**Наследование.**

Хочется, чтобы отдельные части программы были блоками, причем правильно работающими блоками. Повышается уровень абстракции, меньше вещей надо держать в голове. В ООП хотелось бы не только блоки делать, но и

1) при создании блока похожего на уже существующий блок, хотелось бы просто дописать, не переделывая все с нуля и пользоваться

2) взаимозаменяемость блоков – мы привыкли что в жизни много взаимозаменяемого, в розетку в стене можно воткнуть разные электроприборы, причем, когда разрабатывали розетку даже не имели понятия о том, какие будут приборы в будущем, что появятся ноутбуки с блоками питания, мощные настольные ПК, электрогрили и прочее. То есть хочется похожи штуки реализовывать в ООП и программировании.

Связывание (binding). Связывание – это процесс сопоставления имени и адреса в памяти, где лежит код (например, у функции). Обычно такое происходит на стадии линковки, вместо имён функций подставлялись их адреса. С введением наследования кое-что изменилось. Не всегда на этапе компиляции понятно, какой адрес подставлять. Динамическое связывание означает связывание на этапе выполнения программы. Virtual означает, что для этой функции нужно использовать динамическое связывание. Как раз если virtual не написать у базового класса в объявлении функции, то связывание будет статическое и будет вызываться всегда функция базового класса, т.к. указатель мы сделали именно на базовый класс. Фокус с виртуальными функциями называется полиморфизм. По умолчанию в с++ используется статическое связывание. Для использования динамического указываем virtual.

Ключевой момент – чтобы класс можно было расширять, разработчик класса должен позаботиться и внимательно отнестись к его проектировке, где нужно должно стоять protected, где нужно virtual. Почему не ставить всегда везде protected? Нужен хороший пример, но идея такова: не хочется даже наследникам классам предоставлять доступ к некоторым полям и методам, чтобы автор производного класса не мог сломать важные вещи. Плюс вы сможете переделать в базовом классе свои приватные вещи, а производному не надо будет заботиться об этом, он просто скачает ваш переделанный базовый класс.

**Полиморфизм.**

List <- DoubleList. Идея в том, что наследник (производный) класс двусвязный. Причина того, что мы использовали именно наследование, т.к. хотели новый блок построить на основе имеющегося. Оказалось, что у метода addValue пришлось менять поведение у наследника, пришлось применять перекрытие (override). Для того, чтобы метод перекрыть, мы написали слово virtual в базовом классе. Также мы выяснили, что производный класс является в некотором смысле и базовым тоже, поэтому можно передавать объект наследника в некоторые функции, где в качестве параметра ожидается, казалось бы, объект базового класса. Это пример RTTI. Сегодня разберёмся, как это работает.

Связывание – процесс замены имени функции на её адрес для выполнения последующих вызовов. До этого момента мы говорили, что этот процесс происходит на стадии линковки. И такой вид связывания называется статическое связывание (static binding). С виртуальными функциями такого не получается. Связать можно только в момент выполнения программы. Какую из 2 / 3 / 4 функций вызывать в момент выполнения? Напомним, что по умолчанию всегда используется статическое связывание. Если появляется слово virtual, то используется динамическое связывание (dynamic binding). **Как это работает?**

Если в классе есть виртуальная функция, то с классом связывается специальная таблица виртуальных функций на этапе компиляции. Детали реализации упустим, это таблица: первый столбец название функции, второй столбец – адрес. Может быть такое, что от начала объекта первые четыре байта – это указатель на таблицу виртуальных функций.

Если компилятор видит инструкцию “l -> addValue (5);” , он понимает, что эта функция виртуальная и на этапе компиляции он заменяет её на конструкцию вида “Сходи в таблицу виртуальных функций объекта l (указатель – это первые 4 байта), найди функцию addValue и вызови её по адресу из таблицы”. У объекта типа List и у объекта типа DoubleList эти таблицы, естественно, отличаться будут. Статическое связывание быстрее. Но не забываем про бонусы, заменяемость блоков рулит. Какую-то функцию написали с парамтером – объектом базового класса, через 10 лет появились наследники, а функциями с ними работать будет. Одно из определений полиморфизма – это то, что есть один интерфейс, и у него есть много реализаций. Интерфейсом в данном случае называется договор, что должен уметь объект (это неформальное определение). В нашем случае List можно назвать интерфейсом.

Пример: автоматизировать работу бухгалтера на предприятии. Раз в месяц бухгалтер едет в банк, забирает наличность и выдаёт зарплату. Спроектируем рабочее место, которое позволяло бы вести список сотрудников, а в конце месяца выдавало бы сумму, которую надо забрать в банке. Пусть в фирме есть разработчики Developer (несколько уровней с разной фиксированной зп в месяц: salary + level\*500) и отдел продаж Sales (ЗП у них salary + price \* 0.01 \* count, т.е. с каждого заключенного договора по 1% получают). Директор заранее сказал вам, что фирма будет расширяться, планируется больше программистов, также тестеров, сотрудников отдела кадров, у них будут свои алгоритмы расчета ЗП, пока не известно какие, поэтому сделай программу расширяемой. Чтобы нам потом не пришлось переписывать, а только расширять. Какие классы нужны? Worker – базовый класс, т.е. нужен некий договор / интерфейс от которого будем плясать. Все наследники должны уметь все, что умеет интерфейс. И если появятся другие реализации, а программа рассчитана на поддержку интерфейса, то программу не придётся переписывать.

В чем главная идея такой программы? Какой класс мы защитили от изменений? DB ! Мы его спроектировали так, что при появлении других сложных классов сотрудников нам не придётся менять класс DB. Класс DB будет работать с любым, кто реализует метод calculate. В языках интерпретируемых, типа Python надо просто иметь функцию с таким именем (типов можно сказать нету), в языках со статической типизаций типа Java, С++, где есть у переменных тип, этот механизм реализуется через наследование.

По поводу терминологии. Если в классе есть поля и чисто виртуальные функции, он называется абстрактный класс. Если есть только чисто виртуальные функции, а полей нет – называем его интерфейс. В Java есть ключевые слова разные для этого.