Московский государственный университет имени Ломоносова.
Факультет вычислительной математики и кибернетики.
Кафедра системного программирования.

Лекции по Языкам программирования

Студент
Кравец Алексей. 328 группа.
Лектор
Головин Игорь Геннадьевич.

Москва 2008

Лекция 1

1.Литература по курсу
В.Ш.Кауфман Языки программирования — концепции и примеры
Т.Пратт М.Зенкович Языки программирования. Разработка и реализация
Р.Себест Основные понятия языков программирования
Бен Ари Языки программирования
2.Основные языки затрагиваемые в курсе
C++(Страуструп Язык программирования C++, Страуструп Дизайн и эволюция C++)
Ada (Н. Джехани. Язык Ада)
java
C# (C# ускоренный курс)
Модула-2(Н. Вирт Программирования на языке ада)
Оберон
Pascal/C
Модула-2 задумывалась как легкий язык, в противовес языку Ada.

Глава 1
Определение и классификация языков программирования.

Язык программирования — понятие не обязательно связанное с компьютером (например язык APL — язык для математических формул, PLANNER — язык для людей). Можно определить язык программирования экстенсивно, то есть через множество понятий (например сказать что ЯП — это C++,C,Ada, и так далее).
Другое определение: Язык программирования — инструмент планирования поведения исполнителя.

Виды программирования

1.Игровое программирование (программирования для себя)
2.Научное программирование (для себя или очень ограниченного круга лиц)
3.Индустриальное программирование (коммерческое программирование)

Классификация по парадигме программирования

1.Процедурная (императивная) F(S)->S процесс перехода из состояния в состояние некого автомата. наличие оператора присваивания структурирование памяти (наличие типов данных)
2.Функциональная парадигма

Лекция2

Исторический очерк развития языков программирования.

В период с 1960 по 1980 года появилось огромное количество языков программирования (минимум 400). Это было вызвано тем, что каждый ВЧ. каждый серьезный проект считал своим долгом иметь свой собственный язык программирования.

1. Обоснование выбора языков курса

Fortran

Проект разработки этого языка считался неактуальным так как самым дорогим ресурсом считалось компьютерное и память время, следовательно быстродействие и эффективность программы являлись главными приоритетами. Понятно что никакой транслятор не способен генерировать код, по степени эффективности равный коду написанному человеком. Но создатели фортрана четко знали что хотели и добились всех поставленных целей. Главные заказчики на программы - военное ведомство (расчеты из области мат физики). Сначала Математик/физик разрабатывал алгоритм - далее программист его реализовывал. Объяснение шло на основе блок-схем. Программист занимался распределением памяти и переводом алгоритма в машинные коды(то есть фактически выполнял функции транслятора). Отладка так же представляла проблему. Результатом работы программы была распечатка цифровой информации (например дамп памяти). Главный приоритетом было желание исключить из схемы человека который переводить алгоритм в машинный код. Уже тогда существовали программы по переводу программ с ассемблера (авто кодов) на машинный язык. Соответственно необходимо средство по переводу программ с языка понятного и известного математикам на машинный код. Важнейшие операторы - оператор присваивания,операции ввода/вывода, условные операторы, операторы перехода.

Fortran - транслятор формул. Машинная команда IF - прототип оператора IF(e) L1 L2 L3 в фортране. В фортране существовало 4 разновидности операторов перехода. В 1964 был принят первый стандарт языка фортран являющийся самым популярным сегодня.

1977 - новый стандарт. Фортран стал крайне популярным в мире, но при этом во всех книжках по программированию фортран - источник контр-примеров. Проект разрабатывался в спешке - поэтому было сделано достаточно много ошибок. Например фортран - единственный язык в котором хранение матриц организованно по столбцам. Оператор вывода же выводит все построчно (1 вывод - одна строчка). Следовательно

необходимо расширения языка для вывода (список вывода) например для вывода матриц.

Фортран предполагался для работы с математическими моделями, но на нем стали писать все что угодно (например визуальный текстовый редактор).До 1964 фортран был базовым языком IBM. В это время влияние IBM было крайне велико (как Oracle + intel + Ms сейчас). Главный недостаток языка фортран - он провоцировал появление ошибок в программах. Основной пример - взрыв спутника отправленного в космос(ошибка в программе).
Оператор цикла в программе
DO 5 I=1,3
операторы тела цикла
5 CONTINUE
С точки зрения фортрана это корректная программа
Каждый оператор фортрана был на своей строчке (строка перфокарты или ленты(как причина)). В фортране пробелы не играли никакой роли (то есть программа могла быть написана без единого пробела).

Если в приведенном фрагменте заменить запятую на точку(описка) то программа станет такой
DO5I=1.3
Для экономии памяти нет объявления переменных,Но если переменная начинается с букв I-N то она целочисленная, иначе вещественная. Более того данные привязки можно переопределять. У Фортрана существует еще множество недостатков (для индустриального программирования).В данной ошибке виноваты те кто принял решение использовать фортран в таком проекте(запуск спутника).Но для математики (научных проектов) идеален. Существует огромный фонд программ на фортране накопленный со времени его появления. Понятие подпрограммы возникло впервые в Фортране.

Программы на фортране получались очень эффективными .Далее начали разрабатывать новые языки теми кого не устраивал фортран: 19–IFIP решили разработать язык для обмена алгоритмами между собой. Результатом стал язык АЛГОЛ который стал стандартом для ЯП, особенно для университетов Европы.
Достоинства АЛГОЛА:
В фотране были только статические переменные. В Алголе - блочная структура, стековое блочное распределение памяти,рекурсия,(но не было оператора ввода/вывода).
for i:=1 to N step 1 do
begin
s:=s+a[i]

end
Программа на АЛГОЛ60

Ключевые слова выделялось по договоренности - на бумаге подчеркиванием в машине зависит от реализации. Поэтому каждая реализация вводила свой набор операторов ввода/вывода и свой способ выделения ключевых слов - следовательно программы не переносимы. Но при этом в стандарте Фортрана многие вещи не определялись - соответственно многое зависело от тех трансляторов которые уже существовали на этапе разработки стандарта. При этом фортран был проще АЛГОЛА и был переносимым(если программа была написана на стандарте). АЛГОЛ обладает рядом свойств(о них позднее) программы на АЛГОЛЕ не были эффективны, так как основной целью была передать математическую красоту алгоритма но не эффективность.

Неформальный критерий качества компилятора - сравнить отношение эффективности оттранслированной программы и программы на ассемблере (t1 t2). В классическом варианте t1/t2 больше 1. На Алголе оно 7-10, при этом на фортране фирмы IBM (оптимизирующий компилятор) 1.05. Для RISC архитектур отношение может быть меньше 1(сложность архитектуры). Фортран продемонстрировал "принцип экологических ниш каждый вид занимает строго одну экологическую нишу. Таким образом можно сказать что у каждого языка есть своя экологическая ниша - проблемная область и набор навыков, библиотек,людей,алгоритмов и прочего, связанного с этим языком. Людям тяжело забывать старое (навыки,библиотеки..).Фортран занял нишу - научно технические вычисления (он занял ее первым и получил решающее преимущество).С точки зрения научно технических вопросов появилось множество языков после фортрана возможно превосходящих ее, но они не смогли вытеснить фортран. Примером вытеснения является вытеснение ассемблера из ниши системного программирования языком С.

Модула-2 создавался для системного программирования но не смог вытеснить язык С. Приспособившись к нише язык уже почти не способен ее покинуть. Если язык неточно позиционирован для какой-либо ниши(или его ниша уже занята) то он с большой вероятностью не получит популярности.

Периоды
I.1950 - 1960
Фортран
АЛГОЛ (1960)
Алгол - первая разработка серьезного коллектива ученых. Алгол60 по-

влиял на такие языки как например Паскаль, при описании синтаксиса алгола впервые использовались БНФ(разработанные для описания синтаксиса этого языка). Когда были введены грамматики стало ясно что БНФ соответствуют грамматикам класса 2. Со времен алгола60 синтаксис любых языков программирования описывались формально (например с помощью БНФ).

1967 - язык simula67 (имитационное моделирование - создание программный моделей). Понятие объекта/класса. Ключевые слова class new. Объекты классов заводились только в динамической памяти с помощью new. Страуструп работал с этим языком. разрабатывая C++. Когда Страуструп приехал в Америку он стал заниматься имитационным моделированием (для телефонных систем). Он разработал C с классами -который потмо эволюционировал в C++. Итак simula стала знаковым языком в развитии ЯП.

Научная задача - небольшой объем данных и вычислительноемкие операции (идеален фортран). Коммерческие приложения - большие объемы данных и простые операции. Требуется эффективность операций ввода вывода.

1959 COBOL.Был создан спец комитетом - занял свою экологическую нишу(бизнес приложения) + поддержка правительства США(один из членов комитета). Таким образом большинство коммерческих приложений было на этом языке(банки и прочее).Все компьютеры покупаемые за бюджетные деньги должны были иметь компилятор кобола. Даже сейчас в некоторых организациях работают программы на коболе.

Проблема 2000 - следствие использования кобола (в 1959 не думали о 2000 году). На дату (DDMMYY) отводилось минимум памяти (2 цифры на год). В результате огромные суммы были потрачены и на переписывание программ на коболе с учетом проблемы 2000. Кобол тоже не самый хороший язык но он занял свою нишу и до сих пор не совсем оттуда вытеснен.

LISP - "язык искусственного интеллекта"он не так эффективен для коммерческого использования (плохо ложится на Фон Неймовскую архитектуру) Лисп был популярен для экспертных систем (были даже проекты по переводу программ на нем на C),Сейчас интерес к лиспу(и функциональному программированию) возрождается после того как к ним был потерян интерес(из -за потери интереса к экспертным системам)

II 1960 - 1980

Появление огромного числа ЯП (каждый ВЦ хотел иметь свой ЯП) Многие фирмы делали свои яп, для многих проектов создавались новые яп(IBM,военные проекты). Для программы противоракетной обороны

был продуман собственный язык(придуман именно для военного применения). Язык для программы космических полетов (PDP-11) - язык HAL. Далее появилась идея создания универсального языка программирования. Первая попытка - фирма IBM в 1961 анонсировала IBM360 (полный оборот) как универсальную систему для всех классов задач. В 1964 была доделана OS360 (OS для этой системы).В качестве языка программирования было взято понемногу из многих существующих ЯП(рекурсия,блочная структура, оператор DO,..).При этом в язык попали многие не столь хорошие черты, как например список ввода-вывода в фортране. Язык получил название PL/I. Присутствовало двоично-десятичное кодирование(4 бита на каждую цифру) что крайне упростило ввод-вывод чисел. В результате - очень большой, очень интересный ЯП."Язык PL/I был бы очень хорошим языком, если выкинуть из него 80% возможностей"(с). НО для IBM360 он был универсален и он стал универсальным языком программирования. В его поддержку было вложено полтора миллиарда долларов. Но сейчас на этом языке никто не пишет. Он был разработан только для одной архитектуры. Существовали попытки переноса языка PL/I на другие архитектуры но они не были особо удачными. Например его порт под doc.

Язык PL/I был очень сложный. Сейчас большинство компиляторов может генерировать 2 версии - отладочную и релизовую, при этом попытка выпустить 2 вида компиляторов для PL/I провалилась так как результаты из работы отличались даже на корректных программах (была по разному реализована семантика языка).

Отсюда правила - Язык - совокупность взаимосвязанных компромиссов(без компромиссов нельзя).

1968 - пересмотренное сообщение о АЛГОЛ60. разработка языка алгол 60 закончилась и началась работа над новым языком программирования - АЛГОЛ68 (ядро группы - европейцы).Они хотели сделать универсальных хороший ЯП, учтя опыт прошлых разработок ЯП. В отличии от АЛГОЛА60 синтаксис и СЕМАНТИКА АЛГОЛА68 были формальны. W-грамматики для описания семантики. Ортогональность в смысле программирования/дизайна - их независимость. Для ЯП это означает что есть есть конструкция K и G и если они ортогональны то они могут существовать в любой комбинации. Например паскаль не является ортогональным ЯП

for i:=e1 to e2 do

здесь для присваивания доступна только целочисленная переменная,когда как в других случая это не так. Некая не ортогональность заложена уже в концепцию Фон-неймовских ЯП. В Алголе68 была стерта разница между оператором и операцией (все возвращает значение, любой оператор

имеет значение) то есть операторы и операции стали ортогональны. Этот язык повлиял на С.

Принцип ортогональности красив(программы на АЛГОЛЕ68 были очень красивые и компактные), он многие конструкции такого ЯП бессмысленны и потенциально содержат ошибки которые сложно найти, поэтому W-грамматики сейчас почти не применяются. Трансляторы этого языка работали крайне медленно - вплоть до нескольких операторов в минуту.

Так же этот ЯП не нашел себе экологической ниши (не смог вытеснить ни один из старых языков из его ниши).Алгол68 - типичный мертвый язык.

Лекция 3

Алгол был очень сложным языком - Такой сложный язык программирования нельзя предлагать мировому сообществу (с).

Параллельно с разработкой АЛГОЛа шел проект разработки операционной системы MULTICS (мультипрограммирование, использование вычислительной системы многими пользователями) — попытка создания универсальной, мощной, красивой системы. Так же как и с АЛГОЛом эта идея провалилась и часть ученых, участвующих в проекте отказались от своего участия в нем (так же как и с проектом по разработке языка АЛГОЛ).

В качестве противовеса MULTICS Кент Темпсон предложил UNIX, главной идеей которой была простота (так же это первая ОС написанная в основном (кроме небольшой части ядра) на языке высокого уровня — С)

В конце 60-х произошло несколько знаковых событий — появления языка С, языка Паскаль. Язык Паскаль вышел и в 1969 году и вобрал в себя лучшее что было придумано тогда касательно ЯП. Разработанный Н. Виртом для обучения своих студентов программированию, он включал в себя такие возможности как поддержку структурного программирования и структур данных. Паскаль является чисто академическим языком, то есть его стандарт совершенно не годится для реального программирования, при этом он оказал огромное влияние на дальнейшее развитие языков программирования. Его потомки, такие как Object Pascal, Turbo Pascal, Delphi популярны даже сейчас.

Язык С — язык занявший нишу ассемблера. В 70-е годы начали появляться более осмысленные проекты по разработке языков программирования (более концептуальные). Языку С название дал язык CPL(разрабатывался как замена ассемблеру) далее был BCPL, далее В и наконец С. Этот язык разрабатывался под конкретную нишу и не вытеснен из нее до сих пор. Даже сейчас этот язык остается самым популярным (14% по оценкам в интернете).

Язык CLU (язык абстракции данных) — Первый язык с абстракцией данных.

Этот период экстенсивного развития закончился созданием такого знакового языка программирования как Модула — 2.

Н. Виртом были разработаны основные концепции создания ЯП. В качестве примера он разработал язык Модула. Но он не прижился по непонятным причинам. Вирт модифицировал язык и в 1980 получил Модула — 2. На этом языке полностью было написано ПО для одной

графической станции. Но сейчас Модула — 2 тоже мертв, так как не смог сместить язык С из нищи языков системного программирования.

В 1980 году закончился очередной очередной этап развития ЯП.

Что касается Паскаля то тут стоит обратить внимание на книгу Кернигана Software tools in Pascal - в которой он сформулировал 16 причин, по которым паскаль неудобен для промышленного программирования. Главной причиной было отсутствие модульности.

Тогда же начинается третий проект по созданию универсального языка программирования — языка Ада. В данном случае подход к его созданию был совершенно иным. Исследования, проведенные в то время показали, что примерно 50% стоимости ПО - это стоимость его поддержки и сопровождения. Следовательно основная задача — уменьшить эти затраты. В то время в системе Пентагона было около 350 различных ЯП и целью проекта было уменьшить это число до 3.

Далее последовали несколько лет разработки требований к языку,и в 1979 году были готовы оригинальные требования. Выяснилось что возможно создать универсальный ЯП на баз PL/1, ALGOL, или Pascal. Изначально в конкурсе было 12 языков, потом осталось 4 (все 4 на основе паскаля). В итоге победил язык ада(единственный язык разработанный не в США (во Франции)) Язык Ада стал очень популярным в течении короткого времени. Была создана целая система верификации и сертификации компиляторов этого языка. В 1982 был принят первый стандарт языка ада.

Было принято решение заменять стандарт языка каждые 10-12 лет, при этом никаких диалектов и подмножеств языка быть не должно (это требовалось). Был даже издан приказ о том что все правительственное ПО должно быть на аде, но вскоре его отменили. В Аде было принципиальное новшество — механизм рандеву (синхронизация параллельных процессов).

В 1983 году с с классами эволюционировал в С++. В то время основные требования к ЯП ставились с точки зрения промышленного использования и ООП тогда особо не требовалось.

Проблемы языка ада:

1.Отсутствие ООП

2.Подход к языковому дизайну

3.Есть 2 подхода

4.метод сундука (все что нужно)

5.метод чемоданчика (только то без чего невозможно обойтись)

6.В языке ада реализован метод сундука (для любой критической технологической потребности в нем существует конструкция).

Язык Модула-2 — принцип чемоданчика. Например в Ада можно

иметь вложенные модули, а в модула — 2 их нет, так как без них можно обойтись.

К 1988 когда стало ясно, что ООП — основное направления промышленного программирования, Н. Вирт предпринял третью попытку создания ЯП — язык оберон, который представлял из себя урезанную модулу + наследование.

В 1993 появился Оберон — 2. в который были добавлены виртуальные функции (получился в итоге полный ООЯП). При этом компилятор оберона занимал 4,5К строк, а добавление виртуальных функций добавило еще 500 строк. Итак Оберон — 2 крайне простоя ООЯП. Для простых вещей нельзя использовать сложные инструменты. Например компилятор с Ады был очень сложным.

Лекция 4
Современный этап развития языков программирования.

Знаковые события этого этапа - появление C++ (1983), Ada. Именно в языке ада впервые появилось понятие исключения. Далее это понятие перешло в другие языки. Однако, своей цели язык ада не добился (заменить все другие ЯП). 1995 - новый вариант языка ада (обеспеченна максимальная совместимость снизу вверх).

В 2005 году -появилась еще одна версия ада. Ада 95 - пример очень аккуратного расширения процедурного ЯП объектно-ориентированной парадигмой. Главной причиной неудачи языка ада - его сложность. Было запрещено расширять стандарт ада, но каждые несколько лет выпускались новые толкования стандарта (некоторые детали стандарта).Даже требование выпускать все ПО только на сертифицированных компиляторах не могло выполняться так как даже самые совершенные компиляторы не могли пройти все тесты на сертификацию. Добавление ОО парадигмы еще сильнее усложнило язык. Итак третья попытка создания универсального ЯП провалилась, и от дальнейших попыток создания подобный ЯП отказались.

Далее началось развитие ООЯП. Первый чистый ООЯП - smalltalk появился в 1972 году в лаборатории Xerox PARC.В этой же лаборатории были разработаны: мышь, ethernet, станция с графическим интерфейсом.

В 1980 оформилась версия языка (smalltalk 5), ставшая самой популярной версией этого языка. Многие понятия современных ООЯП пришли именно из smalltalk. Например метод - метод доступа - понятие языка smalltalk. Слово super в java так же понятие из smalltalk. Но при этом он был крайне неэффективным.

Еще одна особенность этого языка - для простоты его реализации было разработано специальное представление - байт-код. То есть имелся только один компилятор и интерпретаторы байт кода для каждой целевой машины.

С точки зрения индустриального программирования самым важным ООЯП является язык C++.

В 1986 появилась версия C++ 2.0 - первая широко доступная версия языка.

В 1990 - добавлены шаблоны и исключения. Самой главной ошибкой проектирования Страуструп считает то, что во время подготовки версии 1986 года он сосредоточился на множественном наследовании а не на шаблонах. В smalltalk не было множественного наследования, и многие поборники этого языка считают что его и не должно быть в ООП.

В 1990-х было разработано множество компиляторов C++. При этом вся STL разработана на шаблонах. При этом во всех компиляторах были разные реализации шаблонов, что привело к тому что каждый компилятор должен был иметь свой вариант библиотеки. В 1998 был принят стандарт и была разработана единая библиотека.

В 1990 C++ стал самостоятельным, не зависимым от C языком. До 1990 Страуструп боялся вводить новые ключевые слова так как это может привести к несовместимости с предыдущими версиями. После 1990 появились новые ключевые ключевые слова (шаблоны, namespace, RTTI, STL).Уже к 1994 году все это было разработано, но STL задержало выпуск стандарта до 1998. Даже MSVC++ (Microsoft Visual C++) до сих пор не поддерживает все требования стандарта.

STL - это не только интерфейс, но и реализация которую не совсем полно поддерживает даже .MSVC++. Таким образом платформа STL - это набор шаблонов + компилятор. Таких шаблонов, как в C++, больше нигде нету. Сейчас готовится новый стандарт TR1 и становится видно, что с++ стал очень сложным (не проще АДА). Но C++ один из наиболее мощных ЯП (и самый мощный ЯП на сегодняшний день). Сейчас развитие C++ идет по пути развития STL. C++ - язык в котором даны очень простые базовые средства, а все остальное реализуется с помощью классов, что делает C++ наиболее гибким ЯП.

В 1988 году появился язык оберон, в 1993 - оберон 2, который является самым компактным ООЯП.В его реализации используется принцип минимума языковых конструкций.

Язык Eiffel - широко известен в узких кругах, но у него не появилось своей экологической ниши. Причина - в 1980 активно использовались ООЯП, которые являлись расширениями реально существующих ЯП. При этом все недостатки базовых ЯП проникали в эти языки. Так, C++ - расширение C, получил все его недостатки.

Objective C - язык C в которые добавлены часть конструкцией smalltalk. Object Pascal, Turbo Pascal (ОО начиная с пятой версии), далее - система Дельфи. Сейчас C# замещает Delphi.

Итак большинство ООЯП основаны на C++, при этом такие языки как java, C#, Delphi -сильнее схожи друг с другом чем с C++.Система J++ - реализация java на VS.

Язык Java - анонсирован в 1995 (C# - анонсирован в 1999).Эти языки были разработаны с нуля - то есть без расчета на совместимость.

Принцип java - WORA (write once - run anywhere). Концепция языка близка к концепции универсального ЯП.В его состав входит JVM - виртуальная машина для исполнения байт кода. В 60-е 70-е были проект по реализации машины которые на вход получали ЯП высокого уровня, то

есть являвшиеся вещественными реализациями виртуальных машин.

Symbol — реализация Algol60. В СССР была разработана машина МИР. В 70-х, эти разработки свернули и стали копировать разработки иностранных компаний.

Язык Lisp - язык исследования ИИ и экспертных систем. Была компания которая выпускала аппаратные реализации Lisp машин. Далее была идея интерпретации промежуточного представления программы. Система UCSD-Pascal - требовался единый компилятор для работы на разных архитектурах. Была разработана система P-code (что-то типа ПОЛИЗ). То есть существовал один компилятор с Pascal в P-code и по интерпретатору на каждую архитектуру. Это оказалась простой и эффективной системой работающей на множестве различных архитектур.

Аппаратная реализация ЯП - микропроцессор Katana - микропроцессор реализующий байт-код. Язык java был предложен как чистый ООЯП с синтаксисом похожим на C++. Кроме того существует JVM - интерпретатор байт кода ("языка ассемблера"для JVM).

RTL - real time lib - поддержка программ во время работы.

Java Runtime Environment - JVM + RTL. Все что нужно чтоб реализовать java для некой архитектуры - это реализовать JRE для этой платформы. Существует несколько вариантов java систем, например java2 ME (micro edition), SE (standard), EE (enterprise). Для всех систем один и тот же компилятор - они различаются только набором библиотек, при этом сам язык один и тот же.Существует программа по сертификации java машин (фирмой Sum).

1995 - знаковый год, появилась Win95. Ms хотели сделать свой интернет MSN. Установка WIn95 не допускала возможность работы с открытыми сетями. И тут появился проект который позволял работать на любой платформе. Более тога java - изначально интернет язык, java applet — приложение выполняющиеся в интернет browser. Был придуман язык LiveScript - языке разрабатываемый как замена html. Тогда даже были идеи что скоро все будут писать на java, и только интернет приложения,но этого не произошло.

Главные недостатки java - потери скорости на интерпретации и неэффективность сборки мусора. При этом java - идеальный выход когда нужно обеспечить моно язычную среду на многих платформах. При этом java остается собственностью фирмы Sun. Это привело к некому нежеланию многих переходить на java, но несмотря на это java сейчас очень распространенный ЯП. По сравнению с C++ java более современен (чистый ООЯП, проще чем C++).

Язык C# - разработка MS. Раньше чтобы программировать нужно было изучить ЯП и то как запускается программа в некой вычислитель-

ной среде. Теперь же нужно еще знать платформу под которую будет писаться программа. Сейчас самое сложное - изучение соответствующих API. (В Unix, например, из больше 1000).

Подход MS достаточно сильно отличается от этого. Вместо стандартизации ЯП, они предложили стандартизовать библиотеки. В результате появился стандарт CLI (common language infrostructure). В этот стандарт входит
- CTS общая система типов
- Il промежуточный язык (вариант ассемблера), но не зависит от платформы.
- Pe формат - формат исполняемых файлов - не зависит от платформы.
- стандартизация RTL
то есть всегда один и тот же сборщик мусора, один и тот же набор коллекций. Тепер не важно какой выбрать язык - важно чтоб он входил в эту спецификацию. Ves - виртуальная исполнительная среда. CLR - common language runtime - реализация виртуальной исполнительной системы в .NET.
JScript.NET
VB.NET
FoxPro.NET
C++/CLI
C++ в его обычной реализации нельзя реализовать в этой системе (например сборки мусора в это языке нету) поэтому CLI — "управляемый"C++. То есть к синтаксису C++ добавили новые специфические для .NET ключевые слова. Можно сказать что, к C++ добавили язык C#.

Далее требовался "родной"ЯП для этой системы - самый удобный язык для этой системы,при этом не требовалась совместимость, так как разрабатывался с 0, таким языком стал C#.

JSharp - это чистый язык java, но при этом его RTL берется из CLR (.NET библиотеки).То есть это было указание на возможность перехода с почти любой платформы на платформу .NET. Реализация C++ в .NET страдает большим числом ошибок. В отличии от java систем, есть ECMA — независимая ассоциация производителей компьютеров, в том числе она стандартизует компьютерные системы, в том числе и ЯП.CLI и C# стандартизованы ECMA. Однако стандарты ECMA не совсем полно соответствуют спецификации языка C#, но тем не менее формально владельцем C# является ECMA. Так же ECMA Script - стандартизованные вариант java script.

Сейчас - C# живой и динамичный ЯП - сейчас уже C# 3.0. Во второй версии C# появились обобщения, в третьей версии - lambda функции.

Основное развитие системы java шло по пути развития стандартной библиотеки, но версии языка менялись не сильно, хотя в 2005 году произошла генеральная ревизия языка, Java5 (версия 2005 года), в которую добавились обобщения как в C#.

Пока за пределами рассмотрение остались скриптовые ЯП -принципиально интерпретируемые ЯП, без всякой компиляции. Развитие таких ЯП связанно с развитием веб узлов.Perl,PHP,Python. - примеры таких языков, но с концептуальной точки зрения они не сильно отличаются от тех ЯП о которых говорилось выше.

Лекция 5

Глава 3 Основные понятия ЯП

Пункт 1. Основные позиции при рассматривании ЯП

1.Технологическая.

С точки зрения технологических потребностей, одной из самых важных возможностей языка программирования является возможность разделения программы на модули (это необходимо для индустриального программирования)

2.Реализаторская — позиция автора/ реализатора ЯП.

Существует мнение, что сейчас данная позиция не актуальна, то есть можно объяснить любой ЯП без данной позиции. Но тем не менее в 60-х существовало мнение что ЯП — это то что реализует данный компилятор. Это привело бы к определенным проблемам, поэтому было введено понятие стандарта.

В данном курсе понятие реализации особо не затрагивается, то есть ЯП описываются в основном абстрагировано от реализации, но будем касаться этой позиции в тех случаях когда детали реализации важны с точки зрения оправданности использования той или оной конструкции(C++ виртуальные функции, локальные переменные в стеке и так далее). Например ООЯП не всегда реализуют множественное наследование — причина этому то что динамическое связывание трудно эффективно реализовать.

Авторская позиция.

Любой ЯП — совокупность компромиссов. То есть нельзя добавлять в ЯП все что хочется, так как в результате могут получиться такие «монстры» как АДА. Авторская позиция рассматривает почему данная конструкция была реализована в данном языке в том или ином виде. Например, в C++ не вошли некоторые понятия из Ады (хотя комитет по стандартизации предлагал ввести их), в нем нет виртуальных конструкторов, и так далее.

4.Семиотеческая (семиотика — наука о знаковый системах).Рассмотрение с этой позиции позволяет увидеть проблемы которые могут возникнуть в ЯП с более общего вида. Например проблема неоднозначности сообщений. (Синтаксис, семантика, прагматика). Пример синтаксической неоднозначности — субъект посылает сообщение, а принимающий по-другому интерпретирует его структуру («Привет освободителям востока от Феликса Эдмуновича Держинского»). В ЯП таких неоднозначностей практических нет,но в некоторые они есть. Например

if B1 then if B2

then S1

else S2

Здесь не ясно к какому блоку относится else.

Однако подобные неоднозначности решаются либо на этапе обсуждение стандарта, либо уже после выхода. Заметим что все ЯП имеют почти всегда однозначную синтаксическую структуру.

Семантическая неоднозначность — тоже очень неприятна вещь, при этом проще всего написать компилятор для простого ЯП, который полностью удовлетворяет стандарту. Худший вариант, когда одинаковые конструкции по-разному транслируются или реализуются без никаких сообщений об ошибках.

Прагматическая ошибка (ошибка в понимании «зачем?»). Мы не будем рассматривать такие ошибки.

5.Социальная позиция.

Все ЯП созданы людьми и функционируют в обществе. То есть многие феномены связанные с ЯП необъяснимы с точки зрения первых 4 причин (экологические ниши, успех фортрана и так далее). С социальной точки зрения многие ЯП являются хорошими, даже если они плохие с остальных точек зрения, но тем не менее они популярны именно в силу социальной позиции. Так смена ЯП —процесс довольно тяжелый именно для людей, то есть это социальная позиция.

Еще пример — 2 конкурирующие системы под win32 — Delphi и Visual Basic, по первым 4 позициям Delphi превосходил VB, но по социальной позиции он сильно проигрывал VB (VB был раскручен).

Пункт 2. Основные понятия.

Три основных понятия — данные, операции и связывание (считаются атомарными и не определяются).

В любом ЯП есть, очевидно, данные и операции. Что же касается связывания, то он нем впервые заговорили в 70-х. Понятие связывание играет роль не меньшую чем данные и операции. Пример связывание — механизм связывания формальных и фактических параметров (основное значение придается времени связывания).Например i=12; Важно в какой момент происходит связывание значения i с 12. Например const int i = 12 — на этапе компиляции. const X a; во время запуска конструктора. Выделяют несколько этапов — трансляции, компоновки, выполнения... При этом эти этапы могут быть расширены. Чаще всего выделяются 2 этапа — статическое связывание и динамическое связывание. Статическое — во время трансляции или во время редактирования связей. Динамическое — во время выполнения.

Реализация строк (паскалевский и Сишный).

Некоторые данные проще изображать как операции и наоборот , например получение длины строки для паскалевской строки (данные представляются как операция). Операции представимые как данные называются свойствами. Например визуальное окно имеет привязку (x,y) — обычно координаты левой верхней точки, и установление этих параметров в классе — вызов большого числа процедур, хотя вызывается оно как операции с данными (x = <smth>). Итак — это типичное свойство, то есть это удобно рассматривать как данные, но изменение этого вызывает операции.

Рассмотрим более подробно понятие данных в ЯП. Почти все данные обладают следующими атрибутами — тип данных, имя, значение, время жизни, область видимости. Все данные принадлежать одному из нескольких множеств — типов данных. Тип данных — это множество значений, которое определяется структурой данных + набор операций над этим типом. Только по виду данных нельзя определить что это за объект так как разные типы данных могут иметь одинаковую структуры (важно знание множества операций и их семантика).

Нет безтиповых ЯП — есть языки с динамической типизацией (тип объекта определяется на этапе работы программы), большинство таких ЯП — интерпретируемые. При этом есть ЯП в которых типы данных могут в явном виде не существовать — это так называемые прототипные ЯП.

Например в C# существует такое объявление как var a; то есть определяется просто переменная a, тип которой определяется когда ей будет присвоено значение. В прототипных же ЯП есть определенные объекты, у которых есть слоты, в которые можно положить как объект данных, так и код процедуры. В любом прототипном ЯП есть операция клонирования, которая выдает полную копию объекта. Примером таких ЯП являются javaScript, self. При этом в javaScript это понятие несколько завуалированно. Self стал известен совсем недавно.

Итак в прототипных ЯП тоже есть тип данных, но его определение отличается от определения типа данных в классических ЯП. То есть объект может переходить из одного типа данных в другой. Контроль за поведением объекта во время выполнения исходя из его статических характеристик — квазистатический контроль.

Атрибуты типа данных.

В некоторых случаях имя является единственным и самым важным атрибутом идентификатора (атом a языка LISP). При этом бывают и безымянные объекты.(константа 5 языка паскаль, объекты полученные по new). Так же дополнительную ненадежность вводят ЯП без сборки мусора в которых анонимные объекты не отличаются от именованных.

В C# появился специальный спецификатор readonly, который специфицирует момент связывания (readonly — только статические).

Время жизни.

Связанно с понятием класса памяти. Классы памяти — статическая (все время работы программы), квазистатическая (статический в пределах блока), динамическая (создаются по new malloc и так далее, уничтожение — delete, free, или сборка мусора),persistent(сохраняемый между запусками программы. Реализуется только библиотечными вызовами).

Лекция 6

Область действия (понятие имеет смысл для именованных объектов данных) — та часть программы, в которой имя действительно. Важны два понятие — определяющее вхождение имени (определение или объявление имени), обычно единственно, и использующее вхождение имени. Большинство ЯП конструируется так чтоб существовало единственное определяющее вхождение.

Есть 2 вида областей действия — статические (определяются по тексту программы еще до ее запуска на этапе компиляции) и динамические (встречаются редко). Понятие блока — статическая область действия.

```
var i;

proc p;
var i;
begin
P1;
end p;

proc P1;
begin
i:=1;
end P1;

begin
P;
P1;
end;
```

Такая структура появилась впервые в АЛГОЛ — 60 и перешла потом во все остальные ЯП. В языках с динамической областью действия поиск осуществляется с помощью стека (то есть в при вызове процедуры P1 из P будет изменено значение локальной i(той что определена в P)).

В C++ есть пример динамической области действия - поиск ловушек при обработке исключений. Но в основном в современных ЯП используются статические области действия. Первая причина этому — простота отладки программ, вторая — эффективность реализации (адрес переменной определен и не требуется ее поиск по стеку).

Пункт 3. Концептуальная схема рассмотрения.

Главная проблема — прежде всего сложность. Впервые люди столкнулись с проблемами при программировании в начале 60-х при программировании на OS360. Разработка этой ОС шла три года и когда она вышла, содержала огромное количество неэффективностей, при этом самым главным недостатком была неготовность самой системы. Базис компьютеров, для которых пишутся программы очень велик. При этом разрыв между компьютерами и реальным миром, который нужно описать в программе крайне велик. Для преодолении этого придуман механизм абстракции. При этом почти для каждой проблемной области приходится создавать свой набор моделей. Простейшим средством абстракции является подпрограмма (единственное средство абстракции в ФОРТРАНЕ). С точки зрения базиса ЯП различаются не сильно, но с точки зрения абстракций различие велико.

Схема рассмотрения:
1.Базис (те конструкции и абстракции которые встроены В ЯП)
1.1.Скалярный (неразложимый на более простые операции (типы данных (базовые)) и некоторые операторы)
1.2.Составной (массивы и некоторые операторы)

2.Средства абстракции (механизм создания новых типов данных, классов и подпрограмм).
ВТД — множество операций а не множество значений.
В абстрактных типах данных - понятие интерфейса.
Примем абстракции в языке С — файл stdin stdout. Но в С нет защиты введенных абстракций, что делает его ненадежным ЯП.

3.Средства защиты введенных абстракций. Средства защиты хорошо развиты в языке ада. Надежность ЯП — если программа компилируется, то она семантически верна. Третье свойство очень интересное и не очевидное. 50'% разработок программ шло на сопровождение, 25% на тестирование. То есть третий приоритет — читабельность кода (он пишется один раз и читается много раз, притом не только ее автором).С этой стороны АДА — является хорошим ЯП — много длинных ключевых слов....

Часть 2. Современные ЯП

Сейчас большинство ЯП основаны на ООП парадигме, но при этом используют понятия из традиционных ЯП.
1.Базис традиционных ЯП. Например для языка С и С++ базис почти не различается (в языке С++ появился только ссылочный тип, который тем не менее не имеет никакого отношения к ООЯП).
1.1.Скалярный базис. Простые типы данных.
Классификация:

1. числовые типы данных (основные типы данных)
1.целочисленные
2.вещественные (с плавающей точкой)
-плавающая точка
-фиксированная точка
Такое разделение числовых типов данных (в математике такого нет) — это уже некоторое усложнение. Есть скриптовый ЯП lua в котором нет целых и вещественных типов данных. Есть ТД Number — который универсален (но за это нужно платить скоростью). В языках типа Матлаб есть типы данных для вычислений с произвольной точностью.
2.Логический тип данных (bool)
3.Символьный
4.порядковый тип данных (на его базе возможны перечисления и диапазоны).
5.Ссылки и указатели
6.Функциональные типы данных (его единицами являются процедуры и функции, которые уже являются частью абстракции)

1.Целочисленные типы данных.
Проблемы
1.1 Фиксация представления и связанный с ним набор проблем.
1.2.Без знаковые целые числа. Самый простой в этом плане — язык паскаль.

Но нет разницы между 2-х байтовым, 4-х байтовым и 8-и байтовыми числами. Из соображений эффективности приходится вводить целый набор целочисленных типов данных. Первая версия языка С появился для мини — ЭВМ. Потом был портирован и на другие архитектуры. Единственный размер типа который фиксирован в стандарте — тип char занимал ровно 1 байт. Далее short int, int, long int, long long. При этом представление в языке С не фиксируется. Если требуется фиксированное представление то следует определить typedef новый тип данных. Эта система была унаследована и языком С++. Новые ЯП - C# и Java предназначались для несколько других целей. C# - архитектура CLI в которой существует CommonTypeSystem.

Чтобы сделать язык универсальным, создатели реализовали полную матрицу типов данных (все 8 типов данных 1 2 4 8 знаковые и без знаковые).
1 sbyte byte
2 short ushort
4 int uint
8 long ulong

С развитием 64 битных архитектур, люди не захотели переписывать все программы на С на новую 64-битную архитектуру. Основная проблема 32 битных архитектур — адресация памяти. Поэтому сейчас в основном используются 64x32 — вся базисная арифметика 32 битная, но адресация 64 битная. С языком java несколько другая ситуация — вся переносимость реализуется концепцией Java машины — то есть достаточно реализовать на данной архитектуре JRE машину. В Java есть 4 основных типа с фиксированным представлением (byte, short, int, long). Это позволяет точно детерменировать семантику выполнения программы на любой платформе.

Проблема — сравнение знаковый и без знаковый чисел. Смешивать их опасно так как может пострадать логика программы и такую ошибку тяжело отловить. Пример — язык Модула-2. Там был введен тип данных INTEGER и CARDINAL. Преобразование между ними могло быть только явным. В С++ нет ограничения на такие преобразования.

Зачем нужен без знаковый тип данных для 32 битных архитектур — работа со сдвигами (без знаковый сдвиг), адресная арифметика.В Обероне есть следующие типы данных — byte (0 255), short, integer, longint. Эта система потом перешла в java — тут нет без знаковых чисел и соответственно нет проблем с их представлением. C# позволяет выполнять С код в unsafe(). В Java вместо без знаковых чисел есть без знаковый сдвиг »>.

Лекция 7.
Два подхода к реализации числовых типов данных.

1.Фиксированный базис (примеры ЯП - см прошлую лекцию)
Фиксируется номенклатура числовых типов (языки — C,C++, C#, Delphi, java, Modula -2, Oberon). То есть сказано что к целочисленным типам данных относятся определенные типы для которых многие параметры фиксированы. В C#, M-2, Oberon, Java, Delphi — размерность таких типов фиксирована. Так же зафиксирована семантика Во всех кроме java это было просто, так как они ориентировались на определенную систему. Для java это было сложнее, так как вся работа по реализации семантики ложится на интерпретатор. Фиксация базиса ориентированна на максимальную эффективность. Фиксация номенклатуры, но без фиксации представления создает трудности для переносимости (язык C).Создатели java не могли жертвовать переносимостью, поэтому они пожертвовали эффективностью.

2.Обобщенные числовые типы.
Рассмотрим на примере языка АДА. Программы на этом языке должны были функционировать на различных архитектурах (как на существующих, как и на тех что еще могли появиться), поэтому было невозможно фиксировать базис представления числовых типов.

Перед программами на языке ада ставились несколько требований.
-Эффективность.
-Надежность.
-Читабельность.

Для выполнения этих требований в языке ада был введены обобщенные не именованные числовые типы, а все остальные числовые типы были его потомками. Второй концепцией языка ада была особая концепция типа. Объекты разных типов были несовместимы ни по какому множеству операций, но были совместимы объекты под-типов. То есть разные под-типы совместимы между собой и со своим предком. Например, рассмотрим следующее объявление на языке ада.
type NewInt is new integer; - Ada
type NewInt = integer;Паскаль
typedef int NewInt// C

В языке C typedef не определяет новый тип, а определяет новый синоним типа, когда как в ада и паскале он определяет новый тип. То есть в языке ада нельзя присваивать объекты типа NewInt и integer.

Например type Length is new integer range 0..MaxLen. Теперь это новый тип данных с соответствующими ограничениями на значение. При-

чем это новый тип (new) то есть он не совместим с другими типами.
Type int16 is new integer range -32768..32767
type uint16 in new integer range 0..65535
Это два новых типа, то есть их нельзя смешивать.
То есть
i16 : int16;
ui16 : uint16;
i16 := ui16 ; ошибка
i16 := Int16(ui16) не ошибка так как есть общая база (int).

Все преобразования безопасны, то есть в случае ошибки выдается исключительная ситуация. Если требуются неявные преобразования, то необходимо использовать концепцию под-типа:
subtype POSINT is integer range 1..MAX
subtype naturale is integer range 0..MAX1;

Далее исходя из архитектуры компилятор выбирает оптимальное представление для таких чисел. То есть для переносимости необходимо явно задать диапазон значений (а не представление). То есть проблема представления ложится на компилятор. Теперь возможна такая запись.
n: NATURAL;
p:POSINT;
для присваивания p=n необходим контроль значения n, то есть компилятор либо проверяет значение n на этапе трансляции, если это возможно, либо вставляет код по проверке перед присваиванием. Такой контроль называется квазистатическим. То есть если вся информация есть на этапе трансляции то осуществляется статический контроль, иначе — квазистатический. Такой же контроль есть в языке Паскаль, C# и java. В C# такое ограничение можно обойти с помощью пометки «небезопасного кода».

В аде в случае ошибки выхода за границу возбуждается исключительная ситуация (range error). Безопасными всегда являются операции расширения типа.

В начале все типы данных имели фиксированный базис. Далее с развитием ЯП стали появляться ЯП с обобщенными типами данных. Далее — C# и так далее — возврат к фиксированному базису. Но сейчас начинает развиваться снова подход с обобщенными типами — так как на них проще программировать. Это тенденция современности, когда даже в промышленном программировании стали применяться интерпретируемые ЯП.

Представление вещественных типов.
1.Плавающая точка
$\pm M.B^P$ (знак| мантиса | порядок (B - основание системы счисления)).

В случае вещественных чисел разброс еще больше чем в случае целочисленных. Так же как для целочисленных типов используется 2 подхода — с фиксированным базисом и с обобщенным представлением.

1985 IEEE стандартизовало формат представления вещественных типов.

Представление в виде мантисы и порядка для каждого числа не единственно. Для арифметических операций используется нормализованное представление мантисы. То есть $1/B <= M < 0$. То есть в системе с основанием B все степени B (положительные и отрицательные) представимы точно. Основываясь на представлении, получаем что при операциях порядка 10^6 точность уже в районе 1. Более того при любой мантисе можно выбрать такой порядок что погрешность будет порядка нескольких десятков/сотен.

При переносе программ с одной архитектуры на другую из-за различия в представлении чисел погрешность может принимать недопустимые значения. Более того погрешность имеет свойство накапливаться. То есть нужны либо спец алгоритмы, которые не накапливают погрешности либо улучшить представление. В стандарте IEEE (754) было фиксировано 2 представления для плавающих точек.(float и double). Первый — для 4-байтовых (1 бит на знак, 23 на мантису, 8 на порядок, основание всегда 2). Так же было зарезервировано значение для машинного нуля и машинной бесконечности (плюс и минус).Возникли такие числа как Nan (not a number).

Почти все современные архитектуры реализуют этот стандарт. Все ЯП так же зафиксировали представление. Язык ада сформировался до появления этого стандарта. В нем для вещественного типа фиксируется число точных единиц, а подбор реализации ложится на компилятор. При этом в стандарте ада-83 были зафиксированы модельные числа — способ отображения этих чисел ($B = 2$) фиксирован, то есть стандарт фиксировал как модельные числа представлялись компилятором. Все модельные числе в представлении выбранном компилятором должны быть представлены точно. Дальше от этой идее отказались так как подбор представления был неэффективен и часто был невозможен (из — за модельных чисел). То есть плавающие числа в языке ада считаются потомками обобщенного вещественного типа.
type FLOAT is digit 6;

Современный компилятор должен реализовывать типа float и double.

Отметим что плавающие числа нужны для представления математических расчетов. В других сферах часто требуется несколько другое — то есть например требуется 2^12 значений на промежутке от -M до M. То есть

каждое число по логике является целым, но по виду оно вещественное. И если есть только плавающий тип,то при представлении таких чисел неудобно использовать операции с плавающей точкой, то есть нужно либо преобразовывать в float что дает потерю памяти и скорости операций, либо реализовывать руками все операции.Поэтому в язык ада были добавлены delta — типы.
type DATA is delta 1/4096 range -M..M;
type T is delta H range l..p; - общий способ задания delta чисел. Вся дальнейшая работа по представлению при этом ложится на компилятор. То есть вновь все основная нагрузка ложится на транслятор.

В некоторых языках есть некоторые дополнительные типы данных, например decimal в C#, currency для валюты и так далее, которые являются специализированными delta типами.

Лекция 8
Логические и символьные типы данных.

Логический тип данных (Boolean) — тип данных над которым применимы следующие операции — or not and or. Изначально в языке C не было логических типов, но было логические операции. Любое выражение имело логический смысл, если оно могло было быть преобразовано к int или void* , которые проверялись на равенство 0. Это повышало компактность кода, но понижала надежность и читабельность кода. Единственный язык с неявным преобразованием int в bool — это C и C++. В C-99 логический тип появился. Так же он был стандартизован (был определен базис, определены константы true и fakse). Современные языки, такие как C# и Java имеют тип bool. При этом, в них есть только явные преобразования этого типа. Все реализации логического типа так или иначе наследуют язык АЛГОЛ — 60.

Символьный тип данных. Родина большинства ЯП — США и Западная Европа, то есть для них для представления символов было достаточно одного байта. Одним из первых ЯП с символьным типом данных является ЯП паскаль. С этой точки зрения ЯП в которых символьный тип данных представляется 1 байтом не могут использоваться для решения многих задач (например тех, которые требуют много-язычности). Проблема символьных типов данных — набор символов — мощность набора и какой именно набор символов поддерживается. До 80х с этой точки зрения был полный бардак. Набор символов вообще говоря имеет некоторое официальное название для каждого символа и код символа. Далее появилась кодировка ANSII-7. Теперь она так или иначе используется почти везде как базовый набор символов.[0-127] символов: [0-31] — непечатные символы. Далее были большие и малые английские буквы (большие и маленькие), цифры и знаки пунктуации.

Сейчас все наборы только расширяют этот тип, при этом первые 128 символов всегда совпадают с ANSII.

Одного байта хватало только на кодировку 256 символов (SBCS). Паскаль,C, C++ - одно-байтовый символьные тип. Существует набор символов ISO-Latin1. (ANSII и 128-255 были оставлены для национальных алфавитов, но даже их хватало только для стран западной Европы). Многие алфавиты не покрывались ISO-Latin1, поэтому появлялись новые кодировки (кириллица, турецкий, греческий и так далее). В настоящее время существует несколько кодировок даже кириллицы. Например koi8-r кодировка для интернета/электронной почты. Все почтовые сервера в то время находились в США и использовали кодировку ASCII — 7. Даже не поддерживалась ANSI и ISO-Latin. Koi8-r при удалении стар-

шего бита транслитировала сообщение на русском на латиницу. Кроме этих существует еще множество других кодировок (cp1251 — codePage 1251(MS)).

Проблема стала серьезнее когда она достигла Японии и Китая. Появилось MBCS DBCS. Японцы использовали MBCS, принцип ее был прост — если байт имел значение 0-127 то это бай кодировки ANSII, если больше — то это байт, начинающий кодировку одного из Японских символов (2-3 байта). То есть все программы которые работали с чисто английским текстом могли работать без изменений. При этом, например, переход на DBCS требовал полный передел ПО. 1991 год — появление стандарта Unicode. Примерно соответствующий ISO UCS — universal Char set. Он стандартизовал некоторый набор символов, названия алфавитов, при этом использовался принцип DBCS.

Все символы получили код 0-65535. Он стал называться UCS-2. Первые 128 символов — ASCII 128-255 — Latin1. То есть преобразование от юникода к этим стандартам требовалось только обрубить старший байт. При разработке WinNT на уровне ядра все тексты идут уже в кодировке unicode. То есть MS сделала большой труд по интернационализации системы. Существуют спец системные вызовы — MultyByteToWideChar и наоборот.

Разработка интернациональных приложений достаточно сложна — это не то же самое что и локализация приложения. Для каждой локализации есть своя версия приложения (например текстовый редактор и так далее). Есть еще глобализованные приложения — они работают одинаково в любой среде (современный MS Word).

Во всех современных ЯП принята следующая ситуация — существует тип данных char (!= char C) он занимает 2 байта и использует кодировку unicode. При этом это отдельный тип данных и никак не совместим с целочисленным типом данных, то есть необходимо явное преобразование от char к целочисленным типам данных. Автоматически автоматически такое преобразование не проходит, а явное контролируется. Со старыми ЯП, которые появились до unicode все несколько сложнее. Например в C++ появился тип данных wchar (==unsigned short). Раньше этот тип определялся через typedef. В языке C/C++ char — это не символьный, а арифметический тип данных. В C++ wchar имеет неявное преобразование wchar в insigned short. Определение нового класса (вместо typedef) позволяло перегружать функции (отдельно для unsigned short, отдельно wchar).

Перегрузка функций есть в всех современных ЯП кроме Оберона и модулы-2 (из — за принципе минимальности языковый конструкций). То есть возможно написание спец функций для обработки именно unicode

символов. Проблема такого переноса встала также и перед создателями языка АДА (разные кодировки на разных системах даже для английского языка). При этом требовалась работа на любой системе. Создатели языка Ада решили не привязываться к конкретной кодировки, то есть символьный тип данных рассматривался как частный случай перечислимого типа, для этого требовалось расширить перечислимый тип. Элемент перечисления - так называемый литерал — некий символ, записанный в кавычках.
type enum is (one,two,three);

type CHARS is ('A','B','C');

Ситается что любая программа погружена в пакет STANSART, где были описаны все специальные типы данных, которые покрывали английский язык. Если не требовались преобразования в int, то нам вообще не важно в какой кодировке работает программа. Позже в 95 году появился WIDECHARACTER, который кодировал в Unicode, то есть можно было свободно расширять Ада — программы. Есть еще ucs—4 (4 байта). Он так же мог легко кодироваться в ucs-2 путем обрубания 2 старших байтов. Введение ucs-2 решило проблему Японии, но не Китая, язык которого включает до 40К символов (после реформы их количество стало меньше, но во многих странах остается старая система письменности, требующая множество символов).

За использование unicode приходится платить тратой памяти (хотя сейчас экономия памяти не так актуальна, но все так же актуальны размеры канала передачи данных). Создателей java часто ругали за переход на unicode так как в начале большинство текстов было на английском языке и половина переданной информации являлись нулями (старшие байты). Для решения этой проблемы был разработан utf — формат преобразования универсального набора. Одним из первых таких форматов был формат uft-7 (преобразование в 7 битный код).

Utf-8 — способ представления любого ucs в 8-битных значениях. Способ работы таков. Все номера разбиваются на три диапазона.
0-127
128-2047
2047-65535
Символы из первого диапазона копируются сами в себя. То есть любой английский текст является текстом в utf-8.

Символы из второго диапазона (требуют 11 битов). И представляются в виде (110<5 битов>,10<остальные 6>) В этот диапазон попадают почти все одно-байтовые языки.

В третий диапазон попадают остальные языки. которые не могут ни-

как уместиться в 1 байт (японский, китайский и так далее). Формат — (1110<4> 10<6> 10<6>).

Экономия имеет место так как все управляющие символы и прочее кодируются одним байтом. Японские же тексты уже по типу языка обладают компактностью, то есть им не так страшно разрастание языка. Например, стандартная кодировка для XML — UTF-8. Все современные ЯП имею представление символов в видя unicode, даже Дельфи (начиная с 5 версии). Символьный тип данных имеется во всех перечисленных языках, причем только в двух из них (C/C++) имеется неявное преобразование к числовым типам..

П5.Порядковый тип(диапазоны и перечисления).

Диапазон — первый пример типа, не являющимся типом в себе, то есть он основан на неком другом типе. То есть если есть базовый тип T и две константы этого типа A B — то диапазоном этого типа называется множество значений типа T от A до B.

На базовый тип накладывается ограничение — они должны быть поддерживать операции suck и prev (Pascal). К типам с плавающей точкой нельзя применить операцию succ хотя они и упорядоченны, так как представление этого типа зависит от реализации. То есть базовыми типами для диапазонов могут быть типы, основанные на целых числах (числа, символы и так далее). Диапазоны были уже в языке паскаль. Система опирается на работы Дейксры и Вирта. Так же диапазоны есть в языке Ада.
type NEWINT is new INTEGER range 1..N

В наследнике паскаля — модуле-2 диапазоны включают в себя [|. Диапазон удобен тем, что он ограничивает поведение соответствующего типа, что позволяет осуществлять квазистатический контроль. То есть проверка на выход за значение диапазона происходит либо во время компиляции либо на этапе выполнения.

В языке Оберон(принцип минимальности языковых конструкций) — диапазонов нет. В модификации этого языка — Оберон-2 их так же нет. Так же их нет в java и C#.В чем причина этого объяснил Вирт, когда рассказывал о дизайне Оберон-2. Он сказал, что основное понятие языка оберон-2 — это наследование(расширение). С точки зрения основной концепции расширения типа, диапазоны ничего не дают и более того противоречат идее расширения типа. То есть диапазон является не расширяемым типом данных. Диапазоны использовались в основном для работы с массивами (как индексы).Понятие квазистатического контроля так же связанно с понятием диапазона.

В многих компиляторах Turbo Pascal можно отключать квазистатический контроль. Было сделано наблюдение, что все средства контроля отключаются программистами (в релизовой версии). А раз нет диапазонов, следовательно все массивы удобнее всего начинать с 0. И во всех современных ЯП все массивы похожи на массивы в языке С, то есть объявление требует только длину массива. То есть Диапазон исчез из ЯП потом у что он не соответствовал концепции ООП.

Перечисления были придуманы Виртом (паскаль) потом он же их и вывел из использования, но сейчас они вновь возвращаются.

Лекция 9

Перечислимые типы данных появились впервые в ЯП паскаль (1969). После этого они входили почти во все языки программирования.
type T= (C1,C2,..,CN)Pascal style

Говорят, что перечислимый тип упрощает чтение программы, повышает надежность программы (абстрагируемся от числового значение переменной), может улучшать эффективность программы (путем уменьшения места для хранения таких переменных). При этом в 1988 году, в языке Оберон, Вирт отказался от перечислимого типа данных в ЯП Оберон (принцип минимальности языковых конструкций). Вторая причина — перечислимые типы в том виде как они появились в паскале и соответственно других ЯП, противоречили концепции ООП. Перечислимые типы данных (в виде в котором они появились в Паскале) не могут быть расширенны.

1995 — Java. (Разрабатывался с нуля, частичная синтаксическая совместимость с C). В Java перечислимых типов нет.

1999 — C#. В нем такие типы есть, но в описании языка авторы уже в самом начале «оправдываются» в том почему они включили этот тип данных. Основной аргумент — перечислимые типы данных хорошо интегрируются со средствами визуального программирования (VS). То есть многие свойства компонент удобно представлять в виде перечисления. Текст с использованием таких типов иногда называют «самодокументированным». Третье неприятное свойство перечислимого типа данных — вместе с именем типа неявно импортируются имена констант.

2005 год — java 5.0 — перечислимые типы данных были возвращены. Синтаксически перечислимые типы данных в java похожи на перечислимые типы данных в Паскале, но при этом они представляются как отдельный класс.
Реализация перечислимый типов данных.

1.Классическая схема.(Паскаль, Ада)
type T=(..); Паскаль, модула-2
type T is (C1,...,CN); Ада

При этом нет неявных преобразований из перечислимого типа к целочисленному и наоборот. В стандартном Паскале была функция ord — перевод перечислимого типа к целочисленному. В Модула-2 есть функция val которая принимает перечислимую константу — имя типа и целое значение и возвращает значение соответствующего перечислимого типа или выдает ошибку времени выполнения.

Реализация перечислимого типа в языке Ада немного отличается. Так как символьный тип данных в языке Ада реализован как перечисление,

то есть кроме обычного перечисления есть еще литералы перечисления. В Аде возможно пересечение имен в перечислениях, то есть литералы перечисления могут пересекаться между собой.
type RGBColor in (.. Red...);
type TrafficColor in (Reg,Green,Yellow);
Так как по правилу перегружаться могут только имена функций, то требовались некоторые ухищрения чтоб вписать это в язык Ада. Рассмотрим пример ф-ии на языке Ада.
function Func return T
//объекты локальных переменных
begin
//операторы
end Func;

x:=T;
x:=Func:
Таким образом такие литералы перечисления рассматриваются как имена функций без параметров. Отсюда возникает следующая проблема.
procedure P(X:RGBColor);
procedure P(Y:TrafficColor);
и тогда P(Red) — неоднозначность. Поэтому для этих случаев в Аде существует специальная конструкция указания типа (не является преобразованием). Она используется когда нужно именно указать компилятору что данное значение имеет соответствующий тип. P(RGBColor'Red);

Все эти проблемы возникли из-за того что имена перечислимых констант находятся в той же области видимости что и имя типа. Например в языке C# имена констант локализованы внутри типа. То есть теперь RGBColor.Red. ТО есть теперь одна и та же константа может принадлежать разным перечислениям. Язык C# продолжает традицию традиционных ЯП, то есть константы перечислимого типа в C# - это константы целого типа. По умолчанию константы целоги типа данных представляются значением целого типа данных, при это значения буду (0 .. N-1) при этом представления можно выбирать явно (по умолчанию 0 int). И при этом не существует неявного преобразования, кроме константы 0, которая может преобразовываться в любую перечислимую константу неявно. При явном преобразовании всегда осуществляется контроль.

Особняком стоят языки C и C++. В некотором смысл в этих языках перечислимый тип не является типом в полном смысле слова тип.
emun(C1,C2,C3);
Теперь C1=0,C2=1,C3=2.
При этом

enum c x;
x=-25; - Семантически допустимо. То есть перечислимый тип всего лишь способ задание некоего набора констант. То есть const int C1=0 или #define C1 0 эквивалентны определению перечислимого типа. Поэтому этот тип не особо популярен программистами языка С. При разработке языка C++ по поводу перечислимого типа возникло несколько проблем. 1-Проблема тегов. В языке С возможно задание безымянного enum. В C+ такой enum можно наблюдать такой enum в STL. То есть enum <ios modes> и при этом open(..,int); То есть enum полностью совместим с целочисленным типом данных. То есть Страуструп встал перед следующей проблемой — с одной стороны требовалось поддерживать полную совместимость с С, с другой стороны для перегрузки функций было бы удобно сделать его отдельным типом. Таким образом существует явное преобразование из int -> enum(отсутствие контроля) а преобразование из emun ->int осуществляется неявно. В C++ так как перечисления неявно совместимы с целочисленными константами, то программисту дана возможность управлять значениями соответствующих констант. То есть
enum Mode
reed = 0x1,
write = 0x2,
RW = reed|write;

В большинстве случаев над целочисленными константами требуется выполнять некоторый набор битовых операций. Подход языка C# позволяет избавиться от недостатков перечислимого типа языка С — от ненадежности и неявного импорта имен. В C# появилась возможность выбора базиса представления.

Атрибутирование — частный случай более общей конструкции C# - рефлкексии. То есть это отражение структуры кода программы в результирующем бинарном коде — рефлексия. Атрибутирование — приписывание константам перечислимого типа данных некоторых атрибутов. Рассмотрим один из таких атрибутов — flags. Например enum в атрибутом [flags] позволяет выполнять над константами перечислимого типа логические битовые операции.

Упаковка/распаковка.
В чисто объектно ориентированных ЯП все является объектом. То есть 3+5 — это операция с двумя объектами. Структуры, перечисления и простые типы данных в C# называются value types (типы значений). То есть эти элементы представляются именно как собственные значения а не как ссылки на объект. Для каждого такого типа существует его коробка (box) — класс. Итак для каждого такого класса существует свой класс — обертка. То есть если речь идет об операциях над примитивны-

ми классами то операция проводится над объектом, если же операция классовая — то над обертке. Например для enum существует обертка — класс Enum который является оберткой для любого перечислимого типа данных.

В языке java перечислимый тип данных вообще никак не связан с целочисленным типом данных. enum colorred,green,blue; имена констант так же локализованы внутри имени перечисления. java.lang — пакет java который описывает специфические классы java. Например в java.lang содержится класс Enum который является базовым для всех перечислимых типов.
enum RGBColor
public RGBColor(byte r,byte g,byte b);
private int color();
Red(255,0,0),
Green(0,255,0),
Blue(0,0,255);

Пункт 6 — ссылки и указатели.

Указатель — абстракция машинного адреса. Вопрос — надежность указателей. ЯП делятся на 2 больших класса — строгие с точки зрения указателей и нестрогие. В Стандартном паскале указатели появились только для манипуляции с объектами в динамической памяти. То есть в стандарте паскаль нельяз описать указатель на integer. То есть только ссылки на объекты структур.

Итак строгие языки — Паскаль, Ада — 83, Модула-2, Оберн. Указатели только для ссылок на объекты в динамической памяти.
type PT is access T;
TYPE PT = pointer to T;
При этом T не обязан быть определен ДО определения указателя. В языке ада все несколько сложнее.
Type PT is access;
type T is word;
type PT is access T;
То есть в таких языках все очень похожее. В языке ада работа с указателем.
X:PT;
X := new T;//семантика аналогична new (x) в паскале.
Нет никакого иного способа завести динамический объект. Операции над указателями — присваивание и разыменование. Далее в языке Оберон
X:PT;

X^.next // Модула-2
X.next //Оберон
То есть в Обероне исчезает разница между указателем на объект и самим объектом.В языке Ада операция точка так же применима к указателю на объект. Для доступа ко всему объекту — X.all.

Лекция 10.

Указатели и ссылки. Это разные типы данных но при этом оба являются абстракцией адреса в памяти. Как уже было сказано языки делятся на 2 категории — строгие и не строгие. Строгие — ссылки только на объекты в динамической памяти. Примеры таких языков — Оберон, Модула-2, стандарт Паскаль, Ада — 83.

Нестрогие языки — указатели работают как и в строгих, плюс можно получить адрес любого объекта. Это & в C++, @ в Delphi. Такой подход чреват некоторыми проблемами. Например

```
char * f(){
    char buff[1024];
    return buff;
}

char * p = f();
*p = 0;
```

Такие ошибки компилятор не отлавливает.
Аналогично.

```
static int i;
int *p;
p=&i;
free (p);
```

Для избежание подобных ошибок существуют специальные программы — верификаторы, но даже они не всегда могут найти такие ошибки. Некоторые интегрированные среды поддерживают 2 режима компиляции — отладочный и релизовый. Отладочный работает медленнее за счет использование отладочных библиотек работы с памятью. Так же проблема неконтролируемого преобразования. То есть любой адрес можно преобразовать в адрес void* и его в свою очередь можно явно преобразовать в любой другой тип. void* - это некоторый абстрактный адрес. В таких ЯП нельзя обойтись без такого абстрактного типа указателя.

Одно из решение проблемы — создание строгих языков, но даже строгостью языка не решаются все проблемы. Например если существует операция явного освобождения памяти, то возникает проблема обращения к несуществующему объекту. Такая операция присутствует в Паскале, Ада — 83, Модула -2. Явное освобождение приводит к двум проблемам.
1- мусор в памяти.

```
T * x = new T();
t * x1 = new T();
x=x1; //теперь первый объект является мусором.
```

Ошибка в программе — трудно дать ей точное определение. Ни одна из программ не соответствует своей спецификации (спецификации устаревают). С профессиональной точки зрения, мусор в памяти — огромная проблема, так как она проявляется сильнее всего в реальных условиях. Так же такие ошибки крайне тяжело отловить на этапе тестирования программы.

Вторая проблема — висячие ссылки. Использование адреса несуществующего объекта.

```
T * x = new T();
T * x1 = x;
delete x;
```

Теперь x и x1 ссылаются на один и тот же объект, причем его уже не существует. То есть delete x1; приводит теперь к ошибке. Однако многие менеджеры памяти спокойно отрабатывают на таких ошибках. В случае много - поточности же все еще сложнее так как, например, один поток может освободить память, которую использует другой поток.

Системы с автоматической сборкой мусора. Создатели языка понимали что наличие динамической памяти может привести к множеству проблем, при этом они не могли ввести в свой язык автоматическую сборку мусора. Самый простой и эффективный алгоритм автоматической сборки мусора — подсчет количества ссылок на данную область памяти. Но такой алгоритм можно использовать только для достаточно высоко-уровневых ЯП, так же проблема кольцевых ссылок.

Второй способ основан на регулярном просмотре всей памяти, на которую указывают все указатели, видимые в данный момент. То есть во время сборки мусора система занимается только сборкой мусора. Если памяти хватает, то сборка может и не потребоваться, но в случае если памяти мало, то мы не можем гарантировать скорость отклика системы на любое действие, так как в любой самый важный момент может включиться сборка.

Создатели языка Ада не включили в стандарт ЯП стандартных средств освобождения памяти. То есть присутствовало ключевое слово new для выделения памяти. Но при этом все освобождение в стандарте не оговаривается., то есть допустимы реализации со сборкой мусора.

Более того, для большей переносимости существует стандартный пакет UNCHECKED_DEALLOCATION и соответствующая функция DEALLOCATE применимая к любому типу указателя. Если компилятор не поддерживает автоматическую сборку мусора требовалось использовать эту функцию.

В языке модула-2 имеется 2 процедуры — ALLOCATE и DEALLOCATE.

Для реализации этих функций был введен особый тип данных ADDRESS (аналог void*). Этот тип обладал набором свойств, как например приведение к нему неявно любого указателя.

В языке С примером использование void* - написание контейнеров, в которых хранятся указатели на некоторые объекты. Единственный ЯП который сохранил чистоту с точки зрения указателей — язык Оберон. Он использует простой алгоритм сборку мусора, что делает его достаточно надежным при сохранении нормальной производительности.

В java и C# полностью отсутствует понятие указателя. Если запретить программисту явно освобождать память то понятие указателя трансформируется в понятие ссылки. То есть все классы java кроме простых классов — ссылки на соответствующие объекты в динамической памяти. То есть все объекты являются анонимными и располагаются в динамической памяти. При этом для размещения объектов используется оператор new.
X a = new X();
a = new B();
a = b — копирование ссылок. В java реализована автоматическая сборка мусора.

Язык Ада-83 является строгим языком. То есть закрыта большая дыра в потенциальных ошибках в программах. В языке Ада — 83 не было никаких адресных операций, и реализация ее с автоматической сборкой мусора была не менее надажна чем Оберон.

Ада — 95 — там появилась в некотором виде адресная операция — a'access. Изменения были внесены по нескольким причинам — в определение языка все таки были допущены некоторые ошибки. Далее в 90-х стало популярна концепция ООП. Далее Ада ранее не допускал использование на других ЯП, но к 95 году приоритеты изменились и его создатели уже не могли расчитывать на монопольное использование Ады. Большинство системных вызовов в всех ОС привязаны к языку С. То есть требовалось передавать адреса переменных, чего не было при Ада — 83. Создатели Ада — 9 сумели расширить функционирование Ада, при этом сохранив надежность программ на нем.

Была разрешена операция взятия адреса, но нельзя была применять эту операцию к любым переменным. Если переменная описана как i: alias integer — то к ней применима операция взятия адреса. С точки зрения указателей TYPE PT is access all integer может использоваться только для доступа как к объектам из динамической памяти, как и на объекты из обычной. То есть была сохранена надежность при введении новых возможностей.

Возвращаясь к понятию ссылки, если язык является строгим то все

объекты находятся в динамической памяти, а если существует еще и автоматическая сборка мусора, то понятие указателя вообще не требуется. В java появилось понятие JNI (Java native interface) — возможность запуска родных системным вызовам данной системы. Это ускоряет работу программы, но при этмо противоречит принципу WORA.

В C# специально оставлена «дырка» - спецификатор unsafe, который позволяет запускать код на любом языке, то есть можно использовать любые вызовы системы или стандартной библиотеки (malloc, free). Далее существует спецификатор fixed. Например его можно использовать для перевода массива байтов в указатель на него. Данный код естественно ненадежен. Рассмотрим сборку мусора по алгоритмы Mark and scan. Данный алгоритм при проведении сборки производит сдвиг области памяти, то есть значение ссылки может измениться. Конструкция fixed фиксирует адрес памяти и не позволяет менять адрес ссылки.

Все объекты в java,C#,delphi являются ссылками, при этом в java есть сборка мусора, в delphi — нет. X a;
a = new X(); - единственный способ порождения нового объекта в java / C# / Delphi.
В Delphi все объекты наследуют класс Tobject при этом этот объект обладает конструктором create();. То есть для каждого объекта существует метод create();
В C#, java, delphi — ограниченный вариант указателя.
В C++ при этом ссылка — это именно имя некоего объекта. К ссылки языка C++ применима единственная операция — ее инициализации.
T &a;
Может быть описана как
1- переменная соответствующего типа, должна быть инициализирована через имя соответствующего объекта.
2 -формальные параметры процедур и функций.
void swap (T& a, T& b);
3 — возвращаемое значение функции.
X& f();
4 — как член класса. Инициализируется в конструкторе.

Ссылочный тип данных оказался очень удобным.

Итак рассмотрение примитивных типов данных закончено. Единственный тип данных, добавленный в C++ изначально — ссылочный тип. В современных библиотеках C++ присутствует спец тип данных — smart pointer. Он может например проверять валидность объекта доступа, автоматическое удаление объекта на который он указывает и так далее.

Лекция11.

Глава 2. Составные типы данных.

Типы данных, встроенные в язык программирования, но при этом имеют внутреннюю структуру.
Массивы.
Записи / структуры / объединения.
Множества.(Стандартный Паскаль)
Файлы (стандартный Паскаль)
Строки (не совсем тип (в языке Паскаль) packed array of char).

Классы языка C++ не являются составными типами данных.Отдельно можно добавить ассоциативные массивы и Hash таблицы.

В современных ЯП многие такие структуры данных вместо добавления их в компилятор, добавляется в стандартную библиотеку языка. Для множества из них не существует универсального, одновременно эффективного средства представления данных. Вместо этого выбирается некоторая реализация которая входит в стандартную библиотеку. При этом сохраняется возможность выбора оптимальной реализации под конкретную задачу. Для примера рассмотрим сортировку. Для большинства задач используется quick Sort, но во все библиотеки ЯП включают как минимум еще один алгоритм, например heap Sort. То есть встраивая подобные вещи в компилятор, мы становимся зависимыми от метода реализации, который может быть и не оптимален. С точки зрения простых типов данных, базис ЯП постоянно расширяется (начиная с начала развития ЯП), при это с точки зрения составных типов, но постепенно сужается.

Пункт 2.

В C# есть классы string и dictionary. При этом компилятор знает про класс string (выглядит как библиотечный класс, но встроен в компилятор), а про класс dictionary он не знает ничего (просто библиотечный класс).

Рассмотрим составные типы данных.
1.Массивы.
Абстракция последовательности однотипных элементов.
DxDxDxD — массив.
D^n — массив как декартово произведение одномерных типов данных.
При этом эта последовательность должна представляться в памяти в виде непрерывной последовательности элементов. Такая непрерывность расположения в памяти требуется для оптимизации доступа. Самая распространенная операция для массивов — операция индексирования. Существует 2 способа изображения этой операции.

1.A[i]
2.A (i)
Индексация возвращает ссылку на соответствующий объект.

В C# и java обязательна инициализация массива. К операции индексирования зачастую добавляется набор атрибутов.

Атрибуты массивов:
1.Атрибуты всех объектов данных (базовые атрибуты любого типа данных).
2.Базовый тип.
3.Тип индекса. Должен быть некоторой непрерывной последовательностью значений. При этом оптимален дискретный тип. Дискретный тип — такой тип данных, что для него определены операции succ и pred, причем их работа на всех архитектурах определена однозначно.
4.Границы / Длина массива.

Когда мы знаем базовый тип массива и он непрерывно расположен в памяти, то компилятор может эффективно определить операцию индексирования. В ряде языков тип индекса фиксируется статически во время трансляции. Пример — язык Паскаль array [Index] of T. Так же происходит в языке Ада. В языке С например, так же как и во многих подобных языках, тип индекса — всегда integer. Итак тип индекса фиксируется самое позднее на этапе трансляции. Массив — это всегда компромисс между тремя вещами — надежность, эффективность, гибкость. С точки зрения этих требований оптимальный вариант — массивы из языка Ада. Две крайности — массивы в С и в Паскале. Между ними — Модула -2 и Оберон а так же C# и java.

В Языке паскаль все атрибуты статические, следовательно он надежен. То есть при индексации производится статический/квазистатический контроль. Более того все атрибуты не только статические, но и являются атрибутами типа. Такая реализация дает надежность и эффективность, но сильно вредит гибкости. При этом Паскаль не гнался за гибкостью, то есть для своих нужд он был достаточно гибок. При этом теряется только универсальность, в угоду простоте.

Язык С. Ориентирован не на гибкость или надежность, но на эффективность. С этой точки зрения, паскаль и С оба ориентированны на эффективность, но есть и различия.В качестве типа индекса берется всегда integer. Фиксируется левая граница — 0. Тип элемента и длина являются статическими атрибутами типа. Таим образом мы немного уменьшили гибкость. В С адрес начала массива и его размер однозначно определяет массив. При этом длина требуется только для распределения памяти. То есть чаще всего массив задается адресом своего начала. Задание индекса от 0 позволяет так же максимально эффективно вычислять адрес

элемента при индексировании.

То есть все что нужно чтоб оперировать с массивом — это адрес его начала == адрес его первого элемента что является D*. Следовательно любой массив сопоставим с D*.То есть:

```
int a[20];
a[1] = *(a+1);
```

При этом никакого квазистатического контроля выполнено не будет. Таким образом обеспечена максимальная эффективность и некоторая гибкость сточки зрения универсальности.

Что же касается языка Модула-2, то там так же как и в языке паскаль фиксируется базовый тип и тип индекса. При этом процедуры и функции могут иметь особый тип данных — открытый массив. Например:

procedure P(var A:T) где T — открытый массив.

ARRAY OF TO. То есть мы не фиксируем тип индекса.

Процедура, выдающая сумму элементов массива.

```
PROCEDURE SUM (VAR A:ARRAY OF T):T;
    var R:T;
Begin
    T:=A[0];
    for I:=1 to HIGH(A) DO
    R:=R+A[i];
    return R; end SUM;
```

С точки зрения эффективности доступа, Модула-2 — оптимальный компромисс.

В языке Оберон осталось понятие открытого массива. И фиксируется только базовый тип массива и его длина.

```
Type T = array N of D;
```

Аналогично D T[N];

Максимальный компромисс достигнут в языке Ада. Разные типы данных абсолютно несовместимы между собой. При этом элементы подтипов приводимы к базовому типу. Type TNew is new T [ограничение]. Тут может быть ограничение диапазона. В массивах есть понятия неограниченного массива и ограниченного массива. В неограниченном фиксируется только тип данных и тип индекса.

```
Type arr is array (Index range<>) of D;
Type limarr is array (Index range 1..N) of D;
```

В качестве индексного берется любой дискретный тип данных. Неограниченные массивы могут быть формальными параметрами процедур и функций.

X:limarr; нормально.

X: arr; нельзя.
При этом
Z:arr range 0..10 — нормально.
Z- подтип типа arr.
W:arr range 1..11;
U:arr range 0..11;

Все эти переменные принадлежат разным подтипам одного и того же типа. Неявная совместимость в случае если компилятор проверит диапазон значений. Требование — длины массивов должны совпадать. То есть
Z:=W;
W:=Z; можно.
Но Z:=U — нарушение во время проверки на этапе квазистатического контроля.

```
function SUM(A:arr) return D is
    R:D;
begin
    for i in A'RANGE loop
    R:=R+A(i);
    end loop;
    return R;
    end SUM;
A'LENGTH
A'RANGE
```

Лекци 12.

Как уже говорилось, наилучшего компромисса между надежностью и гибкостью удалось достичь в языке Ада. Повторное описание массивов языка Ада (было в лекции 11). Перепишем пример с прошлой лекции.

Если в объявлении функции явно указать полностью определенный тип массива, то теряется свойство гибкости. Поэтому пишем так.

```
function SUM (A:ARRUNL) return real;
    R:real:=0.0;
begin
    for i in A'RANGE loop
    R:=R+A(i);
    end loop;
    return R;
end SUM;
```

```
A'LENGTH
A'FIRST
A'LAST
A'RANGE <=> A'FIRST..A'LAST
```

```
X:ARRUNL range 0..9;корректно
Y:ARRUNL range -10..10;корректно
i:=SUM (X);корректно
j:=SUM (Y);корректно
```

Длины массивов могут быть любыми. Но требуется чтоб тип индекса и тип элементов массивов совпадали. Для того чтобы избавиться от этих ограничений необходимо использовать некий аналог шаблонов (например их аналог в языке Ада).

Рассмотрим современные языки программирования — C#, Java, Delphi. В Delphi есть все что было в стандартном Pascal плюс, добавились неограниченные массивы. Современный подход к индексации — тип индекса всегда int и начинается с 0. Длина является свойством не типа, но экземпляра. Массивы — объекты особого типа, которые выводятся из некоторого класса или поддерживают некоторый класс/интерфейс. Итак стирается различие между массивом и объектом. Но при этом сохраняется свойство непрерывности расположения в памяти.

Объявление:
T[] x;теперь x — ссылка на массив. Длина при этом не указывается.

x = new T[len]; Длина указывается как свойство конкретного объекта. len не обязана быть константой. Место под массив отводится в динамической памяти. При этом массивы ведут себя как классы — они имеют не свойства, а методы и поддерживают некоторые стандартные методы всех классов. После выделения памяти под массив, мы не можем изменить его длину, мы только можем произвести перераспределение памяти, то есть выделить память под новый массив. Такой подход реализован в C# и Java.

Для экземпляра длина — квазистатический атрибут, то есть задается во время выполнения, но нет возможности менять ее во время выполнения. Аналогичная ситуация со строками. В C# и Java есть специальные классы для создание строк — StringBuilder. Во время работы со строками, например во время создания новой строки, требуется частое изменение размера выделенной памяти, что приводит к огромной фрагментации памяти.

StringBuilder — более оптимальным образом распределяет память под строки, пользуясь тем, что во время его работы количество требуемой памяти может только увеличиваться. То есть такие массивы реально являются квазистатическими.

В Аде есть понятие динамического массива, фактически не похожего на свой аналог в C# или Java.

```
procedure P(N:integer) is
    A:array (1..N) of real;
begin
...
end;
```

В данном примере происходит некоторое отступление от правил языка Ада, так как границы массива не заданы статически. При этом границы массива являются атрибутами именно конкретного объекта, а не всего типа.

Теперь немного про многомерные массивы и операцию вырезки (slice).

Срез массива — некоторый диапазон его элементов. Такая операция есть в PL/I, Fortran — 90, Ada.
A:Array (1..10) of T;
A(2..5); Вырезки одной длины могут быть совместимы между собой. В современных языках программирования — операция copy(i,j) — аналог вырезки.
A(2..5,1..3) — таким образом вырезки уже не всегда могут располагаться в памяти непрерывно.

Для реализации многомерных массивов чаще всего применяется следующий способ. Многомерный массив — это просто непрерывная последовательность массивов меньшей размерности.

```
Type T2 = array [1..N] of T1;
    T1 = array [1..M] of T;
X[i][j]   X[i,j];
```

Аналогично

```
Type T2 = array [1..N,1..M] of T;
```

Язык C/C++;

```
int i[3][3];//i ≡ int**
```

Единственное исключение — язык фортран — там массив располагается не по строкам, а по столбцам. При этом во всех остальных языках — по строкам.

Ступенчатые массивы — есть в Java, могут моделироваться в C (ссылками).

Двумерный массив языка Java трактуется как массив массивов или как массив указателей/ссылок.

В C# есть 2 варианта объявления.

```
int [,] = new int [3,5];
```

И классическое объявление (массив массивов). Тогда получаем ступенчатый массив.

В Java:

```
int [][]X = new int[][3];
```

Далее инициализация каждого из 3-х элементов массива.

```
for (int i=0; i < 3; i++)
    x[i] = new int [i+1];

int [N]x = {{1,2,3}, new int [5],{0}};
```

Явная инициализация.

Пункт 2. Записи.

Современное понятие записи — практически не измененная запись языка Pascal.

```
type R = record
    x,y:T;
    i,j: real;
end;
```

То есть если массив — DxDxDx..xD; то запись — это декартово произведение разнотипных элементов. Основная операция — доступ по имени поля.

Язык C:
```
struct tags{
    int i,j;
};
```

Ада:
```
record
    //поля записи.
end record;
```

В современных языках программирования понятие записи превратилось в понятие класса.

Oberon:
```
RECORD
    последовательность полей
END;
```

Запись с вариантами <=> union //C. ≡ объединение C/C++ ≡ параметризованные записи (Ада).

Объединение типов.

Традиционные ЯП — каждому объекту данных соответствует единственный тип данных. ООЯП — все объекты относятся к некоторому классу объектов.

```
int i = 0;
object o = i;//корректно в C#
```

Янус — проблема — различные объекты могут выступать в разных ролях, то есть каждый объект может иметь несколько ролей / типов. Ярче всего эта проблема проявила себя при проектировании пользовательских интерфейсов. Например окно может содержать в себе некоторый набор элементов, зависящий от конкретного окна.

record
постоянная часть.
вариантная часть.
end;
постоянная часть — последовательность объявления переменных.
вариантная часть
-размеченное распределение.
поле, где есть дискриминант, определяющий вид поля.
-не размеченное распределение.

```
type event = record
    T:time;
    case Evtype:ETEvent
    KBEvent: (scancode : integer,up:Boolean);
    MouseEvent :(....); end:
```

Вариантная часть — все переменные располагаются с одного и того же адреса. Не размеченное распределение — отсутствует поля дискриминанта.

```
type BitCell = record
    case Boolean of
    true:(i:integer)
    false:(bits:packed array [1..48] of Boolean)
end;

X:BitCell;

X.i= 249;
X.bits [48] := 1;
```

union — не инкриминированное объединение без постоянной части.

Лекция 13.

Итак, объединение типа — синтаксически обобщение понятие записи или структуры. Наличие объединений говорит что традиционная система типов имеет некоторую «ущербность». Существует инкриминированные объединения — в них у каждого поля есть определенное значение — дискриминант, по которому определяется тип поля. Объединение типов реализовано так, что все элементы начинаются с одного и того же адреса в памяти.

Паскаль, Modula-2 имеют почти эквивалентные объединения типов. АДА — все объединения различны. С — только чистые не размеченные объединения.

Описание объединений — предыдущая лекция.

Различия в объявление малы. При этом различия в объявлении вызваны различиями в операторе выбора (switch) в соответствующих ЯП. Дискриминант может быть 2-х видов:
1.имя: тип.
2.тип.
При этом первый способ в размеченных и второй в не размеченных. В языке С размер объединения — размер максимального элемента. В Паскале существует и другой синтаксис.
NEW (P, t1, t2, t3 ...);
Распределение память под запись с вариантами, причем выделение памяти идет по типу данного варианта t1. При этом последовательность t_n идет когда есть n вложенных под-записей. При этом объединение типов несет в себе проблемы связанные с ненадежностью. Если запись не размеченная, то надежность крайне мала, если же запись размечена, то изменив дискриминант мы можем сделать дальнейший код полностью нерабочим.
Язык Ада.

```
type varrec (D:DT) is record
    постоянная часть.
    case D:DT of
    when v1 => вариант 1;
    when v2|v3 => вариант 2;
    when others => вариант6;
    end record;

type stack (N:integer) is
    record
```

```
    top : Integer range 1..N = 1;
    body : array (Integer range 1..N) of real;
end record;

X:stack (50);

type PVARREC is
    access VARREC;

p:PVARREC;
p= new VARREC (v1); //обязательно явно указать параметр.
```

В ООП и чистых ООЯП нужда в записи с вариантами отпадает. Они остаются только для совместимости со старым кодом. То есть постоянная часть — base класс. Вариантная часть — классы — потомки. Существует вариант — наследование с полем типа, но приводит к всем проблемам с ненадежностью, которые есть при использовании объединений.

Важна динамическая обработка методов. В идеале для модификации некоторого класса при использовании динамического связывания требуется только исходники самого класса.

Итак запись с вариантами сейчас (из - за ООП) можно считать устаревшей. Понятие записи обобщается понятием класса. То есть добавляются функции члены, которым неявно передается адрес записи. В C++ структура стала классом. Отличие между структурой и классом — в доступе по умолчанию. Совместимость C++ и C — образец хорошей совместимости с прошлыми версиями. При этом было решено множество проблем (например проблема с именованием структур в C). При этом C++ сильно усложнился. У тегов структур в C имеется особое пространство имен, при этом C++ не мог этого допустить.

Основная проблема со структурами — заголовочные файлы UNIX написанные на C, в которых активно используются структуры.

Пример
```
struct C{
    struct B{
    }b;
}
struct B b;
C::b == b;
```

Страуструп утвердил что структуры в C++ почти полностью анало-

гичен классам, при этом если есть конфликт с существующими именами, то он допустим, при этом доступ к структуре далее возможен только через модификатор struct.

В Delphi вопрос был решен проще — есть

```
record
...
end;
```

Которые являются аналогами записей языка паскаль. И есть

```
object
...
end;
```

Аналоги классов.

Для java. В нем нет записей в их обычном смысле, но есть классы.

Отдельно рассмотрим язык C#. Struct в C# это не совсем то, что есть в Java. Им требовалась структура, как просто последовательность объектов последовательно размещенных в памяти. Например:

```
class Point{
    public int x;
    public int y;
};
void DrawLine (Point[] points);
```

То есть передается массив указателей на объекты класса. Для передачи требуется создать массив, потом уничтожить массив, что требует огромных затрат памяти. При этом этот массив требуется только для хранения точек, то есть не оптимально использовать классы просто как контейнеры для объектов. Поэтому было введено понятие структуры, синтаксис объявления структуры и класса аналогичен, при этом нет никаких отличии от класс (в том числе и права доступа), кроме того что структура не имеет референциальности, то есть относится к типам значений и размещается не в динамической памяти. То есть
point[] points = new point[1000]; То в случае структуры это будет именно массив объектов но не указателей. При этом распределение памяти идет одним куском.

В C# не существует глобальной переменных, за исключением их некоторых аналогов — статических членов класса. Так же нет глобальных функций. Для таких структур определены операции развертки / завертывания в классы, что позволяет работать с ними как с обычными классами.

```
Point p1 = o[1];//нельзя
```

Point p1 = point(o[1]);// нормально

При этом для структур нельзя перегружать конструктор умолчания. При создании массива всегда запускается конструктор по умолчанию для структур. Для объектов класса такой проблемы нет, так как там можно явно вызвать конструктор.

Другие типы данных.

По идее их нет. Единственный ЯП который имеет доп типы — Паскаль, в котором есть встроенные типы и операции ввода — вывода. В остальных ЯП это является частью стандартной библиотеки.

Множества, таблицы.
В Паскале было понятие множества, в модуле было так же понятие множества а так же понятие bitset. Bitset был удобен для манипуляции битами внутри слова, при этом в Обероне остался только тип данных set (абстракция понятия множества).

Лекция 14.

Описание ввода вывода требует наличие функций с переменным числом параметров. Критерий мощности языка — возможность написать на нем библиотеку ввода вывода для него самого.

Множества, таблицы и прочее реализуются именно стандартной библиотекой, так как нет универсального представления таких объектов для всех архитектур. Та же ситуация и с алгоритмами, например алгоритмами сортировки.

Несколько другая ситуация со строками. В паскале строка — упакованный массив символов, в С — char * , то есть массив символов, заканчивающийся символом с кодом 0. Для них определены операции сравнения на равенство и на больше /меньше. В ранних языках программирования работа со строками была в стандартной библиотеке, далее с развитием ЯП строки стали переходить в базис ЯП.

В языке Ада есть тип данных STRING — неограниченный массив, то есть чтоб объявить строку требуется указать диапазон.

Для строк есть набор часто встречающихся операций, которые легко реализуются на языке ада. Например часть операций реализуется на вырезке массивов. Так же есть неприятная операция — конкатенация, (operator+). В языке Ада есть возможность переопределения операций, то есть тип STRING реализован как класс стандартной библиотеки, как частный случай массива. Отличие базиса ЯП от стандартной библиотеки в том, что базис языка встроен в компилятор, а стандартная библиотека — просто набор документов.

С++ - один из немногих ЯП в котором строки не включены в базис языка, а есть стандартный библиотечный класс std::string. В С++ есть возможность перекрытия всех операций, что являлось необходимым требованием к реализации стандартного класса.

На самом деле, строка — очень частный случай динамического массива, но из-за различия в требованиях к строкам и к массивам, строка не реализована как std::vector<char>. В основном к строкам применимы операции копирования и конкатенации/вырезки. В С++ есть три вида массивов — std::vector, std::valarray, std::string. В С++ все это могло быть и было реализовано исключительно на механизме классов. ТО же самое нельзя так же сделать на Java или Delphi. Так как там все это реализуется компилятором. Аналогично в языке C#, при этом примера стандартного класса Java — встроенный в Java библиотеку архиватор.

Таким образом сейчас стараются упрощать базис, вынося все в стандартную библиотеку, исключение — строки, так как для них важна скорость работы.

Пункт 5. О единстве СТД.

Массивы — операция [] (индексация)
Может быть переопределена в C++, Delphi (нетривиально), C#(через индексатор).
Записи — операция . (Доступ по имени поля)

Нельзя переопределять в C++. Как определить аргумент и возвращаемое значение этой операции? По хорошему аргументом должно быть смещение от начала записи. В некоторых языках для всех объектов определена операция []. Точка тогда трактуется как [«name»]. То есть таким образом были отождествлены записи и массивы. Таким образом в Javascript obj [0] — обращение самому первому члену записи.

В чисто интерпретируемых языках можно позволить массивам быть полностью динамическими. Таким образом можно добавить некоторые свойства записям, что увеличит гибкость языка, но за счет гибкости и надежности.

Глава 3.
Управления последовательностью действий. Операторный базис ЯП.

Фундаментальные базисные понятия — данные (состояния) и управление данными (переходом из состояния в состояние). Традиционные ЯП ориентированны на поток управления.

Эволюция понятия потока управления. Традиционно различают три уровня потока управления
1.Внутри выражения.
2.Между операторами.
3.Между модулями.

Над каждым ТД есть набор операций. Поток управления — порядок вычисления операций, определяется приоритетом операций. Более интересный вопрос, когда приоритетов нет. То есть например в a+b в каком порядке будут вычисляться операнды выражений. Большинство ЯП говорят что если программа зависит от порядка вычисления операндов, то такая программа не считается стандартизованной. Это не распространяется на логические выражения. Например:
While (A[i] <> X) and (I < N) do Inc(I) end;
Здесь ошибка — выход заграницу индекса.

Если компилятор имеет право вычислять выражение в произвольном порядке, то программа становится менее красивой. Иногда имеет смысл

зафиксировать «ленивость вычислений». В Описании языка Ада предлагалось ввесли ленивые логические операции — andthen orelse. Но в результате просто была зафиксирована ленивость вычисления всех логических выражений.

Программирование без побочных эффектов позволит полностью устранить эту проблемы для арифметических вычислений. При этом в функциональных ЯП побочных эффектов быть не может по определению.

Эволюция — в начала под потоком управления подразумевали произвольную передачу управления в другую точку программы по оператору goto. Ситуация стала меняться когда стали писать более сложные программы — например ОС. В 1967 году Дейкстра опубликовал свою статью - «О вреде оператора goto». Это была революционная статья. До этого разработчики ЯП старались дать программистам как можно больше возможностей,при этом основная мысль Дейкстры - программирования с помощью меньшего количества средств может стать более осмысленным и простым. Он ввел понятие структурного программирования, то есть программирования на основе блоков. Идея программирования не просто изобретения алгоритма или его записи, но собственно в «изобретении». Эта статья положила начало дисциплине, названной Технологией программирования.

В 1968 появилась статья «Заметки о структурном программировании», статья о подпрограммах, который произвел революцию в программировании. Любую операторную схему можно реализовать с использование оператора
a;b
while B do S

Остальные могут быть реализованы на базе этих абстракций. Далее все языки восприняли эти принципы — принципы Структурного программирования.

Альтернативы goto — циклы, ветвления.
Двустороннее ветвление — if(<smth>) S1 else S2;
Проблема — if B then if B2 then S1 else S2;
То есть

```
if(B)
{
    if (B2) S1
}
else S2
```

или

```
if (B)
{
    if (B2) S1
    else S2
}
```

По стандарту верна вторая структура. Для первой нужно вводить доп скобки. C/C++ C#Java приняли такую идеологию. Альтернатива — язык Ада, модула-2, VB, Оберон. Там отказались от понятия составного оператора, то есть требуется явные терминатор каждого оператора. <oper>...end<oper>.

Лекция 15.
Операторный базис ЯП.

Основная проблема операторов ветвления - проблема вложенности. Правила вложенности - см прошлую лекцию. Эта проблема является частным случаем проблемы, которая возникае когда по синтаксису требуется один оператор, а по смыслу - несколько. Для этого было введено понятие составного оператора.

Альтернативное решение - замыкание оператора. То есть любой оператор, который может содержать другие операторы, требует явного объявления конца оператора.

И тогда проблема вложенности легко решается путем явного указания конца оператора.
Например

```
if B1 then
    S11;S12;S13;
endif
```

Потом многие ЯП переняли эту структуру, разработанную для Алгола - 60. При этом, в языке Ада ет слова end, там обязательно требуется явное указание того, что именно заканчивается. Например, endif.

Многовариантное ветвление

Во многих языкам многовариантное ветвление выполняется за счет вложенности. То есть может моделироваться с помощью последовательности вопросов.

```
if B1 then
    S1
else
    if B2 then
    S2
    else
//...
```

При этом вложенные структуры крайне плохо воспринимаются человеком. Ему проще воспринимат структуры типа

```
if B1 then S1;
if B2 then S2;
if B3 then S3;
```

Поэтому обычно используется такая запись

```
if (B1)
    S1;
else if (B2)
    S2;
else if (B3)
    S3;
else
```

При этом чтоб закончить подобную структуру на языке Ада, требуется написать огромное количество endif, что усложняет написани и читабельность программы. Таким образом, требуется оператор выбора. В языке Ада есть спецциальные оператор elsif, в языке модула - 2 — ELIF

$if \;\; B_1 \;\; then$
seq_1
$elsif \;\; B_2 \;\; then$
seq_2
...
$elsif \;\; B_n \;\; then$
seq_n

Рассмотрим отдельно один частный случай - оператор дискретного выбора (оператор - переключатель).

```
case Expr of
//список вариантов
end
```

При этом поведение программы, когда Expr не удовлятворяет ни одному варианту не определенен. Но во всех реализация обычно специфицируют, что просто ничего не выполняется (эквивелентно пустому оператору, если Expr не имеет побочных эффектов).

В модула - 2 синтаксис несколько другой

```
CASE EXPR OF
список вариантов через|
{ELSE
операторы }
end
```

В языке Ада

```
case Expr of
```

```
when условия => операторы
//....
{when other => операторы}
end case;
```

В языке C
switch (Expr) S

При этом case 0:case 1:case 2: - метки с константами. Если отыскалась метка с соотвествующим знечением константы, переходит переход к этой метке, но не выбор соотвествующего фрагмента кода. То есть, основная проблема такого оператора, забытые операторы break, из - за чего происходят интересные ошибки в программах.

Интересное решение было принято в языке C#, там в случай отсутствия break, компилятор выдает ошибку, то есть семантика оператора switch эквивелетна своему аналогу в Паскале.

Если в список констант не входит значение Expr то выполняется метка default, и если ее нет, то получаем пустой оператор.

Остались еще опеарторы цикла. Структура операторов цикла стабилизировалась. При этом принцип структурного программирования - программирования в принципах управляющих структур. Рассмотрми стандартный набор циклов (например языка паскаль).

```
while B do S;
repeat S1;..SN; until B;
for
```

При этом в 1974 году - "Структурное программирования с оператором goto"
Идея статьи - пструктура программы должна соотвествовать структуре алгоритма. То есть можно структурно программировать даже на Фортране, использую лишь только goto. То есть многие задачи даже более удобны с использованием goto.

Аналогично в Модула - 2

```
WHILE B DO ... END;
REPEAT ... UNTIL B;
LOOP ... END;
```

Третий оператор имеет смысл либо в параллельном программировании, либо вместе с оператором EXIT - аналог break только для оператора LOOP (принцим минимализма). Эта же ситуация сохранилась в Обероне

и Обероне - 2.

В других языках программирования похожая ситуация. Язык Ада
while B loop
....
end loop;

for

loop .... end loop;
Так же там присутствует оператор exit, при этом там допустима вторая форма
when условие => exit;
И это условие можно поставить в конце цикла.

Рассмотрия язык C. Там практически тот же набор операторов.
while (B) S
do S while (B);
for
Присуттвуют так же операторы break и continue (особые случаи оператора goto).

Рассмотрми так же оператор return - частный случай оператора передачи управления. Далее в языке java есть возможность выхода по break по метке. То есть выход по break <label> выхдит из того цикла, который помечен меткой. Другой вариант - использование исключений (очень плохо - нельзя так писать).

Без goto можно программировать, так как за 13 лет он так и не появился в java.

Рассотрим цикл for, он был изобретен создателями языка Алгол. Цикл for появился в Паскале в виде for v:= e1 to e2 do S;Причем e1 и e2 вычислялись один раз. Интересно, что в языке модула - 2 цикл for больше соотвествовал варианту Алгола, чем Паскаля, то есть там позволялось задавать шаг цикла.

1988 - Оберон - убрали цикл for.
1993 - Новая версия языка Оберон - 2. И в нем появились многомерные открытые массивы, аналоги виртуальных методов и вернули в оберон цикл for.

Рассмотрим отдельно цикл for. В Аде он имеет вид

for v in диапазон loop
for i in A'RANGE loop S:= S + A (i); end loop;

При этом переменная i локализованна внутри цикла, и описывать ее не требуется. Похоже сделанно и в C++
for (int i = 0;...)

В языке C семантика цикла for несколько другое, там это самый мощный цикл, который может использоваться для любых целей.
for (e1;e2;e3)

В языках C# и java всегда осуществляется квазистатический контроль даже, если мы идем в цикле for по элементам массива, что может быть не эффективным, поэтому в языке C# появилась особая форма этого цикла
foreach (T o in C) S;
int []a ;
foreach (int x in a) S+=x;

Где C - произвольная коллекция (поддерживает интерфейс IEnumerable), при этом коллекция нельзя модифицировать с помощью такого перебора. В языке Java такой цикл появился только в 2005 году
for (T o:c)

TimerTask cansel();
collection<TimerTask> c;
for (TimerTask t:c) t.cansel();

Лекция 16.
Глава 4.
Процедурные абстракции ЯП.

Процедурные абстракции с точки зрения:
- Потока управления
- Потока данных

Процедурные типы данных.

Пункт 1. Передача управления в ПА. Подпрограммы и сопрограммы. Всегда есть один вход (оператор Call) и один выход (Return). Оператр возврата не во всех ЯП усть и может быть в самых разных частях программи. Оператор return - обобщение оператора break. Таким образом, все операторы return идут в коней процедуры.

В C++ оператор return - оператор перехода на метку. Только в C++ есть автоматический вызов деструкторов локальных объектов.

Первые библиотеки были созданны на языке фортран. В некоторых ЯП была возможность вызывать подпрограммы из нескольких входов, то есть существовало несколько входов в процедуру. Так же у процедуры могло существовать несколько выходов, причем возврат отпеределялся при вызове.

SUBROUTINE P (N, *)

call P (N, 25)

Как уже говорилось, были попытки определеить дополнительные точки входа и выхода в процедуру, то есть была возможность входить в процедуру не с начала, и возвращать управление не в точку вызова. Эта концепция совершенно не привилась, то есть во всех современных ЯП у подпрограммы имеется одна точка входа и одна точка выхода.

С точки зрение передачи управления, существует четкая несимметрия между вызывающей программов и вызванной. Выполнений подпрограммы всегда начинается с точки входа и заканчивается возвратом в точку вызова. Такая система была еще в АЛГОЛ - 60.

Альтернативный вариант - сделать программу симметричной путем перехода от понятия subroutine к coroutine, то есть есть несколько как бы параллельно работающих подпрограмм. То есть при повторном входе в процедуру вход будет в ту точку, из которой был выход при прошлом обращении к этой подпрограмме. Это некоторый аналог механихма параллельных процессов.

Таким образом отсутствует понятие возврата (есть только передача управления) и понятия главной программы.

В явном видет такое понятие есть только в языке МОДУЛА - 2. Там был реализован механизм квазипараллельных процессов. Понятие подпрограммы в этом ЯП:
ADDRESS
PROCEDURE TRANSFER (var cor1,cor2: ADRESS);

Таким образом для каждого вызова необходимо запоминать не только адрес возврата, но и все локальные переменные, то есть при операторе перехода необходимо восстановить весь локальный контекст для вызываемой процедуры и сохранить его для вызывающей. Так же требуется некоторая процедура инициализации контекста для первого вызова сопрограммы. Для этого используется
procedure NEWPROCESS (p: PROC, VAR cor: ADDRESS; N : INTEGER);

Где третий параметр - размер памяти дял сохранений контекста. То есть это довольно незкоуровненый вызов. Таким образом, не удалось создать достаточно простой высокоуровневый механизм создания сопрограмм.

Так же была процедура iotransfer, одним из аргументов которой был номер некоторого прерывания, с которым был связан вызов продеруры.

Однако, в современных ЯП отсутствует такая концепция. ОНа была заменена концепцией параллельного процесса. При этом остается вопрос о включении этого механизма имеено в базис ЯП. Тогда возникает множество проблем связанных с производительностью, при работе со средствами парарллелизма уже имеющимися в системе.

В такой проблемой столкнулись и создатели языка Java, но они ввели понятие параллельных программ на уровне JVM. Более того возможен вызов родных вызовов ОС с помощью JNI.

Механизм передачи данных в подпрограммы.

Вариант передачи данных в сопрограммы - глобальные переменные.

Есть три типа передачи данных.
1. Возврат из функции.
2. Передача параметров.
3. Глобальны переменные.
Мы будем рассматривать второй пункт.

Существуе две вещи
– in-out семантика передачи параметров

– способы или механизмы передачи параметров.

CALLER CALLE
In ->
OUT <-
In_Out <-> нам нужно и входное значение, и оно будет меняться.

Механизм передачи параметрво - реализация (низкоуровневая). Семантика - передача параметров с точки зрения ЯП, не вникая в способы реализации. Таким образом, в ЯП высокого уровня принято говорить именно о семантике, но не о реализации.

Способ языка Ада - каждый параметр имог иметь один из трех спецификаторов - in, out, inout.
function ABS(D : In REAL) return REAL;

procedure (A : in ARR; MAX : out REAL; IND: out integer);

procedure swap (x,y :inout T);

При этом в современных ЯП фиксируется именно механизм передачи параметров.
- по значениеи.
- по результату.
- по значению / результату.
- по имени.
- по ссылке.

При передачи параметров возникает 2 сущьности - формальный и фактиченский параметр. При передачи параметров во момент вызова функции происходит связывание формальных параметров с фактическими. Есть 2 момента связывания - момент вызова и момент возврата.

Связывание по ссылке - в момент вызова, и связь не разрывается до момента выхода. Под формальными параметрами всегда понимаются некоторые переменные, которые почти эквиваленты локальным переменным. Самый простой способ их расположения - расположение на регистрах процессора или стеке. При передачи параметров через стек становится возможным механизм рекурсивных вызвов. То есть в Фортране параметры располагались в сегменте данных, что исключало возможность вызова рекурсивных функций.

Передача по значению - копирования фактического параметра в формальные при вызове (например запись параметров в стек). В таких языках как C и Java все передается по значению.

Связывание по результату - в фактический параметр копируется значение формального параметра при возврате. Такой способ реализован в всех ЯП при возврате из функции.

Связывание по результату - копирование фактического в формальный при вызове и обратное копирование при возврате.

Самый простой способ реализации - передача по значению, особенно если в аппаратуре реализован аппаратный стек. Этих трех операций достаточно для реализации всех семантик передачи параметров, но с точки зрения эффективности их не достаточно.

Передача по ссылке - передача адреса по значению.

Рассмотрми последний способ передачи параметров - по имени. Такой способ эквивалентен тому что соотвествующий фактический параметр подставляется на место соотвествующего формального параметра. То есть это некоторый аналог макроподстановок. Таким образом в АЛГОЛ-60 при вызове P(A[i]) значение A[i] может измениться если меняется значение i.

Рассмотрми процедуру swap на алголе.

```
procedure swap (a,b);
integer a,b;
begin
    integer tmp;
    tmp = a;
    a = b;
    b = tmp;
end;
```

Такая реализация не работает. Например

```
array 1..N o ff integer;
i := 1;
swap (A[i], i); будет работать.
swap (i ,A [i]); не заработает.
```

Именно из - за такого механизма передачи параметров АЛГОЛ - 60 не смог стать промышленным ЯП. То есть для каждого вызова требовалось по имени переменной определить ее адрес.

Лекция 17
28.10.2008

In/Out семантика и способы передачи параметров с прошлой лекции. Почти во всех ЯП реализован способ передачи данных по значению. В С только по значению, в С++ появилась возможность передавать по ссылке.

Pascal, Modula - 2, Oberon - по значению и по ссылке (параметры переменные). In гарантирует что объект не изменится, при этом при передаче по ссылке объект может измениться. Вирт посчитал защиту от подобного излишней, хотя в языке С++ такая защита есть - константная ссылка.

Если объект описан с модификатором const, то он получив значение не может изменить его. То есть он постоянен в пределах некоторого блока. То есть если не константному объеку сопоставленна константная ссылка, то это значит что в пределах блока, в котором определена ссылка данный объект не меняется (передача в функцию константной ссылки).

В С++ Т& - передача объектов out/inout, для in следует использовать const T&. При этом Т - передача по значению. В С++ любые временные обекты считаются константными.

Ада-83 и Ада ввобще. Там на самом деле не указывается способ передачи параметров, а указывается семантика, то есть при передаче параметров указывается одно из ключевых слов -in, out или inout

```
procedure P(X: in T;y: out TT; z : inout TTT);
```

При этом компилятор сам выбирает способ передаич параметров. При этом в стандарте Ада - 95 вернулись к способу, который был реализован почти во всех ЯП. Все ЯП реализуют передачу либо по ссылке, либо по значению.

Однако, в некоторых случаях программы на языке Ада работали по разному на разных компиляторах.Например рассмотрим такую программу.

```
procedure P(x,y : out T) is
begin
    X:=...;raise ERROR;
    Y:=...;
end P;
```

P(A,A);

Если параметры передаются по ссылке, то значение A меняется, а если по результату - то меняется. Таким образом, нормальная программа ведет себя по разному в зависимости от компилятора.

Java - передача по значению, но из-за референциальной семантики объектов - фактическая передача по значению только простых типов даннных. Все остальные объекты (в том числе и массивы) - по ссылке. Таким образом для всех простых объектов - in-семантика, для всех остальных - inout семантика.

В Java есть ключевое слово final - аналог в данном контексте модификатора const. А для простых типов данных остается только in семантика. Расмотрим понятие побочного эффекта функции. Побочнй эффект означает, что функция может менять либо какие - то глобальные объекты и/или меняет свои фактические параметры. Таким образом, основная задача процедуры - имеенно побочные эффекты, когда как функция по хорошоме не должна иметь оных.

Глобальная переменная - признак дурного дизайна программы в современном понимании.

function F(x: T) return T; { in }

procedure FF(x: inout T): return T; { inout } не вошло в окончательную версию Ада

procedure P(x: out T; r: inout T);

В Java для каждого простого типа данных существует класс - обертка в пакете java.lang.
int a;
Integer v = a;
f(v); //inout

Все что находится в пакете java.lang - встроено в компилятор.

Рассмотрми язык C#. Для него были крайне актуальны вопросы эффективности. Все параметры там передаются по значению по умолчанию, то есть передача полностью аналогична Java. Но есть два отдельных мождификатора - ref и out, которые реализуют передачу параметров по ссылке.То есть
void f(ref int a) {a = -1};// передача по ссылке
int a = 0;

f(ref a);{a == -1}

Однако здесь требуется передача инициализированной переменной. Для передачи не инициализированной переменной используется модификатор out.

В языке Delphi
procedure P ([var] a: X);

Список аргументов переменной длины

printf (const char *, ...);
va_list
va_start
va_next
va_end

Недостаток - ненадежность. То есть нет возможности контролировать количество переданных функции аргументов.
CString.format (const char *, ...);
Несмотря на то что в C++ существует свой механизм ввода - вывода, многие до сих пор используют C-шный printf из-за его удобства.

В языке оберон или модула-2 нет списков переменной длины, то есть операции вывода намного менее удобны, чем в C.

printf ("Count = %d i); // C
InOut.WriteString ("Count = ");
InOut.WriteInt (i); модула - 2

В C++ остались сишные макросы для списков аргументов переменной длины.

В Java и C# проблема обеспечения надежности была решена. В этих языках реализация опиралась на то, что все объекты имели общего предка - класс Object. Таким образом, список переменной длины - массив объектов типа object.
void f(params int[] integer);
Таким образом, write (String format, params Object[] objs);

Надежность обеспечивается на уровне выполнения, путем контроля параметров.

Следует различать объявления
void f(params int [] elements);
void f(int [] integers);

Аналогичная проблема была в языке java, причем создатели языка не хотели расширять язык, вводя новые ключевые слова. Вместо этого они просто синтаксически расширили язык. То есть

```
void f(int ... integers)

{ for (int i:integers) {..} }
```

На этом закончим с передачей параметров.

Пункт 3. Подпрограмные типы данных

```
TYPE PROCINT = PROCEDURE (INTEGER);
VAR P: PROCINT;

TYPE F = PROCEDURE (REAL):REAL;

PROC PP = PROCEDURE (VAR T);
V:= const {константы - имена процедур}

P(5);
```

В C/C++ имеется механизм указателей на функцию. В Ада - 95 появился как подпрограмный тип, так и подпрограмные переменные. Когда как в Ада 85 их не было. За время с Ада 83 до Ада 95 изменились как взгляди на программирование, так и условия, в которых должны были работать программы на этом языке.

-референция
-композиция
-наследование
-делегирование

Делегирование послужило причиной введениея подпрограмных типов. В языке Оберон обработчик (handler) - аналог подпрограмного типа. Невозможно полностью реализовать ОО поведение без процедурного типа.

Лекция 18.
30.10.2008

Рассмотрми реализацию подпрограмных типов данных.
1. Подпрограммы - указатели (CALL P). Примеры языков - C, C++, Оберон, Модула-2, Паскаль (Delphi), Ada-95.
M-2, Oberon, Delphi: type PRCI = procedure (integer).

Над объектами такого типа есть только две операции - присваивание и вызов. В Ада - 83 не было подпрограмного типа, он появился лишь в Ада - 95.

В Ада - 95 был введен подпрограмный тип данных, который описывался аналогично указателю.

```
type PRCI is access procedure PRCi (i : in integer);

procedure FCALL (p: PRCi) is
begin
    P(0);
end FCALL;
```

Появляется специальная разновидность формальных параметрова, которые явно передавались по ссылке. Таким образом попытка использовать задание семантики, а не способа передачи провалилась.

Отдельно рассмотрим C++ и Delphi. В этих языках есть понятие класса, что дабавляет к понятию процедуры или функции понятие метода класса, то есть процедуры или функции, которой неявно передается объект класса (this в C++, self в Delphi).

```
void foo (int);
typedef void (*footype)(int)
footype f;
f = foo;
(*f)(0);
f(0);

class Bar{
    void foo (int);
    int i;
};
```

Таким образом, появляется понятие указателя на член класса.

```
int Bar::*pb;
pb = Bar::i;//ошибка
Bar b;
```

```
pb = &b.i;//указатель на член класса типа int

typedef void (Bar::*)foom(int);
foom f;
f = Bar::foo;
```

Операции .* или ->*

```
type MEMPRC = procedure (integer) of Bar;
```

Отдельно рассмотрми C# и java. В языке java понятие функционального типа отсутствует, то есть передавать функции как параметры нельзя. При этом в java можно реализовать подобный механизм на основе наследования.

Рассмотрим пример - процедуру интегрирования, то есть вычисления определенного интеграла f(x)

```
typedef doube (*FI)(double);
double Integer (double a, double b, double eps, IF f);
```

В языке java все несколько иначе, подобная реализация возможна только с помощью классов.

```
class integral{
    public double IF (double) ..
    public double integrate (double a, double b, double eps);
}
```

Теперь чтобы передать в класс нужную функцию, нужно унаследовать класс integral и переопределить в новом классе функцию IF. Такое решение было вызвано тем, что из - за соображений безопасности разработчики java полность отказались от всех указателей, в том числе и указателей на функции.

Что же касается языка C#, то там есть понятие delegate - специализированного фиункционального типа данных, который сводится к указателю на функции.

```
public delegate void Operation(int);
```

Делегату можно присваивать новые занчениея.

```
public Operation dlgt;
```

```
class Bar{
public Foo (int x) {..}
public static void Foo2 (int i) {...}
};

Bar b;
dlgt = new Operation (b.Foo);
```
Теперь dlgt (0) == b.Foo (0);
При этом можно написать и так:
dlgt = new Operation (Bar.Foo2);// теперь указывает на статическую функцию.

Более того для делегатов определениа операция += и -=, создающие цепочки операций.
```
dlgt += new Operation (b.Foo);
dlgt += new Operation (Bar.Foo2);
dlgt += new Operation (new X().b);
```
То есть мы привязывем к делегату функцию от нового класса, определенного при вызове.

При вызове dlgt (0), будут применены все эти операции в том порядке, в котором они были заданы. Все классы delegate происходят от класса
```
System.Delegate {
    static Delegate Combine (Delegate first, Delegate second);
    static Delegate Remove (Delegate first, Delegate second);
    static Delegate GetInvocationList (Delegation smth); }
```
Понятие события.

subcribe - желание получать уведомление о том что произошло событие.
distribute - генерация события.
Пример обратного - таймер или почтовая система.
EventSource - ему принадлежит переменная дилагатского типа, а EventConsumer - подписчик, он использует += и -=.
События - частные случаи делегатов, то есть Event - частный случай экзампляра делегата.

```
public event Operation;
```
При этом у события ограниченны операции.

Для большинства ЯП единицей инкапсуляции является тип, однако это не так в случае событий, так как в зависимости от объекта различа-

ется набор операций (+= и -= или создание события (вызов)).

```
Operation op;
public delegate int Operation2(int x, int y);
Operation2 op;
op = delegate (int x, int y){return x+y}

int i;
op = delegate (int x, int y){return x + y + i};

Oper0 o;
public Delegate[] Create(){
    Oper0 [] ops = new Oper0 [3];
    int i = 0;
    for (int k = 0; k < 3; k ++) ops [k] = delegate ()return i ++;
    return ops;
}
```

Анонимный делегат может захватывать не только локальные переменные, но и например свой контекст - класс. То есть существует возможность реализации связывания.

```
class Binder{
    private Operation2 d;
    private int arg;
    public Buiner (Operation2 a, int x)del = a, arg = x
    public Operation1 Get_Bind (){return delegate(int a){return d (a, arg)}}
}
```

Все это является аналогом замыкания и Lambda функций.

Лекция 19
6.11.2008

Глава 5. Логические модули традиционных ЯП.
Определение новых типов данный с помощью логических модулей.
Инкапсуляуция и АТД.

Пункт 1. Понятие ЛМ.

Логический модуль - языковая конструкция, объединяющая взаимосвязанные ресурсы в именованный набор, который позволяет согласованно
- использовать
- распостранять
- изменять
- транслировать

эти ресурсы

Понятие логического модуля тесно связанно с понятием физического модуля. Но в общем случае они не равнозначны.

Главная задача логического модуля в большинстве традиционных ЯП - связать во елино структура данных и соотвествующие множество операций, то есть создание нового типа данных.

Понятие класса (в C++)- мягкая эволюция понятие структуры. Теперь класс является взаимосвязанным набором ресурсов. Таким образом, класс является как типом данных, так и логическим модулем.

При этом основной задаче класса все же остается задача создания нового типа.

В языке Фортран -все подпрограммы - физические модули.

Логические модули в случаях М-2, Оберон, Delphi

В Модула - 2:
- Главный модуль
- Библиотечный модуль
- Локальные модули.

```
MODULE имя;
Объявления
BEGIN
Операторы
END имя.
```

Библиотечные модули:

DEFINITION MODULE M;
Объявления;
END M;
- Модуль определениея
- объекты данных
- типы данных
- заголовки процедур и функций

- Модуль реализации:
IMPLEMENTATION MODULE M;
Объявления и реализации
END M.
- Определения

```
DEFINITION MODULE STACK;

    CONST N = 256;

        TYPE Stack =

            RECORD
            TOP:INTEGER
            BODY:ARRAY [0..N] of REAL

    END;

PROCEDURE INIT (VAR S:Stack);
PROCEDURE POP (VAR S: Stack) : REAL;

VAR Done : Boolean;
```

В модуле реализации должны быть реализации объявленных функций. Все структуры данных, объявленные в модуле реализации не доступны извне.
.DEF => .SYM
.MOD => .OBJ

DM
  => unit => DFV
IM

DM

=> unit (unit stack)

interface

...

implemenation

...

end (stack);

Взаимодействие модулей между собой.

Каждый библиотечный модуль экспортируе все имена, которые объявлениы в модуле определения.

Модуль видимости - имя либо видимо, либо не видимо.

Видимость (видимые имена) делятся на два класса

- потенциальная видимость (требует уточнения)
- непосредственная видимость (может быть использованно без всяких уточнений)

Имеется специальная конструкция IMPORT, которая имеет 2 вида. 1.IMPORT <список имен БМ> (делает потенциально видимывсе все имена, которые объявленны)

VAR S:Stacks, Stack;

FROM Stacks IMPORT Stack, Pop

Пусти имеется имя X - тогда это либо:

- стандартный идентификатор
- локальный объект
- имя из импортируемого модуля, тогда имеет вид M.x
- имя импортированное с помощью импорта во второй форме

В условиях использования разнородных модулей возможен конфликт имен. Возможен либо конфликт между именем из импортируемого модуля и именем описанным, в нашем модуле.

В Модуа - 2 нет перегрузки имен, то есть в пределах одного модуля все имена уникальны.

Если есть конфликт локального имени с именем, полученным по uses, то приоритет имеет локальное имя.

Другой вариант конфликта - конфликт 2 импортируемых имен, то есть

uses M1, M2;

Причем в обоих модулях есть имя X, тогда обращение может быть только по M1.x или M2.x

Самый неприятный момент - конфликт двух модулей с одинаковыми названиями. Один из вариантов решения - использование уникальных комбинаций имен в именах в модулях.
GL_.. для openGL
glu.. для glu

Модули делятся на сегменты экспортируемых и неведимых имен.

В языке Оберон Вирт отказался от понятия главного и локального модуля. Предполагается, что программа на языке Оберон погружена в консольныу систему. Есть специальная комманда для загрузки модуля M в память, после чего пользователь может выполять любую процедуру из этого модуля.
Load M;
M.p

В 1988 году появился Оберон - 2. В самом первом обероне было всего 2 модуля - DEFINITION (аналогично модулю в модула -2) и MODULE (просто модуль).

В пересмотренной версии языка оберон, Никлаус Вирт решил, что программы на языке оберон не могут быть написанны в отрыве от системы программирования или операционной системы. Что привело к тому, что модуль определения может быть сгенерирован автоматически, на основе модуля с описаниями. В модуле все, что требуется поместить в экспортируемое, помечается звездочкой, то есть если требуется экспортировать имя, то его определение помечается звездочкой.

А модуль определения может быть сгенерирован автоматически на основе исходного модуля с помощью специальных утилит.

В данном случае, возникает проблема с доступом на чтение / запись к структурам. Для спецификации того, что переменная экспортируется только на чтение служит символ *-. При этом inline процедур нет впринципе.

Далее в яэыке оберон осталось только единственная форма импорта
IMPORT <список имен модулей>
То есть
IMPORT InOut;
InOut.writeln;

То есть нет непосредственной видимости глобальных импортируемых имен.

Пункт 3. Логические модуле в случае языка Ада.

```
package имя is
    объявления;
end имя;
package body имя is
    определения
end имя;
```

Первая часть называется пакетом, а вторая - телом пакета.

В языке Ада пакеты могут быть вложенны друг в друга. Считается что есть специальный пакет STANDARD.

```
packade STANDARD is
    объявления из STANDARD
    объявления пользовательский пакетов
end STANDARD;
```

В модула - 2 загрузка модуля - загрузка соотвествующей таблици имен, в Ада вс сложнее.

```
STANDARD
...
package P is
    package P1 is
    end P1;
end P;
package PP is
    package P2 is
    ...
    end P2;
    package P3;
    ...
    end P3;
end PP;
```

Видимость тел пакетов точно такая же что и у вложенностей тел объявлений. Всегда видимы имена из пакета STANDARD. Имена становятся видимы после объявления и перестают быть видимы по окончании спецификации пакета. Все имена становятся видимыми потенциально, аналогично языку Модула.

Лекция 20
11.11.2008

Основная возможность логического модуля - возможность определение нового типа данных.
Библиотечный модуль - Модула - 2
Модуль - Оберон
Unit - Delphi

При этом в языке Ада возможны вложенные модули и возможна сложная структра вложенности спецификации пакетов. Причем вложенность тел пакетов имеет такую же вложенность, что и спецификация.

Модуль не обязам иметь тело, но всегда обязан иметь интерфейс. Например, если модуль имеет только определения объектов данных или констант, то implementation часть не требуется.

```
package P is
    package P1 is
    package P12 is
    end P12
end P1
```

Тогда тела вложенны аналогично.

```
package body P is
    package body P1 is
    package body P12 is
    package AUX is
    ....
    end AUX
    package body AUX is
    ...
    end AUX
    end P12
    end P1
end P
```

Имеется 2 подхода к проектированию программ.
Первый - сверху вниз, то есть начинать с главного модуля (самого абстрактного) и идти вниз. Недостаток - до программирования модуля самого нижнего уровня не получается запустить систему. (Top - bottom).

Второй - снизу вверх (Bottom-up).

Для упрощение работы была введена концепция вложенных модулей, то есть не для каждого модуля на данной ступени иерархии трубеются не все модули из нижнего уровня иерархии. Это сильно упрощает проектирования программы, но сильно усложняет язык.

Перегузка или полиморфизм - одному имени соответствует несколько объявления. Все традиционные ЯП придерживаются идеи что в одной области видимости одному имени соотвествует только единственные объект. В современных ЯП полиморфизм распостраняется только на имена процедур и функций.

Не имеют перегрузки только "Виртовские"ЯП - Оберон и Модула. В языке Ада есть перегрузка процедур и функций, причем учитываются и возвращаемые значения. То есть допустима такая перегрузка:

```
function P:boolean;
function P:integer;
```

В java, C++ и аналагичных ЯП такая перегрузка недопистима. Далее возникает вопрос и перегрузке стандартных операций. Тут существует 2 течения.

В первое входит Ада, C#, C++ — позволяющие перегружать стандаротные операции. Требование перегрузки стандартных операций было в стандарте Ада, причем перегрузка не должна была влиять на синтаксический анализ.

То есть если перегрузка не должна была влиять на местность операций.

Классический пример перегрузки операторов - матричная арифметика. То есть

```
package Matrices is
    type Matrix is ...;
    function"+"(M1,M2: Matrix):return Matrix;
    .....
end Matrices;

x,y,z:Matrix;
z:=x + y;
z:="+"(x,y)
```

Однако все имена являются лишь потенциально видимыми, то есть такой код не верен. Требуется специфицировать пакет, то есть писать так: x,y,z:Matrices.Matrix;

z:=Matrices."+"(x,y)

Таким образом, мы приходим к требованию конструкции, которая снимает потенциальную видимость. Такой конструкцией является именованный импорт, который снимает потенциальную видимость, если нет конфликта имен.
use <список имен - объектов>;
То есть возможно написать так.

```
use Matrices;

x,y,z:Matrix;
z:=x + y;
```

Все проблемы начинаются, когда появляется конфликт имен. И в общем случае главные проблемы связанны со сложной структурой пространства имен, и с возможностью использования процедуры неконтролируемого импорта (use - делает всю спецификацию пакета непосредственно видимой).

Таблица имен языка Модула - 2 намного проще чем таблица имен языка Ада. В Модула - 2 требуется только стек таблиц, в языке Ада стек остается,но каждая из таблиц в стеке может содержать в себе другие подтаблицы.

Пункт 4. Инкапсуляция и абстрактные типы данных.

Понятие абстрактного типа данных вошло в обиход вместе с понятием инкапсуляции. Абстрактный тип данных - тип данных, у которого вся структура инкапсулированна.

Говорят что объектно—ориентированный язык - язык в котором имеется 3 понятия - инкапсуляция, наследование и динамический полиморфизм. Инкапсуляция - некоторое скрытие деталей. Основная задача при проектировании структуры программы - проектирования интерфейса, для чего требуется именно механих инкапсуляции.

Понятие единицы защиты почти во всех ЯП - тип. То есть все объекты одного типа данных обладают одинаковыми свойствами с точки зрения зашиты данных. Атомом же защиты является либо тип целиком, либо часть типа - переменная, метод и так далее. В языках Модула - 2 или Ада атомом защиты является тип, то есть либо все закрывается либо все открывается. То есть требуется закрывать все члены данных.

В языках Оберон и языках с классами атомом защиты является член класса.

Рассмотрим пример на модула - 2.

```
DEFINITION MODULE M;
    ....
END M; EMPLEMENTATION MODULE M;
    ....
END M;
```

В Ада так же есть спецификация и тело пакета. Все что объявленно в модуле определения видно извне, все что объявлено в модуле реализации закрыто.

Оберон.

```
TYPE T* = RECORD
    v1:T1;
    v2:T2;
    end;
```

С этой точки зрения оберон поход на языки с классовой структурой, то есть импортируется имя типа, но поля данных являются закрытыми.

Итак абстрактный тип данных в общем случае - это просто множество операций. Рассмотрим реализации АТД.

В языке Модула - 2 АТД определяется как скрытый тип данных.Он объявлен в модуле определения просто как TYPE T;.То есть напрмиер

```
DEFINITION MODULE stacks;
    TYPE STACK;
    PROCUDERE Init (var s:stack);
    PROCUDURE Push (var s:stack, x:integer);
    ....
END stacks;
```

Над таким типом даннх применимы операции, определенные в модуле, а так де операции присванивания и сравнения на равенство / неравенство. Причем семантика этих операций исключительно поверхностная, то есть производится исключительно поверхностное копирования. Структуры АТД должна быть описанна внутри модуля реализации, причем на определение накладываются некоторые ограничения.

Таким образом, для всех объектов АТД используется референцальная модель доступа. Напишем определение стека.

```
TYPE PLINK = POINTER TO LINK;
TYPE LINK = RECORD
```

NEXT:PLINK;
INF:INTEGER:
END: Stack = plink;

Таким образом, в любом случае все объекты АТД - указатели на что - либо.

Таким образом Модула - 2 заставляет программировать в терминах АТД и более того в терминах референциальной модели данных.

Лекция 21.
13.11.2008

Ссылочная модель работы была связанна с механизмом раздельной компиляции и принципом РОРИ.

Принцип РОРИ - разделение определения, реализации и использования.В структуре таких ЯП, как модула - 2, Ада, Delphi, четко видно, что определение, реализация и использования четко разделены. В Модула - 2 - DEFINITION, IMPLEMENTATION. В Ада - спецификация пакета и определение пакета, причем они могут транслироваться отдельно. В Модула - 2 так же модули определения и описания могут транслироваться отдельно.

С точки зрения раздельной трансляции, компилятору необходимо знать только ту часть, которая относится к определению, поэтому эта часть вынесена в отдельную языковую конструкцию.

Для того чтобы откомпилировать клиентский модуль, компилятору необходим только DEFINITION модуль сервиса, который использует клиент. То есть если поменялся модуль определения, то требуется перекомпилировать все модули, которые его используют.

Аналогично сделано в и ЯП Ада и Delphi.
uses Service;

В C/C++ роль модулей определения играют заголовочные файлы, но при незначительном изменении одного заголовочного файла требуется перетранслировать все файлы, которые используют данный заголовочный файл.

При этом в Delphi играют роль только серьезные изменения в интерфейсной часть модуля.То есть если изменить модуль то он будет перетранслирован, а остальные только в том случае если был изменен интерфейс модуля.

Вернемся к реализации АТД с помощью скрытого типа. Допустим компилятор встречает
import service
и подгружает таблицу имен, и тут выясняется что есть
TYPE HANDLER;
после чего компилятор видит
VAR H:HANDLER;
Другой информации у него нет, то есть он не может произвести распределения памяти. Поэтому было введено ограничение, что это может быть либо целый тип, либо указатель, то есть имеет место исключительно референциальная структура.

Создатели языка Ада понимали эту проблему. В Аде абстрактный тип данных - действительно абстрактный тип. Существует 2 концепции - приватные тип данных и ограниченные приватный тип данных.

Основной недостаток с точки зрения инкапсуляции - требование употреблять исключительно референциальную модель данных.

Рассмотрим пример на языке Ада

```
public Stacks is
  type Stack is private;
  -- операции над стеком
  procedure push (S:inout stack, X:integer);
  procedure pop (S:inout stack, X:out integer);
private:
  type stack is record
             top:integer := 0;
             body: array (0..N) of integer;
  end record;
end Stacks
```

В Языке Ада побочный эффект от функций запрещен, то есть нельзя оформить POP как функцию. Если хотя бы один из типов данных объявлен как приватный, то в спецификации пакета появляется дополнительная приватная часть, где должна быть расписана структура типа.

Теперь возможно использовать методы приватного типа, но доступ к его закрытым структурам запрещен.
Push (S,1);
Pop (S,X);
S.top := 1; Ошибка

```
type PLINK is access;
type link is record
   next: PLINK;
   X: integer;
end record;
type PLINK is access link;
```

В приватной части описываются все структуры, нужные данному объекты. Ее полностью видит компилятор, но она не доступна программисту. То есть, чтобы заменить реализацию стека с массива на список, то следует добавить в текст

type Stack is access PLINK;

Теперь необходимо перетранслировать все модули, которые используют этот тип данных. Такой тип данных не является полностью абстрактным. По сравнению с Модула - 2 к данному типу применимы операции присваивания и сравнения.

Как видно, в случае первой реализации стека стандартное присваивание и сравнение на равенство (поверхностные операции) работают корректно, когда как в втором случае эти операции могу быть некорректными.

Для решения этой проблемы были введены ограниченные приватные типы данных.

```
type T is limited private
```

Теперь к этому типу применимы только те операции, которые описаны в самом пакете.

Оберон

```
Record
  top:Integer;
  body:ARRAY N OF INTEGER;
END;

TYPE Stack=
 RECORD
END;
```

Реализация в Обероне не уступает в точки зрения скорости реализации Модула - 2, но улучшает гибкость языка. Но Вирт отошел от принципа РОРИ, то есть совмещены определения и реализация.Причиной этого стала уверенность вирта в том, что программирования на Обероне всегда будет происходить в некоторой среде программирования.

То есть принцип разделения определения и реализации был переложен на документирующие системы. В C# и Java та же схема.

Итак раздел про инкапсуляцию закончен.

Глава 6
Класс. Определение нового типа данных с помощью классов.

Класс - это всегда новый тип данных. Класс может быть реализован в стандартной библиотеке или встроен в компилятор. Например в .NET классы Object и String встроены в компилятор.

Модуль является более универсальным понятием чем класс. Например в C++ все что принадлежит классу, содержится в this. В C# можно вводить операторы преобразования для соответствующего класса. В C++ это решалось с помощью операции преобразования, которая являлась членом класса T1 и приводила к классу T2.

В C# это реализовано с помощью статических функций членов либо T1 либо T2.

В C# и Java отсутствуют глобальные функции. Их заменяют статических функции закрытых классов. Его нельзя наследовать и объект такого класса нельзя породить. То есть это является не классом, а скорее модулем. В C# есть понятие статического класса - полный аналог модуля. Таким образом, модуль более универсален, но менее удобен.

Пункт 1. Члены класса.

class name: наследование{
список членов
};

Такой синтаксис присутствует в Java, C#, C++. В C#, Java, Delphi если родитель не указан, то родителем считается класс Object.

В Delphi:
type name = class (наследование)
список членов-данных
end;

C++ и Delphi поддерживают принцип РОРИ. В C++ мы может описывать тело функции внутри тела класса, что позволяет компилятору сделать функцию встраиваемой. В Delphi запрещено описывать тело функции в описании и интерфейса класса.

```
class X{
  int i;
  void f(int);
};

void X::f(int){..}

Delphi:
type T = class
         procedure P;
```

```
end;
procedure T.P;
begin
end;
```

Пункт 1. Члены классов.
-члены-данные
-члены-функции
-члены-типы

Все члены класса можно разделить на статические и не статические.

Не статические члены - данные — обобщение понятие члена записи. Обращение к ним идет <имя объекта>.<имя члена>. В C++ так же есть операция ->.

Статические члены - данные с точки зрения распределения памяти - глобальные переменные. В C++ Доступ к ним по имени класса C::i. В C# и Java, C.i. В C++ можно обращаться к статическим членам и через объект класса.

Лекция 22
18.11.2008

Для статических членов класса имя класса выступает как имя модуля. Обращение к члену пространства имен черех имя пространства имен и к статическому члену класса через имя класса синтаксически эквивалентны.

Рассмотрим распределение памяти. Для нестатических членов помять отводится там же, где и и память для объекта класса. Со статическими членами совсем другая ситуация. В C# и Java память отводится системой в сегменте данных. В C++ сам программист должен указать в какой файле будет размещен член данных.

```
/*X.h*/
class X{
  static int a;
};

/*X.cpp*/
int X::a;
```

В *.h лучше не включать, так как возможно переопределение имени.

Поговорим о статических и не – статические члены - функции. Не – статическая функция - особая функция, которая отличается следующими вещами:
1. Класс - область действия.(то есть функции члены транслируются в контексте своего класса)
2. Неявный параметр (адрес объекта). В C++ - это this (указатель).В C# и Java - this (ссылка). В D и Smalltalk - self.

То есть

```
class X{
  int i;
  void f(){
    this -> i = 0;
    i = 2;
  }
};
```

У каждой функции есть несколько областей действия. То есть тело функции является вложенным по отношению к телу класса. Наледование так же дает вложенность (потомки вложенны в предков).

Отдельно рассмотрим формальные параметры. В Delphi параметры имеют область действия класса (то есть их имена не могут совпадать с именами класса). В C/C++ они принадлежат телу функции.

Для доступа к закрытым полям класса следует использовать указатель this->. Вариант разрешения конфликта имен - накладывать ограничения на имена.

В статических функция - членах нельзя обращаться к нестатическим членам класса через this. Обращение возможно только через явное указание имени объекта.

Все функции-члены имеют максимальные права доступа к членам класса.

Для статических членов данных нет никакой разницы между указателем на глобальные данные и указателен на статический член класса.

```
int * pi;
int a;
pi = &a;
int X::stat = -1;
pi = &X::stat;
```

Аналогично с именами функций

```
void (*pf)(void);
pf = X::f;
```

Если речь идет о указателях на члены данные, то все не так. int X::* p; - указатель на член класса X.

```
class X{
  int i;
  int y;
  void f();
  void g();
};

p = &X::i; Тогда здесь хранится смещение поля i в классе X.
p = &X::y; Тогда здесь хранится смещение поля y в классе X.

void (X::*)(void) PFX;
PFX = & X::f;
PFX = & X::g;

X anx;
```

```
anx.*p;
anx.*PFX();
px -> p;
px -> * PFX ();
```

Во всех языках допускеатся размещать константные и не константные объекты.
const int i;

Этот член является контантой и может быть инициализирован прямо в определении (так как объект целого типа). Но такие члены могут быть только static. Иначе пришлось бы хранить на каждый объект класса много лишних данных.

В C# если речь идет о константе, то слово const говорит о том, что выражение может быть вычисленно статически и что объект является статическим. Так же в C# есть модификатор readonly. Он специфицирует то, что после инициализации объекта он не может потерять свое значение, но он не обязан быть статическим и может быть инициализирован любым выражением во время выполнения.

Перейдем к третьему варианту данных - типам.

```
class X{
  class Y{
  };
};

Например:
STL
vector{
  iterator{
  };
};
```

В Java внутренний класс - нестатический вложенный класс. Статический вложенный класс - обычный вложенный класс. Внутренний класс очень глубоко связан с объемлющим классом. Например любой объект вложенного класса имеет ссылку на объект объемлещего класса.

```
class BankAccount{
 int Account;
 class Action{
```

```
  int k;
  void f(){k,account}
 }
}

BankAccount X;
BankAccount.Action a = X.new Action();
```

Внутри класса BankAccount можно писать короче

```
class BankAccount{
 int Account;
 class Action{
  int k;
  void f(){k,account}
 }
 void dept ()
 {
  Action depth = new Action ();
 }
}
```

В C# тоже есть понятие статического класса. При этом нестатический класс C# - это аналог статического класса языка Java И обычного вложенного класса языка C++. Если же класс является статическим, то он может содержать только статические членые данных и статические функции.

Например класс math, содержащий определения математических функций и математических констант.

Статический класс нельзя наследовать.

В языке Delphi нет понятия вложенных классов и статических функций и процедур.

```
unit M;
interface
  function GLB (a : integer):integer;
  type X = char;
    var
    i: integer;
    constructor ();
    procedure t;
implementation
end;
```

Инкапсуляция

Следует различать управление доступом и управление видимостью.В C++ есть всего 3 уровня доступа - public (любая функция имеет доступ), protected (только объекты потомки) и private (только функции - члены).

```
T operator+(const T&, const T&);
T T::operator+(const T&);
T& T::operator+=(const T&);

T operator+(const T& t1, const T& t2)
{
  T temp = t1;
  return temp += t2;
}

friend void f();
friend void V::g (T&);
friend class Z;
```

Дружба не наследуется, она не транзитивна и не может быть получена.

В Java есть четвертая категория областей доступа - пакет. То есть функции из классов, находящихся в тмо же пакете. В C# вместо пакета - assembly.

Соответственно есть и 4 метода доступа.
public ++++
protected +++-
internal +±+-
private +—

Лекция 23
20.11.2008

В C++ структуры по умолчанию имеют права доступа public. В C++ все отличия структур от классов заключаются только в правах доступа по умолчанию. Класс же по умолчанию имеет права доступа private.

Правила C++ были не удобны для работы с понятием модуля (набора ресурсов). Языки в которых есть понятия модуля (Delphi, Java, C#) имеют особые спецификации доступа для объектов из того же модуля.

В Java есть понятие пакета (не путать с пакетом Ада). Package - единица управления контекстом и единица дистрибуции.

В C# namespace - единица управления контекстом, сборка - единица управление дистрибуцией. При этом сборка - это не понятие уровня языка. То есть нет ключевого слова assembly, но когда происходит сборка, то она помещается именно в assembly.

В Java появляется пакетный доступ (модификатора нет, является доступом по умолчанию). В C# понятие internal (доступ для всех классов из той же сборки).

По умолчанию в C# - private.

	свой	потомок	производный	мир(чужой)
public	+	+	+	+
protected	+	+	+*	-
internal	+	+**	+	-
private	+	-	-	-

*В Java protected является и пакетные доступ. В C# если protected, то доступ только из производных.В C# есть двойной модификатор, то есть protected internal - доступ и для потомков и для объектов той же сборки.

**В Java если класс описан в том же пакете (той же сборке C#), то он имеет доступ, иначе - нет.

Как видно, правила усложнились, но понятия друга больше не требуется.

В Delphi есть 3 ключевых слова (работают как переключатели) - public, private, protected. В случае описания класса используется объявления по умолчанию.

Доступ по умолчанию совпадает с пакетным доступом C# и Java.

```
type  MyClass = class
                объявления по умолчанию
```

Так как нет ключевого слова - то должны размещаться в начале. Пакетный доступ - public для своего модуля, private для других. Есть так же ключевое слово published - аналог private, но видны в визуальном редакторе ресурсов.

Абстрактный тип данных - тип данных, с которым можно работать только через определенный интерфейс. С точки зрения класса, абстрактный тип данных - класс, у которого нет публичных членов данных.

Есть понятие рефакторинга - изменение кода программы без изменения его работы. Задача рефакторинга - улучшение качества кода.

Чтоб сделать класс по настоящему абстрактным, требуется позаботиться о смысле операции инициализации, копирования, присваивания.

Фундаментальным понятием ООП является понятие абстрактного типа данных и абстрактного класса. Класс является типом данных, но абстрактный класс не является абстрактным типом данных.

Разновидностью абстрактного типа данных является понятие интерфейса. Интерфейс - просто перечисление операций, которые можно выполнять.

Рассмотрим семантику protected."Наивная семантика если член помечен как protected, то к нему возможен доступ из любых функций членов.

```
class X{
private:
  int i;
  void f() {i = 0;}
};
class Y:public X{
  public:
  void f(){i = 0;}//Ошибка
  void g(X& x) {x.i = 0}//Ошибка
};
//===========================================
class X{
protected:
  int i;
  void f() {i = 0;}
};
class Y:public X{
  public:
  void f(){i = 0;}//Correct
  void g(X& x) {x.i = 0}//ERROR
};
```

Итак, если член объявлен как protected в base, то функции - члене производного класса возможен доступ через ссылку на этот же класс,или производный от него.

```
class Z:private X{
};
Y y;
Z z;
y.f (z);//Доступ открыт.
```

Каждый класс, при объявлении прав доступа, объявляет контракт. Private - все это класс оставляет за собой.Public - все открывает. Protected - производным членам можно менять контракт.

Дополнительный свойства классов.
Специальный функции.

Специальные функции - о них компилятор знает что - то дополнительное и способен сам вставлять вызовы таких функций неявно.
Пункт 3. Создание и уничтожение объектов (классов).

При создании объекта всегда вызывается особая функция - конструктор. Конструктор может быть пустым, тогда его вызов может быть опущен компилятором.

В Delphi, Java, C# – исключительно референциальная модель объекта. В C++ объекты делятся по классу памяти на 3 типа - статические, квазистатические и динамические

```
C++
class X{
  X (){}
};

C#, Java
class X{
  X (){}
}
```

В Delphi все несколько иначе, там конструктор - процедура с произвольным именем, помеченная как constructor.

```
type X = class
         constructor Load;
```

В Delphi конструкторы наследуются, каждый объект имеет в качестве предка объект класса Object, который имеет конструктор create.
X a; вызов конструктора по умолчанию.
X a();прототип функции.
Возможно и так
X px = new X;

В C++ объект может быть объявлен как под-объект. Для статических объектов - параметры передаются в в вызове конструктора, для динамических - через оператор new.
X px = new X;

В C# и Java единственный метод создание объекта - new
px = new X;

Вопрос - инициализация под-объектов объекта. В C++ они инициализируются в конструкторе, причем если явно не указанны конструкторы под-объектов, то вызывается конструктор по умолчанию.

```
class Y;
class X{
  Y y;
};
```

Таким образом, необходимо обеспечить вызов конструктора базового класса и конструкторов под-объектов. Это делается в конструкторе базового класса.

```
class Y;
class X:public Base{
  Y y;
  X (){}
};
```

При вызове конструктора класса X будет произведен вызов конструктора класса Base и Y.

Рассмотрим язык C#

```
class X: Base{
  Y a;
  X (){}
};
```

В C++ сначала вызывается конструктор базового класса, потом конструкторы под-объектов, и только потом тело конструктора. В C# конструкторы под-объектов не вызываются (по умолчанию будет nil). Но для типов значений будет вызван конструктор по умолчанию (который для структур нельзя переопределить). Для Base будет вызван конструктор по умолчанию.

И в Java и в C# есть механизм инициализаторов, то есть возможно написать.

```
class X: Base{
  Y a = new Y();
  X (){}
};
```

В C++ в случае нежелания (невозможности) вызова конструктора по умолчанию требуется использовать список инициализации.

```
<конструктор>::=
     имя_класса ([аргументы])[:список инициализации]{<тело>}

X ():Y(0), Base (1){}
```

Список инициализации отрабатывает раньше тела конструктора. Причем, если в списке нет конструкторов некоторых под-объектов, то будут вызваны конструкторы по умолчанию.

В C# и Java вызов конструктора по умолчанию был сведен до вызова конструктора только базового класса.

```
C#
X():this (0), base (-1)
```

Base (-1) - вызов конструктора базового класса (отличного от конструктора по умолчанию). This (0) - для под-объекта.

```
B Java:

class X extends Base{
  X(){
   super (0)// вызов конструктора базового
   this (-1)
}
```

Это должны быть обязательно первыми операторами конструктора.

Все конструкторы вызываются начиная с самого базового. В C++ сначала вызывается конструктор базового класс, потом конструкторы под-классов и только потом тело конструктор основного объекта.

Таким образом, нельзя вызывать виртуальные методы в конструкторе в C++, но это возможно в Java и C#

```
C#
class X:Base {
  Z z = new Y();
  X();
}

Z -> Y -> Base -> X

C++
class Base {
  Y y;
  Base ();
};
class X:Base {
  Z z;
  X();
};

Y -> Base -> Z -> X
```

В C++ дополнительно появляются конструкторы копирования.
X (const X&) {...}
Какой список инициализации будет сгенерирован. Будет вызван конструктор умолчания. Это парадоксально, так как конструктор копирования по умолчанию производит почленное копирование под-членов.

X (const X&):base (x) {...} – явный вызов.

В C# и Java нет конструктора копирования (ибо референциальная модель данных). Так де в них нет конструктора преобразования, так как в Java запрещены неявные преобразования, а в C# они описываются с помощью статических методов.

Лекция 24
25.11.2008

Рассмотрим язык Delphi - там наиболее простая ситуация. Там наследуются конструкторы и деструкторы. Более того, они всегда вызываются явно, то есть там нет особых форм конструкторов.

```
type X = class
  constructor load;
  destructor Destroy;
end;
```

Если конструкторов или деструкторов нет, то они не создаются. Так как конструкторы наследуются, то все объекты имеют конструктор create и деструктор destroy. Когда вызывается конструктор, мы уверены что таблица виртуальных методов имеет корректный вид (в C++ это не так).

Итак конструкторы и деструкторы наследуются но не генерируются. Никаких конструкторов умолчания нету и для вызова конструктора базового класса требуется вызывать его явно. Для этого служит специальное ключевое слово
inherited - Delphi
super - java
base - C#

В Delphi для вызова конструктора базового класса требуется первым оператором в объекте написать inherited Create. При этом это должен быть именно первым оператором в конструкторе, так как конструктор create забивает объект нулями. Для получения размера объекта, create использует виртуальную функцию получения размера объекта.

Важно что имя объекта не используется, то есть при смене имени не требуется переписывать наследуемый класс.

В Delphi нет автоматической сборки мусора в общем виде, но при работе через стандартные интерфейсы она может быть. Это приводит к тому, что программы на Delphi Провоцируют большое числу утечек памяти.
var a: X;
a := X.create;
a := X.Create (1);

a.Free; - не деструктор, но уничножение объекта, который может вызывать деструктор.

Деструктор не требуется явно вызывать явно, так как функция Free не только вызывает деструктор, но и вызывает менеджер памяти для очистки. Так как функция не может менять значение self, то вызов Free не может инициализировать a := nil.

Статические объекты

В C++ все статические объекты создаются до вызова функции main. Это единственное что известно про него.
X a(0);
X b(1);

Все что известно, это то что a и b будут созданы до вызова main и удалены после выхода из нее. Но порядок их вызова не определен. В общем случае C++ не дает способов управления процессов инициализации статических данных, и требуется изобретать дургие способы.

В C# и Java все несколько иначе. В Java для объекта могут стоять инициализаторы.

```
class X{
  static Y a = new Y ();
}

//Java
class X{
  static int []a;
  static {
    a = new int [N];
    for (int i = 0; i < N; i ++) a[i] = i;
  }
}
```

В C# есть возможность простой статической инициализации, а так же есть статический конструктор - полный аналог статического блока в Java. При этом это должен быть конструктором по умолчанию.

В Delphi статический объектов вообще нет. Но в interface можно описать не только методы.

Так же у каждого модуля присутствует инициализационная и финализируюшщая часть, которые выполяются соотвественно при входе и выходя из модуля.

В C++ не сборки мусора, в его реализации в системе .NET она присутствует, но в стандарте его нет. Вызов конструткра обязательно привязывается к инициализации и вызов деструктора - к удалению.

Важная черта C++ - свертка стека, что гарантирует вызов деструкторов для локальных объектов.
RAII - Приобретение ресурса is initialisation

Если класс захватывает ресурсы - то конструктор - это захват, деструктор - освобождение. В C++ единственная возможность утечки памяти - работа с динамической памяти.

В C# и Java есть сборка мусора, но для нее нет универсального эффективного алгоритма. Первый вариант алгоритма - счетчик ссылок. Такая методика очень проста, но она некорректно работает на кольцевых ссылках.

Другая методика - Mark And Sweep. Его недостаток - возможна внезапное неуправляемое замедление программы (в любой момент может включиться сборка мусора).

В C++ проблема при работе с памятью - фрагментирование памяти. M&S позволяет избежать фрагментации, но он меняет адрес объекта.Но это не единставенная проблема этого алгоритма. В C# и Java вызов деструктора вообще не гарантирован, так как он вызывается только при уничтожении объекта, то есть когда не хватает динамической памяти.

Сборка мусора не позволяет явно контролировать очистку ресурсов.
В классе object есть protected метод
protected void finalise ();
Сейчас сборка мусора - параллельный поток с низким приоритетом.

```
if (!close) closed ();
```

Так должен выглядеть метод finalise.

В C# все еще хуже. В C# позволяется писать деструктор, причем синтаксис его точно такой же как и в C++, но работает он не так. В деструкторе вызывается метод метод finalise класса object. То есть деструкторя по сути является тем же финализатором.

В Java метод очистки ресурсов либо вызывается явно, либо в финализаторе.

```
c = new X();// захват ресурса
try {
   return;
}
finally {
  c.close ();//очистка ресурсов
}
```

При этом finally - это не ловушка.

В C# ввели впециальный интерфейс - IDisposible, который состоит из единого метода void Dispose (), который говорит, что объект не обязательно финализировать, замещая собой финализацию.

```
using ( Image im = Imaj.FromFile ())
{
   //...
}
```

Теперь после using блока вызовутся методы dispose для всех объектов из using блока (который (Image im = Imaj.FromFile ()))

В Java есть два вида ссылок. Сильная ссылка - обычная ссылка, слабая ссылка:
WeakReference
Object getReference ();

Слабые ссылки - ссылки на уже освобожденную память с точки зрения программиста, но память еще не освобождена. getReference для слабой ссылки "реанимирует"объект.

Пример - кеш броузера, при переходе на след страницу требуется освободить кеш броузера. Ссылкам на них присваиваются нули и они переходят в разряд слабых ссылок. Если вернуться назад, то вместо запрашиваия ресурсов повторно можно рассмотреть очередь слабых ссылок и попробовать достать объекты. Если объект недоступен, то просто вернется null.

Итак, мы получаем механизм кеширования ресурсов.

Пункт 4. Дополнительные проблемы, связанные с типами.

- копирование, сравнение
- неявные преобразования

Проблема копирования и сравнения объекта - могут отличаться от стандартных операций копирования или сравнения. Как сравнивать и можно ли ограничиться простым почленным копированием. Отсюда 2 понятия - глубокое копирование и поверхностное копирование. Deep copy и shallow copy.

На вопрос о том, как именно копировать может ответить только программист. В C++ стандартное - поверхностное копирование. В других языках стандартное присваивание - поверхностное копирование. В C# и

Java есть методы clone (). В обоих языках есть специальный интерфейс Icloneable.

С C# в классе Object есть метод MemberwiseClone (), реализующий поверхностное почленное копирование, чтоб избежать этого объект должен реализовывать интерфейс ICloneable, в котором должен быть метод clone ().

Лекция 24
25.11.2008

Рассмотрим язык Delphi - там наиболее простая ситуация. Там наследуются конструкторы и деструкторы. Более того, они всегда вызываются явно, то есть там нет особых форм конструкторов.

```
type X = class
  constructor load;
  destructor Destroy;
end;
```

Если конструкторов или деструкторов нет, то они не создаются. Так как конструкторы наследуются, то все объекты имеют конструктор create и деструктор destroy. Когда вызывается конструктор, мы уверены что таблица виртуальных методов имеет корректный вид (в C++ это не так).

Итак конструкторы и деструкторы наследуются но не генерируются. Никаких конструкторов умолчания нету и для вызова конструктора базового класса требуется вызывать его явно. Для этого служит специальное ключевое слово
inherited - Delphi
super - java
base - C#

В Delphi для вызова конструктора базового класса требуется первым оператором в объекте написать inherited Create. При этом это должен быть именно первым оператором в конструкторе, так как конструктор create забивает объект нулями. Для получения размера объекта, create использует виртуальную функцию получения размера объекта.

Важно что имя объекта не используется, то есть при смене имени не требуется переписывать наследуемый класс.

В Delphi нет автоматической сборки мусора в общем виде, но при работе через стандартные интерфейсы она может быть. Это приводит к тому, что программы на Delphi Провоцируют большое числу утечек памяти.
var a: X;
a := X.create;
a := X.Create (1);

a.Free; - не деструктор, но уничножение объекта, который может вызывать деструктор.

Деструктор не требуется явно вызывать явно, так как функция Free не только вызывает деструктор, но и вызывает менеджер памяти для очистки. Так как функция не может менять значение self, то вызов Free не может инициализировать a := nil.

Статические объекты

В C++ все статические объекты создаются до вызова функции main. Это единственное что известно про него.
X a(0);
X b(1);

Все что известно, это то что a и b будут созданы до вызова main и удалены после выхода из нее. Но порядок их вызова не определен. В общем случае C++ не дает способов управления процессов инициализации статических данных, и требуется изобретать дургие способы.

В C# и Java все несколько иначе. В Java для объекта могут стоять инициализаторы.

```
class X{
  static Y a = new Y ();
}

//Java
class X{
  static int []a;
  static {
    a = new int [N];
    for (int i = 0; i < N; i ++) a[i] = i;
  }
}
```

В C# есть возможность простой статической инициализации, а так же есть статический конструктор - полный аналог статического блока в Java. При этом это должен быть конструктором по умолчанию.

В Delphi статический объектов вообще нет. Но в interface можно описать не только методы.

Так же у каждого модуля присутствует инициализационная и финализируюшщая часть, которые выполяются соотвественно при входе и выходя из модуля.

В C++ не сборки мусора, в его реализации в системе .NET она присутствует, но в стандарте его нет. Вызов конструткра обязательно привязывается к инициализации и вызов деструктора - к удалению.

Важная черта C++ - свертка стека, что гарантирует вызов деструкторов для локальных объектов.
RAII - Приобретение ресурса is initialisation

Если класс захватывает ресурсы - то конструктор - это захват, деструктор - освобождение. В C++ единственная возможность утечки памяти - работа с динамической памяти.

В C# и Java есть сборка мусора, но для нее нет универсального эффективного алгоритма. Первый вариант алгоритма - счетчик ссылок. Такая методика очень проста, но она некорректно работает на кольцевых ссылках.

Другая методика - Mark And Sweep. Его недостаток - возможна внезапное неуправляемое замедление программы (в любой момент может включиться сборка мусора).

В C++ проблема при работе с памятью - фрагментирование памяти. M&S позволяет избежать фрагментации, но он меняет адрес объекта.Но это не единставенная проблема этого алгоритма. В C# и Java вызов деструктора вообще не гарантирован, так как он вызывается только при уничтожении объекта, то есть когда не хватает динамической памяти.

Сборка мусора не позволяет явно контролировать очистку ресурсов.
В классе object есть protected метод
protected void finalise ();
Сейчас сборка мусора - параллельный поток с низким приоритетом.

```
if (!close) closed ();
```

Так должен выглядеть метод finalise.

В C# все еще хуже. В C# позволяется писать деструктор, причем синтаксис его точно такой же как и в C++, но работает он не так. В деструкторе вызывается метод метод finalise класса object. То есть деструкторя по сути является тем же финализатором.

В Java метод очистки ресурсов либо вызывается явно, либо в финализаторе.

```
c = new X();// захват ресурса
try {
   return;
}
finally {
  c.close ();//очистка ресурсов
}
```

При этом finally - это не ловушка.

В C# ввели впециальный интерфейс - IDisposible, который состоит из единого метода void Dispose (), который говорит, что объект не обязательно финализировать, замещая собой финализацию.

```
using ( Image im = Imaj.FromFile ())
{
   //...
}
```

Теперь после using блока вызовутся методы dispose для всех объектов из using блока (который (Image im = Imaj.FromFile ()))

В Java есть два вида ссылок. Сильная ссылка - обычная ссылка, слабая ссылка:
WeakReference
Object getReference ();

Слабые ссылки - ссылки на уже освобожденную память с точки зрения программиста, но память еще не освобождена. getReference для слабой ссылки "реанимирует"объект.

Пример - кеш броузера, при переходе на след страницу требуется освободить кеш броузера. Ссылкам на них присваиваются нули и они переходят в разряд слабых ссылок. Если вернуться назад, то вместо запрашиваия ресурсов повторно можно рассмотреть очередь слабых ссылок и попробовать достать объекты. Если объект недоступен, то просто вернется null.

Итак, мы получаем механизм кеширования ресурсов.

Пункт 4. Дополнительные проблемы, связанные с типами.

- копирование, сравнение
- неявные преобразования

Проблема копирования и сравнения объекта - могут отличаться от стандартных операций копирования или сравнения. Как сравнивать и можно ли ограничиться простым почленным копированием. Отсюда 2 понятия - глубокое копирование и поверхностное копирование. Deep copy и shallow copy.

На вопрос о том, как именно копировать может ответить только программист. В C++ стандартное - поверхностное копирование. В других языках стандартное присваивание - поверхностное копирование. В C# и

Java есть методы clone (). В обоих языках есть специальный интерфейс Icloneable.

С C# в классе Object есть метод MemberwiseClone (), реализующий поверхностное почленное копирование, чтоб избежать этого объект должен реализовывать интерфейс ICloneable, в котором должен быть метод clone ().

Лекция 25
27.11.2008

Основная проблема поверхностного копирования - возможность побочного эффекта при изменении копии объекта. Такой проблемы в принципе нет для тех объектов, которые являются константными.

Современные ЯП позволяют программисту решать эту проблему. В C++ по умолчанию используется поверхностное копирования, но программист может переопределить конструктор копирования. Все вопросы копирования решаются только на уровне класса, так как конструктор копирования и оператор присваивания не наследуются.

Например пусть есть класс X, в котором переопределены оператор присваивания и конструктор копирования.

```
class X{
  X& operator= (const X&);
  X (const X&);
};

class Y:public X{
  int i;
  Z z;
};
```

Теперь в классе Y вызываются конструкторы по умолчанию, если не будет явно указанно вызван конструктора, пи этом для класса X, так как там нет конструктора по умолчанию, то возникнет ошибка.

В C# и Java все несколько иначе. Любая операция присваивания там - присваивание ссылки. При этом там присутствует операция побитовое копирование, которая всегда является protected.

В Java присутствует операция clone(), которая возвращает копию объекта.
protected Object clone ();

В C# есть интерфейс интерфейс Icloneable. То есть для того, чтоб разрешить клонирование объекта, требуется унаследовать этот интерфейс и определить publick Object clone ();

Тут возникает несколько проблем, например если коллекция является клонируемой, то добавление к ней любого не клонируемого объекта делает ее не клонируемой.

Более предпочтительным является подход языка Java. Он основан на понятии интерфейса - маркера. В Java это интерфейс, у которого

вообще нет никаких методов. В Java есть понятие рефлекции, то есть возможности получения информации о объекте на основе его кода. Там присутствует интерфейс - маркер Icloneable. То есть класс, которые реализует такой интерфейс, обязуется реализовать интерфейс глубокого клонирования.

В C++ есть 3 стратегии клонирования
- глубокое
- поверхностное
- приватное

В Java есть 4 варианта.
1.класс реализует интерфейс Icloneable, то есть класс должен реализовать
Public Object clone ();

Если класс не поддерживает этот интерфейс, то есть если вызывается недопустимая конструкция копирования, то выбрасывается исключение.

2. Класс запрещает клонирование, то есть он реализует public метод clone, который выбрасывает исключение cloneNotSupported.
3. Условная поддержка клонирования. Класс явно реализует интерфейс Icloneable, но метод clone поддерживает глубокое копирования объектов класс, но не под объектов, то есть при попытке глубокой копии копирования под объекта, который не поддерживает интерфейс Icloneable выбрасывается исключение.
4.Неявная поддержка. Класс не реализует интерфейс, но реализует protected Object clone (), которая возвращает побитовую копию, то есть поддержка дается только производным классам.

Так же на Java можно выразить запрет клонирования как наследование операции clone от класса Object.

Операция сравнения

В C++ по умолчанию сравнивать объекты нельзя, для сравнения необходимо переопределять операцию сравнения.

В Java и C#, есть методы
C#: bool Equals (Object);
Java: bool equals (Object);

Пусть есть 2 объекта, a и b
a, b;
a.Equals (b)

Если b == null, то получим false, но есть a - null, то будет ошибка. Поэтому присутствует статическая функция сравнения от двух параметров.

Для строк проблема сравнения еще более сложна, так как требуется сортировка строк в алфавитном порядке.

Лекция 26

2.12.2008

Перегрузка операторов. Неявные преобразования.

int i;
double d;
d = i;
i = d;
Вторая строчка эквивалентна i = (int)d;

Почти во всех языках есть расширяющие неявные преобразования. Сужающие только в C. При этом единственных ЯП в котором запрещены неявные преобразования - язык Ада.

Современные ЯП как правило допускают неявные преобразования между базовыми типами. Но вопрос о пользовательских неявных преобразованиях не столь очевиден.

В "C с классами"не было пользовательских неявных преобразований, но потом они появились в C++. В Delphi и Java запрещены пользовательские неявные преобразования. В C++ и C# можно перегружать стандартные операции, в Delphi и Java - нет.

Неявные преобразования

Любое неявное преобразование - это потеря информации или ненадежность. Более того, семантика неявного преобразования компилятора может отличаться от той, которую предполагал пользователь.
$A = B \times \exp(-k \times i)/D$

На Fortran такое выражение записываться просто, когда как на C, требуется определить все операции для работы с комплексными числами.
A = mult (B, Div (cexp (mult (-k, i)), D));

Причем на языке C эта формула не откомпилируется, так как требуется преобразование от integer к complex. На C++ требуется либо написать пользовательские преобразования в тип complex, либо перегрузить операцию для всех числовых типов.

Таким образом, ясно что без неявных преобразований требуется 16^2 операций для всех возможный вариантов вызова Div. Для решения данной проблемы в C++ был введен механизм неявных преобразований и понятие конструктора преобразования.

complex(double)

Тогда определение оператора могло выглядеть так.
const complex operator+(const complex&, const complex &);

Аналогично для строк string (const char *);

Далее в C++ появился оператор преобразования. operator(T)() - определение неявного преобразования из класса X в тип T. Например operator(const char*)() const

Но тут возникает проблема

```
class Vector{
  int * body;
  int size;
public:
  vector (int sz){body = new int [size = sz];}
  ~vector() {...}
};
//....
Vector v(20);
v = 3;
v = Vector (3);
```

Последние две строчки могут быть эквивалентны. Но такая семантика реализуется за счет неявного вызова конструктора, что могло не ожидаться. Для решения этой проблемы было введено ключевое слово explisit.

В C# есть и ключевое слово implisit и explisit, которые могут стоять перед определением каждого оператора преобразования, причем в отличии от C++ по умолчанию преобразование явное.

Явное преобразование должно вызываться программистом явно.
T t; X x;
t = x;

Тогда компилятор автоматически вставит t = x.operator(T)(), если такая операция определена программистом.

Создатели Delphi и Java отказались от неявных преобразований вообще, а в C# все операции преобразования - статические функции класса.

```
class X{
  public implisit operator(Y)(X x){...}
  // X -> Y
```

При этом В C# X и Y - только пользовательские классы, функция должна быть статической и быть членом класса X или Y.

Аналогичная проблема с перекрытием стандартных операций. Поэтому класс string встроенный в Delphi. Аналогично в Java, то есть там возможна такая запись.
String s;
s + i;

Причем если i - не строка, то будет вызвано неявное преобразование i к строке. То есть некоторые классы должны быть встроенными, то есть поддерживаться компилятором. В C++ же этого не требуется. В C# нельзя перекрывать операции () и [], но взамен там присутствует особая конструкция - индексатор.

```
class X{
  int this[string x]{}
};
```

Теперь возможно писать
X x;
x ["sss"];

Свойства (property)

Свойство - это специальный член класса, который при использовании выглядит как член данных, а при реализации - как пара функций - get и set.

Свойствами обладают C# и Delphi. В C++, Java и так далее, явных свойств нет, но там можно определить функции доступа для доступа к полям данных.

```
Delphi:
type X = class
  property p: integer read accessor write accessor;

type X = class
  private
    Flenghth : integer;
  property length:integer
      read Flength write Flength;


a.Length = 1; чтение будет из FLength.
```

```
type figure = class
  private a, b : integer;
    Farea:integer;
    property Area: integer read Farea; {доступ на запись ограничен}
```

Более того

```
type figure = class
  private a, b : integer;
  property Area: integer read GetArea;
  property width: integet read b write SetWidth;
  private procedure SetWidth (X:integer);
```

Автогенерируемые свойства.

```
class Point{
public int x;
public int y;
  public Point (int x, int y){this.x = x; this.y = y;}
```

Это можно заменить (в C# 3.0) на

```
class Point{
public int x {get, set};
public int y {get, set};
```

И теперь можно написать
Point p = new Point (x = 0; y = 0);

Многие из того, что было добавлено в C# 3.0 имеет смысл только для работы с SQL запросами.

Для автореализуемый свойств можно писать так, public int y privete get, set, то есть доступ на чтение закрывается.

Глава 7. Раздельная трансляция.

Разделение программы на модули.

Виды трансляции

1. Цельная трансляция (Pascal)
Во первых отсутствует возможность разбиения на модули. Далее, все библиотеки должны подаваться на вход вместе с самой программой. Совершенной не подходит для промышленного программирования.

2. Пошаговая трансляция.(Basic)
3. Раздельная трансляция - раздельная независимая и раздельная зависимая. Программа транслируется по частям, появляется понятие единицы трансляции. раздельная независимая - трансляция исходного кода единицы трансляции в объектный файл
раздельная зависимая - трансляция исходного кода единицы трансляции с использованием набора оттранслированных ранее библиотек.

Появляется понятие контекста трансляции.Он существует всегда (например стандартные имена). В случае независимой трансляции контекст берется из единицы трансляции. Примеры таких языков - Fortran, Assembler, C / C++.

Лекция 27
4.12.2008

При раздельной трансляции возникает понятие модуля компиляции (физического модуля).

Транслятору на вход подается единица компиляции и на выходе получаем объектный модуль. Отсюда возникаем понятие контекста трансляции. В языках с раздельной трансляцией весь контекст берется из единицы трансляции, то есть необходим механизм указания контекста трансляции. Для этого не достаточно директивы extern. То есть необходимо дублировать контекст.

Совершенной другая ситуация при раздельной зависимой трансляции. В этом случае компилятор не только выдает объектный код, но и кладет некоторые данные в трансляционную библиотеку.

Таким образом, то что составляет объектный модуль - это программная библиотека, а трансляционная библиотека - это некоторый аналог таблицы имен.

То есть даже если присутствует единая библиотека, то она все равно обычно делится на 2 части - трансляционную и программную библиотеку.

В почти всех современных ЯП (C#, Delphi, Java) используется именно такой вариант, то есть имеется тенденция к объединению трансляционной и программной библиотеки. Для этих языков характерно понятие рефлексии - получение информации о типе на этапе выполнения. Например, получения типа переменной по ее имени, то есть почти весь исходный код находит отражение в результирующем объектном файле.

При независимой раздельной трансляции возможно большое число ошибок, порождаемых межмодульными связями. В C/C++ требуется дублирование кода, что может порождать ошибки. Для упрощения еще создателями языка была придумана техника на основе заголовочных файлов, где описываются extern имена, прототипы функций, типы.

Таким образом, на C/C++ не рекомендуется описывать внешние имена как extern, вместо этого требуется включать заголовочный файл, где описан данных прототип.

При этом, если модулю требуются имена из нескольких модулей, то он должен включить несколько заголовочных файлов. Отсюда возникает проблема повторного включения, что может вызвать проблемы с объявлением типов.

Таким образом, необходимо использовать "страж включения":

```
# ifndef __M_H__
# define __M_H__
```

....
endif

Что гарантирует, что информация из данного файла будет включена только 1 раз.Таким образом объем кода, который подается транслятору возрастает в 10 - 20 раз за счет включения заголовочных файлов.

Такая технология фактически заимствует то, что используют программисты в более развитых ЯП.

Одностороннее и двустороннее связывание.

ЕК - клиент (использует с помощью предложения импорта) → ЕК - сервис.

Delphi, Modula - 2 - модуль определения и модуль описания. DEFINITION MODULE,IMPLEMENTATION MODULE.

Oberon - 2 единый модуль.

В этих языках логический модуль совпадает с единицей компиляции.

Uses почти эквивалентно предложению # include в С.

Так как это исключительно двусторонняя связь, то сервер ничего не знает о клиенте, более того, кольцевые ссылки запрещены.

```
unit M1
  interface
    uses M2;

unit M2
  interface
    uses M1;
```

При этом такая вещь абсолютно корректна.

```
unit M1
  interface
    uses M2;

unit M2
  interface
  implementation
    uses M1;
```

Так как uses требует трансляции только интерфейса. То ест явно видно что у каждого модуля имеется 2 единицы трансляции - интерфейс и

определение, причем они могут даже транслироваться раздельно друг от друга.

Таким образов основной состав библиотеки - независимые библиотеки общего назначения. При этом возникает огромный список uses (до 2 - 3 строчек).

Рассмотрим язык Ада
- односторонне связи
- двусторонние связи

Ада поддерживает два вида трансляции, то есть можно транслировать все целиком, а можно по частям. Есть 2 вида модулей - первичные, который требуются для трансляции других модулей и вторичные, которые нужны сами по себе. Таким образом, первичные единицы как бы сообщают контекст. В качестве первичной единицы может, например выступать процедура.

Все что описано в модуле спецификации первичного модуля является экспортом.

указание контекста

WITH список_первичных_ЕК
use список; {не обязателен}
текст единицы компиляции.

Таким образом, одностороннее связывание в языке ада практически не отличается от одностороннего связывание в других ЯП. При этом спецификация пакета может быть первичным логическим модулем, а его описание - вторичным.

```
package outer is
  ....
  package Iner is
  ....
  end Inner;
  procedure P(X: T1; Y:T2);
end Outer;
```

Первичной единицей компиляции является определение пакета Outer. Тело пакета является вторичной единицей компиляции.

```
package body Outer is
```

```
  package bodu Inner isd separate;
  procedure P(X:T1;Y:T2) is separate;
end Outer;
```

Таким образом, процедура P и внутренний пакет определены вне этого модуля. Рассмотрим оставшиеся 2 модуля (с процедурой P и с внутренним пакетом).

```
WITH Outer;
use Outer;
package body Inner is
....
end Inner;
{Error.Bad definition}
```

Теперь напишем правильный вариант.

```
separate Outer.Inner;
package body Inner is
....
end Inner;
{Correct.Correct definition}

separate Outer
procedure O(X:T1;Y:T2) is
....
end P;
```

Управление пространствами имен.

В языке Ада нет понятия библиотеки, то есть нельзя указать что этот набор модулей является единицей дистрибуции. Все это делается внешними средствами.

Эта проблема была решена в языке Java. В C++ единицей компиляции является файл, в котором может быть все что угодно. В Java единицей компиляции тоже является файл, но в отличии от C/C++ каждый файл должен содержать в себе первым делом предложение package, указывающее к какому пакету принадлежит данный файл.

```
package mine;
public class Outer{
  public static int main (String [] args)
    {
      //....
    }
}
```

Таким образом, пакет служит для указание контекста трансляции и единицы дистрибуции одновременно. Import <имя пакета> делает все имена из пакета видимыми. Более того, можно и не импортировать пакет, если для доступа к имени этого пакета явно задать полное имя пакета.

Более мощное средство языка Java - пространства имен.

Лекция 28
9.12.2008
Пространства имен

В C++ есть 2 стиля - старый стиль (для совместимости со старыми программами) и новый стиль (на основе пространства имен).

То есть возможно использовать как старый стиль программирования <stdio.h>, так и новый стиль std::iostream.

И C++ и C# поддерживают вложенность пространств имен. При этом в C# следующие объявления эквивалентны

```
namespace N{
  ....
  namespace Ninner{
    ....
  }
}

namespace N{
}
namespace N.Ninner{
}
```

В C/C++/C# единицей компиляции является файл,но единицей контекста в C# является именно пространство имен. В C++ все осложняется тем, что требуется включение заголовочных файлов.

После включения заголовочного файла к объектам из пространства имен можно обращаться через квалификатор (уточнение), то есть std::cout « i; В C# не требуется подключения заголовочных файлов, то есть обращение к объектам некоторого пространства имен идет просто по имени пространства имен.
System.Windows.Forms.

Отсюда возникает понятие единицы дистрибуции (сборки). Похожая ситуация в Java (единица компиляции - файл, единица дистрибуции и контекста - пакет). Более того, пространства имен реально вложены друг в друга, тогда как пакеты нет. То есть иерархический поиск имени производится только в случае пространства имен.

Если мы импортируем пакет в Java (import P1.* или import P1.x). В java появился статический импорт (static import Math.*) что позволяет обращаться к членам данного пакета без явного указания его имени. То есть

x = exp (i);

Все пространства имен открыты снизу, то есть к ним можно добавлять контекст. В C# так же можно сделать имена из пространства имен видимыми без уточнения.
using Sistem.Windows.Forms

Причем имена из текущего пространства имен перекрывают имена из импортируемых пространств имен. Если же 2 импортируемых имени конфликтуют, то имя становится невидимым.

Также появляется квалификатор доступа для класса в пространстве имен.

```
namespace N1{
  class X;
};

namespace N2{
  class Y;
};



N1::X a;
N2::Y b;
f (a, b);
```

Даже если нет директивы using (она не рекомендуется), то вопрос в том, где искать функцию f - N1 или N2.Если f не найдена в текущем пространстве имен, то поиск будет продолжен и в пространствах имен аргументов, если будет найден не один вариант функции f, то будет выдана ошибка.

Такое правило для имен функций требуется для того, чтобы стало возможным перекрытие имен имен стандартных операций.

Объектно - ориентированные языки программирования.

Выделяют ООЯП и просто объектные. ООЯП должны поддерживать 3 свойства - инкапсуляцию, наследования и динамический полиморфизм.

Наследование

Базовый класс → производный класс. Производный класс наследует все данные базового класса. C#, Java, Delphi
C++, Oberon, Aga 95

С точки зрения наследования, у языков первой группы вся иерархия наследования начинается с класса Object, то есть все объекты так или иначе имеют класс Object в качестве базового.

В языках второй группы любой класс может стать корнем дерева наследования.

```
C++:
class Derived: public Base{
   объекты новых членов.
};
```

При этом C++ единственный ЯП, который поддерживает модификацию прав доступа. Приватное наследование наиболее часто используется при реализации интерфейсов.

```
C#:
class Derived: Base{
   объекты новых членов.
};
```

Модификатор доступа запрещен. В C# запрещено множественное наследование, но можно наследовать множество интерфейсов. Настоящее множественное наследование реализовано только в C++ (из тех ЯП, которые рассматриваются в курсе)

```
Java:
class Derived extends Base implements i1, i2{
   объекты новых членов.
};
```

При этом, если не указан базовый класс, то в C# и Java класс становится потомком класса Object.

```
Delphi:
class Derived class (Base);
```

```
Oberon:
TYPE BASE  = RECORD
```

```
END;
TYPE DERIVED = RECORD (BASE)
END;
```

Язык Ада.

```
type base is tagged record
    ....
end;

type Derived is new Base with record
    ....
end;

type Derived is new Base with null record;
```

Вторая запись - если не требуется добавлять новые члены данных.

Все языки позволяют линейно распределять память,исключение - smalltalk. Там объект подкласса может быть отделен от части суперкласса. Таким образом, появляется возможность варьировать набор методов выполнения во время выполнения. Это дает языку огромную гибкость, но сильно замедляет работу программы, так как при каждом обращении к методу класса требуется поиск по всей цепочке наследования.

Таким образом, все языки промышленного программирования реализуют линейное распределение памяти.

Все функции класса существуют в единственном экземпляре. При наследовании каждый класс - своя область действия, то есть

```
class X{
  void f();
};

class Y:public X{
  int f;
};
```

То есть в классах X и Y имена могут совпадать. С точки зрения я области действия имеет место 2 принципа
- перегрузка
- скрытие

Почти во всех случаях речь идет именно и скрытии. То есть мы должны различать имена функций (допускают перегрузку) и имена остальных объектов.

```
Y y;
y.f();
```

Такое обращение - обращение к int f. Так как void f() скрыто. То есть такое обращение приведет к ошибке. Рассмотрим другой вариант

```
class X{
  void f();
};

class Y:public X{
  void f(int);
};
```

В данном случае речь идет так же о скрытии, то есть f() будет перекрыта f(int). То есть класс y обладает только функцией y.f(int).

Перегрузка возможна только в одной области действия. Для имен функций появляется и третий вариант - переопределение. Это только для функций, которые обладают динамическим связыванием.

```
class X{
  virtual void f();
};

class Y:public X{
  void f();
};
```

Функцию из класса - предка можно вызвать путем явного указания имени класса. y.X::f().

Более хитрая ситуация в языке Java.

Рассмотрим теперь модульные ЯП - Ада - 95 и Оберон.

```
<= Oberon =>

MODULE M;
  TYPE BASE* =RECORD
    I:INTEGER;
    J*:REAL;
  END;
END M;
```

```
MODULE M1;
IMPORT M;
TYPE DERIVED *= RECORD (M.BASE)
  K: INTEGER;
END;

PROCUDURE P(VAR X:DERIVED);
BEGIN
  X.K := 0;
  X.J := 1;
  X.I := 1; Ошибка;
  END:
```

То есть, есть либо публичные члены данных (*), либо пакетные члены данных, которые видны только в пределах пакета.

Рассмотрим язык Ада.

```
package M is
  Type Base is tagged private;
  ...
priate
  type Base is tagged record
  ...
  end record;
end M;

package M1 is
use M;
type Derived is new Base
    with record
      K:integer;
    end record;
procudere B(VAR X:inout Derived) is ...
```

Далее,в Ада появляется понятие дочерних пакетов.

```
package M.M1 is
  type Derived is new Base with
                      record
                       K:Integer;
                     end record;
  procedure P(X: inout Derived);
```

Все имена, которые описаны в M доступны в M1(даже приватные).

Множественное наследование

Проблемы
1. Конфликт имен
2. Реализация
3. Присваивание объектов разных классов.
4.

```
Base * pb = new Base;
Derived * pd = new Derived;
pb = pd;
pb -> f();
Здесь все нормально.


class X:public Y, public Z{
};

X * px = new X;
Y * py = new Y;
Z * pz = new Z;

px = pz; Error
py = px;
pz = px;

pz -> f ();
```

В случае виртуальных методов вопрос еще сильнее усложняется, так как компилятор на этапе компиляции не знает какой именно метод вызывать.

Лекция 29
11.12.2008

Итак было отмечено три проблемы связанные с множественным наследованием
1. Конфликт имен.

C++ допускает полное множественное наследование.C++ Один из немногих языков, допускающих полное множественное наследование. При этом ограниченное множественное наследование (интерфейсы) есть почти во всех современных ЯП. При наследовании интерфейсов конфликтовать могут только имена методов.

2. Сложности реализации.

Требование коррекции указателей в случае линейного размещения памяти (присутствует почти во всех языках промышленного программирования).

3.Идеологические соображения (не рассматриваем)

Пример иерархии
$X- > W < -Y$
$L- > X- > W < -Y < -L$
$L- > W < -L$ Такое недопустимо

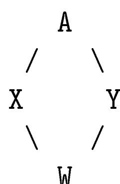
Поиск имен происходит сначала в своем пространстве имен, а потом параллельно в высших порядках (механизм, схожий с механизмом поиска в пространствах имен).

Для получения доступа к членам данных классов - предков может потребоваться явная спецификация.

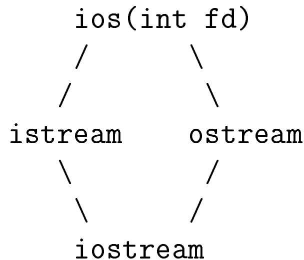
В таких языках, как C#, Java, Delphi, в которых все объекты наследуют класс Object, имеются обобщенные (абстрактные) контейнеры, в которых могут храниться объекты любого типа.

Если требуется хранить один и тот же объект в качестве двух сущностной, то это можно реализовать с помощью множественного наследования. Такие иерархические структуры изучается в особом разделе математики - теории решеток или теории категорий.

Бриллиантовое (ромбовидное) наследование



Примером такого наследования может служить проектирование модуля <iostream>. Упрощенная иерархия выглядит так.




```
class A{
};

class X:public virtual A{
};

class Y:public virtual B{
};

class W:public X, public Y{
};
```

Если опустить хотя бы одно слово virtual, то "ромбовидного"наследования не будет. Недостатком такой схемы служит то, что решение о типе наследования должно приниматься на этапе единичного наследования.

Это не является недостатком C++, но свойством наследования, так как вся иерархии должна разрабатываться на этапе проектирования.

В C++ нет никаких ограничений с точки зрения наследования, то есть нет никаких языковых методов запретить наследования данного класса. Конечно, можно сделать конструктор или деструктор закрытыми, но это уже не извороты.

В Java можно запретить наследование, с помощью ключевого слова final. Если это слово стоит перед именем метода, то этот метод не может переопределяться в производных классов. Если же оно стоит перед определением класса, то класс нельзя наследовать. В C# такую же роль играет слово sealed.

Почти все библиотека .NET является запечатанной, то есть из нельзя наследовать, а те классы, из которых можно наследовать обычно являются абстрактными.

Запечатывание может повысить эффективность программы.

Динамическое связывание методов.

Без динамического связывания методов понятие наследования практически теряет свой смысл. Настоящее наследование проявляется только при использовании динамической типизации. В языках индустриального программирования (со строгой типизацией) замена объекта супер-класса на объект производного класса ничего не меняет с точки зрения надежности.

base=derived
void foo (base b);
f(base)
f(derived)

Другое дело, когда вместо объекта передается ссылка на объект. В языках где есть наследование, есть понятие статического типа, то есть типа, присвоенного объекту при создании.

var X = new System.Window.Forms.Form

При этом тип объекта X статически определим.

В Объектных ЯП для некоторых объектов ТП появляется динамический тип - тип объекта данных, на который ссылается указатель или ссылка.

Base * pd;

Статической тип - Base, а динамическим типом может быть любой из потомков Base, так как pb может указывать на любого объекта-потомка Base.

Статический тип объекта данных изменить нельзя. Каждый объект обладает куском памяти, для которого был вызван конструктор.

В C#, Java и Delphi, объект размещается в динамической памяти, и как только он был там размещен его тип зафиксирован.

Не-статический метод класса - метод, вызываемый через ссылку на объект. Метод называется статически привязанным, если его вызов определяется из статического типа ссылки, и динамически если его вызов определяется динамическим типом ссылки.

То есть определяется динамический тип ссылки, и для этого типа вызывается метод.

В C++ это реализуется на основе виртуальных функций. На такие функции наложены некоторые требования.
Определение, если тип X является производным или совпадает с Y, то тип X ковариантен с Y.

У динамически связанных методов должен быть один и тот же профиль параметров, но ковариантные возвращаемые значения. То есть

```
class Y{
  public Y f(){}
}

class X extends Y{
  public X f(){}//Ok
  public int f(){}//Err
}
```

В Java все методы динамически связанные, в C++, C#, Delphi для этого существует специальное ключевое слово virtual. То есть виртуальность - свойство метода.

Если метод объявлен с ключевым словом virtual, то он становится динамически связанным. Динамическое связывание - это свойство вызова, но в языках оно привязано к методу.

$A \to B \to C \to D \to$
B* pb;
pb -> f();
Пусть f - статический метод.

Сначала поиск идет в классе B, если он его не находит, то он идет в объемлющую область видимости, пока не найдет метод или не выдаст ошибку.

void B::g(){
g()
}

Это неявные вызов через ссылку this. Сначала f отыскивается в классе B, потом в классе A, потом в пространствах имен, вплоть до самого внешнего пространства имен.

Поиск нужного метода осложняется еще и перегрузкой функции.

Далее, пусть f - виртуальный метод.

Это означает, что отыскивается динамический тип, то есть требуется знать на что указывает bp. Например
bp = new D

То есть мы смотрим на динамический тип, а потом начиная с него ищем соответствующее переопределение метода. То есть если метод объявлен как виртуальный, то каждый производный класс может либо унаследовать метод, либо переопределить его.

Аналогичная ситуация и для языка C#. То есть для виртуального метода отыскивается динамический тип, и для него вызывается метод.
$A \to B \to C \to D$

Немного другая ситуация в языке java.
pb -> f()

Сначала компилятор идет в определение B и отыскивает определение f(), там он узнает способ связывание и тип параметров. Эта ситуация осложняется неявными преобразованиями. Так же выводится доступ. В Java другая модель - там идет управление видимостью, то есть на закрытою функцию никто не смотрит.

То есть компилятор даже не видит приватные методы при вызове извне.

Вопрос - что есть определена приватная виртуальная функция. Ее можно переопределять, но что с доступом?

```
class A{
private:
  virtual void f(){..}
};

class B{
private
  virtual void f(){..}
};
```

Причем в переопределении функция обязаны так же быть private. Переопределяя приватную функцию, мы даем базовому классу вызывать приватную функцию нашего класса.

Переопределяя функцию, мы отказываемся от наследования реализации. Пример

```
    Figure
```

```
     /  |   \
    /   |    \
Point Circle  Rect
```

```
virtual void Draw ();
```

Наследовать не обязательно,но допустим, что Circle → Ellipse и есть список из фигур. Тогда нарисовать все фигуры можно так

```
while (L != NULL)
{
  L-> f ->Draw ();
  L = L -> next;
}
```

Чтоб создать новый тип данных, нужно просто определить новый класс, и там переопределить соответствующие методы. Например

```
void PatternRect::Draw()
{
  Full (pattern);
  Rect::Draw ();
}
```

То есть вызывается унаследованный, а не переопределенный метод. Такая конструкция - снятие свойства виртуальности метода.
Rect *f = new PatternRect;
f -> Rect::Draw (); вызов метода Draw для Rect;
Для приватных виртуальных функций возможно либо полное замещение либо полное наследование, но не использование в замещении.

Лекция 30
16.12.2008

В C++ метод является виртуальным, если он помечен ключевым словом virtual. При этом замещение метода может идти только при наследовании. Для замещения необходимо, чтоб совпадали имя и сигнатура параметров (профиль параметров должен совпадать), тип возвращаемого значения должен быть ковариантен.

Таким образом, если функция возвращает один из базовых типов языка (int, double) то тип возвращаемого значения должен совпадать. В случае, если она возвращает объект класса, то при переопределении функция может возвращать производный от него тип.

```
class Base{
public:
  virtual base * clone ();
};

class Derived: public Base{
public:
  Derived * clone ();
};
```

Все заместители виртуальной функции по определению будут виртуальными.

При этом остается вопрос с правами доступа. То есть если открытая функция переопределена как приватная, то она может быть вызвана из интерфейса базового класса.

```
class Base{
public:
  virtual base * clone ();
};

class Derived: public Base{
private:
  Derived * clone ();
};

Base * pb = new Base;
pb -> clone ();
```

```
Derived  * pd = new Derived;
pd  -> clone ();//нельзя

pb = pd;
pb -> clone ();//Ok
```

Теперь рассмотрим Java, там идет управление видимостью, а не доступом.

```
X:
public void f() {}


class Y extends X{
private void f(){}
}
```

В данном случае приватная функция вообще не видна, то есть ее вызвать можно только внутри класса Y. Извне будет вызываться функция класса X.

В Java все функции по умолчанию динамические, единственный способ запретить динамическое замещение - ключевое слово final. Если в java не совпадает имя - то это просто новая функция, если совпадает имя, но не совпадает профиль, то это скрытие. Если же совпадает имя и профиль, но тип возвращаемых значений не ковариантен, то это ошибка.

```
class Y  extends X{
  public final void f(){}
};

Y y = new Y_Derived;
y -> f();
```

При этом будет всегда вызван именно метод из класса Y, так как она объявлена как final.

Рассмотрим теперь C# и Delphi. Как и в C++ там есть как виртуальные, так и не виртуальные методы. Если в C# метод объявлен как виртуальный, то в классе наследнике возможна следующая ситуация - функция, у которой совпадает профиль, имя, и ковариантен тип возвращаемого значения не обязана замещать.

Таким образом требуется еще модификатор override.

```
class Y::X{

  public override void f(){..}
};
```

Если слова override нет, то свойство динамического связывания теряется. То есть.

```
class Y:X{

  public  void f(){..}
};
```

В данном примере для функции f нет динамического связывания.

В Delphi

```
type X = class
          procedure P; virtual;
         end;

type Y = class (X)
          procedure P;override;
         end;
```

То есть ситуация аналогичная C#.

В C# в случае отсутствие override или new в определении функции выдается предупреждение.

```
class Z: Y{
  public virtual void f();
};
Y y = new Z();
y.f ();
Теперь будет вызвана функция из Z, так как в классе Z опять указанно слово virtu
```

Так же возможна ситуация, когда функция не была виртуальной в базовых классах, но стала виртуальной в классе потомке.

Рассмотрим язык Оберон - 2. В одной из версия появились процедуры, динамически привязанные к типу.

Рассмотрим сначала вариант с полем типа.

```
TYPE FIGURE* = RECORD
              ....
              END;

TYPE LINE* = RECORD (FIGURE)
            ...
            END;

TYPE CIRCLE* = RECORD (FIGURE)
              ....
              END;


PROCEDURE DRAW (VAR F: FIGURE); (VAR X: LINE);
IF F IS LINE THEN
   DRAWLINE (LINE.F);
ELSEIF F IS CIRCLE THEN
   DRAWCIRCLE (CURCLE.F);
```

Теперь рассмотрим вариант с динамической привязкой типа.

```
PROCEDURE (VAR F: FIGURE) DRAW();

PROCEDURE (VAR C: CIRCLE) DRAW();
```

Это единственный случай, когда в Оберон - 2 разрешена перегрузка имен. Вопрос - как это вызывать?

```
PROCEDURE DRAWALL (VAR F.FUGURE);
...
F.DRAW ();
```

В псевдо модуле определения будет сгенерирован следующий код.

```
TYPE FIGURE =
     RECORD
       Объявление видимых членов.
       PROCEDURE DRAW();
END;
```

DRAW ^ () - вызов унаследованной реализации (снятие виртуальности).

```
FIGURE INTERSECT (F1, F2: figure);
```

Для каждой пары типов Требуется написать свою процедуру INTERSECT. Вопрос опять же как ее вызвать?

Один из вариантов - f1@f2.intersect. Но те языки, которые мы сейчас рассматриваем не имеют мультиметодов.

Теперь рассмотрим Ада - 95. Там нету виртуальных методов в явном виде.

```
type Base is tagged record ... end record;

type Derived is new Base with record <новые члены и объявления> end record;

procedure P( x: Base);
procedure P( y: Derived);
```

Пока речь о замещении или динамической привязки к типу нету. То есть

```
b: Base;
d: Derived;
P(b);{P(Base)}
P(d);{P(Derived)}

P(Y'Base);{P(Base)}
```

В Ада - 95 были введены CW - переменные (class wide) и CW - Типы. CW - Типы (классовые типы).

T'class - объединение T и всех объектов, производных от него.
X: Base'class - полный аналог неограниченных переменных. Такой переменной можно присваивать любой объект класса Base или производных от него.
P(X) - динамический вызов.

```
P(b);   static
P(d);   static
P(X);   dynamic
```

FIGURE → LINE
FIGUGE → CIRCLE

procedure DrawAll (p: Figure'class)

Любая процедура, которая имеет хотя бы одним параметром объект тегированного типа может быть вызвана динамическим образом, причем тегированный объект должен быть заменен на объект конкретного типа.

```
function Intersect (X, Y: Figure) return Figure;

function Intersect (A, B: Figure'class) return Figure;{Bad}
```

Но мы не можем вызвать процедуру, у которой более одного параметра является классовыми типами. То есть только один параметр может быть классового типа. Но возможно написать Intersect (X, L) или Intersect (C, X), то есть указывая один классовый параметр.

```
class Figure {
  public:
    virtual void Draw ();
};

class circle: public Figure {
  public:
    void Draw ();
};
```

Такая реализация вызовет ошибку (не описать Figure::Draw()), что не позволяет инициализировать таблицу виртуальных методов. Класс Figure - абстрактный класс.

Во всех ООЯП есть понятие абстрактного класса. В C++ есть понятие число виртуальной функции. Класс, который имеет хотя бы одну чисто виртуальную функцию называется абстрактным. Нельзя создать объекты такого класса. Синтаксически это реализуется как virtual void Draw() = 0;

Это означает что соответствующая функция может иметь реализацию, а может и не иметь.

```
Figure f;//Err
Figure * p;//Ok
p = new Figure;//Err
p = 0; //Ok
p = new Line;//Ok
```

В C# и Java есть ключевое слово abstract, что означает что метод является виртуальным и притом чисто виртуальным. Если у класса есть хоть один абстрактный метод, то класс является абстрактным, то перед его определением должно стоять ключевое слово abstract.

Лекция 31
18.12;2008

Глава 3. Интерфейсы

Интерфейс - особая языковая конструкция, которая содержит объявление набора публичных методов. Интерес не содержит членов данных, он может содержать только статические члены.

Интерфейс может наследоваться (при наследовании рассматриваются как абстрактные классы).

В C++ интерфейсы моделируются с помощью абстрактных классов. В C# и Java присутствуют явно (interface).

Рассмотрим 2 вопроса
1. Реализацию методов интерфейса (явная или неявная)
2. Интерфейсы и виртуальные методы.

Интерфейсы в C++

```
class Interface{
  public:
    virtual void f() = 0;
    virtual int g() = 0;
    virtual void ~Interface (){}
};
```

В Интерфейсе не должно быть не виртуальных методов, членов данных или приватных членов. Таким образом интерфейс состоит только из указателей на таблицу виртуальных методов.

Единственный метод, который может иметь реализацию в интерфейсе - деструктор.

Возможна ситуация, когда в классе интерфейса отсутствует деструктор (деструктор по умолчание). Если деструктор не виртуальный, то мы не можем пользоваться им как интерфейсом. В таком случае требуется, чтоб наследующий класс содержал в себе удаление объекта - интерфейса.

Другой вариант использование такого - запрет на удаление объектов - интерфейсов.

В технологии COM интерфейсы - всегда чисто абстрактный класс. Эта технология подразумевает использование самых разных контекстов выполнения (в отличии от .NET), то есть знать как уничтожить объект может знать только сам объект.

```
class IUnknown{
  public:
    virtual int QueryInterface (IID id, IUnknown ** punk) = 0;
    virtual int AddRef () = 0;
    virtaul int Release () = 0;
};
```

В этом классе нет деструктора. Тот деструктор, который будет сгенерирован по умолчанию бесполезен, и никогда не будет вызван. Удаление объекта реализовано на основе подсчета ссылок на объект через методы Release и AddRef.

При этом любой конкретный класс, который реализует этот интерфейс должен реализовать эти методы.

Второй вариант использования интерфейсов в C++ - это обеление контракта.

```
class Iset{
  public:
    virtual void Include (T&) = 0;
    virtual void Exclude (T&) = 0;
};

class BitScale {};
class SList {};

class SetList:
  public Iset,
  private SLIst{
  ....
};
```

При этом мы всегда может заменить SList на BitScale, причем потребуется только перетранслировать модуль.

Интерфейсы в языке Java

```
Interface IDrawable{
  void Draw ();
  static int final SomeConst = 1;
}
```

В интерфейсе могут содержаться только объявления методов, а так же статических членов и констант. По умолчанию все интерфейсы имеют правило видимости public.

Если класс наследует некоторый интерфейс, то он должен либо реализовать все методы данного интерфейса, либо быть абстрактным.

```
abstract class Receiver implements IMsg{
   public void Receive (Message M){}
}
```

В C++ при множественном наследовании может существовать несколько таблиц виртуальных методов. В языках с единичным наследованием все виртуальные методы объединяются в одну большую таблицу.

Реализация интерфейса

В Java имеется неявная реализация интерфейса, то есть класс реализует интерфейс, когда каждому методу интерфейса соответствует один публичный метод класса.

```
Interface IDrawable{
  void Draw ();
  static int final SomeConst = 1;
};

interface Card{
  void Draw ();
};

class CartGame implements IDrawable, Card{
};
```

В случае неявной реализации интерфейса мы не можем реализовать Draw для обоих интерфейсов.

```
interface I{void f();}
interface A implements I{}
interface B implements I{}
class X implements A, B {}
```

Вопрос - какое здесь наследование (C# и Java) ромбовидное или обычное?

Последняя особенность Java - интерфейсы - маркеры. Каждый интерфейс определяет "контракт при этом интерфейсы - маркеры определяют самый чистый контракт, то есть не имеют методов вообще.

Такие интерфейсы служат для целей компилятора, то есть программист не может сам определять такие интерфейсы. Пример - интерфейс маркер Cloneable.

Такие интерфейсы требует поддержку некоторых определенных методов в классе.

В C++ явное понятие интерфейса не требуется, так как он поддерживает множественное наследование. В C++ явная или нее явная реализация интерфейсов?

Рассмотрим теперь язык C#. Там есть явное понятие интерфейса, там могут содержаться объявления методов, свойств, индексаторов.

```
interfase Intf{
  void Insert (T x);
  int Counter {get ;}
  T this [int];
  event MyDelegate Ev;
};
```

Интерфейс не разрешает указывать в себе операции преобразования типа, потому что он является статическим. Если же мы программируем на C++, то мы можем разрешить интерфейсам содержать члены данных, когда как в C# каждый такой случай требуется явно оговаривать.

C# допускает одновременно и явную и не явную реализацию интерфейсов.
Явная - просто реализация публичного метода (для каждого интерфейса). Неявная - одна реализация на всех.

```
I1:IDrawable;
I2:IDrawable;
class X:I1, I2{
  public void Draw(){}
};

X x = new X();

X.Draw ();
((I1)X).Draw ();
((I2)X).Draw ();


classX:I1, I2{
  void I1.Draw () {...}
  void I2.Draw () {...}
}
```

```
X x = new X();
x.Draw ();//Error
I1 i1 = (I1)x;
i1.Draw ();//Ok I1.Draw ();

I2 i2 = (I2)x;
i2.Draw ();//Ok I2.Draw ();
```

Заметим что в явной реализации отсутствует модификатор, то есть такую реализацию можно вызывать только при приведении интерфейсов.

Либо возможен такой вариант
public void Draw()((I1)this).Draw ()

Интерфейс и виртуальность. Одна из задач интерфейса - одинаковая работа компонент, их реализующий. Если в классе реализован некоторый метод интерфейса, то наследующие этот класс классы не могут переопределить этот метод. Таким образом, интерфейсы в C# отличаются от абстрактных классов в C++ (с точки зрения наследования).