**1. Понятие модуля. Принципы модульного программирования. Понятие объекта как динамического модуля.**

Модуль в программировании представляет собой функционально законченный фрагмент программы, оформленный в виде отдельного файла с исходным кодом, предназначенный для использования в других программах. Модули позволяют разбивать сложные задачи на более мелкие в соответствии с принципом модульности.

Модульное программирование – это организация программы как совокупности небольших независимых блоков, называемых модулями, структура и поведение которых подчиняются определенным правилам.

Использование модульного программирования позволяет упростить тестирование программы и обнаружение ошибок. Аппаратно-зависимые подзадачи могут быть строго отделены от других подзадач, что улучшает мобильность создаваемых программ.

Основные концепции модульного программирования:

-каждый модуль имеет единственную точку входа и выхода;

-размер модуля по возможности должен быть минимизирован;

-вся система построена из модулей;

-каждый модуль не зависит от того, как реализованы другие модули.

Классы очень удобно собирать в модули. При этом их описание помещается в секцию

interface, а код методов — в секцию implementation. Создавая модули классов, нужно

придерживаться следующих правил:

-все классы, предназначенные для использования за пределами модуля, следует

определять в секции interface;

- описание классов, предназначенных для употребления внутри модуля, следует

располагать в секции implementation;

- если модуль B использует модуль A, то в модуле B можно определять классы,

порожденные от классов модуля A.

**2. Понятие класса. Понятие метода. Представление метода в виде обычной процедуры. Понятие конструктора и деструктора.**

Каждый объект всегда принадлежит некоторому классу объектов. Класс объектов — это

обобщенное (абстрактное) описание множества однотипных объектов. Объекты являются

конкретными представителями своего класса, их принято называть экземплярами класса.

Например, класс СОБАКИ — понятие абстрактное, а экземпляр этого класса МОЙ ПЕС

БОБИК — понятие конкретное.

Для поддержки ООП в язык Delphi введены объектные типы данных, с помощью которых

одновременно описываются данные и операции над ними. Объектные типы данных

называют классами, а их экземпляры — объектами. Классы объектов определяются в секции type глобального блока. Описание класса начинается с ключевого слова class и заканчивается ключевым словом end. По форме объявления классы похожи на обычные записи, но помимо полей данных могут содержать объявления пользовательских процедур и функций. Процедуры и функции, предназначенные для выполнения над объектами действий, называются методами. Предварительное объявление методов выполняется при описании класса в секции interface модуля, а их программный код записывается в секции

implementation. Однако в отличие от обычных процедур и функций заголовки методов

должны иметь уточненные имена, т.е. содержать наименование класса.

Особой разновидностью методов являются конструкторы и деструкторы. Напомним, что

конструкторы создают, а деструкторы разрушают объекты. Создание объекта включает

выделение памяти под экземпляр и инициализацию его полей, а разрушение — очистку

полей и освобождение памяти. Действия по инициализации и очистке полей специфичны для каждого конкретного класса объектов. По этой причине язык Delphi позволяет

переопределить стандартный конструктор Create и стандартный деструктор Destroyдля

выполнения любых полезных действий. Можно даже определить несколько конструкторов и деструкторов (имена им назначает сам программист), чтобы обеспечить различные процедуры создания и разрушения объектов. Объявление конструкторов и деструкторов похоже на объявление обычных методов с той лишь разницей, что вместо зарезервированных слов function и procedure используются слова constructor и destructor.

**3. Понятие свойства. Методы получения и установки значений свойств. Свойства-массивы (в некоторых языках программирования). Индексаторы (в некоторых языках программирования).**

Помимо полей и методов в объектах существуют свойства. При работе с объектом свойствавыглядят как поля: они принимают значения и участвуют в выражениях. Но в отличие отполей свойства не занимают места в памяти, а операции их чтения и записи ассоциируются собычными полями или методами. Это позволяет создавать необходимые сопутствующиеэффекты при обращении к свойствам. Например, присваивание свойствуActive значения True вызовет открытие файла, а присваивание значения False — закрытиефайла. Создание сопутствующего эффекта (открытие или закрытие файла) достигается тем,что за присваиванием свойству значения стоит вызов метода.Объявление свойства выполняется с помощью зарезервированного слова property,например:

type

TDelimitedReader = class

...

FActive: Boolean;

...

// Методзаписи (установкизначения) свойства

procedureSetActive(constAActive: Boolean);

propertyActive: Boolean read FActivewrite SetActive; // Свойство

end;

Ключевые слова read и write называются спецификаторами доступа. После слова read

указывается поле или метод, к которому происходит обращение при чтении (получении)

значения свойства, а после слова write — поле или метод, к которому происходит обращениепри записи (установке) значения свойства. Например, чтение свойства Active означаетчтение поля FActive, а установка свойства — вызов метода SetActive.

Свойство-массив — это индексированное множество значений. Например, в классе

TDelimitedReader множество элементов, выделенных из считанной строки, удобно

представить в виде свойства-массива:

type

TDelimitedReader = class

...

FItems: array of string;

...

functionGetItem(Index: Integer): string;

...

propertyItems[Index: Integer]: string read GetItem;

end;

functionTDelimitedReader.GetItem(Index: Integer): string;

begin

Result :=FItems[Index];

end;

Элементымассива Items можнотолькочитать, посколькуклассTDelimitedReader

предназначен только для чтения данных из файла.

В описании свойства-массива разрешено использовать только методы, но не поля. В этом

состоит отличие свойства-массива от обычного свойства.

Основная выгода от применения свойства-массива — возможность выполнения итераций с помощьюциклаfor, например:

var

Reader: TDelimitedReader;

I: Integer;

...

forI := 0 to Reader.ItemCount - 1 do

Writeln(Reader.Items[I]);

...

Свойство-массив может быть многомерным. В этом случае методы чтения и записи

элементов должны иметь столько же индексных параметров соответствующих типов, что и свойство-массив. Свойства-массивы имеют два важных отличия от обычных массивов:

- их индексы не ограничиваются диапазоном и могут иметь любой тип данных, а не

толькоInteger. Например, можно создать свойство-массив, в котором индексами

будут строки. Обращение к такому свойству могло бы выглядеть примерно так:

Reader.Items['FirstName']:= 'Alexander';

-операции целиком со всем свойством-массивом запрещены; разрешены операции

только с его элементами.

Свойства иногда называются "умными полями", а индексаторы — "умными массивами", а значит, стоит использовать для них один синтаксис. Действительно, определение индексатора во многом напоминает определение свойств, кроме двух крупных отличий. Во-первых, индексатор принимает аргумент индекс. Во-вторых, поскольку сам класс применяется как массив, в качестве имени индексатора используется ключевое слово this. Взгляните на такой пример индексатора:

classMyClass

{

public object this [intidx]

{

get

{

// Возврат нужных данных.

} set

{

// Установка нужных данных.

} }

}

Это лишь часть примера, иллюстрирующего синтаксис индексаторов, так как внутренняя реализация способа определения данных, их получения и установки к индексаторам не относится. Имейте в виду, что независимо от внутреннего способа хранения ваших данных (т. е. в виде массива, набора и т. д.) индексаторы — всего лишь средства, позволяющие программисту создавать экземпляр класса для написания, например, такого кода:

MyClassels = new MyClassO;

cls[0] = someObject;

Console.WriteLine("{0}", cls[0]);

Что именно вы делаете в пределах индексатора — ваше личное дело, пока клиент класса получает при обращении к объекту как к массиву ожидаемые результаты.

**4. Расширение класса путем создания производного класса. Термин «наследование». Существование «прародителя» всех классов. Перекрытие элементов класса в производных классах. Совместимость объектов различных классов. Контроль и преобразование типов. Информация о типе времени выполнения программы.**

Наследованием называется возможность порождать один класс от другого с сохранением всех свойств и методов класса-предка (прародителя, иногда его называют суперклассом) и добавляя, при необходимости, новые свойства и методы. Набор классов, связанных отношением наследования, называют иерархией. Наследование призвано отобразить такое свойство реального мира, как иерархичность. Класс, который наследует атрибуты другого класса, называется порожденным классом или потомком. Соответственно класс, от которого происходит наследование, выступает в роли базового, или предка. Очень важно, что в отношениях наследования любой класс может иметь только одного непосредственного предка и сколь угодно много потомков. Поэтому все связанные отношением наследования классы образуют иерархию. Примером иерархии классов является библиотека VCL; с ее помощью в среде Delphi обеспечивается разработка GUI-приложений.

В языке Delphi существует предопределенный класс TObject, который служит неявным предком тех классов, для которых предок не указан. Это означает, что объявление

type

TTextReader = class

...

end;

эквивалентно следующему:

type

TTextReader = class(TObject)

...

end;

Класс TObject выступает корнем любой иерархии классов. Он содержит ряд методов,

которые по наследству передаются всем остальным классам. Среди них конструктор Create, деструктор Destroy, метод Free и некоторые другие методы.

 Свойство базового класса можно перекрыть (от англ. override) в производном классе,

например чтобы добавить ему новый атрибут доступа или связать с другим полем или

методом.В наследнике можно вызвать перекрытый метод предка, указав перед

именем метода зарезервированное слово inherited. Когда метод предка полностью совпадает с методом потомка по формату заголовка, то можно использовать более короткую запись.

 Для классов, связанных отношением наследования, вводится новое правило совместимости типов. Вместо объекта базового класса можно подставить объект любого производного класса. Обратное неверно. Например, переменной типа TTextReader можно присвоить значение переменной типа TDelimitedReader:

var

Reader: TTextReader;

...

Reader :=TDelimitedReader.Create('MyData.del', ';');

Объектная переменная Reader формально имеет тип TTextReader, а фактически связана с

экземпляром класса TDelimitedReader.

Правило совместимости классов чаще всего применяется при передаче объектов в

параметрах процедур и функций. Например, если процедура работает с объектом класса

TTextReader, то вместо него можно передать объект класса TDelimitedReader или

TFixedReader. Заметим, что все объекты являются представителями известного вам класса TObject. Поэтому любой объект любого класса можно использовать как объект класса TObject.

 В языке Delphi существуют операторы is и as, с помощью которых выполняется соответственно проверка на тип (typechecking) и преобразование к типу (typecasting).

Например, чтобы выяснить, принадлежит ли некоторый объект Obj к классу TTextReader или его наследнику, следует использовать оператор is:

var

Obj: TObject;

...

ifObj is TTextReader then ...

Для преобразования объекта к нужному типу используется оператор as, например

withObj as TTextReader do

Active := False;

Стоит отметить, что для объектов применим и обычный способ приведения типа:

withTTextReader(Obj) do

Active := False;

Вариант с оператором as лучше, поскольку безопасен. Он генерирует ошибку (точнее

исключительную ситуацию) при выполнении программы (run-timeerror), если реальный экземпляр объекта Obj не совместим с классом TTextReader. Ошибку приведения типа можно обработать и таким образом избежать досрочного завершения программы.

**5. Классы в программных модулях. Атрибуты доступа к элементам объектов. Термин «инкапсуляция».**

Классы очень удобно собирать в модули. При этом их описание помещается в секцию

interface, а код методов — в секцию implementation. Создавая модули классов, нужно

придерживаться следующих правил:

- все классы, предназначенные для использования за пределами модуля, следует

определять в секции interface;

- описание классов, предназначенных для употребления внутри модуля, следует

располагать в секции implementation;

- если модуль B использует модуль A, то в модуле B можно определять классы,

порожденные от классов модуля A.

Программист может разграничить доступ к атрибутам своих объектов для других

программистов (и себя самого) с помощью специальных ключевых слов: private, protected,

public, published.

- Private. Все, что объявлено в секции private недоступно за пределами модуля. Секция

private позволяет скрыть те поля и методы, которые относятся к так называемым

особеностям реализации.

-Public. Поля, методы и свойства, объявленные в секции public не имеют никаких

ограничений на использование, т.е. всегда видны за пределами модуля. Все, что

помещается в секцию public, служит для манипуляций с объектами и составляет программный интерфейс класса.

- Protected. Поля, методы и свойства, объявленные в секции protected, видны за

пределами модуля только потомкам данного класса; остальным частям программы

они не видны. Так же как и private, директива protected позволяет скрытьособенности реализации класса, но в отличие от нее, разрешает другимпрограммистам порождать новые классы и обращаться к полям, методам и свойствам,которые составляют так называемый интерфейс разработчика. В эту секцию обычнопомещаются виртуальные методы.

- Published. Устанавливает правила видимости те же, что и директива public.

Особенность состоит в том, что для элементов, помещенных в секцию published,

компилятор генерирует информацию о типах этих элементов. Эта информация

доступна во время выполнения программы, что позволяет превращать объекты в

компоненты визуальной среды разработки. Секцию published разрешено

использовать только тогда, когда для самого класса или его предка включена

директива компилятора $TYPEINFO.

Перечисленные секции могут чередоваться в объявлении класса в произвольном порядке,

однако в пределах секции сначала следует описание полей, а потом методов и свойств. Если в определении класса нет ключевых слов private, protected, public и published, то для

обычных классов всем полям, методам и свойствам приписывается атрибут видимости

public, а для тех классов, которые порождены от классов библиотеки VCL, — атрибут

видимостиpublished. Внутри модуля никакие ограничения на доступ к атрибутам классов, реализованных в этом же модуле, не действуют. Кстати, это отличается от соглашений, принятых в некоторых других языках программирования, в частности в языке C++.

Инкапсуля́ция — свойство языка программирования, позволяющее объединить и защитить данные и код в объектe и скрыть реализацию объекта от пользователя (прикладного программиста). При этом пользователю предоставляется только спецификация (интерфейс) объекта. Пользователь может взаимодействовать с объектом только через этот интерфейс. Сокрытие реализации целесообразно применять в следующих случаях:

-предельная локализация изменений при необходимости таких изменений;

-прогнозируемость изменений (какие изменения в коде надо сделать для заданного изменения функциональности) и прогнозируемость последствий изменений.

**6. Понятие виртуального метода. Перекрытие виртуального метода в производном классе. Абстрактный виртуальный метод. Механизм вызова виртуального метода. Методы обработки сообщений. Термин «полиморфизм».**

 Виртуальный метод (виртуальная функция) — в объектно-ориентированном программировании метод (функция) класса, который может быть переопределён в классах-наследниках так, что конкретная реализация метода для вызова будет определяться во время исполнения. Таким образом, программисту необязательно знать точный тип объекта для работы с ним через виртуальные методы: достаточно лишь знать, что объект принадлежит классу или наследнику класса, в котором метод объявлен.

Объявление виртуального метода в базовом классе выполняется с помощью ключевого слова virtual, а его перекрытие в производных классах — с помощью ключевого слова override. Перекрытый метод должен иметь точно такой же формат (список параметров, а для функций еще и тип возвращаемого значения), что и перекрываемый.

 При построении иерархии классов часто возникает ситуация, когда работа виртуального

метода в базовом классе не известна и наполняется содержанием только в наследниках.

Конечно, тело метода всегда можно сделать пустым или почти пустым, но лучше воспользоваться директивой abstract. Директива abstract записывается после слова virtual и исключает необходимость написания кода виртуального метода для данного класса. Такой метод называется абстрактным, т.е. подразумевает логическое действие, а не конкретный способ его реализации. Абстрактные виртуальные методы часто используются при создании классов-полуфабрикатов. Свою реализацию такие методы получают в законченных наследниках. Работа виртуальных методов основана на механизме позднего связывания (latebinding). В отличие от раннего связывания (earlybinding), характерного для статических методов, позднее связывание основано на вычислении адреса вызываемого метода при выполнении программы. Адрес метода вычисляется по хранящемуся в каждом объекте описателю класса.

 Вызов виртуального метода осуществляется следующим образом:

1. Через объектную переменную выполняется обращение к занятому объектом блоку

памяти;

2. Далее из этого блока извлекается адрес таблицы виртуальных методов (он записан в

четырех первых байтах);

3. На основании порядкового номера виртуального метода извлекается адрес

соответствующей подпрограммы;

4. Вызывается код, находящийся по этому адресу.

Специализированной формой динамических методов являются методы обработки

сообщений. Они объявляются с помощью ключевого слова message, за которым следует

целочисленная константа — номер сообщения. Метод обработки сообщений имеет формат процедуры и содержит единственный var-параметр. При перекрытии такого метода название метода и имя параметра могут быть любыми, важно лишь, чтобы неизменным остался номер сообщения, используемый для вызова метода. Вызов метода выполняется не по имени, как обычно, а с помощью обращения определен в классе TObject).Методы обработки сообщений применяются внутри библиотеки VCL для обработки команд пользовательского интерфейса и редко нужны при написании прикладных программ.

Полиморфи́зм (в языках программирования) — возможность объектов с одинаковой спецификацией иметь различную реализацию.

Язык программирования поддерживает полиморфизм, если классы с одинаковой спецификацией могут иметь различную реализацию — например, реализация класса может быть изменена в процессе наследования[1].

Кратко смысл полиморфизма можно выразить фразой: «Один интерфейс, множество реализаций».

**7. Понятие ссылки на метод объекта (или делегата – в зависимости от языка программирования). Понятие события. Применение ссылок на методы для расширения объектов.**

 В языке Delphi существуют процедурные типы данных для методов объектов. Внешне

объявление процедурного типа для метода отличается от обычного словосочетанием of

object, записанным после прототипа процедуры или функции:

type

TReadLineEvent = procedure (Reader: TTextReader; const Line: string) of object;

Переменная такого типа называется указателем на метод (methodpointer). Она занимает в

памяти 8 байт и хранит одновременно ссылку на объект и адрес его метода.

type

TTextReader = class

private

FOnReadLine: TReadLineEvent;

...

public

propertyOnReadLine: TReadLineEvent read FOnReadLine write FOnReadLine;

end;

Методы объектов, объявленные по приведенному выше шаблону, становятся совместимы по типу со свойством OnReadLine.

type

TForm1 = class(TForm)

procedureHandleLine(Reader: TTextReader; const Line: string);

end;

var

Form1: TForm1;

Reader: TTextReader;

Если установить значение свойства OnReadLine:

Reader.OnReadLine := Form1.HandleLine;

ипереписатьметодNextLine,

functionTTextReader.NextLine: Boolean;

var

S: string;

N: Integer;

begin

Result := not EndOfFile;

ifResultthen // Если строки для считывания еще есть, то

begin

Readln(FFile, S); // Считывание очередной строки

N :=ParseLine(S); // Выделение элементов строки (разбор строки)

if N <>ItemCount then

SetLength(FItems, N);

if Assigned(FOnReadLine) then

FOnReadLine(Self, S); // уведомление о чтенииочереднойстроки

end;

end;

то объект Form1 через метод HandleLine получит уведомление об очередной считанной

строке. Обратите внимание, что вызов метода через указатель происходит лишь в том

случае, если указатель не равен nil. Эта проверка выполняется с помощью стандартной

функции Assigned, которая возвращает True, если ее аргумент является связанным

указателем.

Описанный выше механизм называется делегированием, поскольку он позволяет передать

часть работы другому объекту, например, сосредоточить в одном объекте обработку

событий, возникающих в других объектах. Это избавляет программиста от необходимости

порождать многочисленные классы-наследники и перекрывать в них виртуальные методы.

Делегирование широко применяется в среде Delphi. Например, все компоненты делегируютобработку своих событий той форме, в которую они помещены.

**8. Понятие метакласса (в некоторых языках программирования). Методы, применяемые к классам. Виртуальные конструкторы (в некоторых языках).**

Язык Delphi позволяет рассматривать классы объектов как своего рода объекты, которыми

можно манипулировать в программе. Такая возможность рождает новое понятие — класс

класса; его принято обозначать термином метакласс. Для поддержки метаклассов введен специальный тип данных — ссылка на класс (classreference). Он описывается с помощью словосочетания classof. По аналогии с тем, как для всех классов существует общий предок TObject, у ссылок на классы существует базовый тип TClass, определенный, как:

type

TClass = class of TObject;

Переменная типа TClass может ссылаться на любой класс. Практическая ценность ссылок на классы состоит в возможности создавать программные модули, работающие с любыми классами объектов, даже теми, которые еще не разработаны.

Метаклассы привели к возникновению нового типа методов — методов класса. Метод

класса оперирует не экземпляром объекта, а непосредственно классом. Он объявляется как обычный метод, но перед словом procedure или function записывается зарезервированное слово class.

 Методы классов могут быть виртуальными. Например, в классе TObjectопределен

виртуальный метод класса NewInstance. Он служит для распределения памяти под объект и автоматически вызывается конструктором. Его можно перекрыть в своем классе, чтобы

обеспечить нестандартный способ выделения памяти для экземпляров. МетодNewInstance

должен перекрываться вместе с другим методом FreeInstance, который автоматически

вызывается из деструктора и служит для освобождения памяти. Добавим, что размер памяти, требуемый для экземпляра, можно узнать вызовом предопределенного метода класса InstanceSize.

 Особые возможности ссылок на классы проявляется в сочетании с виртуальными

конструкторами. Виртуальный конструктор объявляется с ключевым словом virtual. Вызов виртуального конструктора происходит по фактическому значению ссылки на класс, а не по ее формальному типу. Это позволяет создавать объекты, классы которых неизвестны на этапе компиляции. Механизм виртуальных конструкторов применяется в среде Delphi при восстановлении компонентов формы из файла. Восстановление компонента происходитследующим образом. Из файла считывается имя класса. По этому имени отыскивается ссылка на класс (метакласс). У метакласса вызывается виртуальный конструктор, который создает объект нужного класса.

**9. Ошибки и исключительные ситуации. Классы исключительных ситуаций. Создание и обработка исключительных ситуаций. Защита от утечки ресурсов в случае возникновения исключительных ситуаций. Приемы надежного программирования.**

 Исключительная ситуация (exception) — это прерывание нормального хода работы

программы из-за невозможности правильно выполнить последующие действия.

Представим, что подпрограмма выделяет область динамической памяти и загружает в нее

содержимое некоторого файла. Если в системе окажется недостаточно памяти, то данные

будет негде разместить и попытка загрузить файл приведет к ошибке. Скорее всего, вся

программа будет аварийно завершена из-за того, что оператор загрузки данных обратится по

недоступному для программы адресу. Как этого избежать? При обнаружении проблемы

подпрограмма должна создать исключительную ситуацию — прервать нормальный ход

своей работы и передать управление тем операторам, которые смогут обработать ошибку.

Как правило, операторы обработки исключительных ситуаций находятся в одной из

вызывающих подпрограмм.

Механизм обработки исключительных ситуаций лучше всего подходит для взаимодействия

программы с библиотекой подпрограмм. Подпрограммы библиотеки обнаруживают ошибки,но в большинстве случаев не знают, как на них реагировать. Вызывающая программа,наоборот, знает, что делать при возникновении ошибок, но, как правило, не умеет ихсвоевременно обнаруживать. Благодаря механизму обработки исключительных ситуацийобеспечивается связь между библиотекой и использующей ее программой при обработкеошибок.Механизм обработки исключительных ситуаций довольно сложен в своей реализации, но дляпрограммиста он прост и прозрачен. Для его использования в язык Delphi введеныспециальные конструкции try...except...end, try...finally...end и оператор raise.

 Исключительные ситуации в языке Delphi описываются классами. Каждыйкласс

соответствует определенному типу исключительных ситуаций. Когда в программе возникает исключительная ситуация, создается объект соответствующего класса, который переносит информацию об этой ситуации из места возникновения в место обработки.

Классы исключительных ситуаций образуют иерархию, корнем которой является класс

Exception. Класс Exception описывает самый общий тип исключительных ситуаций, а его

наследники — конкретные виды таких ситуаций (таблица 4.1). Например, класс

EOutOfMemory порожден от Exception и описывает ситуацию, когда свободная

оперативнаяпамятьисчерпана.

 Идея обработки исключительных ситуаций состоит в следующем. Когда подпрограмма

сталкивается с невозможностью выполнения последующих действий, она создает объект с

описанием ошибки и прерывает нормальный ход своей работы с помощью оператора raise.

Так возникает исключительная ситуация.

raiseEOutOfMemory.Create('Маловато памяти');

Данный оператор создает объект класса EOutOfMemory (класс ошибок исчерпания памяти) и прерывает нормальное выполнение программы. Вызывающие подпрограммы могут эту исключительную ситуацию перехватить и обработать. Для этого в них организуется так называемый защищенный блок:

try

// защищаемые от ошибок операторы

except

// операторы обработки исключительной ситуации

end;

Между словами try и except помещаются защищаемые от ошибок операторы. Если при

выполнении любого из этих операторов возникает исключительная ситуация, то управление передается операторам между словами except и end, образующим блок обработки исключительных ситуаций. При нормальном (безошибочном) выполнении программы блок except...end пропускается. Программы, построенные с использованием механизма исключительных ситуаций, обязаны придерживаться строгих правил распределения и освобождения таких ресурсов, как память, файлы, ресурсы операционной системы. Представьте ситуацию: подпрограмма распределяет некоторый ресурс, но исключительная ситуация прерывает ее выполнение, и ресурс остается не освобожденным. Даже подумать страшно, к чему может привести такая ошибка: утечка памяти, файловых дескрипторов, других ресурсов операционной системы. Следовательно, ресурсы нуждаются в защите от исключительных ситуаций. Для этого в среде Delphi предусмотрен еще один вариант защищенного блока:

// запрос ресурса

try

// защищаемые операторы, которые используют ресурс

finally

// освобождение ресурса

end;

Особенность этого блока состоит в том, что секция finally...end выполняется всегда

независимо от того, происходит исключительная ситуация или нет. Если какой-либо

оператор секции try...finally генерирует исключительную ситуацию, то сначала выполняется секция finally...end, называемая секцией завершения (освобождения ресурсов), а затем управление передается внешнему защищенному блоку. Если все защищаемые операторы выполняются без ошибок, то секция завершения тоже работает, но управление передается следующему за ней оператору. Обратите внимание, что секция finally...end не обрабатывает исключительную ситуацию, в ней нет ни средств ее обнаружения, ни средств доступа к объекту исключительной ситуации.

**10. Понятие интерфейса. Описание интерфейса. Поддержка интерфейса классом.**

**Механизм подсчета ссылок в интерфейсах. Расширение интерфейса. Глобально-уникальный идентификатор интерфейса. Совместимость интерфейсов и классов. Получение интерфейса через другой интерфейс. Представление интерфейса в памяти. Механизм вызова метода объекта через интерфейс. Применение интерфейса для доступа к объекту динамически-подключаемой библиотеки.**

Останется лишь интерфейс — заголовки методов и описания свойств. В отличие от объекта интерфейс сам ничего “не помнит” и ничего “не умеет делать”; он является всего лишь "разъемом" для работы с объектом. Объект может поддерживать много интерфейсов и выступать в разных ролях в зависимости от того, через какой интерфейс вы его используете. Совершенно различные по структуре объекты, поддерживающие один и тот

же интерфейс, являются взаимозаменяемыми. Не важно, есть у объектов общий предок или нет. В данном случае интерфейс служит их дополнительным общим предком.

 В языке Delphi интерфейсы описываются в секции type глобального блока. Описание

начинается с ключевого слова interface и заканчивается ключевым словом end. По форме

объявления интерфейсы похожи на обычные классы, но в отличие от классов:

- интерфейсы не могут содержать поля;

- интерфейсы не могут содержать конструкторы и деструкторы;

- все атрибуты интерфейсов являются общедоступными (public);

- все методы интерфейсов являются абстрактными (virtual, abstract).

Новый интерфейс можно создать с нуля, а можно создать путем расширения уже

существующего интерфейса. Во втором случае в описании интерфейса после слова interface указывается имя базового интерфейса:

type

IExtendedTextReader = interface(ITextReader)

procedureSkipLines(Count: Integer);

end;

Определенный таким образом интерфейс включает все методы и свойства своего

предшественника и добавляет к ним свои собственные. Несмотря на синтаксическое

сходство с наследованием классов, расширение интерфейсов имеет другой смысл. В классах наследуется реализация, а в интерфейсах просто расширяется набор методов и свойств. В языке Delphi существует предопределенный интерфейс IInterface, который служит неявным базовым интерфейсом для всех остальных интерфейсов.

 Интерфейс является особым типом данных: он может быть реализован в одной программе, а использоваться из другой. Для этого нужно обеспечить идентификацию интерфейса при межпрограммном взаимодействии. Понятно, что программный идентификатор интерфейса для этого не подходит — разные программы пишутся разными людьми, а разные люди подчас дают одинаковые имена своим творениям. Поэтому каждому интерфейсу выдается своеобразный «паспорт» — глобально-уникальный идентификатор (GloballyUniqueIdentifier— GUID). Глобально-уникальный идентификатор — это 16-ти байтовое число, представленное в виде заключенной в фигурные скобки последовательности шестнадцатеричных цифр:

{DC601962-28E5-4BF7-9583-0CE22B605045}. Константы с типом TGUID разрешено инициализировать строковым представлением глобально-уникального идентификатора. Компилятор сам преобразует строку в запись с типом TGUID. Пример:

const

InterfaceID: TGUID = '{DC601962-28E5-4BF7-9583-0CE22B605045}';

Если глобально-уникальный идентификатор назначается интерфейсу, то он записывается

после ключевого слова interface и заключается в квадратные скобки.

 Интерфейсной переменной можно присвоить значение объектной переменной при условии, что объект (точнее его класс) реализует упомянутый интерфейс:

var

Intf: ITextReader; // интерфейсная переменная

Obj: TTextReader; // объектная переменная

begin

...

Intf := Obj; // В переменную Intf копируется ссылка на объект Obj

...

end;

Такая совместимость сохраняется в производных классах. Если класс реализует некоторый

интерфейс, то и все его производные классы совместимы с этим интерфейсом. Через интерфейсную переменную у объекта всегда можно запросить интерфейс другого

типа. Для этого используется оператор as, например:

var

Intf: IInterface;

begin

...

withIntf as ITextReader do

Active := True;

...

end;

Если объект действительно поддерживает запрашиваемый интерфейс, то результатом

является ссылка соответствующего типа. Если же объект не поддерживает интерфейс, то

возникает исключительная ситуация EIntfCastError.

В действительности оператор as преобразуется компилятором в вызов метода

QueryInterface:

var

Intf: IInterface;

IntfReader: ITextReader;

...

IntfReader :=Intf as ITextReader; // Intf.QueryInterface(ITextReader,

IntfReader);

Напомним, чтометодQueryInterfaceописанвинтерфейсеIInterfaceипопадает

автоматически во все интерфейсы. Стандартная реализация этого метода находится в классе TInterfacedObject.

Интерфейс по сути выступает дополнительной таблицей виртуальных методов, ссылка накоторую укладывается среди полей объекта (рисунок 6.5). Эта таблица называется таблицейметодов интерфейса. В ней хранятся указатели на методы класса, реализующие методыинтерфейса.

**11. Понятие компонента. Понятие визуального программирования. Инструментальные средства визуального компонентного программирования. Современные библиотеки компонентов.**

**12. Понятие расширяемой программы и расширяемого программирования. Принципы расширяемого программирования. Применение методологии объектно-ориентированного программирования для построения расширяемых программ. Языки программирования Никлауса Вирта: Oberon, ComponentPascal.**

**13. Имитация модульного программирования в языке C++. Понятие пространства имен.**

 В языке С++ очень бедные средства модульного программирования, поэтому для

достижения модульности программ, следует придерживаться определенных принципов.

Роль программного интерфейса модуля играет h-файл, а cpp-файл — роль реализации этогомодуля. Внутрь h-файла включаются h-файлы других модулей, необходимые для

компиляции интерфейсной части. Внутрь cpp-файла включаются h-файлы других модулей,необходимые для компиляции cpp- и h-файлов интерфейсной части модуля.

В больших проектах наблюдается серьезная проблема — конфликт идентификаторов. Она

решается с помощью пространства имен.

namespaceSys

{

intvar;

voidProc();

}

Внутри пространства имен обращение к определенным внутри переменным и

подпрограммам можно осуществлять, используя неполную форму записи:

var = 10;

Proc();

за пределами – надо использовать полную форму записи:

Sys::var = 10;

Sys::Proc();

Для того чтобы избежать возможного конфликта идентификаторов, все определения внутримодуля следует помещать в пространство имен. Следует давать небольшой буквенныйидентификатор, который будет соответствовать префиксу файла**.**

**14. Классы в языке C++. Наследование. Конструкторы и деструкторы. Стандартные конструкторы. Создание объектов по значению (на стеке) и по ссылке (в динамической памяти). Операторы new и delete. Размещающий оператор new. Порядок конструирования и разрушения объектов. Вложенные определения классов. «Друзья» класса. Статические члены класса.**

 Классы в С++ определяются с помощью одного из ключевых слов: class или struct.

В С++ доступны атрибуты доступа в классах:

-public

-protected

-private

Вработесекций protected и private в Delphi и C++ естьразличия:

- В Delphi классы внутри одного модуля могут обращаться к данным и подпрограммам

друг друга без ограничений. А действие секций protected и private распространяется

только за пределами данного модуля.

-ВС++ действие этих секций распространяется на любые два класса.

Для того, чтобы объекты некоторого класса могли получить доступ в private и protected

полям другого класса, используется оператор friend, который разрешает доступ ко всем

записям и методам класса для того класса, который указан в операторе. Данный оператор

используется внутри класса, с его помощью нельзя разрешать доступ к членам класса извне,иначе это нарушит принцип сокрытия данных.

Наследование класса выполняется следующим образом:

classTDelimitedReader: publicTTextReader

{

...

};

При наследовании указываются атрибуты доступа к элементам базового класса (public,

protected, private). Для того чтобы понять смысл атрибута доступа к базовому классу,

базовый класс следует рассматривать, как поле производного класса.

Конструктор создает объект и инициализирует память для него (деструктор — наоборот).

classTTextReader

{

public:

TTextReader();

~TTextReader();

};

ВDelphi стандартный деструктор является виртуальным. В С++ это определяет

программист, если планируется создавать объекты в динамической памяти (по ссылке),

деструктор необходимо делать виртуальным:

virtual ~TTextReader();

Создание объектов:

-по значению (на стеке):

TTextReaderreader;

-по ссылке (в динамической памяти):

TTextReader\*reader = new TTextReader(); //оператор new служитдляразмещения

объекта в динамической памяти

-с помощью оператора new:

TTextReader\*reader = new (адрес )TTextReader;

Таким способом объект создается по ссылке по указанному адресу памяти.

Разрушение объектов:

- если объект создан по значению (на стеке), его разрушение выполняется

автоматически при выходе переменной за область видимости

{

TTextReaderreader;

...

} // здесь происходит разрушение объекта reader при автоматическом вызоведеструктора

-если объект создан в динамической памяти (по ссылке), он должен быть уничтожен с

помощьюоператора delete:

delete reader;

При этом сначала происходит вызов деструктора, а затем — освобождениединамической памяти.

Так выглядит динамическое создание и разрушение объектов:

new:

malloc();

TTextReader();

delete:

~TTextReader();

free();

Если программист не определяет в классе конструкторы, то компилятор создаетавтоматически два конструктора:

-конструктор без параметров

-конструктор копирования

Пример:

classTTextReader

{

public:

TTextReader(); // конструктор без параметров

TTextReader(constTTextReader&R); // конструктор копирования

}

Внимание! Если программист определил хотя бы один конструктор в класс — компилятор несоздаст никаких стандартных конструкторов.Конструктор без параметров создается для того, чтобы можно было написать:

TTextReader R;

Конструктор копирования нужен для следующей записи:

TTextReader R1 = R2; // означаетTTextReaderR1(R2);

Конструктор копирования вызывается в том случае, когда создаваемый по значению объектсоздается путем копирования другого уже существующего объекта.Следует отметить, что запись:

TTextReader R1 = R2;

и два оператора:

TTextReader R1;

R1 = R2;

имеют схожий синтаксис с вызовом конструктора копирования, но разную семантику: в

первом случае объект создается конструктором копирования, во втором — конструктором

без параметров, а затем с помощью оператора ‘=’ выполняется присваивание одного объектадругому (данный вариант требует перегрузки оператора ‘=’ для класса TTextReader). Работа стандартного конструктора копирования, создаваемого компилятором, заключается втом, чтобы выполнить полное копирование памяти с помощью функции memcpy.

В конструкторе производного класса конструктор базового класса вызывается автоматическидо выполнения первого оператора в теле конструктора.В деструкторе производного класса деструктор базового класса вызывается автоматическипосле последнего оператора в теле деструктора.Если базовый класс содержит конструктор с параметрами или несколько конструкторов, товозникает неопределенность в том, какой конструктор базового класса будет вызван.

Если поле объявлено с ключевым словом static, то это — обычная глобальная переменная,для которой имя класса используется как пространство имен.

Если метод объявляется с этим словом, то это — обычная глобальная функция, которая

является другом класса. Такая функция не имеет псевдо-параметра this.

**15. Множественное наследование. Проблема повторяющихся базовых классов. Замена множественного наследования наследованием от интерфейсов в других языках объектно-ориентированного программирования. Типовой пример применения множественного наследования — «наблюдатель» (observer).**

В С++ поддерживается множественное наследование. В этом случае конструктор базовыхклассов вызывается автоматически в порядке их упоминания в описании класса. Деструкторже базовых классов вызывается строго в обратном порядке.Каждый конструктор перед началом своей работы инициализирует указатель vtable (в Delphi

он называется VMT). Конструктор базового класса тоже инициализирует этот указатель. В

результате этого объект как бы "рождается", сначала становясь экземпляром базового класса,а затем производного. Деструкторы выполняют противоположную операцию.

В результате этого в конструкторах и деструкторах виртуальные методы работают как

невиртуальные.

В С++ множественное наследование подразумевает, что у одного класса может быть

несколькобазовыхклассов:

classTDelimitedReader : public TTextReader, public TStringList

{

...

};

Объект класса TDelimitedReader содержит все поля и методы базовых классов TTextReader иTStringList. При этом в классе TDelimitedReader можно переопределять виртуальные методыкаждого базового класса.

Множественное наследование имеет ряд проблем:

- отсутствие эффективной реализации (неэффективность скрыта от программиста);

- неоднозначность, возникающая из-за того, что в базовых классах могут быть одноименные поля, а также методы с одинаковой сигнатурой;

- повторяющийся базовый класс в иерархии классов.

Таким образом, множественное наследование таит следующую проблему: заранее

неизвестно от каких классов программист захочет унаследовать свой класс. Однако при

создании класса использовать виртуальное наследование неэффективно, если наследуются

поля, так как доступ к полям всегда будет осуществляться через дополнительный указатель.Вывод: одинарное наследование в стиле Java, C++, Delphi допустимо только от классов,множественное — от интерфейсов. Иначе можноосуществлять множественное наследованиелишь от классов, в которых отсутствуют поля.

**16. Виртуальные методы в языке C++. Недостатки синтаксиса определения и перекрытия виртуальных методов в языке C++. Понятие константного метода. Проблемы, порождаемые наличием константных методов. Операторы приведения типа в языке C++: const\_cast, reinterpret\_cast, static\_cast, dynam-ic\_cast.**

В С++ виртуальные методы определяются при помощи ключевого слова virtual:

class TTextReader: virtual public TObject

{

...

};

При перекрытии виртуального метода ключевое слово virtual можно записать, а можно и

опустить. Синтаксис перекрытия виртуальных методов не предусматривает такие проблемы,

как версионность и рефакторинг кода (упрощение программного кода с сохранением

функциональности).

Если метод виртуальный следует всегда писать ключевое слово virtual.

В С++ абстрактный класс объявляется следующим образом:

class TTextReader

{

protected:

virtual void NextLine() = 0;

...

};

Такой метод называется абстрактным и класс, содержащий данный метод, тоже называется абстрактным. Виртуальные методы следует объявлять в секции protected.

Существует четыре оператора преобразования типа в С++:

reinterpret\_cast<тип>(переменная)

static\_cast<тип>(переменная)

const\_cast<тип>(переменная)

dynamic\_cast<тип>(переменная)

Первый оператор (reinterpret\_cast) позволяет отключить контроль типов данных на уровне

компилятора, с помощью него любой указатель может быть интерпретирован, как любой

другой указатель, а также любая память или переменная может быть интерпретирована

иначе.В программах этот оператор преобразования типа использовать не следует, так как оннарушает переносимость программ. Его наличие свидетельствует о том, что программа

является кросс-платформенной. Пример использования:

int i;

char \*p;

p = reinterpret\_cast<char>(&i);

Второй оператор (static\_cast) используется вместо преобразования тип(переменная),

(тип)переменная и (тип)(переменная) при работе с классами, структурами и указателями на них. Оператор static\_cast был задуман по причине того, что в С++ выражение тип(переменная) может оказаться вызовом конструктора. Если в программе требуется преобразовать тип, а не вызвать конструктор типа, используется данный оператор. Кроме того, оператор (тип)переменная или (тип)(переменная) может в некоторых случаях оказаться преобразованием reinterpret\_cast<тип>(переменная), а при разработке кросс-платформенных программ оператор reinterpret\_cast всегда содержит потенциальную опасность неправильной работы программы на другой платформе. Поэтому вместо операторов тип(переменная), (тип)переменная и (тип)(переменная) следует использовать операторы reinterpret\_cast и static\_cast, которые убирают не явность из преобразования.

Так как оператор static\_cast является громоздким, то для простых типов данных допустимо использование форм: (тип)переменная и (тип)(переменная). Форма тип(переменная) не должна использоваться для преобразования типа.

Третий оператор (const\_cast) используется для приведения не константных указателей к

константным и наоборот. При объявлении переменных и параметров функций в описании типа может быть указано ключевое слово const. Объявление f(const char \*s); означает, что символы, адресуемые указателем s, изменять нельзя. Наличие константных объектов порождает проблему — огромная избыточность программного кода. Заранее программист не знает, будет ли пользователь (другой программист) его класса создавать константные объекты. Вследствие того, что это не исключено, программист начинает записывать слово const в объявление всех методов, в которых его можно записать. Многие методы являются виртуальными или вызывают виртуальные методы. Случается так, что в производных классах виртуальные методы, вызванные константными методами, модифицируют поля объектов (это требуется по условию задачи). Это приводит к логической проблеме, которая решается либо за счет применения оператора const\_cast к указателю this в производных классах, либо за счет объявления полей в производных классах с модификатором mutable (записывается при описании полей класса в том случае, если они должны модифицироваться константными методами).

 Ключевое слово volatile запрещает кэшировать значение переменной. Если в программе

происходит считывание значения переменной, объявленной с этим ключевым словом, то

значение считывается из памяти, а не из регистров, а запись всегда производится в память, в которой размещается данная переменная. Если ключевое слово volatile не указано, то оптимизатор C++ имеет право выполнять регистровые операции (оптимизации) при чтении и записи переменных, а так же размещать их в регистрах. Четвертый оператор (dynamic\_cast) соответствует оператору as в Delphi. Для работы этого оператора нужно в опциях компилятора включить опцию RTTI. Если это выполнено, то оператор ynamic\_cast работает, как static\_cast. Оператор dynamic\_cast работает по-разному в зависимости от того, применяется он к ссылке на объект (&) или указателю на объект (\*). Если оператор применяется к ссылке на объект, то преобразование не может быть выполнено и возникает исключительная ситуация. Если он применяется к указателю на объект и преобразование не может быть выполнено, оператор возвращает NULL.

**17. Ссылки в языке C++. Рекомендации по работе со ссылками. Типичные ошибки при работе со ссылками.**

 Ссылка является альтернативным именем объекта и объявляется следующим образом:

int i;

int &r = i;

Использование ссылки r эквивалентно использованию переменной i. Основное применение ссылок — передача параметров в функцию и возврат значения. В случае, когда ссылка используется в качестве параметра функции, она объявляется неинициализированной:

void f(int &i);

 Во всех остальных случаях ссылка должна инициализироваться при объявлении, как

показано ранее.

Если ссылка является полем класса, она должна инициализироваться в конструкторе класса в списке инициализации до тела конструктора. При использовании в качестве параметров функций ссылки соответствуют var-параметрам в языке Delphi:

procedure P(var I: Integer)

begin

...

end;

Константные ссылки соответствуют const-параметрам в языке Delphi:

procedure P(const I: Integer)

begin

...

end;

При передаче ссылочного параметра в стек заносится адрес переменной, а не ее копия.

Ссылку следует рассматривать, как псевдоним переменной, которой она инициализирована. Ссылки отличаются от указателей тем, что позволяют компилятору лучше оптимизировать программный код. Для возврата значений из процедур (через параметры) предпочтение следует отдавать указателям, а не ссылкам. Ссылки следует использовать лишь в тех случаях, когда известно, что возвращаемый объект должен создаваться не в динамической памяти, а на стеке, то есть ссылки применяют при возврате value-type-объектов. При работе со ссылками существует типовая ошибка — возврат через ссылку переменной, созданной на стеке. Пример ошибочной записи:

void f(int \*p)

{

int i;

p = &i;

}

Следующая запись тоже будет ошибочной:

void f(int &r)

{

int i;

r = i;

}

Следующий пример тоже ошибочен, так как нельзя возвращать адрес объекта, созданного на стеке:

std::string& GetName(Object\* Obj)

{

const char\* str = Obj->GetName();

return std::string(str);

}

**18. Обработка исключительных ситуаций в языке C++. Защита от утечки ресурсов. Имитация оператора try-finally. Понятие автоматического указателя (auto\_ptr). Использование автоматических указателей для защиты от утечки ресурсов.**

 В С++ отсутствует аналог блока try…finally…end.

На платформе Windows благодаря структурной обработке ОС существуют следующий блок:

\_\_try

{

...

}

\_\_finally

{

...

}

Но следует отметить, что для переносимых программ он не подходит. В С++ существует аналог блока try…except…end:

**try**

**{**

**...**

**}**

**catch(std::ios\_base::failure)**

**{**

**...**

**}**

**catch(std::exception)**

**{**

**...**

**}**

**catch(...)**

**{**

**...**

**}**

Распознавание исключительных ситуаций происходит последовательно блоками catch, поэтому их последовательность должна быть от частного к общему. Последний блок catch в примере выше ловит любую исключительную ситуацию. Создание исключительных ситуаций выполняется с помощью оператора throw (аналог raise в Delphi):

throw std::exception("Ошибка");

Внутри блока catch оператор throw возобновляет исключительную ситуацию, как и raise в

Delphi. При создании исключительной ситуации при помощи оператора throw объект,

описывающий исключительную ситуацию, может быть создан в динамической памяти:

throw new std::exception("Ошибка");

Если применяется такой способ создания исключительной ситуации, ее уничтожение должно происходить следующим образом:

try

{

...

throw new std::exception("Ошибка");

}

catch(std::exception \*e)

{

delete e;

}

catch(...)

{

...

}

Если же записать так:

try

{

...

throw new std::exception("Ошибка");

}

catch(...)

{

...

}

то возникнет утечка ресурсов из-за того, что объект std::exception, созданный в

динамической памяти, не будет освобожден. В С++ отсутствует общий базовый класс для исключительных ситуаций, поэтому на верхнем уровне работы программы нужно отлавливать все возможные базовые классы исключительных ситуаций. Это является препятствием на пути построения расширяемых систем.

 Поскольку в С+ отсутствует блок try…finally, его приходится эмулировать.

Object \*p = new Object();

try

{

...

}

catch(...)

{

delete p;

throw;

}

delete p;

Данный код эквивалентен следующему:

Object \*p = new Object();

\_\_try

{

...

}

\_\_finally

{

delete p;

}

за исключение того, что второй пример не является переносимым. Согласно стандарту С++ в деструкторах и операторах delete не должно быть исключительных ситуаций, если же исключительная ситуация произошла, то поведение программы не определено. Если исключительная ситуация происходит в конструкторе объекта, объект считается не созданным и деструктор для этого объекта не вызывается, но память, выделенная для

объекта, освобождается. Если внутри объекта агрегированы другие объекты, то вызываются деструкторы лишь для тех объектов, которые были полностью созданы к моменту возникновения исключительной ситуации. Если объект создается в динамической памяти и освобождается в той же самой процедуре, то для защиты от утечки ресурсов можно применять оболочечные объекты — wrapper (содержит указатель на динамический объект, который уничтожается в деструкторе оболочечного объекта). Оболочечный элемент создается на стеке, поэтому его деструктор вызывается автоматически, гарантируя тем самым уничтожение агрегированного динамического объекта. Такие оболочечные объекты в библиотеках программирования называются AutoPtr, SafePtr и т.д.

**19. Перегрузка бинарных операторов. Перегрузка унарных операторов. Перегрузка операторов преобразования типа.**

Перегрузка операторов позволяет заменить смысл стандартных операторов (+, –, = и др.) для пользовательских типов данных. В С++ разрешена перегрузка операторов, выраженных в виде символов, а также операторов:

new delete

new[] delete[]

Запрещена перегрузка следующих операторов:

:: . .\* ?:

Перегрузка операторов таит угрозу: она резко усложняет понимание программы, поэтому ей пользоваться нужно очень осторожно. Для стандартных типов данных перегрузка запрещена, хотя бы один из операторов должен принадлежать пользовательскому типу данных.

 Бинарный оператор можно определить либо в виде нестатической функции членов с одним аргументом, либо в виде статической функции с двумя аргументами.

Для любого бинарного оператора @ выражение aa@bb интерпретируется как

aa.operator@(bb) или operator@(aa, bb). Если определены оба варианта, то применяется

механизм разрешения перегрузки функций.

 Унарные операторы бывают префиксными и постфиксными.

Унарный оператор можно определить в виде метода класса без аргументов и в виде функции с одним аргументом. Аргумент функции — объект некоторого класса.

Для любого префиксного унарного оператора выражение @aa интерпретируется как:

aa.operator @();

operator @(aa);

Для любого постфиксного унарного оператора выражение aa@ интерпретируется, как:

aa.operator @(int);

operator @(aa, int);

Запрещено перегружать операторы, которые нарушают грамматику языка.

Существует три оператора, которые следует определить внутри класса в виде методов:

operator =

operator []

operator ->

Это гарантирует, что в левой части оператора будет записан lvalue (присваиваемое

значение).

 В С++ существуют операторы преобразования типов. Это является хорошим способом

использования конструктора для преобразования типа. Конструктор не может выполнять

следующие преобразования:

-неявное преобразование из типа, определяемого пользователем в базовый тип. Это

связано с тем, что базовые типы не являются классами.

- преобразование из нового класса в ранее определенный класс, не модифицируя

объявление ранее определенного класса.

 Оператор преобразования типа возвращает значение типа T, однако в сигнатуре оператора он не указывается. В этом смысле операторы преобразования типа похожи на конструкторы. Хотя конструктор не может использоваться для неявного преобразования типа из класса в базовый тип, он может использоваться для неявного преобразования типа из класса в класс. В программе следует избегать любых неявных преобразований типов, так как это приводит к ошибкам. С помощью ключевого слова explicit можно запретить неявное преобразования типа к конструкторам. Слово explicit записывается лишь для тех конструкторов, которые могут вызываться лишь с одним параметром. Если же они вызываются с несколькими параметрами, то неявное преобразование типов невозможно.

Если объект создается на стеке, то неявное преобразование типа часто бывает необходимо, тогда слово explicit писать надо. Так же его надо писать, когда объект создается динамически. При перегрузке операторов нужно быть внимательным к типу возвращаемого значения: для некоторых операторов объект возвращается по ссылке, для некоторых — по значению.

**20. Шаблоны функций. Перегрузка шаблонов функций. Шаблоны классов. Специализации шаблонов. Стандартная библиотека шаблонов Standard C++ Library. Строки. Итераторы. Потоки ввода-вывода.**

 Шаблоны обеспечивают непосредственную поддержку обобщенного программирования. Они представляют собой параметризованные классы и параметризованные имена функций. Шаблон определяется с помощью ключевого слова template:

template <class T>

class basic\_string

{

public:

basic\_string();

basic\_string(const T\*);

basic\_string(const basic\_string&);

private:

T\*str;

};

typedef basic\_string<char> string;

typedef basic\_string<unsigned int> wstring;

Вместо слова typename часто записывают слово class, но параметром шаблона может быть

любой тип данных. С точки зрения компилятора, шаблон является макроподстановкой,

поэтому шаблонные классы определяются целиком в заголовках файлов (в h-файле, а не в

cpp-файле).

Методы шаблона описываются следующим образом:

template <class T>

basic\_string<T>::basic\_string(const \*T)

{

...

}

 Допускается применение шаблонов с целью реализации абстрактных алгоритмов, то есть шаблонов функций.

template <class T>

void sort(vector<T>& v);

При вызове шаблонных функций компилятор подставляет тип данных и создает новый

вариант функции. Если один и тот же тип данных используется несколько раз, то на все типы данных используется несколько раз, то на все типы данных создается один шаблон функции. При использовании шаблонов существует три больших недостатка:

-шаблоны невозможно отлаживать.

-существенно замедляется время компиляции. В больших проектах оно может

доходить до 30-60 минут.

- очень быстро растут размеры объектных модулей и библиотек на диске.

 Шаблонные функции могут вызываться с явным указанием параметра шаблона:

sqrt<int>(2);

или без него:

sqrt(2);

В этом случае применяется механизм разрешения перегрузки:

- ищется набор специализации шаблонов функций, которые примут участие в

разрешении перегрузки;

- если могут быть вызваны два шаблона функций и один из них более специализирован,

то только он и будет рассматриваться;

- разрешается перегрузка для этого набора функций и любых обычных функций. Если

аргументы функции шаблона были определены путем выведения по фактическим

аргументам шаблона, к ним нельзя применять “продвижение” типа, стандартные и

определяемые пользователем преобразования.

- если и обычная функция, и специализация подходят одинаково хорошо, предпочтение

отдается обычной функции;

- если ни одного соответствия не найдено, или существует несколько одинаково хорошо подходящих вариантов, то выдается ошибка. В параметрах шаблонов допустимы стандартные значения, принимаемые по умолчанию.

 Как правило, шаблон представляет единственное определение, которое применяется к

различным аргументам шаблона. Это не всегда удобно, иногда существует необходимость

использовать различные реализации в зависимости от типа. Например, надо для всех указателей использовать особую реализацию шаблона, а для всех базовых типов данных — обычную реализацию. Это делается с помощью специализации шаблона. Специализация шаблонов, как правило, используется для сокращения объема программного кода. Если шаблон создается для указателей на какие-то объекты и класс объекта не так важен, то при использовании обычных шаблонов без специализации возникает многократное дублирование одного и того же кода. Это связано с тем, что в машинных кодах работа со всеми указателями строится одинаково. Чтобы избежать дублирования кода в случае использования указателей следует создавать специализации шаблонов.

 Перечислим, что содержится в стандартной библиотеке шаблонов:

- Классы и шаблоны для организации потоков ввода/вывода

В языке С++ вместо функций printf и scanf предлагается использовать объекты потоковых

классов:

std::cout;

std::cin;

Вывод осуществляется с помощью оператора сдвига:

std::cout << "Hello!";

int n;

std::cin >> n;

Чтобы перевести строку надо сделать следующую запись:

std::cout << "Hello!" << std::endl; // или "\n"

Объекты cout и cin являются экземплярами классов ostream и istream. Существуют также

классы iostream (класс для ввода/вывода) и streambuf (позволяет выполнить

буферизованный ввод/вывод). В программе не следует смешивать потоковый ввод/вывод с функциями printf и scanf. Если все же это происходит, То между блоками кода, использующими тот или иной подход, надо выполнять вызов функции fflush — сброс буферов.

 - Итераторы

#include <iterator>

Итератор — абстракция указателя на элемент контейнера.

Пример использования:

#include <iterator>

void strlen(const cahr \*str)

{

const char \*p = str;

while(\*p != 0)

{

...

++p;

}

...

}

Указатель в строке — это итератор по строке. Можно сказать, что в примере выше типы данных char\* и const char\* являются итераторами строки (обычной 0-терминированной). Внутри каждого контейнера стандартной библиотеки С++ определены два типа данных:

iterator и const\_iterator, которые фактически являются указателями на элемент контейнера. В стандартном контейнере существуют функции begin() и end(), которые возвращают

соответственно итераторы на первый и последний элементы контейнера. Физически функция end() возвращает NULL. Если итератор адресует объект, то доступ к полям следует осуществлять с помощью оператора \*. Допустимо использование и оператора ->, но он может быть переопределен и поэтому работа операторов \*. и -> может отличаться. Переход к следующему элементу контейнера выполняется префиксным инкрементом ++it. Допустимо использование постфиксного оператора it++, но в последнем случае может возникнуть неоднозначность в том, что увеличивается на единицу — итератор или значение, возвращаемое итератором.

**21. Деление типов данных на типы-«значения» (value-types) и типы-«ссылки» (reference-types) в языке C#. Автоматическое управление памятью ссылочных данных.Упаковка и разупаковка данных. Типы данных со значением null.**

К первой категории (значение) относятся участки памяти, распределенные под переменные и предназначенные для хранения значений этих переменных. Например, следующий код

int x = 5;

объявляет переменную - целую со знаком, размером 32 бита, с именем x и начальным значением, равным 5.

Переменные-ссылки содержат адреса объектов, размещенных в динамической памяти. Следующий код объявляет переменную с именем y, типа object и инициализирует ее с помощью оператора new. Таким образом она получает адрес экземпляра object, размещенного в динамической памяти (object -- это базовый класс для всех типов в C#, но об этом чуть ниже).

object y = new object();

 А сам объект становится "мусором". Как я уже упоминал ранее, в C# встроена поддержка "сборщика мусора", это означает, что система автоматически освобождает память, занимаемую такими "мертвыми" объектами. Другие языки, такие как C++ и Pascal, не поддерживают автоматическую "сборку мусора". Поэтому программисты, пишущие на этих языках, вынуждены явно освобождать блоки динамической памяти по мере необходимости. Недостатком такого способа управления памятью является возможность появления "утечек" памяти. Как показывает опыт -- управление памятью "вручную" слишком громоздко и является потенциальным источником ошибок. Вот почему многие современные языки программирования (такие как Java, Python, Scheme, Smalltalk) имеют встроенную в окружение времени исполнения поддержку автоматической "сборки мусора".

 Тип, допускающие значения NULL, являются экземплярами структуры System.Nullable(Of T). Тип, допускающий значения NULL, может представлять правильный диапазон значений для своего базового типа значений и дополнительное пустое значение null. Например, для Nullable<Int32>, называемого "тип Int32, допускающий значения NULL", можно назначить любое значение от -2 147 483 648 до 2 147 483 647 или значение null. Для Nullable<bool> можно назначить значения truefalse или null. Возможность назначения значения null для числовых и логических типов особенно полезна при работе с базами данных и другими типами данных, содержащих элементы, которым может быть не назначено значение. Например, логическое поле в базе данных может хранить значения true или false или может быть не задано. Типы, допускающие значения NULL, имеют следующие характеристики.

Типы, допускающие значения NULL, представляют переменные типа значения, которым можно назначать значение null. Нельзя создать тип, допускающий значения NULL, на основе ссылочного типа. (Ссылочные типы уже поддерживают значение null.)

Синтаксис T? является краткой формой для Nullable(Of T), где T — это тип значения. Две эти формы взаимозаменяемы.

Назначение значения для типа, допускающего значения NULL, выполняется так же, как и для обычного типа значения, например int?. x = 10; или double? d = 4.108. Однако типу, допускающему значения NULL, также можно присваивать значение NULL: int? x = null.

Используйте метод Nullable(Of T).GetValueOrDefault для возврата назначенного значения или значения по умолчанию для базового типа, если значением является null, например int j = x.GetValueOrDefault();

Используйте свойства HasValue и Value, доступные только для чтения, для выполнения проверки на наличие значения null и извлечения этого значения, например if(x.HasValue) j = x.Value;

Свойство HasValue возвращает значение true, если переменная содержит значение, или значение false, если она пуста.

Свойство Value возвращает значение, если оно назначено. В противном случае создается исключение System.InvalidOperationException.

По умолчанию для свойства HasValue используется значение false. Свойство Value не имеет значения по умолчанию.

С типом, допускающим значение null, можно также использовать операторы == и !=, например if (x != null) y = x;

Используйте оператор ??, чтобы назначить значение по умолчанию, которое применяется в том случае, если тип с текущим значением null, допускающий значения NULL, назначен типу, не допускающему значения NULL (например, int?) x = null; int y = x ?? -1;

Вложенные типы, допускающие значения NULL, использовать нельзя. Компиляция следующей строки невозможна: Nullable<Nullable<int>> n;

**22. Классы в языке C#. Отличие структур (записей) в языке C# от классов. Поля. Методы. Конструкторы и деструкторы. Свойства. Индексаторы. Атрибуты доступа. Сборки. Пространства имен. Частично определяемые классы и их назначение.**

**23. Делегаты и события в языке C#. Механизм вызова событий.**

**24. Обобщенные классы в языке C# и их отличие от шаблонов классов в языке C++. Установка ограничений на параметры обобщенных классов. Обощенные делегаты.**

**25. Понятие итератора в языке C#. Оператор foreach. Оператор yield.**

**26. Понятие атрибутов в языке C#. Создание пользовательских атрибутов. Анализ атрибутов во время выполнения программы. Понятие рефлексии (reflection) в языке C#.Сериализация объектов.**