ООП в языке Delphi

1. Парадигмы объектно-ориентированного программирования: инкапсуляция, наследование, полиморфизм.

Объектно-ориентированное программирование основано на идеях объектно-ориентированной парадигмы (ООП), основными принципами которой являются:
- Инкапсуляция (объединение данных и обрабатывающих методов внутри объекта. Тип объекта называется классом. Класс представляет собой единство трех сущностей - полей, методов и свойств.);
- Наследование (процесс порождения новых объектов от уже существующих. При этом потомок берет от родителей все его поля, свойства и методы, которые потом можно оставить в неизменном виде или переопределить);
- Полиморфизм (свойство, которое позволяет методам родительских классов и потомков иметь одинаковые имена, но разное содержание. Выполнение каждого конкретного действия будет определяться типом данных).

2. Формула объекта. Природа объекта.

В их программном мире все является объектом (с некоторыми исключениями в языке Delphi), а программы представляют собой взаимодействие (или разговор) объектов между собой.
Формула объекта:
Объект = Данные + Операции
На основании этой формулы была разработана методология объектно-ориентированного программирования (ООП).
Природа объекта:
В общем случае каждый объект "помнит" необходимую информацию, "умеет" выполнять некоторый набор действий и характеризуется набором свойств. То, что объект "помнит", хранится в его полях. То, что объект "умеет делать", реализуется в виде его внутренних процедур и функций, называемых методами. Свойства объектов аналогичны свойствам, которые мы наблюдаем у обычных предметов. Значения свойств можно устанавливать и читать. Программно свойства реализуются через поля и методы.

3. Классы. Объекты.

ООП — идеология программирования, основанная на объединении данных и процедур, которые могут работать с этими данными в совокупности, называемые классами. Сутью ООП является использование привычной нам в обыденной жизни объектной модели. Каждый объект имеет свои свойства и с ним можно совершить характерные для него действия. Класс — это тип объекта. Класс описывает и реализует те самые свойства и действия. Объект в нашем понимании будет являться переменной, типом, которым и будет являться какой-то класс. Полями класса - его свойства, а методы — действия, которые можно совершить с экземпляром этого класса (объектом). В некоторой степени классы можно сравнить с обыкновенными записями, однако классы имеют гораздо больше возможностей.
Описание класса:

```
type
TBook = class
public
  PagesCount: integer;
  Title, Author: string;
  function CompareWithBook(OtherBook: TBook): integer;
  procedure ShowTitle;

  constructor Create(NewTitle, NewAuthor: string; NewPagesCount: integer);
end;
```

Создание и удаление объекта:

```
var
  MyBook: TBook;
...
MyBook := TBook.Create('Delphi для начинающих', 'Cyberexpert', 1000);
MyBook.Destroy;
```

4.Конструкторы и деструкторы. Методы.

Методы — это процедуры и функции, которые принадлежат объекту. С помощью своих методов объект выполняет возложенные на него обязанности. Методы описываются внутри объявления объекта и бывают нескольких типов: static (статические) — это простые процедуры и функции; virtual (виртуальные) — такие методы могут быть переопределены в потомках объекта; dynamic (динамические) — эти методы схожи с виртуальными, но для определения адреса используется другой способ; message (сообщения) — такие методы реагируют на события операционной системы; abstract (абстрактный) — такой метод будет только объявлен в объекте, а реализации у него не будет. Если вы объявили метод как virtual или dynamic, то можно переопределить их действия в наследниках.
Методы, которые предназначены для создания и удаления объектов называются конструкторами и деструкторами соответственно. Описание данных методов отличается от обычных тем, что в их заголовках стоят ключевые слова constructor и destructor. В качестве имен конструкторов и деструкторов в базовом классе TObject и многих других классах используются имена Create и Destroy.
Прежде чем обращаться к элементам объекта, его нужно создать с помощью конструктора. Например: ObjectA:=TOwnClass.Create;
Create – это конструктор объекта; он всегда присутствует в классе и служит для создания и инициализации экземпляров. При создании объекта в памяти выделяется место только для его полей. Методы, как и обычные процедуры и функции, помещаются в область кода программы; они могут работать с любыми экземплярами своего класса и не дублируются в памяти. Конструктор выделяет память для нового объекта, задает нулевые значения для порядковых полей, значение nil – для указателей и полей классов, строковые поля устанавливает пустыми, а также возвращает указатель на созданный объект. После создания объект можно использовать в программе: получать и устанавливать значения его полей, вызывать его методы.
Если объект становится ненужным, он должен быть удален вызовом специального метода Destroy (освобождение памяти, занимаемой объектом), например: ObjectA.Destroy;
Destroy – это деструктор объекта; он присутствует в классе наряду с конструктором и служит для удаления объекта из динамической памяти. После вызова деструктора переменная math становится несвязанной и не должна использоваться для доступа к полям и методам уже несуществующего объекта. Чтобы отличать в программе связанные объектные переменные от несвязанных, последние следует инициализировать значением nil.
Вызов деструктора для несуществующих объектов недопустим и при выполнении программы приведет к ошибке. Чтобы избавить от лишних ошибок, в объекты ввели предопределенный метод Free, который следует вызывать вместо деструктора. Метод Free сам вызывает деструктор Destroy, но только в том случае, если значение объектной переменной не равно nil: ObjectA.Free;

5. Свойства. Понятие свойства. Методы получения и установки значений свойств.

Помимо полей и методов в объектах существуют свойства. При работе с объектом свойства выглядят как поля: они принимают значения и участвуют в выражениях. Но в отличие от полей свойства не занимают места в памяти, а операции их чтения и записи ассоциируются с обычными полями или методами. Это позволяет создавать необходимые сопутствующие эффекты при обращении к свойствам. Объявление свойства выполняется с помощью зарезервированного слова property.

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 обращение при записи (установке) значения свойства. Чтобы имена свойств не совпадали с именами полей, последние принято писать с буквы F (от англ. field).
Обращение к свойствам выглядит в программе как обращение к полям.
Если один из спецификаторов доступа опущен, то значение свойства можно либо только читать (задан спецификатор read), либо только записывать (задан спецификатор write). В следующем примере объявлено свойство, значение которого можно только читать.
Методы получения (чтения) и установки (записи) значений свойств подчиняются определенным правилам. Метод чтения свойства — это всегда функция, возвращающая значение того же типа, что и тип свойства. Метод записи свойства — это обязательно процедура, принимающая параметр того же типа, что и тип свойства. В остальных отношениях это обычные методы объекта.
Использование методов для получения и установки свойств позволяет проверить корректность значения свойства, сделать дополнительные вычисления, установить значения зависимых полей и т.д.

```
type TDelimitedReader = class
  FActive: Boolean;
  ...
  procedure SetActive(const AActive: Boolean);
  function GetItemCount: Integer;
  ...
  property Active: Boolean read FActive write SetActive;
end;
```

6. Свойства-массивы. Свойство-массив как основное свойство объекта. Методы, обслуживающие несколько свойств.
Кроме обычных свойств в объектах существуют свойства-массивы (array properties). Свойство-массив — это индексированное множество значений.
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 помощью цикла for.
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 только Integer (например, можно создать свойство-массив, в котором индексами будут строки); операции над свойством-массивом в целом запрещены; разрешены операции только с его элементами.
Свойство-массив можно сделать основным свойством объектов данного класса. Для этого в описание свойства добавляется слово default. Такое объявление свойства позволяет рассматривать сам объект класса как массив и опускать имя свойства-массива при обращении к нему из программы. Следует помнить, что только свойства-массивы могут быть основными свойствами объектов; для обычных свойств это недопустимо.
Один и тот же метод может использоваться для получения (установки) значений нескольких свойств одного типа. В этом случае каждому свойству назначается целочисленный индекс, который передается в метод чтения (записи) первым параметром.

```
type TDelimitedReader = class
  ...
  property Items[Index: Integer]: string read GetItem; default;
  ...
  property FirstName: string index 0 read GetItem;
  property LastName: string index 1 read GetItem;
  ...
end;
```

7. Наследование. Понятие наследования. Прародитель всех классов. Перекрытие атрибутов в наследниках.
Классы инкапсулируют (т.е. включают в себя) поля, методы и свойства; это их первая черта. Следующая не менее важная черта классов — способность наследовать поля, методы и свойства других классов. Механизм наследования используется для того, чтобы избавиться от дублирования общих атрибутов (полей, свойств и методов) при определении новых классов.
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. 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.
В механизме наследования можно условно выделить три основных момента: наследование полей; наследование свойств; наследование методов. Наследование свойств и методов имеет свои особенности. 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. Метод базового класса тоже можно перекрыть в производном классе, например, чтобы изменить логику его работы. В наследнике можно вызвать перекрытый метод предка, указав перед именем метода зарезервированное слово inherited.

8. Совместимость объектов различных классов. Контроль и преобразование типов.
Для классов, связанных отношением наследования, вводится новое правило совместимости типов. Вместо объекта базового класса можно подставить объект любого производного класса. Обратное неверно. Правило совместимости классов чаще всего применяется при передаче объектов в параметрах процедур и функций.
Поскольку реальный экземпляр объекта может оказаться наследником класса, указанного при описании объектной переменной или параметра, бывает необходимо проверить, к какому классу принадлежит объект на самом деле. Чтобы программист мог выполнять такого рода проверки, каждый объект хранит информацию о своем классе. В языке Delphi существуют операторы is и as, с помощью которых выполняется соответственно проверка на тип (type checking) и преобразование к типу (type casting). Например, чтобы выяснить, принадлежит ли некоторый объект Obj к классу TClass или его наследнику, следует использовать оператор is. Для преобразования объекта к нужному типу используется оператор as.

9. Виртуальные методы. Понятие виртуального метода. Механизм вызова виртуальных методов.
Это методы, которые могут быть переопределены в потомках объекта. Для определения адреса Delphi строит таблицу виртуальных методов, которая позволяет во время выполнения программы определить адрес метода. В такой таблице хранятся все методы текущего объекта и его предка.
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.
Суть виртуальных методов в том, что они вызываются по фактическому типу экземпляра, а не по формальному типу, записанному в программе.

Работа виртуальных методов основана на механизме позднего связывания (late binding). В отличие от раннего связывания (early binding), 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.
Благодаря механизму наследования и виртуальных методов в среде Delphi реализуется такая концепция ООП как полиморфизм. Полиморфизм существенно облегчает труд программиста, поскольку обеспечивает повторное использование кода уже написанных и отлаженных методов.
Работа виртуальных методов основана на косвенном вызове подпрограмм. При косвенном вызове команда вызова подпрограммы оперирует не адресом подпрограммы, а адресом места в памяти, где хранится адрес подпрограммы. Косвенный вызов также используется в процедурных переменных. Процедурная переменная и есть то местом в памяти, где хранится адрес вызываемой подпрограммы. Для каждого виртуального метода тоже создается процедурная переменная, но ее наличие и использование скрыто от программиста.
Все процедурные переменные с адресами виртуальных методов пронумерованы и хранятся в таблице, называемой таблицей виртуальных методов (VMT — от англ. Virtual Method Table). Такая таблица создается одна для каждого класса объектов, и все объекты этого класса хранят на нее ссылку.
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.

10.Абстрактные виртуальные методы. Динамические методы. Методы обработки сообщений.
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 (так мы и поступили), 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.
Разновидностью виртуальных методов являются так называемые динамические методы. При их объявлении вместо ключевого слова virtual записывается ключевое слово dynamic. В наследниках динамические методы перекрываются так же, как и виртуальные — с помощью зарезервированного слова override. По смыслу динамические и виртуальные методы идентичны. Различие состоит только в механизме их вызова. Методы, объявленные с директивой virtual, вызываются максимально быстро, но платой за это является большой размер системных таблиц, с помощью которых определяются их адреса. Размер этих таблиц начинает сказываться с увеличением числа классов в иерархии. Методы, объявленные с директивой dynamic вызываются несколько дольше, но при этом таблицы с адресами методов имеют более компактный вид, что способствует экономии памяти. Таким образом, программисту предоставляются два способа оптимизации объектов: по скорости работы (virtual) или по объему памяти (dynamic).
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 к специальному методу Dispatch, который имеется в каждом классе (метод Dispatch определен в классе TObject). Методы обработки сообщений применяются внутри библиотеки VCL для обработки команд пользовательского интерфейса и редко нужны при написании прикладных программ.

11. Классы в программных модулях. Разграничение доступа к атрибутам объектов. Указатели на методы объектов.
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 скрыть особенности реализации класса, но в отличие от нее разрешает другим программистам порождать новые классы и обращаться к полям, методам и свойствам, которые составляют так называемый интерфейс разработчика. В эту секцию обычно помещаются 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 видимости published.
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++.
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. Переменная такого типа называется указателем на метод (method pointer). Она занимает в памяти 8 байт и хранит одновременно ссылку на объект и адрес его метода.
Описанный 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 делегируют обработку своих событий той форме, в которую они помещены.

12. Метаклассы. Ссылки на классы. Методы классов.
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 (class reference). Он описывается с помощью словосочетания class of.
По аналогии с тем, как для всех классов существует общий предок TObject, у ссылок на классы существует базовый тип TClass.
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.
Передаваемый в метод класса неявный параметр Self содержит не ссылку на объект, а ссылку на класс, поэтому в теле метода нельзя обращаться к полям, методам и свойствам объекта. Зато можно вызывать другие методы класса. Методы класса применимы и к классам, и к объектам. В обоих случаях в параметре Self передается ссылка на класс объекта. Методы классов могут быть виртуальными.

13. Виртуальные конструкторы. Информация о типе времени выполнения программы — RTTI.
Особая прелесть 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 происходит следующим образом. Из файла считывается имя класса. По этому имени отыскивается ссылка на класс (метакласс). У метакласса вызывается виртуальный конструктор, который создает объект нужного класса.
Информация о типах времени исполнения (Runtime Type Information, RTTI) — это данные, генерируемые компилятором Delphi о большинстве объектов вашей программы. RTTI представляет собой возможность языка, обеспечивающее приложение информацией об объектах (его имя, размер экземпляра, указатели на класс-предок, имя класса и т. д.) и о простых типах во время работы программы. Сама среда разработки использует RTTI для доступа к значениям свойств компонент, сохраняемых и считываемых из dfm-файлов и для отображения их в Object Inspector,
Компилятор Delphi генерирует runtime информацию для простых типов, используемых в программе, автоматически. Для объектов, RTTI информация генерируется компилятором для свойств и методов, описанных в секции published в следующих случаях:
1) Объект унаследован от объекта, дня которого генерируется такая информация. В качестве примера можно назвать объект TPersistent.
2) Декларация класса обрамлена директивами компилятора {$M+} и {$M-}.
Object Pascal предоставляет в распоряжение программиста два оператора, работа которых основана на неявном для программиста использовании RTTI информации. Это операторы is и as. Оператор is предназначен для проверки соответствия экземпляра объекта заданному объектному типу. Следует отметить, что определенная проверка происходит еще на этапе компиляции программы. если фактические объект и класс несовместимы, компилятор выдаст ошибку в этом операторе. Перейдем теперь к оператору as. Он введен в язык специально для приведения объектных типов. Посредством него можно рассматривать экземпляр объекта как принадлежащий к другому совместимому типу. Использование оператора as отличается от обычного способа приведения типов наличием проверки на совместимость типов. Так при попытке приведения этого оператора с несовместимым типом он сгенерирует исключение EInvalidCast.
Определенным недостатком операторов is и as является то, что присваиваемый фактически тип должен быть известен на этапе компиляции программы и поэтому на месте TSomeObjectType не может стоять переменная указателя на класс.

14. Ошибки и исключительные ситуации. Классы исключительных ситуаций. Обработка исключительных ситуаций.
В любом работающем приложении могут происходить ошибки. Причины этих ошибок бывают разными. Некоторые из них носят субъективный характер и вызваны неграмотными действиями программиста. Но существуют и объективные ошибки, их нельзя избежать при проектировании программы, но можно обнаружить во время ее работы. Примеров таких ошибок сколько угодно: недостаточный объем свободной памяти, отсутствие файла на диске, выход значений исходных данных из допустимого диапазона и т.д.
Хорошая программа должна справляться со своими ошибками и работать дальше, не зацикливаясь и не зависая ни при каких обстоятельствах. Для обработки ошибок можно, конечно, пытаться использовать структуры вида if <error> then Exit. Однако в этом случае ваш стройный и красивый алгоритм решения основной задачи обрастет уродливыми проверками так, что через неделю вы сами в нем не разберетесь. Из этой почти тупиковой ситуации среда Delphi предлагает простой и элегантный выход — механизм обработки исключительных ситуаций.
Исключительная ситуация (exception) — это прерывание нормального хода работы программы из-за невозможности правильно выполнить последующие действия.
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 их своевременно обнаруживать. Благодаря механизму обработки исключительных ситуаций обеспечивается связь между библиотекой и использующей ее программой при обработке ошибок.
Механизм обработки исключительных ситуаций довольно сложен в своей реализации, но для программиста он прост и прозрачен. Для его использования в язык Delphi введены специальные конструкции try...except...end, try...finally...end и оператор raise.
Исключительные ситуации в языке Delphi описываются классами. Каждый класс 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. Например, класс EOutOfMemory порожден от Exception и описывает ситуацию, когда свободная оперативная память исчерпана.
Стандартные классы исключительных ситуаций объявлены в модуле SysUtils. Они покрывают практически весь спектр возможных ошибок. Если их все-таки окажется недостаточно, вы можете объявить новые классы исключительных ситуаций, порожденные от класса Exception или его наследников.

15. Создание исключительной ситуации. Распознавание класса исключительной ситуации.
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. Вызывающие подпрограммы могут эту исключительную ситуацию перехватить и обработать. Для этого в них организуется так называемый защищенный блок. 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.
Распознавание класса исключительной ситуации выполняется с помощью конструкций
on <класс исключительной ситуации> do <оператор>;
которые записываются в секции обработки исключительной ситуации.
Поиск соответствующего обработчика выполняется последовательно до тех пор, пока класс исключительной ситуации не окажется совместимым с классом, указанным в операторе on. Как только обработчик найден, выпоняется оператор, стоящий за словом do и управление передается за секцию except...end. Если исключительная ситуация не относится ни к одному из указанных классов, то управление передается во внешний блок try...except...end и обработчик ищется в нем. Обратите внимание, что порядок операторов on имеет значение, поскольку распознавание исключительных ситуаций должно происходить от частных классов к общим классам, иначе говоря, от потомков к предкам.

16. Возобновление исключительной ситуации. Доступ к объекту, описывающему исключительную ситуацию.
В тех случаях, когда защищенный блок не может обработать исключительную ситуацию полностью, он выполняет только свою часть работы и возобновляет исключительную ситуацию с тем, чтобы ее обработку продолжил внешний защищенный блок. Если ни один из внешних защищенных блоков не обработал исключительную ситуацию, то управление передается стандартному обработчику исключительной ситуации, завершающему приложение.
При обработке исключительной ситуации может потребоваться доступ к объекту, описывающему эту ситуацию и содержащему код ошибки, текстовое описание ошибки и т.д. В этом случае используется расширенная запись оператора on:
on <идентификатор объекта> : <класс исключительной ситуации> do <оператор>;
Например, объект исключительной ситуации нужен для того, чтобы выдать пользователю сообщение об ошибке:

```
try
  // защищаемые операторы
except
  on E: EOutOfMemory do
    ShowMessage(E.Message);
end;
```
Переменная E — это объект исключительной ситуации, ShowMessage — процедура модуля DIALOGS, отображающая на экране небольшое окно с текстом и кнопкой OK. Свойство Message типа string определено в классе Exception, оно содержит текстовое описание ошибки. Исходное значение для текста сообщения указывается при конструировании объекта исключительной ситуации. Обратите внимание, что после обработки исключительной ситуации освобождение соответствующего объекта выполняется автоматически, вам этого делать не надо.

17. Защита от утечки ресурсов. Приемы надежного программирования.

Программы, построенные с использованием механизма исключительных ситуаций, обязаны придерживаться строгих правил распределения и освобождения таких ресурсов, как память, файлы, ресурсы операционной системы. 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.

Блок try...finally...end обладает еще одной важной особенностью. Если он помещен в цикл, то вызов из защищенного блока процедуры Break с целью преждевременного выхода из цикла или процедуры Continue с целью перехода на следующую итерацию цикла сначала обеспечивает выполнение секции finally...end, а затем уже выполняется соответствующий переход. Это утверждение справедливо также и для процедуры Exit (выход из подпрограммы).

Как показывает практика, подпрограммы часто распределяют сразу несколько ресурсов и используют их вместе. В таких случаях применяются вложенные блоки try...finally...end. Кроме того, вы успешно можете комбинировать блоки try...finally...end и try...except...end для защиты ресурсов и обработки исключительных ситуаций.

18. Понятие интерфейса. Описание интерфейса. Расширение интерфейса. Глобально-уникальный идентификатор интерфейса.

Если из объекта убрать поля и код всех методов, останется лишь интерфейс — заголовки методов и описания свойств. Схематично понятие интерфейса можно представить в виде формулы:

Интерфейс = Объект – Реализация

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).

Приведем пример интерфейса и сразу заметим, что интерфейсам принято давать имена, начинающиеся с буквы I (от англ. Interface):
```
type
  ITextReader = interface
  // Методы
  function NextLine: Boolean;
  // Свойства
  property Active: Boolean;
  property ItemCount: Integer;
  property Items[Index: Integer]: string;
  property EndOfFile: Boolean;
end;
```
Интерфейс ITextReader предназначен для считывания табличных данных из текстовых источников. Зачем вообще нужен интерфейс для доступа к табличным данным, если уже есть готовый класс TTextReader с требуемой функциональностью? Объяснение состоит в следующем. Не определив интерфейс ITextReader, невозможно разместить класс TTextReader в DLL-библиотеке и обеспечить доступ к нему из EXE-программы. Создавая DLL-библиотеку, мы с помощью оператора uses должны включить модуль ReadersUnit в проект библиотеки. Создавая EXE-программу, мы должны включить модуль ReadersUnit и в нее, чтобы воспользоваться описанием класса TTextReader. Но тогда весь программный код класса попадет внутрь EXE-файла, а это именно то, от чего мы хотим избавиться. Решение проблемы обеспечивается введением понятия интерфейса.

Поскольку интерфейс не может содержать поля, все его свойства отображены на его методы.

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.

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 (Globally Unique Identifier — GUID).

Глобально-уникальный идентификатор — это 16-ти байтовое число, представленное в виде заключенной в фигурные скобки последовательности шестнадцатеричных цифр:

    {DC601962-28E5-4BF7-9583-0CE22B605045}

В среде Delphi глобально-уникальный идентификатор описывается типом данных TGUID.

Генерация глобально-уникальных идентификаторов осуществляется системой Windows по специальному алгоритму, в котором задействуется адрес сетевого адаптера, текущее время и генератор случайных чисел. Можете смело полагаться на уникальность всех получаемых идентификаторов.
Наличие глобально-уникального идентификатора в описании интерфейса не является обязательным, однако использование интерфейса без такого идентификатора ограничено, например, запрещено использовать оператор as для преобразования одних интерфейсов в другие.

19. Реализация интерфейса. Использование интерфейса. Реализация нескольких интерфейсов.
Интерфейс бесполезен до тех пор, пока он не реализован. Реализацией интерфейса занимается класс. Если класс реализует интерфейс, то интерфейс может использоваться для доступа к объектам этого класса. При объявлении класса имя реализуемого интерфейса записывается через запятую после имени базового класса:
```
type
  TTextReader = class(TObject, ITextReader)
    ...
end;
```
Такая запись означает, что класс TTextReader унаследован от класса TObject и реализует интерфейс ITextReader.
Класс, реализующий интерфейс, должен содержать код для всех методов интерфейса. Если класс содержит только часть методов интерфейса, то недостающие методы придется добавить.
Для доступа к объекту через интерфейс нужна интерфейсная переменная:
```
var
  Intf: ITextReader;
```
Интерфейсная переменная занимает в оперативной памяти четыре байта, хранит ссылку на интерфейс объекта и автоматически инициализируется значением nil.
Перед использованием интерфейсную переменную инициализируют значением объектной переменной:
```
var
  Obj: TTextReader;  // объектная переменная
  Intf: ITextReader; // интерфейсная переменнаябegin
  ...
  Intf := Obj;
  ...
end;
```
После инициализации интерфейсную переменную Intf можно использовать для вызова методов объекта Obj. Через интерфейсную переменную доступны только те методы и свойства объекта, которые есть в интерфейсе.
Один класс может содержать реализацию нескольких интерфейсов. Такая возможность позволяет воплотить в классе несколько понятий. Например, класс TTextReader — "считыватель табличных данных" — может выступить еще в одной роли — "считыватель строк". Для этого он должен реализовать интерфейс IStringIterator:
```
type
  IStringIterator = interfacefunction Next: string;
    function Finished: Boolean;
end;
```

20. Реализация интерфейса несколькими классами. Связывание методов интерфейса с методами класса. Реализация интерфейса вложенным объектом.
Несколько совершенно разных классов могут содержать реализацию одного и того же интерфейса. С объектами таких классов можно работать так, будто у них есть общий базовый класс. Интерфейс выступает аналогом общего базового класса.
```
type
  TTextReader = class(TInterfacedObject, ITextReader, IStringIterator)
    ...
  end;
  TIteratableStringList = class(TStringList, IStringIterator)
    ...
end;
```
Метод интерфейса связывается с методом класса по имени. Если имена по каким-то причинам не совпадают, то можно связать методы явно с помощью специальной конструкции языка Delphi.
```
type
  TTextReader = class(TInterfacedObject, ITextReader, IStringIterator)
    ...
    function NextItem: string;
    function IStringIterator.Next := NextItem; // Явное связывание
end;
```
При работе с объектами класса TTextReader через интерфейс IStringIterator вызов метода Next приводит к вызову метода NextItem:
```
var
  Obj: TTextReader;
  Intf: IStringIterator;
begin
  ...
  Intf := Obj;
  Intf.Next; // -> Obj.NextItem;
  ...
end;
```
Очевидно, что связываемые методы должны совпадать по сигнатуре (списку параметров и типу возвращаемого значения).
Случается, что реализация интерфейса содержится во вложенном объекте класса. Тогда не требуется программировать реализацию интерфейса путем замыкания каждого метода интерфеса на соответствующий метод вложенного объекта. Достаточно делегировать реализацию интерфейса вложенному объекту с помощью директивы implements:
```
type
  TTextParser = class(TInterfacedObject, ITextReader)
    ...
    FTextReader: ITextReader;
    property TextReader: ITextReader read FTextReader implements ITextReader;
    ...
end;
```
В этом примере интерфейс ITextReader в классе TTextParser реализуется не самим классом, а его внутренней переменной FTextReader.
Очевидно, что внутренний объект должен быть совместим с реализуемым интерфейсом.

21. Совместимость интерфейсов. Совместимость класса и интерфейса. Получение интерфейса через другой интерфейс.

Совместимость интерфейсов подчиняется определенным правилам. Если интерфейс создан расширением уже существующего интерфейса:

```
type
  IExtendedTextReader = interface(ITextReader)
    ...
end;
```

то интерфейсной переменной базового типа может быть присвоено значение интерфейсной переменной производного типа:

```
var
  Reader: ITextReader;
  ExtReader: IExtendedTextReader;
begin
  ...
  Reader := ExtReader; // Правильно
  ...
end;
```

Но не наоборот:

```
ExtReader := Reader; // Ошибка!
```

Правило совместимости интерфейсов чаще всего применяется при передаче параметров в процедуры и функции. Например, если процедура работает с переменными типа ITextReader,

```
    procedure LoadFrom(const R: ITextReader);
```

то ей можно передать переменную типа IExtendedTextReader:

```
LoadFrom(ExtReader);
```

Заметим, что любая интерфейсная переменная совместима с типом данных IInterface — прародителем всех интерфейсов.

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 переменнаябegin
  ...
  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:

```
type
  TTextReader = class(TInterfacedObject, ITextReader)
    ...
  end;

  TDelimitedReader = class(TTextReader)
    ...
  end;
var
  Intf: ITextReader;        // интерфейсная переменная
  Obj: TDelimitedReader;    // объектная переменнаябegin
  ...
  Intf := Obj;
  ...
end;
```

Однако, если класс реализует производный интерфейс, то это совсем не означает, что он совместим с базовым интерфейсом:

```
type
  ITextReader = interface(IInterface)
    ...
  end;
  IExtendedTextReader = interface(ITextReader)
    ...
  end;
  TExtendedTextReader = class(TInterfacedObject, IExtendedTextReader)
    ...
  end;
var
  Obj: TExtendedTextReader;
  Intf: ITextReader;
begin
  ...
  Intf := Obj;  // Ошибка! Класс TExtendedTextReader не реализует// интерфейс ITextReader.
  ...
end;
```

Для совместимости с базовым интерфейсом нужно реализовать этот интерфейс явно:

```
type
  TExtendedTextReader = class(TInterfacedObject, ITextReader, IExtendedTextReader)
    ...
end;
```

Теперь класс TExtendedTextReader совместим и с интерфейсом ITextReader, поэтому следующее присваивание корректно:

```
  Intf := Obj;
```

Исключением из только что описанного правила является совместимость всех снабженных интерфейсами объектов с интерфейсом IInterface:

```
var
  Obj: TExtendedTextReader;
  Intf: IInterface;
```

```
begin
  ...
  Intf := Obj;  // Правильно, IInterface – особый интерфейс.
  ...
end;
```
Через интерфейсную переменную у объекта всегда можно запросить интерфейс другого типа. Для этого используется оператор as, например:
```
var
  Intf: IInterface;
begin
  ...
  with Intf 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, что метод QueryInterface описан в интерфейсе IInterface и попадает автоматически во все интерфейсы. Стандартная реализация этого метода находится в классе TInterfacedObject.

22. Механизм подсчета ссылок. Представление интерфейса в памяти.

Механизм подсчета ссылок на объект предназначен для автоматического уничтожения неиспользуемых объектов. Неиспользуемым считается объект, на который не ссылается ни одна интерфейсная переменная.
Подсчет ссылок на объект обеспечивают методы _AddRef и _Release интерфейса IInterface. При копировании значения интерфейсной переменной вызывается метод _AddRef, а при уничтожении интерфейсной переменной — метод _Release. Вызовы этих методов генерируются компилятором автоматически:
```
var
  Intf, Copy: IInterface;
begin
  ...
  Copy := Intf; // Copy._Release; Intf._AddRef;
  Intf := nil;  // Intf._Release; end;          // Copy._Release
```
Стандартная реализация методов _AddRef и _Release находится в классе TInterfacedObject.
Обратите внимание, что объектные переменные не учитываются при подсчете ссылок. Поэтому мы настоятельно рекомендуем избегать смешивания интерфейсных и объектных переменных. Если вы планируете использовать объект через интерфейс, то лучше всего результат работы конструктора сразу присвоить интерфейсной переменной:
Если интерфейс является входным параметром подпрограммы, то при вызове подпрограммы создается копия интерфейсной переменной с вызовом метода _AddRef:
Копия не создается, если входной параметр описан с ключевым словом const.
Интерфейсная переменная уничтожается при выходе из области действия переменной, а это значит, что у нее автоматически вызывается метод _Release:
Глубокое понимание работы интерфейсов требует знания их технической реализации. Поэтому вам необходимо разобраться в том, как представляется интерфейс в оперативной памяти компьютера, и что стоит за операторами Intf := Obj и Intf.NextLine.
Интерфейс по сути выступает дополнительной таблицей виртуальных методов, ссылка на которую укладывается среди полей объекта. Эта таблица называется таблицей методов интерфейса. В ней хранятся указатели на методы класса, реализующие методы интерфейса.
Интерфейсная переменная хранит ссылку на скрытое поле объекта, которое содержит указатель на таблицу методов интерфейса. Когда интерфейсной переменной присваивается значение объектой переменной,
```
Intf := Obj; // где Intf: ITextReader и Obj: TTextReader
```
к адресу объекта добавляется смещение до скрытого поля внутри объекта и этот результат заносится в интерфейсную переменную. Чтобы убедиться в сказанном, посмотрите в отладчике значения Pointer(Obj) и Pointer(Intf) сразу после выполнения оператора Intf := Obj. Эти значения будут разными! Причина в том, что объектная ссылка указывает на начало объекта, а интерфейсная ссылка — на скрытое поле внутри объекта.
Алгоритм вызова метода интерфейса такой же, как алгоритм вызова метода класса. Когда через интерфейсную переменную выполняется вызов метода,
```
Intf.NextLine;
```
реализуется следующий алгоритм:
1) Из интерфейсной переменной извлекается адрес (по нему хранится адрес таблицы методов интерфейса);
2) По полученному адресу извлекается адрес таблицы методов интерфейса;
3) На основании порядкового номера метода в интерфейсе из таблицы извлекается адрес соответствующей подпрограммы;
4) Вызывается код, находящийся по этому адресу. Этот код является переходником от метода интерфейса к методу объекта. Его задача — восстановить из ссылки на интерфейс значение указателя Self (путем вычитания заранее известного значения) и выполнить прямой переход на код метода класса.
Вся эта сложность скрыта в языке Delphi за понятием интерфейса. Причем несмотря на такое количество операторов в примере, вызов метода через интерфейс в машинном коде выполняется весьма эффективно (всего несколько инструкций процессора), поэтому в подавляющем большинстве случаев потерями на вызов можно пренебречь.

23. Применение интерфейса для доступа к объекту DLL-библиотеки.

Если вы поместите свой класс в DLL-библиотеку, то при необходимости использовать его в главной программе столкнетесь с проблемой. Подключение модуля с классом к главной программе приведет к включению в нее кода всех методов класса, т.е. задача выделения класса в DLL-библиотеку не будет решена. Если же не подключить модуль с описанием класса, главная программа вообще не будет знать о существовании класса, и воспользоваться классом будет невозможно.
Эта проблема решается с помощью интерфейсов.
Сначала вынесем описание интерфейса в отдельный модуль, чтобы этот модуль в дальнейшем можно было подключить к главной программе. Затем удалим описание интерфейса из первого модуля, а вместо него подключим модуль с интерфейсом. Наконец включим скорректированный модуль в DLL-библиотеку.
Вроде бы все готово, и теперь в главной программе достаточно подключить модуль и работать с объектами через интерфейс. Но постойте! А как в программе создавать объекты классов, находящихся в DLL-библиотеке? Ведь в интерфейсе нет методов для создания объектов! Для этого определим в DLL-библиотеке специальную функцию и экспортируем ее. В главной программе импортируйте эту функцию, чтобы с ее помощью создавать объекты класса.
.

ООП в языке С++

1. Принципы модульного программирования на языке С++. Пространства имен.

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.

Модульность в языке С++ поддерживается с помощью директив препроцессора, пространств имен, классов памяти, исключений и раздельной компиляции (строго говоря, раздельная компиляция не является элементом языка, а относится к его реализации).

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.

Поскольку пространства с глобальной областью видимости добавляются к системе, то имеется возможность возникновения коллизии имен. Это становится особенно актуальным при использовании библиотек, разработанных различными независимыми производителями. Использование namespace позволяет разбить глобальное пространство имен с тем, чтобы решить подобную про-блему. По существу, namespace определяет область видимости. Общая форма namespace представлена ниже:

```
namespace имя {
// объявление объекта
}
```

Кроме того, можно использовать безымянное пространство имен, как показано ниже:

```
namespace {
// объявление объекта
}
```

Безымянное пространство имен позволяет определить уникальность идентификаторов с областью видимости в пределах единственного файла.

Ниже приведен пример использования namespace:

```
namespace MyNameSpace {
int i, k;
void myfunc(int j) { cout << j; }
}
```

Здесь i, k и myfunc() составляют область видимости пространства имен MyNameSpace.

2. Перегрузка идентификаторов. Предопределенные аргументы в подпрограммах.

Имена, использующиеся для переменных, функций, меток и других определяемых пользователем объектов, называются идентификаторами. Идентификаторы могут состоять как из одного, так и из нескольких символов. Первым символом должна быть буква или знак подчеркивания, а за ним могут стоять буквы, числа или знак подчеркивания. В С прописные и строчные буквы трактуются по-разному. Следовательно, count, Count и COUNT — это три различных идентификатора. Идентификатор не может совпадать с ключевым словом С и не должен иметь такое же имя, как функция, уже содержащаяся в библиотеке С.

В С и других языках программирования один и тот же идентификатор может быть связан в данный момент времени с более чем одним объектом. Такая ситуация называется перегрузка имени (name overloading/name hiding). Создание двух объявлений одного имени в одном классе перегрузки в одном блоке видимости или на верхнем уровне является ошибкой.

Предопределенные аргументы в подпрограммах – это необязательные аргументы или аргументы, имеющие значение по умолчанию:

```
void Foo::bar(bool enable = true)
{
   // реализация метода
}
```

3. Классы в языке С++. Наследование. Конструкторы и деструкторы.

Классы в С++ — это абстракция описывающая методы, свойства, ещё не существующих объектов. Объекты — конкретное представление абстракции, имеющее свои свойства и методы. Созданные объекты на основе одного класса называются экземплярами этого класса. Эти объекты могут иметь различное поведение, свойства, но все равно будут являться объектами одного класса. В ООП существует три основных принципа построения классов:

- Инкапсуляция — это свойство, позволяющее объединить в классе и данные, и методы, работающие с ними и скрыть детали реализации от пользователя.
- Наследование — это свойство, позволяющее создать новый класс-потомок на основе уже существующего, при этом все характеристики класса родителя присваиваются классу-потомку.
- Полиморфизм — свойство классов, позволяющее использовать объекты классов с одинаковым интерфейсом без информации о типе и внутренней структуре объекта.

Конструктор (от слова construct — создавать) — специальная функция, которая выполняет начальную инициализацию элементов данных, причём имя конструктора обязательно должно совпадать с именем класса. Важным отличием конструктора от остальных функций является то, что он не возвращает значений вообще никаких, в том числе и void. В любом классе должен быть конструктор, даже если явным образом конструктор не объявлен (как в предыдущем классе), то компилятор предоставляет конструктор по умолчанию, без параметров.

В отличии от конструктора, деструктор (от слова destruct — разрушать) — специальный метод класса, который служит для уничтожения элементов класса. Чаще всего его используют тогда, когда в конструкторе, при создании объекта класса, динамически был выделен участок памяти и необходимо эту память очистить, если эти значения уже не нужны для дальнейшей работы программы.

```
classTTextReader
{
public:
TTextReader();
~TTextReader();
};
```

4. Создание объектов по значению (на стеке) и по ссылке (в динамической памяти). Стандартные конструкторы.
Создание объектов:
-по значению (на стеке):
TTextReaderreader;
-по ссылке (в динамической памяти):
*TTextReader*reader = new TTextReader(); //оператор new служит для размещения*
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 вызове деструктора
-если объект создан в динамической памяти (по ссылке), он должен быть уничтожен с помощью оператора delete:
delete reader;
При этом сначала происходит вызов деструктора, а затем — освобождение динамической памяти.
Так выглядит динамическое создание и разрушение объектов:
new:
malloc();
TTextReader();
delete:
~TTextReader();
free();
Если программист не определяет в классе конструкторы, то компилятор создает автоматически 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 не создаст никаких стандартных конструкторов. Конструктор без параметров создается для того, чтобы можно было написать:
TTextReader R;
Конструктор копирования нужен для следующей записи:
TTextReader R1 = R2; // означает TTextReaderR1(R2);
Конструктор копирования вызывается в том случае, когда создаваемый по значению объект создается 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 объекта другому (данный вариант требует перегрузки оператора ‘=’ для класса TTextReader). Работа стандартного конструктора копирования, создаваемого компилятором, заключается в том, чтобы выполнить полное копирование памяти с помощью функции memcpy.

5. Порядок конструирования и разрушения объектов. Агрегирование объектов.
В конструкторе производного класса конструктор базового класса вызывается автоматически до выполнения первого оператора в теле конструктора. В деструкторе производного класса деструктор базового класса вызывается автоматически после последнего оператора в теле деструктора. Если базовый класс содержит конструктор с параметрами или несколько конструкторов, то возникает неопределенность в том, какой конструктор базового класса будет вызван.
В объектно-ориентированном программировании под агрегированием (или как его еще называют - делегированием) подразумевают методику создания нового класса из уже существующих классов путём их включения. Об агрегировании также часто говорят как об «отношении принадлежности» по принципу «у машины есть корпус, колёса и двигатель».

6. Операторы new и delete. Размещающий оператор new.
C++ поддерживает динамическое выделение и освобождение памяти объектов с использованием операторов new и delete. Эти операторы выделяют память для объектов из пула, называемого свободным хранилищем. Оператор new вызывает специальную функцию operator new, а оператор delete вызывает специальную функцию operator delete.
Существует особая форма оператора new, называемая Placement new. Данный оператор не выделяет память, а получает своим аргументом адрес на уже выделенную каким-либо образом память (например, на стеке или через malloc()). Происходит размещение (инициализация) объекта путём вызова конструктора, и объект создается в памяти по указанному адресу. Часто такой метод применяют, когда у класса нет конструктора по умолчанию и при этом нужно создать массив объектов.

7. Вложенные определения классов. «Друзья» класса. Статические члены класса.
Класс, объявленный внутри другого класса, называется вложенным. Он является членом объемлющего класса, а его определение может находиться в любой из секций public, private или protected объемлющего класса.
Функция, которая не является членом класса, но получившая право доступа к его закрытой части, называется другом класса (friend). Функция становится другом класса после описания как friend.
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.

8. Множественное наследование. Проблема повторяющихся базовых классов.

В C++ поддерживается множественное наследование. В этом случае конструктор базовых классов вызывается автоматически в порядке их упоминания в описании класса. Деструктор же 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 C++ множественное наследование подразумевает, что у одного класса может быть несколько базовых классов:

```
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 методы каждого 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 можно осуществлять множественное наследование лишь от классов, в которых отсутствуют поля.

9. Типовой пример применения множественного наследования — observer.

Шаблон OBSERVER фиксирует связи между примитивными данными и их всевозможными представлениями следующим образом.
1. Каждый фрагмент данных инкапсулируется в некотором объекте Subject (субъект, который соответствует модели в MVC).
2. Каждый новый пользовательский интерфейс определенного субъекта инкапсулируется в некотором объекте Observer (наблюдатель, который соответствует представлению в MVC).
3. Субъект может иметь сразу несколько наблюдателей.
4. Изменяясь, субъект уведомляет об этом своих наблюдателей.
5. Наблюдатели, в свою очередь, опрашивают свои субъекты с целью получения информации, которая влияет на их внешний вид, и обновляются в соответствии с этой информацией.

Субъект хранит основную информацию, а наблюдатели обновляются всякий раз при изменении информации субъекта. Когда пользователь сохраняет результаты своей работы, сохраняется именно субъект; наблюдатели сохранять не нужно, поскольку отображаемая ими информация поступает из субъекта.

10. Виртуальные методы. Абстрактые методы и классы. Подстановочные функции.

В C++ 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 C++ 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.
Подставляемые или встраиваемые (inline) функции – это функции, код которых вставляется компилятором непосредственно на место вызова, вместо передачи управления единственному экземпляру функции.
Если функция является подставляемой, компилятор не создает данную функцию в памяти, а копирует ее строки непосредственно в код программы по месту вызова. Это равносильно вписыванию в программе соответствующих блоков вместо вызовов функций. Таким образом, спецификатор inline определяет для функции так называемое внутреннее связывание, которое заключается в том, что компилятор вместо вызова функции подставляет команды ее кода.
Подставляемые функции используют, если тело функции состоит из нескольких операторов.
Этот подход позволяет увеличить скорость выполнения программы, так как из программы исключаются команды микропроцессора, требующиеся для передачи аргументов и вызова функции.

11. Операторы приведения типа в языке C++. Информация о типе времени выполнения программы — RTTI.

Хотя C++ полностью поддерживает традиционное приведение типов языка C, для него определе-ны четыре дополнительных оператора приведения типов. Ими являются: const_cast, dynamic_cast, reinterpret_cast и static_cast. Общая форма записи имеет следующий вид:

```
const_cast <тип> (объект)
dynamic_cast <тип> (объект)
reinterpret_cast <тип> (объект)
static_cast <тип> (объект)
```

Здесь тип определяет целевой тип, а объект является объектом, приводимым к новому типу.
Оператор const_cast используется для явного переопределения const и/или volatile в операции приведения типа. Целевой тип должен быть тем же самым, что и исходный тип, за исключением его атрибута const или volatile. Наиболее общеупотребительным использованием const_cast является удаление атрибута const.

Оператор dynamic_cast выполняет приведение типов во время исполнения с проверкой возможности приведения типа. Если такое приведение не может быть сделано, то выражение принимает значение NULL. Его основным применением служит приведение полиморфных типов. В общем случае оператор dynamic_cast выполняет успешное приведение типа, если требуемое полиморфное приведение типа допустимо, т. е. если целевой тип может быть применен к исходному типу объек-та. Если приведение типа невозможно, то dynamic_cast приобретает значение NULL.
Оператор reinterpret_cast изменяет один тип в фундаментально иной тип. Например, он может быть использован для преобразования указателя в целое число.
Оператор reinterpret_cast предназначен для использования при приведении типов указателей, несопоставимых в отношении наследования.
Оператор static_cast выполняет не полиморфное приведение типов. Например, он может использоваться для приведения указателя на базовый класс к указателю на производный класс. Он также может использоваться для любого стандартного указателя. Никаких проверок времени ис-полнения не выполняется.
Только оператор const_cast может убрать атрибут const, что не может выполнить ни оператор dynamic_cast, ни static_cast, ни reinterpret_cast.
Используя идентификацию типа времени исполнения (run-time type identification — RTTI) можно определить тип объекта во время исполнения программы. Для этого используется функция typeid. Для использования этой функции необходимо включить заголовочный файл typeinfo.h. Общая форма записи функции typeid имеет следующий вид:
typeid (объект)
Здесь объект является объектом, чей тип требуется определить. Функция typeid возвращает ссылку на объект типа typeinfo, который описывает тип объекта объект. (В проекте стандарта C++ этот тип называется type_info.) Класс typeinfo определяет следующие публичные члены:
bool operator== (const typeinfo &ob) const;
bool operator!= (const typeinfo &ob) const;
bool before(const typeinfo &ob) const;
*const char *name() const;*
Перегруженные операторы == и != обеспечивают сравнение типов. Функция before() возвращает истину, если вызываемый объект стоит выше в иерархии объектов, чем объект, используемый в качестве параметра. Функция before() предназначена большей частью для внутреннего использо-вания. Возвращаемое ею значение не имеет ничего общего с иерархией классов или наследованием. Функция name() возвращает указатель на имя типа.
Когда функция typeid применяется к указателю на базовый класс полиморфного класса, она автоматически возвращает тип объекта, на который указывает указатель, в том числе любой класс, выведенный из базового класса. (Как уже говорилось, полиморфным классом называется класс, содержащий хотя бы одну виртуальную функцию.)

12. Ссылки. 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);
}

13. Обработка исключительных ситуаций на языке C++.
В C++ 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 C++ 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 C++ 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.

14. Защита от утечки ресурсов. Оболочечные объекты (auto_ptr).

Поскольку в C+ 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 C++ 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.

15. Перегрузка операторов. Перегрузка бинарных операторов.

Перегрузка операторов позволяет заменить смысл стандартных операторов (+, –, = и др.) для пользовательских типов данных. В C++ 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.

16. Перегрузка унарных операторов. Перегрузка операторов преобразования типа.

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 C++ 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.

17. Шаблоны функций. Перегрузка шаблонов функций.

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

18. Шаблоны классов. Специализации шаблонов. Создание новых типов данных на базе шаблонов.
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).
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.
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.

19. Стандартная библиотека шаблонов. Потоковый ввод-вывод.
Перечислим, что содержится в стандартной библиотеке шаблонов:
- Классы и шаблоны для организации потоков ввода/вывода
В языке C++ 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 C++ 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.

20. Строки. Контейнеры. Итераторы.

Стандартная библиотека включает в себя следующие разделы:
- Строки. Шаблоны строк в стиле C++. Также в этот раздел попадает часть библиотек для работы со строками и символами в стиле C.
- Стандартные контейнеры. В стандартную библиотеку входят шаблоны для следующих контейнеров: одномерные массивы, списки, одно- и двунаправленные очереди, стеки, ассоциативные массивы, множества, очереди с приоритетом.
- Итераторы. Обеспечивают шаблоны итераторов, с помощью которых в стандартной библиотеке реализуется стандартный механизм группового применения алгоритмов обработки данных к элементам контейнеров.

21. Алгоритмы. Утилиты. Диагностика. Локализация.
Стандартная библиотека включает в себя следующие разделы:
- Алгоритмы. Шаблоны для описания операций обработки, которые с помощью механизмов стандартной библиотеки могут применяться к любой последовательности элементов, в том числе к элементам в контейнерах. Также в этот раздел входят описания функций bsearch() и qsort() из стандартной библиотеки C.
- Основные утилиты. В этот раздел входит описание основных базовых элементов, применяемых в стандартной библиотеке, распределителей памяти и поддержка времени и даты в стиле C.
- Диагностика. Определения ряда исключений и механизмов проверки утверждений во время выполнения (assert). Поддержка обработки ошибок в стиле C.
- Локализация. Определения, используемые для поддержки национальных особенностей и форматов представления (дат, валют и т. д.) в стиле C++ и в стиле C.

22 Определения для языка программирования C++. Числовые шаблоны.
???
Шабло́ны (англ. template) — средство языка C++, предназначенное для кодирования обобщённых алгоритмов, без привязки к некоторым параметрам (например, типам данных, размерам буферов, значениям по умолчанию).
В C++ возможно создание шаблонов функций и классов.
Стандартная библиотека включает в себя раздел числа – определения для работы с комплексными числами, математическими векторами, поддержка общих математических функций, генератор случайных чисел.

23. Перспективные технологии объектно-ориентированного программирования.
Объектно-ориентированное программирование (ООП) - это метод программирования, при использовании которого главными элементами программ являются объекты.
Современными языками объектно-ориентированного программирования являются C++ и Java. С середины 90-х годов многие объектно–ориентированные языки реализуются как системы визуального программирования, в которых интерфейсная часть программного продукта создается в диалоговом режиме, практически без написания программных операторов. К объектно – ориентированным системам визуального проектирования относятся Visual Basic, Delphi, C++ Builder, Visual C++.
От любого метода программирования мы ждем, что он поможет нам в решении наших проблем. Но одной из самых значительных проблем в программировании является сложность. Чем больше и сложнее программа, тем важнее становится разбить ее на небольшие, четко очерченные части. Чтобы побороть сложность, мы должны абстрагироваться от мелких деталей. В этом смысле классы представляют собой весьма удобный инструмент.
ООП дает возможность создавать расширяемые системы (extensible systems). Это одно из самых значительных достоинств ООП и именно оно отличает данный подход от традиционных методов программирования. Расширяемость (extensibility) означает, что существующую систему можно заставить работать с новыми компонентами, причем без внесения в нее каких-либо изменений. Компоненты могут быть добавлены на этапе выполнения.
Многоразового использования программного обеспечения на практике добиться не удается из-за того, что существующие компоненты уже не отвечают новым требованиям. ООП помогает этого достичь без нарушения работы уже имеющихся клиентов, что позволяет нам извлечь максимум из многоразового использования компонент.
Выживет ли объектно-ориентированное программирование, или оно лишь модное поветрие, которое скоро исчезнет?
Классы нашли свое место в большинстве современных языков программирования. Одно лишь это говорит о том, что им суждено остаться. Классы в самом ближайшем будущем войдут в стандартный набор концепций для каждого программиста, точно так же, как многие сегодня применяют динамические структуры данных и рекурсию, которые двадцать лет назад были также в диковинку. В то же время классы — это просто еще одна новая конструкция наряду с остальными. Нам нужно узнать, для каких ситуаций они подходят, и только здесь мы и будем их использовать. Правильно выбрать инструмент для конкретной задачи — обязательно для каждого мастерового и в еще большей степени для каждого инженера.
ООП ввергает многих в состояние эйфории. Пестрящая тут и там реклама сулит нам невероятные вещи, и даже некоторые исследователи, похоже, склонны рассматривать ООП как панацею, способную решить все проблемы разработки программного обеспечения. Со временем эта эйфория постепенно уляжется. И после периода разочарования люди, быть может, перестанут уже говорить об ООП, точно также как сегодня вряд ли от кого можно услышать о структурном программировании. Но классы будут использовать как нечто само собой разумеющееся, и мы сможем, наконец, понять, что они собой представляют: просто компоненты, которые помогают строить модульное и расширяемое программное обеспечение.