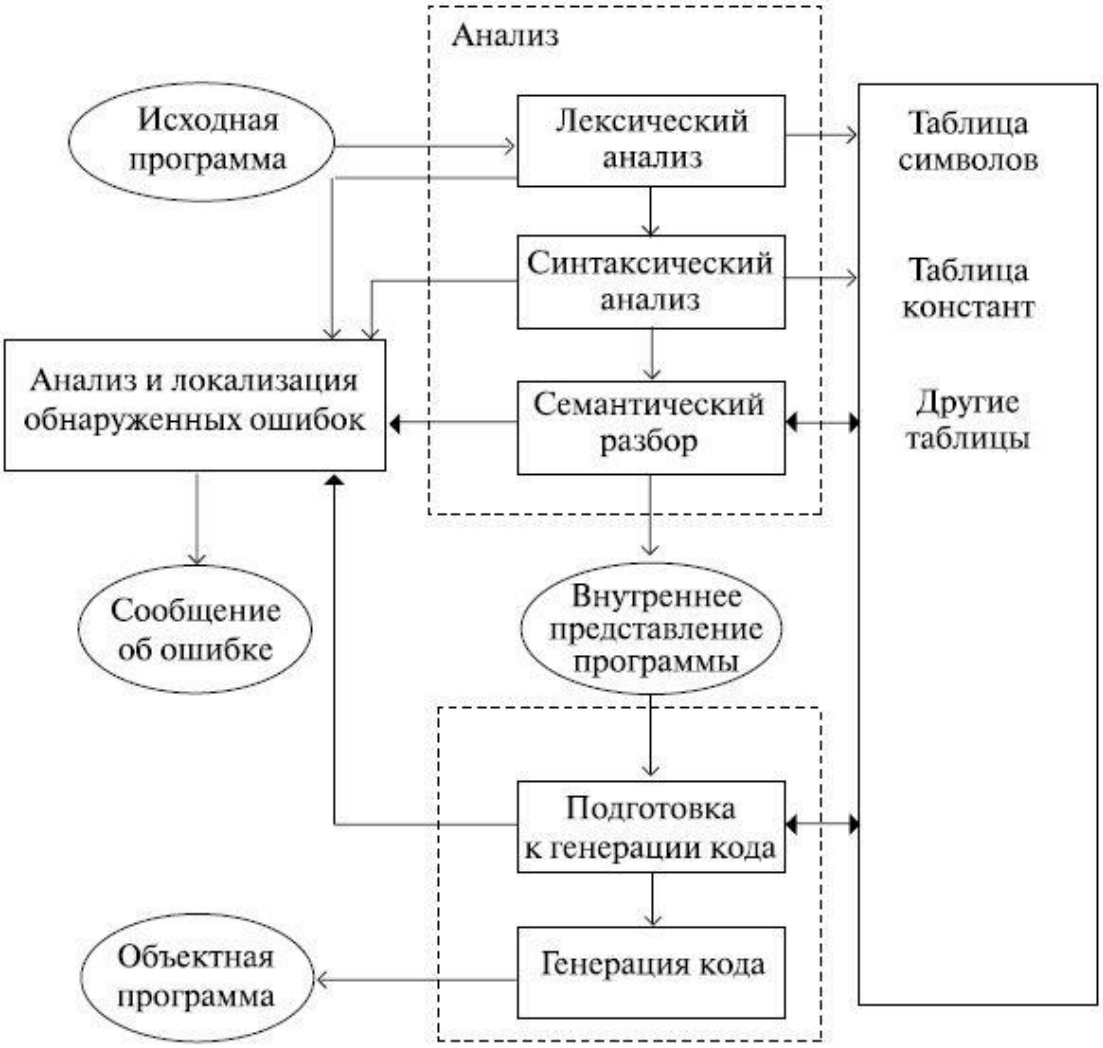

Рисунок 1 – Структура процесса компиляции

Лексический анализатор (*сканер*) – просматривает *литеры* исходной программы слева направо и строит символы программы – целые числа, идентификаторы, служебные слова и т. д. (символы передаются затем на обработку фактическому анализатору). *Сканер* также может заносить идентификаторы в таблицу символов и выполнять другую простую работу, которая фактически не требует анализа исходной программы.

Синтаксический и семантический анализаторы выполняют сложную работу *по* разделению исходной программы на составные части, формированию ее внутреннего представления и занесению информации в таблицу символов и другие таблицы.

Внутреннее *представление* исходной программы в значительной степени зависит от его дальнейшего использования. Это может быть *дерево*, отражающее *синтаксис* исходной программы и т.д.

Перед генерацией команд обычно необходимо некоторым образом обработать и изменить внутреннюю программу. Кроме того, должна быть распределена *память* под переменные готовой программы. Одним из важных моментов на этом этапе является *оптимизация* программы с целью уменьшения времени ее работы. По существу, на этом этапе происходит перевод внутреннего представления исходной программы на *автокод* или на *машинный язык*. В интерпретаторе эта часть компилятора заменяется программой, которая фактически выполняет внутреннее *представление* исходной программы. Само внутреннее *представление* в этом случае мало чем отличается от того, которое получается при компиляции.

Первоначально компиляторы представляли собой обособленные программные модули, решающие исключительно задачу перевода исходного текста программы на входном языке в язык машинных кодов. Компиляция выполнялась с помощью последовательности команд, инициировавших запуск соответствующих программных модулей с передачей им всех необходимых параметров. Параметры передавались каждому модулю в командной строке и представляли собой набор имен файлов и настроек, реализованных в виде специальных ключей. Со временем разработчики компиляторов объединили всё необходимое множество программных модулей в составе одной поставки компилятора. Кроме того, были унифицированы форматы объектных файлов и файлов библиотек подпрограмм. Теперь разработчики, имея компилятор от одного производителя, могли, в принципе, пользоваться библиотеками и объектными файлами, полученными от другого производителя компиляторов.

Поскольку процессу компиляции всегда соответствует типичная последовательность команд, был предложен специальный командный язык, который обрабатывался интерпретатором команд компиляции – программой make. Командный язык Makefile позволял в достаточно гибкой и удобной форме описать весь процесс создания программы от порождения исходных текстов до подготовки её к выполнению. Язык Makefile стал стандартным средством, единым для компиляторов всех разработчиков. Это было удобное, но достаточно сложное техническое средство, требующее от разработчика высокой степени подготовки и профессиональных знаний, поскольку сам командный язык Makefile был по сложности сравним с простым языком программирования.

Следующим шагом в развитии средств разработки стало появление интегрированной среды разработки (IDE). Интегрированная среда объединила в себе возможности текстовых редакторов исходных текстов программ и командный язык компиляции. Разработчик исходной программы теперь не должен был выполнять всю последовательность действий от порождения исходного кода до его выполнения, от него также не требовалось описывать этот процесс с помощью системы команд в Makefile. Теперь ему было достаточно только указать в удобной интерфейсной форме состав необходимых для создания программы исходных

модулей и библиотек. Ключи, необходимые компилятору и другим техническим средствам, также задавались в виде интерфейсных форм настройки.

Создание интегрированных сред разработки стало возможным благодаря бурному развитию персональных компьютеров и появлению развитых средств интерфейса пользователя (сначала текстовых, а потом и графических). Развитие интегрированных сред несколько снизило требования к профессиональным навыкам разработчиков исходных программ. Теперь в простейшем случае от разработчика требовалось только знание исходного языка (его синтаксиса и семантики).

Дальнейшее развитие средств разработки также тесно связано с повсеместным распространением развитых средств графического интерфейса пользователя. В их состав были сначала включены соответствующие библиотеки, обеспечивающие поддержку развитого графического интерфейса пользователя и взаимодействие с функциями API операционных систем. А затем для работы с ними потребовались дополнительные средства, обеспечивающие разработку внешнего вида интерфейсных модулей (системы визуального программирования). Такая работа была уже более характерна для дизайнера, чем для программиста.

Для описания графических элементов программ потребовались соответствующие языки. На их основе сложилось понятие ресурсов (resources) прикладных программ. Ресурсами прикладной программы называют множество данных, обеспечивающих внешний вид интерфейса пользователя этой программы, и не связанных напрямую с логикой выполнения программы. Характерными примерами ресурсов являются тексты сообщений, выдаваемых программой; цветовая гамма элементов интерфейса; надписи на таких элементах, как кнопки и заголовки окон и т.п.

Обычно среда разработки включает в себя текстовый редактор, компилятор и/или интерпретатор, средства автоматизации сборки и отладчик. Иногда она также содержит средства для интеграции с системами управления версиями и разнообразные инструменты для упрощения конструирования графического интерфейса пользователя. Многие современные среды разработки также включают браузер классов, инспектор объектов и диаграмму иерархии классов для использования при объектно-ориентированной разработке ПО. Хотя и существуют среды разработки, предназначенные для нескольких языков программирования, такие как Eclipse, NetBeans, или Microsoft Visual Studio, но есть среды разработки предназначается для одного определенного языка программирования – как, например, Delphi, Qt Creator, Spyder.