Языки программирования

Головин Игорь Геннадиевич

Вт – 3ей, Чт – 4й парой, отчетность – экзамен

Лекции записывали: Михайлишин Алексей, Жбанков Денис, Щербинин Виктор, Чеботарев Павел

[В.Ш. Кауфман «Языки программирования: концепции и примеры», 1993]
[Т.П. Пратт, М. Зенкович «Языки программирования: разработка и реализация»]
[Р. Себеста «Основные понятия языков программирования»]
[Бен Ари «Языки программирования»]

Мы будем рассматривать:

- **C++** (много разных книг, стоит выделить Страуструпа)
- **Ада** (мало литературы на русском языке, первая версия 1983 – Н. Джеханн «Язык Ада», затем в 1995 и 2005)
- **Java** (Страуструп, Шилдт «Новые черты в языке Java 2005»)
- **C#** (язык появился в 1999, сам автор языка книгу не написал, но можно порекомендовать «C# - ускоренный курс»)
- **Modula2**, **Оберон** (сейчас не используются, но очень красивы и просты; Н. Вирт «Программирование на языке Модула 2»)
- **Паскаль**, **C** (иногда будем ссылаться на них)

Цель курса – не выучить языки программирования (для этого нужно на них писать), а изучить концепции.
План курса:

1) Введение
2) Основные понятия традиционных ЯП (основное понятие – тип данных)
3) ООП языки программирования

Глава 1. Определения и классификация ЯП.

Примеры ЯП, которые не для компьютера: APL – язык для *короткой* записи векторных операций (с массивами), PLANNER – на нем можно было описать алгоритмически неразрешенные проблемы (первоначально был для людей, затем, урезанно, перенесен на компьютер).

Экстенсиональное определение в данном случае не работает (пример про HTML – язык это или нет?), но есть интенсиональное:

Язык программирования – инструмент планирования поведения исполнителя. В этом смысле HTML тоже является языком программирования. В нашем курсе мы ограничим понятие исполнителя.

COBOL (common business oriented language) – ориентирован на обработку информации, удобные средства для описания plain-data данных. Создан, чтобы языки программирования писали обычные люди, для этого он был похож на естественные языки (assign 5 to j; add 1 to I given j).

Виды программирования:

1) игровое программирование (для себя, напр. Basic)
2) научное (для решения каких-то задач, программа нужна *только* как средство решения задачи)
3) индустриальное (программа отчуждается от ее авторов) – это мы будем изучать

(С.И. Мухин пишет на фортране, его (язык) знают все математики)

Парадигмы программирования:

1) процедурная (императивная), главный оператор *присваивания*, они эффективны
2) объектно-ориентированная, класс (в нем члены-данные и члены-функции)

Языки программирования:

1) процедурные
2) процедурно-ООП
3) функциональные, главная операция *аппликации* (применения функции)
4) декларативные

[lect2] Глава 2. Исторический очерк ЯП.

Существует несколько тысяч языков программирования. Языков много, генеалогию составить тяжело. Обоснуем выбор языков, которые мы выбрали.

FORTRAN (formula translator) – самый первый проект ЯП (1954 - 57). Первоначально затея о написании *транслятора* с некого языка на машинный считалась обреченной на провал т.к. тогда было очень дорого

процессорное время и память, а никто кроме человека так хорошо не сможет запрограммировать задачу. Но потом Бекус понял, что убрать из схемы математик->программист математиков невозможно, следовательно, надо убрать программиста (-ску), которые переводили «блок-схемы» в «двоичный код». Математики от появления FORTRANa были просто в восторге! Он сразу же стал самым популярным языком в мире. Однако позже стало понятно, что FORTRAN – яркий пример того, как делать не надо. Было много ошибок т.к. тогда надо было делать быстрее и не знали, как оно вообще должно быть. До 1964 года FORTRAN был базовым языком IBM. На нем писали все что угодно, несмотря на то, что он был создан для математических вычислений. Самый главный недостаток языка FORTRAN – он сам провоцировал ошибки программистов (DO5I=**1.3** instead of DO 5 I = **1, 3**).

Успех проекта FORTRAN спровоцировал целую кучу проектов.

ALGOL (создан IFIP - International Federation for Information Processing, международная организация):

- блочная структура
- формальные методы (БНФ)
- стек -> рекурсивный вызов процедур
- другие нововведения

Но не было стандартизованного ввода-вывода, на разных машинах свои реализации – следовательно, сложно было переносить программы на другие системы (а вот FORTRAN можно было). А также код, оттранслированный вручную, а не на ALGOLe, работал в 7-10 раз быстрее. Для FORTRANa этот показатель достигал 1.05.

Принцип **«экологической ниши»** языка - набор методик, библиотек, средств, программистов, работающих в одной данной области. Тот, кто первый займет эту нишу, у того и решающее преимущество. Даже если приходят более подготовленные, но не слишком «более», то они не смогут занять это место. Для языков программирования это проявляется ярко. В данном случае нишу занял FORTRAN, другую, например, Си (Си вытеснил ассемблер). Когда язык пытается выйти из этой ниши у него это **не** получается.

Периоды развития ЯП:

1) 50е – начало 60х годов – **эмбриональный** период
   - **FORTRAN**
   - 1960 – **ALGOL60** (предтечи PASCAL) – занял нишу в образовании; на базе него был создан SIMULA67 (где в принципе были классы, но не было наследования)
   - 1959 – **COBOL** (эффективный ввод-вывод), проблема-2000 завязана с COBOLом, так как там был специальный тип данных DATA, формата DDMMYY, а все бизнес-программы успешно проработали до наших дней
   - 1959 – 61 **LISP** – язык для AI, но плохо ложился на машины «Фон-Неймана» (переписывали на Си)
2) 60е – 1980 – период **экспоненциального** роста
   - Ежегодно появлялись *сотни* новых языков, наиболее заметен – Си

   появление универсальных языков

   - 1961 – для IBM/360 создали **NPL**, позже **PL/I**, из каждого языка взяли лучшее (что нравилось людям в языке) -> провал, так как язык слишком сложный (не соблюден принцип *взаимосвязанных компромиссов*)
   - 1968 – пересмотрен ALGOL60 (появился ALGOL68), закончилась его разработка. И IFIP стала работать над новым языком
   - ОС **MULTICS** – идеи использования ресурсов ПК многими пользователями (мультипрограммирование), хотели создать мощную, гибкую и универсальную систему, но она также как и NPL оказалась слишком сложной. В противовес MULTICS создали **UNIX** (Кен Томпсон), где реализовали *простые*, но фундаментальные концепции (написана на Си). Название языку Си дал язык CPL (машинно-независимый ассемблер). CPL породил BCPL. Первую версию языка назвали B, потом уже Си.
   - 1969 – **PASCAL**, академический язык (на основе созданы Object Pascal, Delphi). Паскаль отражал концепцию структурного программирования.
   - В 70-е годы стали появляться попытки создать осмысленные ЯП. Они стали создаваться более концептуально. Отметим язык **CLU** - язык абстракции данных, породивший идеи абстрактного типа данных, которые применяются ныне почти во всех языках.
   - 1980 - **Modula2**, язык почему то не пошел, хотя был очень даже хорош (для системного программирования) [Н.Вирт читал лекцию в П13 в 80м году] Авторское описание MODULA-2 занимает 40 страниц.
   - *Ортогональность*. Если одна конструкция может появиться в одном месте, а другая – в другом, то первая может появиться во втором месте, а вторая в первом. Если конструкции K и G ортогональны, то они независимы.

3) с начала 80х – **современный** этап
   - 1972 – **SmallTalk**, первый ООП
   - 1980 – **Ада**, прошел конкурс Пентагона (для написания встроенных систем реального времени, «стальные требования», код – «зеленый» язык), создан во Франции. Окончательный стандарт принят в 1983, требовали не создавать диалектов, а только обновлять стандарт (каждые 10-12 лет). Принципы критичных технологических потребностей и минимальности сложно соблюсти одновременно. В результате компиляторы с Ады получились сложными. Скорость компиляции 3 строки в минуту.
   - 1988 – **Оберон** (последний язык Вирта), были подвергнуты ревизии все конструкции Модулы2, язык еще более упрошен, но добавлено *наследование* (type extension), 1993 – **Оберон2**, полностью ООП (появился аналог виртуальности в Си++)

[lect4] **Подробнее про современный этап развития ЯП**

Современный этап развития ЯП проходит под знаменем Объектной Ориентации.

Два основных подхода в проектировании языка - принцип сундука и принцип чемоданчика.

- **Принцип сундука** - берем все что может пригодиться.
- **Принцип чемоданчика** - берем только то, без чего нельзя обойтись.

Язык Ада был спроектирован по принципу сундучка. **Modula2** - принцип минимальности языковых конструкций (чемоданчик). (Пример: Разрешать ли вложенные модули? В ада это разрешено. Но без этого можно программировать, и в Modula2 этого нету). Какой принцип лучше? В некотором роде минимальные языки - лучшие (например, си).

Тезис Вирта - сложность языка программирования немедленно переносится на сложность соотв. программных систем.

Знаковое событие – появление **C++**, а также стандарта ЯП **Ада** – 1983.

В Аде появилось понятие исключения. Все еще есть люди, программирующие на Аде, но своей цели, замены всех остальных языков, Ада не добилась.

- 1995 – новый вариант Ады (Ада 95) – появление черт ООП
- 2005 – новый вариант Ады (Ада 2005).

В этих версиях Ады поддерживалась максимальная совместимость снизу вверх. Даже самый лучший компилятор Ады не был сертифицирован, т.к. Ада – очень объёмный язык. Больше попыток создания универсального ЯП не было.

Первый чистый ОО ЯП - SmallTalk

- 1972 – SmallTalk – XEROX PARC
- 1980 – SmallTalk5

Многие понятия ООП (например, метод) – из SmallTalk. Он использовался в научных проектах, в индустрии распространения не получил. SmallTalk использовал bytecode (как Java). В нем не было множественного наследования. Некоторые считают SmallTalk образцом ОО ЯП, и что множественное наследование не присуще истинному ООП.

Самый важный ОО ЯП – C++

C++ унаследовал много недостатков от C (раздельную трансляцию).

Развитие C++:

- 1986 – C++ 2.0, он стал распространяться по миру
- 1990 – еще одна версия, появились шаблоны и *исключительные ситуации*. Только в C++ есть полное множественное наследование, во всех остальных ОО ЯП оно либо обрезано, либо его нет вообще. 1989 году в каждой СП был компилятор C++. Но в стандарте еще не было шаблонов, и их реализация и библиотеки были разными для каждого компилятора.
- до 90 – появление первых компиляторов
- 90-98 – самостоятельное развитие C++: появление пространств имен, RTTI
- 1998 – единая библиотека шаблонов, новый стандарт

MS Visual Studio до сих пор не поддерживает всю реализацию STL => появления понятия платформы STL. В настоящее время готовится новый стандарт C++ - TR1. Развитие идет в основном в STL. Популярные библиотеки (напр. Blitz) становятся частью STL.

Другие языки:

- 1988 – Оберон

- 1993 – Оберон2
- В 80-е годы появился Eiffel, широко известный в узких кругах, но так и не занял свою нишу
- Objective C – C, в который добавлены конструкции Smalltalk.
- Object Pascal => Turbo Pascal => Delphi
- RAD – Rapid Application Development
- J++ - реализация java на visual studio.
- 1995, весна – Java
- 1999 – C#

Эти языки реализованы с нуля (не надстройки над C). Они используют принцип **WORA** (Write Once, Run Anywhere) и байт-код. Появилось понятие **JVM** (java virtual machine).

Java

В 60-70 годы были популярны машины, имеющие в качестве входного языка язык высокого уровня, от них унаследована идея JVM. Впервые похожая идея реализована в UCSD Pascal. Промежуточный язык там назывался *P-code*. Т. о. был один компилятор из Pascal в P-code, и много интерпретаторов для различных архитектур.

Katana – микропроцессор, понимающий байт-код.

Java – язык+JVM. Компилятор – **javac**, компилирует java-программы в байт-код. **JRE** – java runtime environment, включает интерпретатор и другие вещи, нужные для интерпретации байт-кода.

Типы платформ Java:
- Micro Edition
- Standard Edition
- Enterprise Edition

Javac: .java => .class (байт-код). В байт-коде хранятся не только команды, но и метаданные.

Рефлексия – доступ к коду программы из самой программы – также появилась в Java.

Java позиционировалась как интернет-язык. Java не стал главным языком, т.к. Майкрософт стал интернет-компанией. Не очень хорошая эффективность. Близорукая политика фирмы Sun – она долго «не отпускала» java от себя. Java – чистый ОО ЯП, он проще, чем C++. Основное развитие шло по пути развития библиотек, но в 2005 появилась **Java5**, сильно отличающаяся от предыдущих версий.

C#

Майкрософт не стала пытаться создать универсальный язык, а стандартизовала библиотеки.

- CLI - Common Language Infrastructure – набор спецификаций
- CTS – Common Type System
- IL – Intermediate Language – промежуточный язык, что-то вроде байт-кода
- PE – формат исполняемого файла, не зависящего от архитектуры.
- Библиотека времени выполнения.
- VES (Virtual Execution System) – виртуальная исполнительная система, основанная на CLI
- CLR (Common Language Runtime) - реализация VES в .NET.
- Mono – реализация CLI в Linux.

В .NET есть:

- VB.NET
- Jscript.NET – аналог JavaScript
- FoxPro.NET
- C++/CLI – управляемый C++ – C++, в который добавлены особые понятия (ключевые слова), специфичные для .NET
- C# - «родной» язык .NET. Многие понятия Java близки C#.
- J# - Java для .NET, с .NET – библиотеками.
- ECMA – European Computer Manufacturers Association – также стандартизует ЯП. Она стандартизовала CLI и C#.

Не вся спецификация C# 2.0 стандартизована ECMA. 2008 год – C# 3.0.

На самом деле, программы на C# - чисто компилируемые (JIT(Just In Time)-компиляция) – IL компилируется в исполняемый файл во время работы программы.

Скриптовые языки

Пример: JavaScript, Python, Perl, PHP

На них легче программировать, но они менее эффективны. Они хорошо подходят для веб-

программирования из-за особенностей работы в Интернете.

[lect5] **Глава 3. Основные понятия ЯП.**

I) Основные **позиции** при рассмотрении ЯП:

1) **Технологическая**. *Технологическая потребность* - базовое понятие. Самая критичная потребность - раздельная трансляция модулей, модульность. Для нас ТП будет одной из самых важных.
2) **Реализаторская** (как реализуется та или иная конструкция). *Открытый стандарт* - описывает множество программ, результат которых гарантирован стандартом. *Закрытый стандарт* - результат некоторых программ не определен. [пример: DO 5 I=1,N; при N<1 результат не определен, т.к. на разных компиляторах результат разный]

   Современные ЯП вполне можно описать и без реализации, поэтому мы будем уделять ей не так много внимания. (Например, концепцию динамического связывания методов можно объяснить без всякой реализации). При рассмотрении реализации становится ясно, почему многие ЯП не используют множественное наследование. (Из рассматриваемых нами ЯП мн. наследование полностью реализовано только в C++).
3) **Авторская позиция.** Наиболее удачные языки - авторские, т.е. те, которые были написаны 1-2 людьми для реализации конкретной цели. Когда в написание языка начинают вмешиваться комитеты, компании, и т.д., он "разбухает" и становится громоздким. Одна из самых интересных книг с точки зрения авторской позиции - "Дизайн и эволюция C++" Страуструпа.
4) **Семиотика** (наука о знаковых системах). *Семантика* - часть семиотики. Например, один из вопросов - проблема "недоразумений". Семиотика с разных позиций трактует ошибки в сообщениях между субъектами языка (*синтаксическая*, *семантическая*, *прагматическая* неоднозначность). Передающий и принимающий субъекты могут интерпретировать сообщение по разному.

   Пример из языка: "Джон проектирует колледжи в Лондоне" - неоднозначность трактовки. Но часто мы понимаем неоднозначность, потому что у нас есть система приоритетов. В ЯП неоднозначности практически отключены. [пример: if b1 then if b2 then s1 else s2; Здесь чисто синтаксическая проблема]

   Как правило после выхода ЯП к нему добавляются исправления таких неоднозначностей. *Семантическая неоднозначность* - многие компиляторы не полностью реализуют стандарт C++ (например VS). Прагматические проблемы как правило самые тяжелые и относятся к технологической позиции.
5) **Социальная.** Некоторые феномены необъяснимы ни с одной из предыдущих позиций. Примером могут служить технологические ниши (фортран). Для человека важно понятие мобильности знаний, умений, навыков. *Владение ЯП* - такой навык, т.е. постоянно переходить на новые ЯП очень тяжело, поэтому фортран так популярен до сих пор.

   Пример - **Delphi** лучше других по всем пунктам. Но из-за того что она основана на паскале, у социума сложилось убеждение что Delphi пригоден только для обучения.

II) Основные **понятия** ЯП.

Три основных понятия - **данные**, **операции**, **связывание**. Эти понятия мы будем считать атомарными.

Любые программы обрабатывают данные с помощью операций. Связывание - менее очевидное понятие, но оно играет очень большую роль. *Связывание* - способ передачи параметров. Нас будет интересовать не механизм связывания, а время связывания (на каком этапе происходит связывание).

Пример.

```
I = 1;
```

В какой момент происходит связывание единицы и объекта I? Это зависит от контекста.

- Если const int I = 1;, то на этапе *компиляции*.
- const X a; - в момент *выполнения конструктора*.

Мы будем выделять 2 основных этапа - **статическое** связывание и **динамическое** связывание (binding).

Имеет место **дуализм** операций и данных.

Пример - строка символов S. **length(s)** - взятие длины строки. Что такое длина строки? Это операция или данные? Например в СИ **strlen** - это операция (она вычисляется). В **TurboPascal** длина строки хранится в первом байте строки, поэтому это данные, т.е. длину строки можно узнать, обращаясь к первому байту строки.

Некоторые операции нам проще изображать как данные (и наоборот). Пример: **property** в классах (getters/setters). Например, в визуальной системе окно имеет точку привязки (x, y). При ее изменении происходит не просто замещение данных, но еще и другие сложные операции (перерисовка и т.д.).

Атрибуты данных:

1) тип данных;
2) имя;
3) значение;
4) время жизни;
5) область действия (видимости).

Как правило данные принадлежат к некоторому **классу эквивалентностей**. Вирт говорил, что тип данных - это множество значений. Но это определение не было принято.

Тип данных - множество значений, которое определяется структурой данных + операции над ними.

Принцип инкапсуляции - тип данных скрывает свою структуру и выдает общий набор операций.

Существуют «**безтиповые языки**» (ошибочный термин, есть в многих книгах). В них переменные не связаны с конкретным типом (но тем не менее типы там есть). Безтиповые языки - языки, в которых тип данных может меняться динамически.

Есть **прототипные ЯП** - в них тип данных может отсутствовать, понятие типа данных явно не существует.

В C# есть понятие **var** - переменная.

- a = 5; переменная получает значение целочисленного объекта
- var a = new List<int>(); переменная с выводимым типом (тип выводится, исходя из правой части)

В ЯП есть понятие **Объект данных.** У него есть **слоты** В объекте данных количество слотов варьируется. В слоты можно поместить другие объекты. Есть также понятие **клонирования объектов** (примеры таких языков: **self**, **IO**). Такие ЯП (прототипные) очень гибки.

Квазистатический контроль - контроль данных во время выполнения, исходя из статических характеристик типа. Иногда, имя объекта - важная характеристика, но есть объекты без имен. Пример безымянного объекта в паскале - числовая константа, динамические объекты в памяти (доступ только по указателю).

Пример из C#

- const int i=5; // время связывания - на этапе трансляции.
- readonly X a; // время связывания - при выполнении.

Имя жизни очень часто связывается с понятием класс памяти.

Память бывает:

- *статическая* - создается в самом начале, очищается в самом конце работы программы
- *квазистатическая* - в отличие от статической, память определяется явным образом, есть понятие блока
- *динамическая* - период создания объекта контролируется программистом явно

Есть еще один вид памяти - **persistent** (сохраняемая), но он не используется в ЯП, т.к. нет универсального механизма сохранения данных.

[lect6] **Основные понятия ЯП. Продолжение.**

1) Тип данных
2) Имя
3) Значение
4) Время жизни
5) Область действия (видимости)
   1. Статические области действия (видимы только в своем блоке или вложенных)
   2. Динамические области действия (в современных языках почти нет)
6) Адрес

Пример.

```
var I;

proc P;          proc P1          P();
  var I;           {              P1();
  {                  I = 1;
    P1();            }
  }
```

В Си++ есть и динамические области действия (try…catch ищутся по стеку вызовов). Языки с динамической областью действия переменных менее эффективны (так как надо просматривать стек).

III) Концептуальная схема рассмотрения ЯП.

Наиболее важная проблема современных ЯП – *сложность*. Сложность определяется *семантическим разрывом* (базисные понятия компьютеров примитивны, а моделировать нужно очень сложные реальные процессы). Решение проблемы – **абстракция**. Для каждой конкретной области приходится создавать свою абстракцию. Впервые в фортране, абстракция – *подпрограммы*. Именно благодаря этому скопилось огромное количество библиотек для Fortran'а, и он все еще жив. Т.е. ЯП различаются в основном в абстракции, а не в базисе.

Схема.

1) **Базис**
   - Скалярный (простые ТД и операции)
   - Составной (составные ТД и операторы)
2) **Средства абстракции** (новые ТД, классы, подпрограммы), пример, в языке Си тип данных FILE* - абстрактный тип данных
3) **Средства защиты**, хорошо реализованы в языке Ада. Чем они лучше, тем больше уверенность, что если программа компилируется, то она семантически тоже верна.

IV) <u>Традиционные ЯП.</u>

С точки зрения базиса в языке Си++ (после Си) появились только ссылки.

<u>Глава 1. Простые типы данных.</u>

I) Классификация

1) **Простые** типы данных
2) **Числовые** типы данных (в некоторых языках есть только тип Number (язык **Lua**) – интерпретатор сам выбирает, какой это тип – целый или вещественный)
   - Целочисленные
   - Вещественные (бывают плавающие и фиксированные (для банковских систем))
3) **Логические**
4) **Символьные**
5) **Порядковые** (перечисления, диапазоны)
6) **Ссылки и указатели**

Функциональные типы данных (с одной стороны похожи на указатели, но все же это средство абстракции)

II) Целочисленные типы данных

Проблемы:

1) **Фиксация представления** (набор)
2) **Беззнаковые целые** (самый простой в этом смысле – Pascal, там есть только integer; а вот в Си – много таких типов, для покрытия всех надобностей программиста: char – short int – int – long – long long)

   CLI – common language infrastructure (contain **CTS** – common types system) – *стандарт*: sbyte – byte, short – ushort, int – uint, long – ulong (в порядке от 1 до 8 байт). Это используется в C#.

   Немного о развитии 64бит – архитектур. Процессоры Itanium (Intel) IA-64 – «родная» 64бит архитектура, а x64-32 – адреса 64бит, а все остальное – 32бит, как и было (надо переписать только компилятор, предложено AMD).

   Смешивать беззнаковые и знаковые типы очень опасно. Си, например, сравнивает их «нехорошо». Беззнаковые типы и арифметика нужны в основном для адресации (а не для счетчиков). Поэтому до сих пор много где есть беззнаковые типы (нет в Java – так они решили эту проблему, ввели специальный беззнаковый оператор сдвига >>>).

[lect7] **2 подхода к реализации числовых типов данных**

1) фиксированный базис
2) обобщенные числовые типы

Номенклатура ЦТД:

- C, C++ (фикс. базис, но определен только char, а int зависит от архитектуры)
- C#, D, M-2, Оберон (фикс. базис, ориентированы на конкретную архитектуру)
- Java (фикс. базис, JVM)
- Ада (*обобщенные* числовые типы, можно вывести *новый тип*, можно *подтип*)

Примеры объявления:

- **Ада:** type NewInt is new integer range 0..MAX; //NewInt и integer теперь разные типы данных (i16 = INT16(ui16) – явное преобразование, а вот i16 = ui16 - нельзя)
- **Паскаль:** type NewInt = integer; //NewInt и integer совместимы
- **C++:** typedef int NewInt; //аналогично использованию #define

Квазистатический контроль – (например, в Ада или Pascal) контроль, который компилятор неявно вставляет на этапе компиляции (range check, контроль присвоений разных типов); если *сразу понятно*, что в коде ошибка, компилятор сообщит это, если же ошибка *возможна*, то вставит код контроля.

III) Плавающие вещественные типы данных

Нужны для *математических* расчетов

+/- M * B^p, представление нормализовано **1/B <= M < 1, B = 2**

Существует проблема точности вычислений. Если мы даже не можем точно представить все числа, то как обеспечить точность вычислений? Стандартизировали (стандарт 1985 года, IEEE-754), представления мантиссы и степени (сколько битов выделять). А также ввели машинную бесконечность и ноль (+inf, -inf, +0, -0), все это NaN (not a number). Во всех современных языках теперь есть два типа данных: float (4byte) и double (8byte).

Фиксированные вещественные типы данных

Нужны, например, для представления оцифрованного аналогового сигнала. По сути, число целое (может принимать ограниченное множество значений, например от –M до M), но при вычислениях нужны именно вещественные типы.

type Data is **delta** 1/4096 **range** –M..+M; //пример фикс. типа на Аде, 1/4096 – точность

Также придумывали особые типы данных (например, decimal – для business oriented programs, удобнее переводить в символы) для каких-либо специфических нужд. В языках с классовой структурой те типы, которые можно реализовать классами не вводятся в базис.

[lect8] **IV) Логические и символьные типы данных**

boolean (*операции* or, and, nor, xor, equ, *значения* true = 1, false = 0)

Так сделано практически во всех языках, кроме Си (там нет boolean, для этого используется int, просто значение сравнивается с нулем – *неявное* преобразование int в boolean).

Сначала, для **символьного** типа данных хватало 1 байта. Но для многоязычной поддержки такой размер не годится. Т.е. существует две проблемы: мощность алфавита (сколько байт) и набор символов (character set). Появилось множество самых различных кодировок (**ASCII-7** – был базовым в 70е, 80е годы; включает в себя символы: 0-31 – непечатаемые, A-z, 0..9, пунктуация). **SBCS** (single byte character set) – однобайтовое представление символьных типов данных, первоначально в ЯП было именно так. Для национальных алфавитов предназначались символы 128-256. Но даже для Европы этого не хватало. [Рассказ про то, как появилась КОИ-8Р – отбрасывание старшего бита почтовыми серверами]. Японцы ввели систему на подобии UTF, если первый бит = 0, то байт считался ASCII, иначе добавлялись дополнительные байты. **UNICODE** (USC2 – universal character set, название по ISO; практически то же самое что и UNICODE) – универсальная кодировка, предполагала количество символов от 0 до 65535. Начиная с WinNT, MS использует UNICODE. В **C#** и **Java** тип данных char подразумевает именно UNICODE (2 байта). А в старых языках просто ввели новые типы данных (по сути, просто был define от unsigned short). В C++ ввели wchar_t, который не был просто define (typedef), т.к. надо было поддерживать *перегрузку функций* (вызов от разных типов данных). Параллельно с UNICODE был разработан стандарт **UTF** (UTF-7) для того, чтобы экономить трафик (в основном по сети передается англ. тексты и цифры). Весь UNICODE разбивался на промежутки 0-127, 128-2047, 2048-65535 и кодировался так, что любой ASCII-текст был одновременно UTF из диапазона 0-127 и т.д. Тогда, если нужен расширенный алфавит, то передавалось больше байтов, если же нет – то меньшее.

Порядковые типы данных:

- диапазоны
- перечислимые типы

Если **L** и **R** являются данными типа **T**, то **L..R** – **диапазон**. Должны быть определены функции pred, succ (как в Pascal), определенные только для типов, которые созданы на основе целого типа данных. Диапазоны впервые появились в Pascal и после этого были практически во всех ЯП. Они полезны т.к. используется квазистатический контроль (см. пред. лекцию). Однако диапазонов нет в Обероне, так как без них в принципе можно обойтись. Но как объяснить, что их нет в Java и C#? Диапазоны – **не** расширяемый тип данных (наследование не проходит), что противоречит концепциям ООП.

[lect9] **V) Перечислимые типы**

В Обероне конструкция входит в язык, только если она *необходима* в нем. Перечислимые типы были убраны из языка Оберон, т.к. они противоречат расширению типа. Перечисляемый тип - это просто удобный способ именования целочисленных констант.

- 1995 год - Java, проектировался с нуля
- 1999 - C#, содержал перечислимые типы. В первой главе руководства к C# автор "оправдывается" в том, что в язык введены перечислимые типы данных. Причина - перечислимые типы удачно подходят для визуального проектирования. Например, свойства компонентов удобно представлять в виде перечислимых типов: выравнивание - влево, вправо, по центру. Такой текст кода с перечислимыми типами называют **самодокументированным**.

Еще одно неприятное свойство перечислимых типов: вместе с именем типа происходит *неявный импорт* имен значений, и импорт происходит на том же уровне видимости. Вопросы неявного импорта сложны и могут приводить к труднообнаруживаемым ошибкам.

В **Java 5.0** (Java 2) в 2005 году перечислимые типы вернулись. Подход к ПТД в Java 2 снимает все претензии к ним в других языках. Здесь это не целочисленный тип, а *класс*.

Как перечислимые ТД реализованы в различных языках.

Pascal, Ада, M-2, Delphi:

```
type T = ( ... )
type T is ( ... )
```

Фактически перечислимые типы реализуются с помощью целочисленных констант. К этим типам применима функция **ord()** - явное приведение перечислимого ТД к числовым константам. В языке Ада символьный ТД - частный случай перечислимого типа данных. Кроме идентификаторов констант могут быть литералы перечисления (напр.: 'A').

Таким образом, литералы перечислений могут пересекаться, т.е. иметь несколько форм в одной и той же области видимости (что недопустимо ни в Паскале, ни в других подобных языках):

```
type RGBCOLOR is (... Red ...);
type TrafficColor is (Red, Green, Blue);
```

Ада:

```
function Fun return T is
    объявление локальных имен
begin
    операторы
end Fun
x:T;
x:=Fun;
```

Паскаль:

```
y:=Red;
```

Т.о. присваивание перечислимых значений можно рассматривать как вызов функции, которая все время возвращает одно и то же.

```
procedure P(x:RGBColor);
procedure P(y:TrafficColor);
```

P(Red) - неоднозначность при компиляции. Для этих случаев в языке Ада существует "указание типа":

```
Тип'Значение
```

Для данного случая:

```
P(RGBColor'Red) - указание использовать тип RGBColor.
```

C#:

```
enum E {c1, c2, ..., cN} //здесь видно только имя E, чтобы обратиться например к cK, надо написать: E.cK;
```

Таким образом, проблема неявного типа уходит, более того программы не становятся менее читабельными.

Можно указывать базисные типы:

```
enum E:byte{ ... }
```

Также считается, что 0 может преобразовываться к любому перечислимому ТД.

В языке **C** перечислимый ТД реализован неэффективно.

```
enum C {c1,c2,c3}; // c1=0, c2=1, c3=2
```

```
x = c1;
x = -25; //не ошибка!
```

Можно так же описать константы:

```
const int c1=0; //и т.п. ...
#define c1 0
```

Таким образом, программистам вовсе не обязательно было использовать перечислимый ТД. В **C++** сделан следующий шаг в усовершенствовании перечислимых ТД. В нем можно управлять нумерацией перечислений

Пример:

```
enum Mode
{
  Read = 0x1,
  Write = 0x2,
  ReadWrite = Read|Write
}
```

Недостатки: пониженная надежность.

В **C#** не существует возможности преобразования между enum и int.

Атрибутирование - частный случай рефлексии(отражение информации об исходных текстах в бинарных кодах), приписывание некоторому ТД атрибута.

Мы рассмотрим атрибут flags. Если установлен этот атрибут, то над ТД можно выполнять бинарные логические операции.

Т.о. перечислимый тип в C# получился достаточно мощным и активно используется. Enum, struct, простые ТД - в C# называются value types. Для каждого из таких типов существует "коробка" (box). Каждому примитивному ТД соответствует специальный класс-обертка (Wrapper Class). Например, для int - Int32, для uint - UInt32. Все эти типы наследуются от Object. У каждого класса, наследованного от Object, есть метод toString().

Для перечислимых ТД существует класс System_Enum.

Пример на Java:

```
enum Color {
   Red,
   Green,
   Blue
}
Color c;
c = Color.Red;
```

В Java потеряна вся связь между enum и целыми числами. Любое перечисление - наследник класса Enum.

```
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)
```

Таким образом, все перечислимые типы превратились в классы, поэтому все претензии к ним от Вирта исчезают.

[lect10] **VI) Ссылки и указатели**

Языки делятся на два класса с точки зрения указателей: строгие и нестрогие. Вопрос: можно ли описать в языке Паскаль указатель на integer? Ответ: нет.

- **Строгие языки** – указатели могут в них использоваться только для адресации анонимных объектов в динамической памяти. К ним относятся *Ада*, *Оберон*, *Модула-2*, *Паскаль*, *Ада-83*. В них нельзя смешать статические объекты и динамические объекты.
- **Нестрогие языки** – в них, кроме того, что есть в строгих языках, есть еще адресная арифметика и можно получить адрес любого объекта. Примеры: *C*, *C++*, *Delphi*. Проблемы: например, возвращение функциями указателей на локальные объекты.

Ада:

```
type PT is access T;
```

Модула-2:

```
TYPE PT = POINTER TO T;
```

Описание указателя на необъявленный тип (**Ада**):

```
type PT is access;
type T is record
  next: PT
end record
type PT is access T
....
x:PT
x:= new T;
```

В строгих языках над указателями допустимы только операции присваивания и разыменования.

М-2:

```
x^.next
```

Оберон:

```
x.next //как обращение к св-ву объекта
```

Еще один **пример ошибки** в нестрогих языках:

```
static int i;
...
int *p;
p=&i;
free(p);// – ошибка!!
```

При отладке такие ошибки можно находить, используя «отладочный» менеджер памяти. Как избежать проблемы нестрогих языков? Использовать строгие языки. Но этим все проблемы не решаются, если в языке есть явная операция освобождения памяти.

Проблемы, возникающие из-за явного освобождения памяти:

1) **Мусор**.

```
T* x=new T();
T *x1=new T();
x=x1;// появление мусора
```

Ошибка в программе – любое несоответствие требованиям пользователя. Мусор – серьезная проблема в промышленном ПО.

2) **Висячие ссылки** – использование адреса несуществующего объекта.

```
T* x=new T();
T* x=x1;
x1=x;
delete x; //появление висячей ссылки x1
```

Решение этих проблем – автоматическая сборка мусора. **Способы сборки мусора**:

1) Подсчет ссылок. Каждый выделенный кусок памяти помечается счетчиком. При выходе из области видимости счетчик уменьшается на 1. Когда счетчик=0 – объект уничтожается.
2) Учет памяти (mark&scan). В некоторые моменты времени запускается сборщик мусора, помечающий используемые куски, сдвигающий их к началу кучи и освобождающий неиспользуемые куски. При этом значения указателей и ссылок меняется, и сборщик мусора исправляет эти значения в программе.

Языки с автоматической сборкой мусора не используются в realtime-системах, т.к. они при некоторых условиях (при нехватке памяти) сильно тормозят выполнение программы.

В стандарте языка **Ада** не используется автоматическая сборка мусора (т.к. одним из основных его приоритетов является эффективность), но в реализациях ее компиляторов она может быть. Компилятор, не реализующий сборку мусора, обязан реализовать DEALLOCATE(p). **Модула-2**: в этом языке также нет автоматической сборки мусора, есть спец. модуль ALLOCATOR, в котором есть две процедуры – ALLOCATE и DEALLOCATE, спец. тип данных ADDRESS – к нему может быть неявно приведен любой указатель.

Такие типы данных (ADDRESS, void *) использовались программистами в основном при создании универсальных контейнеров. Так понятие адреса проникло в строгие языки. Только Оберон остался «чистым» от этого понятия. В нем автоматическая сборка мусора.

C#, Java – в них вообще отсутствует понятие указателя. Value types в этих языках называются простые типы и некоторые структуры (известного размера), для них отводится память под них самих. Для всех остальных объектов («большие» объекты – массивы и объекты большинства классов) память отводится для ссылки, а сами объекты располагаются в динамической памяти и являются анонимными. Этот принцип называется преференциальная модель объекта.

```
var X:T; {если T – имя класса, то X – ссылка}
```

Ада83 – строгий язык. В нем автоматическая сборка. В нем есть атрибут A'BASE – адрес объекта, но в стандарте говорится, что программы с его использованием непереносимы. Ада95 – появление атрибута a'access – адрес a. До Ады95 в ЯП Ада нельзя было применять концепцию объектов. Ко времени создания Ады95 изменились технологические предпосылки для этого языка. К этому времени было очевидно, что Ада не будет универсальным языком. В Аде95 можно передавать адрес, но только структур. Этот язык остался надёжным. a'access выдает адрес. Но для статических объектов адрес брать нельзя:

```
A: X;
A'ACCESS; – ошибка.
i: aliased integer; //aliased указывает, что к i применима адресная операция.
TYPE PT IS ACCESS T; //PT можно использовать только для динамических объектов.
TYPE PI IS ACCESS ALL INTEGER; //PI может использоваться как для объектов из
  динамической памяти, так и для ALIASED.
```

В **Java** есть JNI – Java Native Interface – возможность вызывать родные системные вызовы данной системы – противоречие WORA (Write Once, Run Anywhere), но это повышает производительность.

В **C#** оставлена «дырка» - спецификатор unsafe {} , в блоке которого можно использовать указатели. fixed(byte *p=new byte(10)){...} – эта конструкция указывает, что соотв. ссылка не может меняться в процессе сборки мусора. Аналогично - в visual studio c++ есть тип CString – надежная динамическая строка, этот тип появился до STL. В этом типе есть указатель на char *, и его можно получить, но этот класс использует алгоритмы, подобные алгоритмам сборки мусора, и этот указатель может измениться. Вызов S.GerBuffer() запрещает изменения этого адреса, но после использования нужно вызвать S.ReleaseBuffer(), чтобы опять разрешить изменения.

Все объекты *C#*, *Java*, *Delphi* являются ссылками.

C#, Java:

```
X a = new X(); //единственный способ порождения объекта
```

Delphi:

```
a: X;
a := X.create; {в этот момент отводится память и вызывается конструктор}
a. – разыменовывание ссылки.
```

В **C++** ссылка – синоним объекта, аналог другого имени. К ссылке в C++ применима единственная операция – инициализации.

```
T b;
T &a = b; //a – синоним b
int x;
int &i;
x = 0;
i = 1;
cout << x; //будет выведено 1
```

Одному объекту соответствует 2 имени – может получиться путаница.

Где **применяется**:

- X &a = GetSomeObject(); //вместо того, чтобы постоянно вызывать GetSomeObject()
- Параметры функций или возвращаемое значение.
- Член класса – инициализируется в списке инициализации конструктора.

Мы закончили рассмотрение примитивных типов данных в традиционных ЯП.

Зачем Страуструп ввел ссылочный тип? Для того, чтобы можно было переопределять стандартные операции. Например, при переопределении операций = , [] лучше использовать ссылку, а не указатель. Во многих библиотеках есть спец. типы – разумные указатели – классы, в которых переопределена операция -> . Подробнее в книге Андрея Александреску, «Современное проектирование на C++: Обобщенное программирование и прикладные шаблоны проектирования».

[lect12] **Глава 2. Составные ТД**

I) Классификация СТД

- массивы
- записи/структуры -> классы
- множества
- файлы
- строки
- ассоциативные массивы (хеш-таблицы)

Все типы, кроме массивов и структур, не имеют четко определенных эффективных алгоритмов управления и методов хранения. Например, сортировать ассоциативный массив можно с помощью **qsort**, **heapsort**, **пирамидальной сортировки**, каждая из которых имеет недостатки на определенных наборах ключей. Поэтому встраивать фиксированные методы в компилятор не всегда оптимально.

II) Массивы

последовательность однотипных элементов (непрерывная, с точки зрения размещения по памяти)

Атрибуты массива:

- базовый тип => D
- тип индексов (для этого типа должны быть применимы операции *succ* и *pred*, основаны на целых)
- L..R (левая и правая граница)и связанный с ним атрибут Length

Связывание массива с типом его элементов **статическое** во всех языках (т.е. на этапе компиляции). Именно для решения этой проблемы ввели void* (ADDRESS) для указания абстрактного адреса на объект.

При реализации массивов (да и любых вещей) надо находить **компромисс** между:

- надежность
- эффективность
- гибкость (удобство для программиста)

/про сравнение гибкости, надежности и т.д. для разных языков; си – эффективен, например (нумерация массивов всегда идет с нуля, а вот в Паскале может быть как угодно)/

Операции, применимые к массиву:

- индексация (от двух аргументов: массив и индекс) – возвращает ссылку на элемент
  D: **[]**, A,**i** => ref D
- изменение атрибутов (например, размера)

Существуют **открытые массивы** (например, в Modula2) – это, по сути, шаблон:

procedure P(var A:T); //здесь T – открытый массив, формальный параметр функций
A: ARRAY OF T

Вызывая, HIGH(A) мы получаем правую границу, левая граница – 1. Это и есть открытые массивы.

Ада – **неограниченные массивы**.

```
type arr is ARRAY (index range <>) of D; //A'RANGE - статический атрибут
x: ARRUNL range -10..10; // A'RANGE - динамический атрибут для ARRUNL
```

C#, Java, Delphi (совместимость с Pascal оставлена)

Считается, что тип индекса **int**, в диапазоне от **0..N-1**. Length – атрибут *экземпляра*, а не *типа*.

```
T []x = new T[N]; //N - не обязательно константа
0 <= I <= x.Length
```

[lect13] **Многомерные массивы**

Поддержка массивов это всегда компромисс между эффективностью и надежностью (например, проверка out-of-range при каждом обращении). Длина массива - это свойство *не типа*, а *экземпляра*. Название массива - это *ссылка* на *экземпляр*. Поэтому в C#, Java, Delphi нельзя объявить массив с уже прописанной длиной (надо сделать new). Поэтому менять длину массива во время исполнения нельзя, несмотря на то, что массивы выделяются из динамической памяти.

«Вырезка» (Slice) – PL/I, FORTRAN 90, Ада

A: Array (1..10) of T;
A(2..5); //вырезка

A(*, i); //i-й столбец
A(2..5, 1..3); //некоторая матрица – подтип исходной

Многомерный массив – фактически просто последовательность одномерных массивов (массив массивов).

x[i][j] ~ x[i,j] – эквивалентно в Pascal, элементы по строкам

III) Записи

У массивов основная операция – *индексирование* **[]**, а у элементов записи есть операция **name.id**, которая применима слева к объекту типа «запись», а справа – к имени поля. Синтаксис записи во всех языках практически одинаков. Если мы заключаем набор объявлений переменных в скобки, то эта совокупность объявлений считается локализованной в записи, и доступ к ним может быть только по имени записи.

Pascal:

```
type R = record
  s, y: T;
  i, j: integer;
end;
var rec: R;
```

Ада:

```
record
  //последовательность полей;
end record;
```

C:

```
struct tag
{
  //объявления полей;
}
```

Записи - это естественное обобщение классов. Но у класса могут быть еще и методы. Класс носит не только функцию типа данных, но и модуля. Вместо механизма объединения типов есть наследование классов. Пустой базовый класс не является ошибкой. Объединение осталось в C++ для совместимости с Си. А в Аде есть параметрические записи. В Java понятие записи вообще убирается. Структура отличается от класса в C++. В классе наследование и доступ по умолчанию приватные, в структуре - публичные. Имена классов обязаны быть уникальными, а имя структуры - нет (если класс описан при помощи структуры).

В C# понятие записи совершенно другое. Есть понятие value type и reference type структур - нереференциальные типы данных.

```
точка – point x,y
```

Быстрее инициализировать массив point, нежели массив ссылок на структуры point.

Структуры - недокласс в C#. Структура не может наследовать и наследоваться. Не существует виртуальных методов структуры, как любой класс может содержать функции-члены. У структуры нельзя переопределять конструктор умолчания (конструктор без параметров), так как в C# любой объект не имеет начального значения.

В объектных языках (C#, Java) все является объектами. В C++ стандартные типы объектами не являются.

Проблема Януса: разные объекты могут выступать в разных ролях. Ярче всего проявляется при проектировании интерфейсов. Например, окно может содержать набор элементов, зависящий от конкретного окна.

```
record
  постоянная часть;
  вариантная часть;
end record;
```

Постоянная часть - последовательность объявления переменных.
Вариантная часть:

- размеченное распределение (есть дискриминант, определяющий вид поля)
- не размеченное

Как в Паскале получить доступ к биту? Никак? Нет, для этого есть неразмеченная вариантная часть.

```
packed array of boolean; //это и есть последовательность битов
```

Паскаль:

```
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;
```

[lect14] **Объединение типов**

- регулярные - дискриминант
- нерегулярные

C:

```
Union tag { T1 v1, T2 v2, Tn, vn };

Case дискриминант of
 V1: (объявления полей);
 V2: (объявления полей);
 V3: (объявления полей);
```

А каков **размер** объединения? В Си он выбирается абсолютно естественно – по размеру максимального элемента. В Pascal память выделяется строго по тому варианту, который указан в new(P, t). Если память распределена по одному варианту, а мы обращаемся по другому, то возникает ошибка. Такая ошибка не обрабатывается компилятором т.к. это сложно, а в некоторых случаях невозможно.

```
new(P, t); //присваивания не происходит т.к. может быть неразмеченно
P^.D := t; //не стоит это забывать
```

Параметризованная запись (Ада):

```
type VARREC(D: DT) is record
 постоянная часть
 case D: DT of
  when v1 => (вариант 1)
  when v2|v3|v5…v10: (вариант 2)
  when others…
end record

x: VARREC(t); //t: DT;
y: VARREC; //ошибка!
```

Формально, параметр D можно поменять, но на практике – нельзя (выделение памяти происходит при инициализации, и менять значение дискриминанта не получится). Память здесь выделяет некий зачаток конструкторов. В современных ООП нет нужды в записях с вариантами, если их оставили, то только для совместимости (Delphi, C). В Обероне, Java, C# такой нужды нет, так в них нет проблем совместимости, нужные функции здесь выполняются, используя *наследование*.

Основной **недостаток** записей с вариантами, объединений типов: они объявляются один раз, а используются много раз – поэтому при модификации кода (добавление событий) появляется много проблем (сложно добавить новое во все переключатели, можно забыть). Отсюда появляется *ненадежность*. Реально решает эту проблему только **динамическое связывание методов**. В идеале для этого нам нужны исходники только базового класса, и того, который мы добавляем.

Для совместимости в C++ struct – это не совсем класс. По умолчанию это есть класс, но разрешены совпадения имен. Если такие конфликты встречаются, то доступ к структуре осуществляется только по ключевому слову struct.

Пример: если мы объявляем класс Point, содержащий две координаты (+какие то методы), то для передачи массива из Point в функцию мы выделяем память под массив указателей и по два инта на каждый указатель. То же самое при удалении. Эта проблема появилась в C#, Java, но ее нет (и не может быть) в C++ (там нет сборщика мусора и проч.). Для решения этой проблемы (производительности) введено слово struct, но это не то же самое, что в C++. В итоге упрощается процедура распределения памяти и увеличивается производительность. С точки зрения синтаксиса struct не отличается от классов, но не с точки зрения функциональности (нет наследования, другое распределение памяти, запрещено переопределять конструктор по умолчанию).

Итог: запись (с вариантами) – устаревшее понятие. В современных языках понятие записи переросло в понятие класса.

IV) Другие составные типы данных

Эти типы данных в современных ЯП мигрировали в библиотеки.

- абстракция I/O (файлы и т.д.)
  writeln в Паскале – выглядит как процедура, а на самом деле просто ключевое слово
- множества, таблицы (есть только некоторые исключения: SETL, M-2, Паскаль)
- строки (во всех языках тип данных string встроен в базис, исключение – C++, потому что он очень мощный) – особенный типа данных, для него важна эффективная реализация (например, присваивание – просто копирование указателей)

[lect15] **V) О единстве составных типов данных**

- **[]** – операция индексирования, A x I -> Val (в C++ Val&)
- **. –** операция «точка», фактически частный случай индексирования

С точки зрения общего подхода, множества – частный случай массивов, где элементы – объекты.

Если мы обращаемся к объекту от неопределенного индекса, то он создается (мы разрешили неограниченно разрастаться как объектам, так и массивам). Если нам не хватает определенного свойства – то мы можем ввести его динамически. За счет этого сильно увеличивается *гибкость* языка, но становиться затруднительно контролировать объекты => падает *эффективность* и *надежность*.

```
obj.prop=-1; <=> obj["prop"]=-1; //если свойства prop нет, то эта конструкция его добавляет
5.prop; //что делать? придавать свойство prop всему классу Integer или только экземпляру?
```

Т.е. мы можем трактовать массивы и структуры унифицировано.

Глава 3. Управление последовательностью действий

Традиционные ЯП основываются на архитектуре Фон Неймана, основные понятия которой – **состояние** и **поток управления** («control flow» instead of «data flow» in func langs).

I) Эволюция понятия ПУ

Уровни ПУ

- внутри выражения
- между операторами программы
- между модулями

Поток управления определяется, во-первых, приоритетами операций. Интересен другой вопрос: f(a, b) – в каком порядке будут вычисляться операнды? В современных языках программа, зависящая от порядка вычисления операндов (каких-либо операций, кроме логических), считается нестандартизованной и непереносимой:

```
*--*cp++; //даже не все компиляторы могут понять
WHILE (A[I] != X AND I < N) …; //ошибка или нет зависит от порядка вычисления
```

Имеет смысл зафиксировать *ленивость* логических операций (например, a and b – если a == false, то b не нужно вычислять, т.к. результат в любом случае будет false). В языке-предтече Ады предлагалось ввести два вида логических операций – обычные и ленивые (and then, or else). В результате ленивость была зафиксирована. В C логические операции также ленивые. С точки зрения оптимизации и распараллеливания нужно минимизировать побочные эффекты операций.

Первоначально, под *потоком управления* понимали передачу выполнения на любую точку кода (GOTO). Если передашь не туда – твоя проблема. Так и было в Фортране. Ситуация стала меняться, когда стали писать более сложные программы (ОС, например). 1967 – Э.Дейкстра опубликовал статью о вреде GOTO – это была одна из самых популярных статей по CS. Фактически, эта статья положила началу новому направлению – **структурированное программирование** (однако, прижился перевод структурное). Появилось понятие **структуры управления,** т.е. есть некоторые «черные ящики», которые передают друг другу данные, управление и у каждого есть только один вход и выход.

Управляющие структуры:

- ветвления
  - односторонние (**if** B **then** S1;)
  - двусторонние (**if** B **then** S1 **else** S2;)

  Существует проблема замыкания (два **if** подряд – не понятно, какой **else,** к какому **if**) – это синтаксическая неоднозначность. Проблемы была решена в пользу ближайшего **if**. Используйте операторные скобки и все будет хорошо. В некоторых языках (Ада, M-2, VB) решили проблему по-другому – там есть определенные терминанты (endif, end), и нет составных операторов.
- циклы
- составной оператор
- вызов подпрограммы / возвраты (call, return)

[lect16] Есть две основные управляющие структуры - **ветвление** и **циклы**.

I) Ветвления

Оператор ветвления на 2 направления – **if**. Основная проблема - проблема вложенности.

```
if B1 then
  if B2 then
    S2
  else
```

```
    S3
```

В языках появляется составной оператор - частный случай блока. Он применяется там, где по синтаксису подходит 1 оператор, а требуется несколько. Отличие от блока - в блоке могут появляться локальные переменные. В языке паскаль блок - объявление, за которым следует составной оператор.

```
 объявления
begin
  операторы
end
```

ADA, **Modula-2**, **Bash (B-Shell)**, **Оберон** - используют схему явного закрытия операторов.

Проблема вложенности решена:

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

```
if B then
  S1
else
  S2
end
```

Чем мы вынуждены платить за явное закрытие? В некоторых случаях надо употреблять многовариантные развилки.

```
B1 –> S1;
... –> ...
Bn – >Sn;
```

Такие развилки могут моделироваться через много вложенных if ... then .. else.

```
if B1 then
  S1
else
  if B2 then
    S2
  else
  if B3 then
    S3
...
```

Но человеку трудно воспринимать такие структуры. Корректная запись такой конструкции (на си):

```
if (B1)
  S1
else if (B2)
  S2
else if (B3)
  S3
...
else
  Sn+1
```

Но в некоторых языках в конце придется написать еще много "end if", что неудобно. Поэтому в язык надо специально добавить оператор многовариантного ветвления.

В языке **Ада** есть оператор elsif, в **М-2** - ELIF

```
if B1 then
  Seg1
{elsif Bi then
  Segi}
else
  SegN+1
endif
```

Общий **оператор выбора**

```
case Expr of
  описание вариантов
end
```

Варианты

```
константа: оператор
```

Изначально Вирт не предусмотрел действия, если ни одно значение не подходит. Но позже в паскале была добавлена такая конструкция (else оператор).

В **М-2**:

```
CASE EXPR OF
   список вариантов |
[ELSE
   операторы]
END
```

Возможны и **диапазоны**:

```
конст1..конст2
1,3,5..10,12
```

Т.о. семантика везде одинакова, синтаксис варьируется. Оператор **переключателя** (в Си):

```
switch (expr) S
```

Семантика: вычисляется значение выражения expr. Затем выполняется поиск нужной метки.

```
case 1:case2: //Если expr == Ci, то делается goto Ci.
```

Распространенная **ошибка:**

```
case 1:
a = i; // после первого case нужен break;
case 2:
   ...
```

В языке Java синтаксис остался тем же, хотя оператор перехода там отсутствует вообще. Слово goto там зарезервировано. Если не написать break после case, компилятор выдаст ошибку (в C# так же). Если значение не входит в список констант, то нужна метка default (действия по умолчанию).

II) <u>Операторы цикла</u>

```
while B do S
repeat S1; S2; ... Sn; until B
```

В общем случае выход из цикла может быть в начале, в конце, в любом месте цикла. Таким образом, должна быть специальная конструкция выхода из середины цикла.

Модула-2:

```
WHILE B DO ... END
REPEAT .... UNTIL B
LOOP ... END //бесконечный цикл, выход только по EXIT (актуален в задачах
параллельного программирования)
```

Ада: всего 3 разновидности цикла

```
while, loop, for
```

Есть оператор exit:

- укороченная форма - exit;
- полная форма - when усл-е => exit;

Си:

- while (B) S;
- do S while (B);
- for

break - оператор выхода
continue - оператор продолжения

В **Ада**, **Си**, **Модула-2**, **Оберон** - есть оператор выхода из функции - return. В Ада в качестве "break" используется оператор exit.

```
имя:
  цикл или переключатель
 break (имя) – выход из того цикла или переключателя, который  помечен меткой
"имя"
```

<u>Цикл FOR</u>

Цикл for всегда можно смоделировать циклом while.

Паскаль:

```
for v:=l1 to(downto) l2 do S
```

Модула 2:

```
FOR v:= E1 TO E2 [STEP N] DO .. END
```

1988 **Оберон**, 1993 **Оберон-2** - цикл for вернулся.

Ада:

```
for v in диапазон loop
 ...
end loop;
```

```
for i in A'RANGE loop
  for i in A'FIRST..A'LAST loop
...
```

Замечание: в языке Ада переменную i не надо объявлять.

В **Си++** индекс можно объявлять прямо внутри цикла, и его область действия ограничивается этим циклом.

```
for (int i=0; i<N; i++) S
```

С появлениям языков **C#** и **Java** ситуация изменилась, например, нужно проходить циклом по коллекциям. Потребовалось контролировать диапазон индекса:

```
for(int i=0; i< A.GetLength(P); i++)
  .. A[i] // понятно, что i не может превзойти Length(P)
```

Коллекция - обобщение массивов - просто набор элементов. Появилась специальная конструкиця для обработки коллекций: **foreach**

```
foreach (T o in C ) S
foreach (int x in a) s+=x;
```

С помощью такого перебора нельзя модифицировать коллекцию, но просмотр становится эффективным. Этот оператор появился в C# с самого начала, а в Java только с 2005 года.

Java:

```
for (T o: C)
  Timertask
    cancel();
```

```
/////
collection<TimerTask> c;
for (TimerTask in c)
  t.cancel();
```

Что мы не учли в этой главе: оператор **entry**, **select** (**Ада**) - для параллельного программирования.

[lect17]

ПРОПУЩЕНА ЛЕКЦИЯ (23 октября)

[lect18]

Способы **передачи параметров**:

1) по значению
2) по результату
3) по значению результата
4) по имени
5) по ссылке

1)-3) применяются в случае, когда передаются небольшие данные, иначе накладные расходы слишком велики. **Thunk** – переходник.

Как это реализовано в языках:

C - по значению

C++ - по значению + по ссылке, причем можно передавать константные объекты

```
void f(  const T& ); //константная ссылка
```

более того, предполагается что при передачи без "const" объект обязательно должен изменяться в процессе обработки

```
void f(T) ; //по значению
```

Pascal, Modula-2, Oberon-2 - по значению, по ссылке (параметры-переменные). Способ передачи по

ссылке обеспечивает полный доступ к объекту.

Ada - способ передачи параметров не указывается при передаче

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

x меняться не может, y,z - могут. При этом компилятор может выбрать соотв. семантику. Он может выбрать способы 1),2),3),5). В **Ada-95** вернулись к способу, который так или иначе реализован во всех других языках.

```
Procedure P(x,y:out T)
begin
x = ....;
raise ERROR;
end P

A:T;
P(A,A)
```

Если передаем по ссылке и происходит исключение, то значение A меняется. Работа программы может зависеть от компилятора, и это плохо. В ада-95 есть только 2 способа передачи - по значению и по ссылке.

Java - все передается по значению, реально же простые типы передаются по значению (in семантика), а объекты - по ссылке (inout семантика).

Проблема - для простых типов данных - только in семантика, т.е. нельзя поменять в процедуре значение внешней простой переменной. На первый взгляд это недостаток. Но есть такое понятие как "побочный эффект" - изменение в процессе работы процедуры значений глобальных объектов или фактических параметров. В случае изменения внешней переменной получается, что основным назначением процедуры является побочный эффект. Не рекомендуется менять значение глобальных объектов в процессе работы процедур. В современных ООП, таких как Java и C# вообще невозможно существование каких-то глобальных объектов.

В **java.lang** для каждого простого типа данных существует класс-обертка.

Пример:

```
int a;
Integer v = a;
f(v); //здесь мы можем менять v, т.к. он передается по ссылке.
a = v+1; // обратная распаковка

void f(Integer v) {
  v = -1;
}
```

C# изначально проектировался как "моноязык". По умолчанию все параметры передаются по значению, параметры типа объект - по ссылке. Есть 2 модификатора **ref** и **out**, которые реализуют передачу параметра по ссылке.
void f(ref int a) { a = -1; }

```
int i = 0;
f(i); //ошибка, компилятор скажет что нужно добавить ref
f(ref i); //правильно. Эта особенность улучшает документированность кода.
```

ref/out семантика может применяться к любым объектам, в т.ч. и к ссылкам

```
void g(ref X y) {
  y = new X();
}

X a;
g(ref a); // ошибка, неопределенная ссылка. (ссылка передается по ссылке)
```

Delphi

```
procedure P(a:X);
a f();
a = X.create();
```

Ввод-вывод, переменный список параметров

```
printf("%d %d=%s", a, b+1, str)
```

параметры переменной длины:

```
va_list
```

```
va_start
va_next
va_end
```

printf - небезопасная, но очень популярная функция. До сих пор ее часто используют в c++

```
printf("count=%d\n", i);
```

Oberon:

```
InOut.WriteString("count=");
InOut.WriteInt(i);
InOut.Writeln;
```

В современных языках **Java** и **C#**

params - модификатор последнего формального параметра.

```
void f(params int[]integers); // теперь у функции может быть произвольный список
параметров, но либо целого типа, либо приводимого к нему
```

функция write должна выводить любые типы, поэтому:

```
write(string format,params Object[] objects); //все типы приводимы к Object
```

Обратим внимание на следующую функцию:

```
void f(int[] integers); //существен. разница, аргументом может быть только массив
```

в **C# 3.0** введено ключевое слово var - переменная с выводимым именем типа (не бестиповая!)

```
ListVewItem i = new ListViewItem();
var i = new ListViewItem; // то же самое что и выше
```

Такие нововведения связаны с тем, что C# разрабатывался с учетом интеграции с БД, другими языками. Создатели **Java** не пошли по этому пути, и обошлись без добавления новых ключевых слов.

Функцию с переменным числом параметром можно записать так:

```
void f(int ... integers) { // новая лексема "..."
int r=0;
for(int i: integers) {
r+=i;
}
return i;
}

//функция вывода может быть описана так:
void write(string fmt; Object ... objs);
```

III) Подпрограммные типы данных

- передача подпрограмм как параметров других подпрограмм

Вирт ввел еще 2 особых вида параметров - формальные параметры-процедуры, формальные параметры-функции.

Delphi, Modula-2, Oberon - введен подпрограммный тип данных

```
TYPE PROCINT=PROCEDURE(INTEGER);
VAR P:PROCINT; // можно присваивать процедуры, имеющие 1 параметр типа INTEGER

TYPE F=PROCEDURE(REAL):REAL;
PP = PROCEDURE(VAR T);
```

До появления ООП подпрограммные типы рассматривались как средства передачи параметров в другие подпрограммы.

В языке **ада-83** не было подпрограммных типов.
В языке **ада-95** появился программный тип (т.к. за это время изменились взгляды программирования)

- наследование
- композиция
- референция
- делегирование (передача использования части интерфейса)

В **оберон-2** было понятие наследования, при этом наследовалась структура типа (поля класса). Без процедурного типа невозможно полностью реализовать объектную концепцию.

[lect19] **Подпрограммные ТД**

- передача параметров
- делегирование

Сначала рассмотрим подпрограммные ТД как указатели. В таком случае фактически передаем только адрес начала массива данных, затем выполняем call, к таким языкам относятся: C, C++, Паскаль, М-2, Оберон, Delphi, Ада95.

М-2, Оберон, Delphi:

```
type PRCi = procedure( integer );
```

Такие конструкции появились, когда понадобилась связь с другими подпрограммами, т.е. нужно было передавать адреса процедур и функций.

Ада95:

```
type PRCi is access procedure PRC(i: integer);
procedure PP(a: integer);

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

Попытки повысить гибкость языка за счет передачи адресов функций провалилась т.к. все эти языки происходят от Фон-неймановских языков. Особняком стоят только C++ и Delphi так как в них есть понятие *класса* (а также процедуры и функции члены класса), а в остальных есть понятие *контейнера*.

C++, Delphi:

```
void Foo(int);
typedef void (*Footype) (int);

Footype f;
f = Foo;
```

Теперь мы можем вызывать Foo **так**: `*f(0)` **или так**: `f(0)`

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

int Bar::*pB;
pB = &Bar::i; //так нельзя

Bar b;
pB = &b; //а вот так можно

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

Для обращения к указателю на член класса используются операторы: **.*** и **->***. То же самое, но с другим синтаксисом присутствует и в языке Delphi. Т.е. отдельно существует функциональный тип и тип, который реализует указатели на члены класса.

Java:

В Java **нет** понятия функционального типа. Т.е. нельзя передавать указатели на функции. Полезный пример, где нужно передавать указатели на функции: вычисление интеграла.

```
class Integral{
  public double IF(double){…}
  public double Integrate(double a, double b, double EPS){…}
```

Т.е. все, что нам надо – это унаследоваться от класса Integral и перегрузить функцию IF, в дальнейшем процедура Integrate будут вызывать уже нашу функцию. Есть другой способ.

```
class FuncBar{
  public FuncBar(Bar b){ _b = b; }
  private Bar _b;
  public void Caller(int){ _b.Foo(i); }
```

Так как указатель слишком низкоуровневая штука и, как следствие, она небезопасна, то ее и убрали из Java. Как видим, без этого можно обойтись.

C#:

```
public delegate void Operation(int);
  //это прототип, в этот delegate входят все функции, которые имеют такой тип
public Operation dlgt;
class Bar{
  public void Foo(int x){…}
  public static void Foo2(int i){…}
Bar b;
dlgt = new Operation(b.Foo); //можно так
dlgt = new Operation(Bar.Foo2); //а можно и так, указатели на разные функции
```

В delegate включается не только указатель на объект, но и указатель на член функцию. Для delegate определены также операции **+=** и **-=** (фактически это просто вызовы static-функций из класса System.Delegate). Это значит, что delegate – это целые «цепочки». Изначально delegate инициализируется пустым значением.

```
dlgt += new Operation(b.Foo);
dlgt += new Operation(Bar.Foo2);
class X{ public void f(int a); }
dlgt += new Operation(new X().f); //т.е. не важно какие функции, какого типа
```

От delegate можно определить массив функций и вызывать их отдельно. Т.е. delegate – просто более короткий синтаксис. Специально для callback в C# есть особые delegate – механизм событий. Есть операции subscribe, unsubscribe и distribute, т.е. мы подписываемся на некое событие, а когда оно происходит, нас уведомляют. *Пример:* таймер, почтовые сообщения.

Также в языке C#, начиная с версии 2.0, появились анонимные делегаты.

```
public delegate int Operation2(int x, int y);
Operation2 op = delegate(int x, int y) { return x + y; };
int i;
op = delegate(int x, int y) { return x + y + i; }
  //в качестве контекста можно использовать даже локальные переменные,
  //это называется «захват» или замыкание
```

Заметим, что в C# нет функций как таковых, а есть static (или просто) челны класса.

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

Вопрос: что напечатается, если вызовем Create? (0, 1, 2), если i описана как в примере, и (0, 0, 0) если она «совсем локальная» (переместить). Также делегаты можно использовать для Binder. Более общее и «приятное» понятие, заменяющее анонимных делегатов (покрывающее, более общее) – лямбда-выражения (введены в 2008 в C#, известны очень давно).

[lect20] **Средства развития ЯП**

Глава 5. Логические модули в традиционных ЯП. Определение новых ТД с помощью ЛМ

I) Понятие ЛМ

Логический модуль – языковая конструкция, объединяющая взаимосвязанные ресурсы в именованный набор, позволяющая использовать эти ресурсы, изменять и транслировать их.

Фортран, А-60, Стандартный Паскаль: в них были физические модули.

ТД = *множество операций* + множество значений. Одна из основных целей ЛМ – определение новых типов данных. Класс в современных ООП является логическим модулем, а вот структура как таковая не является. Объединение, например, математических функций в один модуль логично и помогает программисту (Math, там только статические члены-функции).

II) ЛМ в языках М-2, Оберон, Delphi

М-2: вообще говоря, весь проект разбивается на модули:

- главный модуль (в нем, собственно, программа)

```
MODULE name; var …; BEGIN operators … END name.
```

- библиотечные модули (БМ)
  - модуль определений

```
DEFINITION MODULE M; var …; END M;
```

    - Объекты данных
    - Типы данных
    - Заголовки процедур и функций
  - модуль реализаций

```
IMPLEMENTATION MODULE M; var …, realization…; END M;
```

- логические модули (пример: мониторы, не будем их разбирать)

Пример реализации модуля объявлений (**интерфейс**) стека на М-2:

```
DEFINITION MODILE Stacks;
  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; //для контроля успешности выполнения
END Stacks;
```

Модуль определений + Модуль реализации => UNIT:

Delphi:

```
unit Stack
interface        //модуль интерфейса
…
implementation   //модуль реализации
…
end Stacks.
```

Рассмотрим взаимодействие модулей между собой. БМ что-то экспортирует и импортирует из модулей реализации. Очень важно понятие **видимости**: *потенциальная видимость* и *непосредственная видимость* (т.е. имя может использоваться без всяких уточнений).

Уточненный идентификатор имеет такую форму:

```
имя_модуля.идентификатор
```

M_1	M_2	…	M_N
DEFINITION_1	DEFINITION_2	…	DEFINITION_N
IMPLEMENTATION_1	IMPLEMENTATION_2	…	IMPLEMENTATION_N

Импорт, нужен для того, чтобы имена стали непосредственно видимыми:

```
IMPORT список_имен_БМ;
VAR S: Stacks.Stack;
FROM Stacks IMPORT Stack, Pop, Push;
```

Если не импортировать, то непосредственно видимыми в М-2 являются *только* стандартные идентификаторы и имена модулей. В языке Си есть проблема с пространством имен – оно одно.

В языке **Оберон** все еще проще. Нет главного модуля. Экспортируемые процедуры без параметров называются командами (М.Р, есть команда загрузки – LOAD M). Т.е. есть только библиотечные модули, которые бывают модулями определений и просто модулями (реализации). В языке **Оберон-2** вообще остался только один модуль т.к. модуль определений в принципе может быть сгенерирован из модуля реализации (в нем нужно как-то помечать экспортируемые имена, например звездочкой). Существуют специальные утилиты для такой генерации. Все глобальные имена здесь тоже видны только с уточнением.

III) Логические модули в языке Ада

```
package имя is
  объявления;                                  //определения пакета
end имя;

package body имя is
   определения всех процедур и функций;       //тело пакета
end имя;
```

Пакеты в **Ада** могут быть вложенными. Это и есть главное отличие от М-2, которое все усложняет. Есть глобальный пакет STANDARD, в который вложены все пакеты проекта. Вложенность тел пакетов должна быть такая же, как вложенность объявлений пакетов. Соответственно определяется и область видимости имен (переменных и пакетов). Непосредственно в «своем» пакете имя видимо непосредственно, а все объявленные ранее имена видны через уточнение. Появляется дерево видимости.

[lect21]

Также могут быть чисто служебные пакеты, которые нужны только для реализации.

```
package P is
    package P1 is
        Package P2 is
    .....
end P2
endP1

...
package body P is
    package body P1 is
        package body P2 is
            package AUX is
          ....
       end AUX
   package body AUX is
    ....
   end AUX
    end P1
end P
```

Пакет AUX виден только внутри пакета P2 и не может использоваться больше нигде.

Top-Down подход к программированию - сначала главный модуль, спецификация, проектирование и т.д. Такой подход очень красив и позволяет, начиная с самых главных проблем переходить к более мелким. Недостатки такой системы: главный модуль начинает работать только тогда, когда написаны самые нижние модули, а это как правило ввод-вывод. ввод-вывод обычно пишется в последнюю очередь, и готовая система выпускается поздно. Лучше, когда можно тестировать модули по мере их написания.

Другой подход - top-down - наоборот. Реально используют комбинации первого и второго метода.

Обозначение: **O,M,D** - **Oberon**, **Modula**, **Delphi**.

Перегрузка операций - частный случай полиморфизма. Одному имени в одной и той же области видимости соответствуют несколько семантик. Раньше в языках был жесткий принцип: каждому имени в одном области видимости могла соотв. только одна семантика. Но в современных языках только Оберон и Модула не имеют перегрузки функций. Во всех остальных она разрешена, и даже в двух формах - статический полиморфизм и динамический полиморфизм.

Оберон - единственный язык, в котором нет статического полиморфизма.

```
// в ада:
function P: boolean;
function P: integer;
if P then // по контексту можно определить тип

//c++
bool f();
int f(); // нельзя, т.к. непонятно какой контекст
```

Являются ли стандартные операции объектом перегрузки? В этом вопросе существует 2 течения:

1) **Ada**, **C#**, **C++** - разрешена перегрузка, стандартные операции считаются функциями +- можно перегружать в двух вариантах: двуместной и одноместной. В прологе можно перегрузить + семиместным предикатом.
2) **Delphi**, **Java** - перегружать стандартные операции нельзя.

Для чего нужна перегрузка стандартных операций? Классический пример - матричная арифметика.

```
package Matrix is
   type Matrix is ....
function "+"(m1,m2: matrix)
return Matrix;
end Matrix;
x, y, z: Matrix;
z := x+y;  // можно
z := "+"(x,y) // тоже можно
```

Использовать такую форму можно только внутри пакета Matrix из-за области видимости.

Другой пример:

```
x,y,z: matrices.Matrix;
z := "+"(x,y); // нельзя, т.к. компилятор не увидит такой операции
z := matrices."+"(x,y)
```

Таким образом, все удобство перегрузки пропадает. Необходимо иметь возможность "вбрасывать" видимость процедуры во внешнюю область.

Возьмем для примера **Модула-2**. Сначала мы видим стандартные имена. Затем имена библиотечных методов, которые загружаются при выполнении import.

```
P
  P1
    P12
  P2
    P21
```

Если заполнять стек имен, находясь в определенном пространстве, то структура пространства может становиться очень сложной.

IV) Инкапсуляция и абстрактные типы данных

Впервые понятие инкапсуляции появилось в 70х годах.
Абстрактный тип данных – множество значений, структура которого полностью инкапсулирована.

Наследование бессмысленно без динамического полиморфизма. По мнению современных авторов, самым важным свойством современного языка программирования. является инкапсуляция. **Инкапсуляция** = "скрытие". В современном программировании очень важны интерфейсы, а хорошие интерфейсы невозможны без сокрытия некоторых свойств. Инкапсуляция "защищает" данные. С точки зрения инкапсуляции все объекты равны.

Атомом защиты является либо тип целиком, либо часть типа (переменная, член класса и т.д.). В **Ада** атомом является тип, т.е. мы можем либо открыть весь тип, либо закрыть весь тип. Как следствие, нужно все закрывать, т.е. программист должен использовать А.Т.Д.

Оберон + языки с классами - атомами защиты служит часть типа.

Модула-2:

```
DEFINITION MODULE M;
...
CNDM
IMPLEMENTATION MODULE M;
...
END M
```

В дельфи модуль также делится на 2 части - объявление и реализацию. Ада - спецификация пакета. Все что объявлено в интерфейсе - видимо извне. Все что в реализации - закрыто, т.е. сама структура логического модуля основана на понятии инкапсуляции.

```
TYPE LINK =
RECORD
END // здесь реализована инкапсуляция; сам тип доступен извне, а его члены – нет
      (ини доступны только из того же модуля что и объявление)
```

В общем случае АТД - это множество операций.

М-2: АТД реализуется через скрытый тип данных. Он объявлен в модуле в таком виде:

```
TYPE T;
DEFINITION MODULE STACKS;
    TYPE STACK; // структура здесь скрыта и нам неизвестна
```

```
PROCEDURE INIT(var s: stack);
          PUSH(var s: stack);
          X:POP()
...
END STACKS;
```

Над скрытым типом определены 2 операции - присваивание (:=) и сравнение (=, !=). Вирт не назвал тип данных абстрактным, поскольку он не совсем абстрактен. Например, как действует семантика копирования применительно к стеку? Возможны 2 семантики: "поверхностная" и "глубокая". В модула-2 - поверхностная семантика. Скрытый ТД внутри реализации должен быть реализован либо в виде целого ТД либо в виде указателя, поэтому он не является абстрактным в чистом виде.

[lect22] **Абстрактные типы данных**

Модула-2 навязывает референциальную модель - реализация типа может быть только integer и pointer. Это связано с механизмом раздельной компиляции и принципом **РОРИ** (разделение, определение, реализация и использование).

М-2, **Ада**, **Дельфи**, **Паскаль** - определение и реализация вынесены в разные модули. В Аде части пакета могут быть помещены в разные файлы. Пусть в М-2 один модуль импортирует другой сервисный модуль, тогда компилятору для компиляции исходного модуля необходимо знать только определение модуля сервиса. В Аде если изменить спецификацию одного пакета, то все пакеты, использующие данный пакет, должны быть перетранслированы. В Дельфи (например, uses Service): что нужно сделать, чтобы вызвать перекомпиляцию модуля сервиса? В **Си** файлы .h это определение, а .cpp - интерфейс. Если хедер меняется, то make перетранслирует весь проект.

Иначе говоря, все языки 70-х годов поддерживают разделения объявлений и реализации.

Скрытый тип только моделирует абстрактный тип, но не является им полностью. В М-2 есть две концепции: полный и ограниченный (полностью моделирует абстрактный) тип данных.

Ада:

```
package Stacks is
   type Stack is private;
   procedure Push (s: inout Stack; x: integer);
   procedure Pop (s: inout Stack; x: out integer);
private //в ней нужно расписать реализацию абстрактного типа
   type Stack in record
         top: integer := 0;
         foo: array (0..N) of integer;
   end record;
end Stacks;
```

Теперь можно написать Pop(s,x), но нельзя написать S.top := 1; поэтому тип является действительно приватным.

```
type Stack is access;
type Link is record
   next: Stack;
   x: integer;
end record;
type Stack is access Link;
```

Операции сравнения и присваивания для составных типов определяются рекурсивно, так вот в Аде эти операции применимы для типов с одинаковой длиной. Копирование может быть поверхностным и глубоким (разница заметна, если в записи есть ссылки).

```
type T is limited private; //ограниченный приватный тип данных
```

Оберон - 1988:

```
DEFINITION Stacks;
   TYPE Stack =
         RECORD
               ...
         END;
MODULE Stacks
   TYPE Stack*=
         RECORD
               ...
         END;
```

Это тоже не совсем абстрактный тип данных, но с точки зрения гибкости этот тип даже превышает аналогичный тип из Ада. Вирт отказался от принципа РОРИ. Сейчас в **Java** и **C#** есть средства, вытаскивающие из классов интерфейс.

Глава 6. Определение новых типов данных с помощью классов.

Класс - языковая конструкция, которая служит для объединения структуры данных и операций, ее обрабатывающих, в один набор. Основная роль класса - тип данных. Класс похож на модуль. Стандартные классы маскируются подклассы ввиду удобства классов (например, Object, string).

Collection - стандартный класс.

Рассмотрим класс **C++** Matrix.

Операция A+B отличается от операции A+=B. В первом варианте возникает неоднозначность в случае использования класса: какому классу принадлежит данная операция? A или B?

Какому классу принадлежат функции exp и sin? В **C#** это класс Math, но он объявлен как закрытый и наследовать его нельзя, боле того нельзя порождать объекты этого класса. Таким образом, это не класс, это модуль. Создатели C# придумали для него название "**статический класс**".

I) Члены классов

C++:

```
class name : наследование
{
   список членов класса;
}
```

Если нет наследования, то класс может быть верхом в иерархии.

Delphi:

```
type name = class (наследование)
   список членов класса
end;
```

Что насчет принципа **РОРИ**? Во время создания **Delphi** он был аксиомой.
В языке **C#** есть понятие частичного класса (partial class).

C++:

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

Delphi:

```
type T = class
   procedure P;
end;
procedure T.P;
begin
   ...
end;
```

Члены классов:

- члены-данные
- члены-функции
- члены-типы

Члены-данные (и члены-функции):

- статические
- динамические

Сначала обсудим **члены-данные**. Нестатические члены-данные - то же самое, что и поля в записи. Они даже располагаются как поля. Некоторые реализации даже точно определяют порядок в памяти. В **Дельфи** все члены нестатические. **Статические члены** - фактически глобальные переменные (C#, Java).

Обращение:

```
C++:         X::i;
Java, C#:    X.i;
```

[lect23] **Члены класса**

static – модификатор для статических членов.

Члены-данные.

Обращение к статическому члену `(C++): Class?name;`
В других языках вместо `? – точка.`
Обращение к нестатическим членам `(C++): ObjectName.name;`

(В C++ также можно обращаться и к статическим членам, что не очень хорошо с точки зрения технологии программирования)

Распределение памяти.
Для нестатических членов-данных – аналогично полям структур.

Статические члены - в C# и Java –автоматически.
В C++ нужно указывать, в каком конкретно модуле нужно отводить память под статический член-данных.

Статические и нестатические члены-функции.

Нестатическая член-функция – особая функция, отличающаяся от обычной в следующем:

- Область действия – класс, в который она входит (в частности, она видна только из класса).
- Неявный параметр – адрес объекта (C++, C#, Java – this (указатель), Delphi, Smalltalk, … – Self (ссылка)).

Вызов: `ObjectName.funcName(params)`

Возникает две области действия – класс и блок-тело функции-члена. Вторая область видимости вложена в первую. Куда относятся формальные параметры метода? В **Дельфи** – они принадлежат области видимости класса. В других языках (C++, …) они относятся к области видимости функции-члена.

```
class X
{
   int i;
   X(int i) { this>i=i; } //здесь можно без this, т.к. класс – область видимости.
                          Это укорачивает код
}
```

Конфликта имен лучше избегать.

Статические функции-члены:

- Область действия – статические члены класса.

В языке C++ статический член с точки зрения указателей похож на глобальную переменную (функцию)

```
void (*pf)();
void f();
class X {
public:
int i, j;
void f(); void g();
static void f2();
};
pf=f; //можно
pf=f2; //тоже можно
```

С указателями на нестатические члены все подругому, т.к. им нужно хранить дополнительную информацию (является ли функция виртуальной, …).

```
int x?* p;
p=&X?i;//можно
p=&X?j;//можно
void (X?* pfX)();
pfX=&X?f; //можно
pfX=&X?g; //можно
X anX, *pX;
anX.*p;//обращение
anX.*pfX(); //обращение
pX>*p;
.*, >* – новые операции.
```

Замечание о константности.

```
class X {
const int ii; //нельзя инициализировать в определении
static const int i;
};
```

ii сохраняет одно и то же значение на протяжении всей жизни объекта.

В C#:

```
const int i = expr;//это говорит о том, что expr должно быть вычислимо на этапе
трансляции.
```

Так определенные константы всегда статические.

```
readonly a = expr;// может быть инициализирован в любое время, но после этого не
может быть изменен
```

В java такого отличия от C++ нет, но есть модификатор final, означающий неизменяемость.

Вложенные типы (члены-типы).

C++:

```
class X {
public:
   class Y {};
};
```

Y – вложенный класс. Он – то же самое, что глобальный класс, но его имя локализовано в классе X и он имеет полный доступ к членам класса X.

Пример: в STL любой контейнер имеет вложенный тип iterator (напр.).

В языках Java и C# есть понятие статического вложенного класса.

Вложенные классы в Java.

Статический вложенный класс – то же самое, что вложенный класс в C++ (аналогичен статической функции). Нестатический вложенный класс – класс, который очень глубоко связан с объемлющим классом. Любой вложенный класс имеет ссылку на объект объемлющего класса. Обратное, в общем случае, неверно.

```
class BankAccount {
int money;
class Action {
int k;
void f () { ... money=k; }// можем обращаться к членам объемлющего класса
void deposit(sum)
{
   this.new Action//вместо этого можно просто new Action()
}
};
```

Создание объекта вложенного класса возможно только в контексте указателя на объемлющий класс.

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

Т.о. объект a имеет как бы 2 this.

С использованием такого механизма и сборки мусора можно программировать сложные структуры. Есть анонимные внутренние классы – внутренние классы блока, могут ссылаться на локальные объекты. В **C#** тоже есть понятие нестатического вложенного класса. Нестатический вложенный класс C# аналогичен статическому вложенному классу Java (и вложенный класс C++).

```
static class X { .. }; – может содержать только статические члены-данные и
статические члены-функции.
```

Зачем нужно такое понятие? Для классов, которым не нужны нестатические члены.

```
class Math {
...
};
Math m = new Math(); // бессмысленно, следовательно нужно использовать static
static class Math { .. }; // ведет себя как модуль – содержит набор глобальных
                          функций и констант, локализованных внутри класса.
```

Нельзя порождать объекты такого класса, нельзя его наследовать.
Замечание о **Delphi**. В нем есть понятие класса, членов-данных и членов-функций. Вложенных типов и статических членов нет. Они не нужны, т.к. Delphi имеет одновременно и модульную, и классовую структуру.

```
unit M;
interface
type X=class;
   x:integer;
   procedure count;
end;
```

Здесь глобальные функции, описанные в unit M с точки зрения доступа - аналог статический членов класса, таким образом, нужды в статических членов нет.

Инкапсуляция.

Для классовых ЯП она заключается в ограничении доступа (в модульных языках (Ада, Оберон, Модула2) – ограничение видимости, а не доступа). Самая простая модель – в **C++**. Есть три уровня доступа – **public**, **private**, **protected**. Рассматриваются три категории функции: функции-члены класса, функции-члены производного класса, остальные функции.

- public – любая функция имеет доступ
- protected – только функции-члены класса и производных классов
- private – только функции-члены класса

Проблема – как переопределить оператор +? Как функцию-член или как глобальную функцию?

```
T T?operator +(const T& t);
T operator +(const T& t1, const T& t2);
```

С точки зрения семантики вторая форма более естественна, с точки зрения доступа – первая. Есть методики, позволяющие реализовывать операцию + через операцию +=, без доступа к членам класса. Однако, иногда нужно реализовывать глобальные функции, тесно связанные с реализацией класса. Поэтому в **C++** введен механизм друзей. Функция-друг обладает такими же правами доступа, как функция-член. Отношение дружбы не наследуется, нельзя набиться в друзья. Это отношение не транзитивно, не симметрично.

В других языках отношения защиты более сложные.

Откуда берутся **друзья**? Обычно, они находятся в рамках одного проекта.
В **C#** есть понятие Assembly – сборка (надязыковое понятие). В **Java** – Package. Т.о. дополнительно вводится еще одна категория функций – функции-члены из того же пакета (сборки). Таким образом, есть 4 категории : функции-члены класса(1), функции-члены производного класса(2), функции-члены из того же пакета (сборки)(3), остальные функции(4).

[lect24]

II) Инкапсуляция в классах

Три типа: **public**, **protected**, **private**. Они задаются жестко, поэтому понадобились friend. По умолчанию в структуре private – для совместимости со старыми структурами в стиле Си, а в классе – public.

C++	свой	наследник	мир
public	+	+	+
protected	+	+	-
private	-	-	-

В **Java** используется пакетный доступ (package, контекст), в **C#** - internal (namespace, assembly).

В таблице * означает, что имеются различия между C# и Java, зависимость в своей сборке мы или нет.

C#/Java	свой	наследник	взаимосвяз.	мир
public	+	+	+	+
protected	+	+	+*	-
internal	+	+*	+	-
private	+	-	-	-

В **Delphi** есть public, private, по умолчанию, protected. Все в привычном понимании, а умолчание - это фактически совпадение с internal (пакетным типом), т.е. по отношению к своему модулю они ведут себя как public, к другому – как private.

Абстрактный тип данных (АТД) – в первом приближении (с точки зрения класса) – это класс, у которого публичными являются только процедуры, функции или, может быть, другие классы.

Рефакторинг – изменение кода без изменения функциональности, но с увеличением качества (например, введение getters/setters).

Заметьте – абстрактный класс **не** является абстрактным типом данных, хотя и является типом данных и абстрактен как класс. Эти понятия ортогонально и не связаны. Наивный смысл protected – если член класса помечен как protected, то к нему есть доступ из любого наследника. Такой смысл реализован в Delphi.

Пример на **C++**:

```
class X {
  private:
    int i;
  public:
    void f() { i = 0; }        //ясное дело работает
    void g(X &x) { x.i = 0; }  //и это тоже работает!
};
class Y: public X {
  public:
    void f() { i = 0; }
    void g(X &x) { x.i = 0; }  //не работает, если i – protected, можно если Y &x
};
```

Ни в одном языке (которые мы изучаем) нет других отношений между классами кроме наследования. Т.е. «братья» и «сестры» являются чужаками. Для соблюдения инкапсуляции в примере нет доступа в наивном смысле, он есть только в Delphi (непонятно почему).

III) Создание и уничтожение объектов (классов)

Всегда у каждого класса есть хотя бы один конструктор, который вызывается при создании классов. В **Delphi**, **C#**, **Java** – референциальная модель (объекты только в динамической памяти).

В **C++** же объекты бывают трех типов:

- статические
- квазистатические
- динамические

В **Delphi** конструкторы выделяются именем constructor, они наследуются и называются обычно Create или Load (для persistent objects – загружаемых объектов). Тут нет таксономии конструкторов (т.к. в любом классе есть конструктор – они наследуются), а во всех других языках – есть (в них нужен конструктор по умолчанию, он есть всегда).

В **C++** инициализация подобъектов происходит в конструкторах объектов (по-другому и быть не может). Если она не задана, то код будет сгенерирован автоматически. Сначала будет вызван конструктор базового класса, потом подобъекта, а лишь потом объекта. В **C#** для подобъектов этого не произойдет, т.к. подобъект – лишь ссылка, и она будет обнулена. Это не страшно, т.к. пока нет подобъекта как такового, а есть лишь ссылка. Можно написать так:

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

Список инициализации – есть в C++, в классах идет перед телом, необязателен для указания, но присутствует всегда (генерируется автоматически). Конструктор по умолчанию так называется, потому что вызывается для базового класса и подобъектов автоматически. Пример такого списка:

```
X(0), Base(1), i(0) //последнее эквивалентно i = 0 с точностью до порядка вызовов
```

В **C#** и **Java** конструкторы подобъектов автоматически **не** вызываются, их надо указывать явно. Синтаксис в этих языках отличается, но смысл тот же (для списков инициализации). В **C++** сначала вызывается конструкторы подобъектов, а лишь затем базовых классов. Поэтому в конструкторе нельзя вызывать виртуальные методы – класса как такового еще не сконструировано и нет таблицы виртуальных методов. Т.е. в C# и C++ разные порядки вызовов конструкторов и инициализаторов. **C++**: Base(), BaseObj(), DerivedObj(), Derived(); **C#**: DerivedObj(), BaseObj(), Base(), Derived().

[lect25] **О создании и уничтожении объектов**

Delphi:

- конструкторы и деструкторы наследуются
- всегда вызываются явно

```
type X = class
constructor load; // называться могут как угодно
destructor destroy;
```

По умолчанию ничего не генерируется, потому что все наследуется (даже неявно из класса TObject, в котором есть методы Create и Destroy). Всегда перед созданием и удалением создается таблица виртуальных методов, поэтому в отличие от си, можно вызывать в конструкторе-деструкторе виртуальные методы.

inherited (соответствует super в Java и base в C# -- это ссылки), но inherited - не ссылка, а ключевое слово. **inherited create**; - наследует конструктор из предка.

По умолчанию конструктор забивает объект нулями, поэтому вызывать его надо первым. В Дельфах нет автоматического сборщика мусора, как C# и Java. Еще в одном смысле Delphi "застряли посередине", например можно работать с интерфейсами (например, IXML Document), в которых автоматическая сборка мусора есть.

Теперь о **синтаксисе**:

1) Создание

```
var a:X; // еще не созданный объект класса X
a:=X.create(1); // конструктор всегда вызывается явно
```

2) Уничтожение

   В TObject есть метод Free, поэтому удалить можно командой a.Free, но это не деструктор, хотя

внутри Free действительно вызывается деструктор. Поэтому деструктор не вызывается явно. Причем, при удалении ссылка (self) не становится == null, потому что ссылки определяются как константы (нельзя писать self = ...).

3) Инициализация статических объектов

В Си мы не можем сказать, в каком порядке и в какой момент будут инициализированы статические переменные (единственное, что можно сказать, что все их конструкторы будут до main, а деструкторы -- после), поэтому в конструкторах нельзя рассчитывать на какой-либо порядок инициализации.

В **C#** и **Java** ситуация лучше, потому что инициализация вызывается строго (C#: static Y a = new Y();) и порядок вызова определяется порядком записи.

Java:

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

Часто статические инициализации выводятся даже в другой поток, поэтому если вдруг там возникнет исключение, то непонятно в какой момент времени оно придет в главный поток.

В **Delphi**:

Статических объектов вообще нет. Но в общем случае unit в Delphi может иметь кроме *interface* части, еще и *implementation* часть, которая может быть поставлена в соответствие статическим переменным. Деструкторов нет, но им может соответствовать блок *finalization* в части *implementation* (перед которым есть блок *initialization*).

C++:

В нем абсолютно точно известно, когда будет вызван конструктор и деструктор (выход из зоны действия, либо вызов delete). И еще насчет понятия "свертка стека": для любого локального объекта, захваченного в конструкторе, C++ гарантирует вызов соответствующего деструктора. Поэтому утечка ресурсов может возникать только при работе с динамической памятью.

C# & **Java**:

Есть сборка мусора.

Но универсальных алгоритмов сборки нет. Один из популярных: контроль за количеством ссылок. Но такой алгоритм не подходит для кольцевых списков. Поэтому используется более продвинутая версия алгоритма (выбирается объект, про который известно, что он точно живой -> находятся ссылки из него на другие объекты и эти объекты тоже помечаются живыми и т. д.). Главная проблема этого алгоритма - неизвестность момента начала и если много объектов, то алгоритм занимает много времени. А бонус этого алгоритма - параллельная дефрагментация памяти. К тому же ссылка в процессе сборки мусора может измениться, поэтому мы не можем отождествить в этих языках ссылку с указателем.

Сборка мусора часто не выполняется вообще, пока памяти продолжает хватать.

Пример на **C#**:

```
Image im = Image.FromFile(filename);
<do smth>
im.SaveToFile(filename);
```

Сохранить в тот же файл не удается, потому что он захвачен функцией FromFile. Это логично, потому что если файл большой, то читать его целиком в память нецелесообразно. Очевидный выбор: освободить ресурс im и сохранить копию в тот же файл:

```
Image im = Image.FromFile(filename);
<do smth>
Image im1 = im;
im = null;
im1.SaveToFile(filename);
```

Но в этом случае все равно не будет гарантии, что этот код заработает, потому что хоть мы и присвоили im = null, но мы не можем сказать, что после этого начнется очистка ресурсов и, следовательно, закроется файл. И это лишь одна из многих проблем.

Еще о минусах сборки:

Сейчас запускается низкоприоритетный процесс сборки мусора, потому что в интерфейсных программах есть куча времени, пока пользователь тупит, а сборщик тем временем тихо собирает мусор.

В **Java** должен быть *финализатор*, потому что он - последняя надежда.

```
Java: protected void finalize()
{
<свой деструктор>
if (!close) Close(); // их деструктор
}
```

В **C#** еще хуже, потому что даже есть понятие деструктора, который на самом деле не работает так, как от него ожидают. На самом деле внутри любого деструктора вызывается просто финализатор класса Object.

Еще метод для **Java**:

```
c = new X();
try
{
   <...>
   return;
}
finally
{
   <этот блок будет гарантированно выполнен в любом случае (будет исключение или нет)>
   c.Close();
}
```

В **Delphi** то же самое, но там между *try* и *finally* нельзя писать catch. То есть либо *try-catch*, либо *try-finally*.

В **C#** ввели специальный интерфейс *IDisposeable* (утилизируемые) объекты. Метод *Dispose();* В примере с картинками надо написать не `im = null`, а `im.Dispose();`

А еще можно так:

```
using (Image im = Image.FromFile(..))
{
   <smth>
}
```

На выходе из *using* вызывается *Dispose* для всех объектов, бывших внутри. Кусок кода полностью эквивалентен этому:

```
Image im = Image.FromFile(..);
try
{
   <smth>
}
finally
{
   im.Dispose();
}
```

Никакой из традиционных языков (**Ада**, **М-2** и т. д.) эти проблемы (создания-уничтожения) не решают, они их игнорируют и предоставляют программисту.

P. S. В **Java** есть термин "слабая ссылка" (*WeakReference*), имеющая непосредственное отношение к сборщику мусора. Содержит метод *Object GetReference*(); **Слабая ссылка** - с точки зрения программиста эта память уже освобождена, но финализатор еще не применен. Теоретически, ресурсы еще можно восстановить методом *GetReference*(), превращая слабую ссылку в живую.

Пример: кэш браузера. Переходим на следующую страницу, очевидно, что ресурсы старой страницы нужно освободить. Мы обнуляем ссылки, и сборщик переводит их в разряд слабых. А если нажимается кнопка Back, можно еще посмотреть очередь слабых ссылок, прежде чем лезть снова запрашивать объекты.

IV) Дополнительные проблемы, связанные с классами

- копирование, сравнение
- неявные преобразования

Есть понятие "**глубокая**" (deep) и "**поверхностная**" (shallow) копия.

В **C++** по умолчанию определяется побитовая копия (поверхностная), а если ее не хватает, то нужно определить конструктор копирования. Сравнение тоже может быть определено по-разному. Например,

сравнивать ссылки или содержимое по ссылкам.

Delphi игнорирует проблему копирования.

В **C#** и **Java** есть метод Clone() интерфейса ICloneable, который можно переопределить для глубокого копирования, но лучше так не делать, потому что не факт, что реализованная копия будет глубокой. В классе Object есть метод MemberWiseClone() - делает поверхностную копию.

[lect26]

В **Java** и **C#** есть специальные средства – **рефлексии**, для того, чтобы узнать поддерживается или нет определенный интерфейс.

В языке **C++** есть три стратегии клонирования:

- глубокое копирование
- побитовое копирование
- запрет копирования вообще

А вот в **Java** – 4 стратегии клонирования:

- Разрешение клонирования - класс сообщает, что он клонирован и в нем обязательно определен *публичный* метод клонирования

```
class X implement IClonable {
  public Object Clone() throw() {};
```

  Если вызывается недопустимая конструкция клонирование, то кидается специальное исключение

- Запрещение клонирования – также кидается исключение
- Условная поддержка клонирования – явным образом реализуется интерфейс *IClonable*, но вот подобъекты могут не клонироваться – тогда исключение
- Неявная поддержка клонирование – не реализуется интерфейс, но делается побитовая копия:

```
Class X { protected Object clone() {return …} }
```

Также существует **проблема сравнений** по операциям равно (==) и неравно (!=). **C#** и **Java** у каждого класса есть функция **E**quals (ну или **e**quals), они возвращают bool. По умолчанию сравниваются ссылки (this == 0). Плюс к этому существует статическая функция от двух параметров, на случай сравнения null с null (чтобы не было разыменования null). Существует интерфейс IComparable, не будем на нем подробно останавливаться.

[lect27]

IV) Перегрузка операций. Неявные преобразования (частный случай перегрузки)

В языке Оберон есть понятие "расширяющие преобразования"

```
int i;
double d;
i=d; // идет преобразование с потерей информации. ~i=(int)d
```

Неявные преобразования есть почти во всех языках, за исключением языка Ада. Архитектор Ада считал, что неявные преобразования это всегда плохо. Как правило, все современные языки разрешают неявное преобразование между простыми типами.

Вопрос: разрешать ли пользователю неявные преобразования? Когда Страуструп проектировал свой C++, в то время была идея на запрет неявных преобразований. Но эта идея оказалась непрактичной, и все же были разрешены пользовательские неявные преобразования.

C++, **C#** - допускают пользовательские неявные преобразования и можно перегружать стандартные операции. **Delphi**, **Java** - запрещают. Перегрузка стандартных операций запрещена.

Аргументы против преобразования:

- любое неявное преобразование может вести к потере информации и => ненадежности.
- семантика н.п. может не совпадать с тем, что ожидает программист.

Классы, которые реализовал Страуструп в своем языке:

- Файлы ввода/вывода. Эти классы можно назвать удачными (но, тем не менее, многие до сих пор пользуются функциями типа printf)
- Complex - класс комплексных чисел.

Пример на **фортране**:

```
A=B*CEXP(-k*i)/D // например B вещ., i – мнимая единица,D – комплексное, k – целая. Все работает.
```

Чтобы такое реализовать на **Си**:

```
Complex Plus(Complex c1, Complex c2); //...
Mult(B, Div(CEXP(-Mult(k,i)), i)))// ...
```

На **C++**:

```
Complex operator +(Complex c1, Complex c2);
```

Если бы не было неявного преобразования, пришлось бы писать все эти операторы плюса для double, float, int, short, char, ... Т.е. в этом случае пришлось бы реализовать несколько сотен функций, суть которых фактически сводится к преобразованию между типами и выполнению уже запрограммированной операции.

```
X
X(T) // конструктор преобразования.
```

Теперь достаточно написать Complex(double), потому что к double неявно преобразуется все остальное.

Другой **пример**:

```
string( const char *)
s = "string"; // компилятор вставит s = string("string");
```

Дополнительно в C++ появился оператор преобразования.

```
operator T() // нестатическая функция-член, определяет преобразование X => T
```

Пример:

```
operator const char *() const
```

Но тут же появилось много проблем. Например, проблема с вектором:

```
class Vector {
   int *body;
   int size;
public:
  Vector(int sz) { body = new int[size=sz];}
  Vector() {...}
}
```

Иногда такой конструктор случайно оказывается конструктором преобразования.

```
Vector v(20); // Все ок.
V = 3; // Компилятор вставит V = V(3), т.е. создаст вектор длины 3. Но по сути
нельзя приводить int к Vector.
```

Позже эту проблему решили так: появилось ключевое слово explicit, которое говорило о том, что конструктор можно вызывать только явно. Для совместимости с прошлым кодом работал конструктор преобразования.

В **C#** учли этот опыт и ввели слова explicit и implicit. По умолчанию подразумевается explicit.

Явное преобразование: T ConvertToT() // X=>T

```
T t;
X x;
t = x; // ~ t = x.operator T()
```

Оператор дает возможность преобразования из базового класса в производный класс.

Создатели **Delphi** и **Java** отказались от неявных преобразований вообще. В **C#** любые операции должны быть статическими функциями класса.

```
class X {...}
class Y {...}
```

```
public implicit operator Y(X x){...} // эту функцию можно было сделать членом
класса Y
```

Если перед функцией стоит implicit, то такая операция может вызываться неявно. Если explicit (по умолчанию) - только явно.

Java:

```
S1+S2
string S;
s+i; //~ s+i.toString(); Таким образом, неявные преобразования в Java все же есть,
но они встроены в компилятор.
```

```
System
java.lang // типы данных, которые встроены в язык.
```

В поздних версиях **C++** компилятор также начинает что-то знать о стандартной библиотеке, например при возбуждении исключения он вызовет std::exception

В **C#** нельзя делать перекрытие () и [], потому что у программистов могут быть разные представления о семантике таких операций. Вместо этого в C# есть специальная функция - индексатор.

```
int this[string s] {...} // аргумент в квадратных скобках.
```

Теперь можно делать что-то вроде этого:

```
int <- x["string"]
```

V) Свойства (property)

Свойство при реализации выглядит как член данных, а при реализации как 2 функции - установка значения и взятие значения. (get и set). Понятие свойства отражает дуализм данных и операций.

C#, **Delphi** - поддерживают свойства.
C++, **Java** не поддерживают свойства.

Функции доступа (accessors) - getter, setter.

Delphi:

```
type X = class
  ....
  property p:integer read accessor write accessor;
// где accessor – либо имя переменной члена-класса, либо имя процедуры или функции.
// для read – имя функции без параметров
// для write – имя процедуры с одним значеннием.
```

Пример:

```
  type X = class
   private
     flength:integer;
property length:integer read flength write flength;
```

Более сложный пример:

```
type Figure = class
  private a,b: integer;
              farea: integer;
  private function GetArea():integer; // в этой функции мы можем вычислить площадь для a,b, а дальше просто возвращать
// уже посчитанное значение farea
   property width:integer read b write SetWidth;
   property area:integer read GetArea; // ограничили доступ на запись.
private procedure SetWidth(x:integer); // здесь мы можем вычилсить новую площадь.
```

таким образом, мы можем "кэшировать" значения классов для экономии вычислений.

Свойства также есть и в **C#**:

```
class Rect {
public:
  int Width {
     get{ return _width; }
     set{ _width = value; _area = _wiidth*_height} // value – ключевое слово, обозначающее устанавливаемое значение.
  }
private int _width, int _height, _area;
}
```

В **C# 3** было добавлено понятие автореализуемого свойства.

```
class Point {
  public int x;
  public int y;

  public Point(int x, int y) {
   this.x = x;
   this.y = y;
  }
}
```

```
// можно это написать так:
class Point {
 public int x {get;set;}
 public int y {get;set;} //у этих переменных будет автоматически реализован метод
set и get
}
Point p = new Point{x=0; y=0;};
...
public int x{get; private set;} // доступ к x может быть только внутри блока
инициализации
...
```

Глава 7. раздельная трансляция и разбиение программы на модули.

Виды трансляции.

1) Цельная трансляция. Программа подается компилятору целиком и выполняется. Для серьезных целей такое использовать нельзя.

2) Пошаговая. Есть понятие конструкций языка. Транслятору дается такая констуркция, он обрабатывает ее и по возможности

выполняет. Пример таких языков: Basic, sh.

3) Инкрементная (устарела)

4) Раздельная трансляция.

4.1) Раздельная незваисимая трансляция.

4.2) Раздельная зависимая трансляция.

Здесь появляется понятие компиляции, т.е. перевода с одного языка на другой. Появляется понятие единицы компиляции.

4.1: объектный модлуь <= Транслятор <= единица компиляции

4.2: транслятор <= Трансляционная библиотека.

<= Единица компиляции.

Контекст трансляции - набор внешних имен, необходимых для компиляции. (Например, стандартная библиотека).

Контекст существует всегда. Если его нет, то он берется из самой единицы компиляции.

Фортран, **Ассемблер**, **C/C++** - есть указание контекста трансляции.

Пример:

```
[asm]
 EXTRN X:WORD
[/asm]

[c]
extern ...
[/c]
```

В языках с зависимой трансляцией подключают контексты. Все современные языки имеют зависимую трансляцию.

[lect28]

Раздельная трансляция:

- Зависимая
- Независимая

Возникает понятие единицы компиляции (ЕК, физический модуль).

ЕК > Транслятор > Объектный модуль

Трансляционная библиотека – таблица имен.

Программная библиотека – объектный модуль.

Контекст трансляции – набор объявлений внешних имен, необходимых для трансляции.

При зависимой трансляции транслятор кроме генерации объектного модуля еще добавляет информацию в трансляционную библиотеку. Во многих языках логические модули совпадают с физическими. Например, в Дельфи в файлах .dcu в первой части – таблица имен, в второй – объектный код.

В Обероне, C#, Delphi, Java есть понятие модуля, но нет понятия объектного файла.

Для современных ЯП характерно понятие рефлексии – доступа к коду программы во время выполнения. Например, можно считать или записать поле класса по имени этого поля.

Так как при зависимой трансляции в модулях фактически записывается текст программы, то для защиты авторский прав существуют средства для т.н. обфускации – запутывания кода модуля.

I) Независимая трансляция. C/C++

При независимой трансляции для задания контекста трансляции используется директива extern и определения типов. Необходимо дублировать все внешние имена из одного модуля в другой. Это ведет к трудно устранимым ошибкам. Есть технология для избегания таких ошибок. Каждый модуль разбивается на заголовочную часть (.h, .hpp – хедер-файл) и реализационную часть (.c, .cpp). В заголовке описываются прототипы функций, extern – конструкции и объявления типов. Для использования некоторого имени из контекста трансляции не нужно писать extern-конструкции, а нужно написать #include соответствующего файла, в котором описано это имя.

Проблема в том, что это средство реализовано тривиальными средствами препроцессора. Отсюда проблема : m.h >m1.h >m2.h, m.h>m2.h (повторное включение, дублирование объявлений типов - ошибка). Поэтому, шаблон хедер-файла следующий:

```
#ifndef __M__H__
#define __M__H__
//текст объявлений
#endif
__M__H__ – некоторое уникальное имя
```

Благодаря такой конструкции содержимое хедер-файла может быть включено только один раз.

Так как при использовании больших библиотек некоторые хедеры, имеющие большой объем, включаются директивой #include много раз, то для ускорения трансляции современные компиляторы C++ используют предкомпилированные хедеры.

II) Зависимая трансляция

При зависимой трансляции возникают понятия одностороннего и двустороннего связывания.

Одностороннее связывание.

Одностороннее связывание: ЕК – клиент запрашивает внешнее имя у ЕК – сервера.

В Дельфи, М-2, Обероне ЕК==Лог. модуль.

Дельфи:

```
uses ...; – сразу делает все имена из перечисленных модулей непосредственно видимыми.
M2:
import ... – для имен, импортированных таким образом, нужно всегда писать имя модуля.
from ЕК import … – непосредственный импорт (имена делаются непосредственно видимыми, не нужно писать имя модуля)
```

В Обероне второй конструкции нет.

При таком механизме связывания серверный модуль ничего не знает о клиентском. Кольцевые связи в таком случае – ошибка. Пример в Дельфи:

```
unit m1;
interface
uses m2;

unit m2;
interface
uses m1;
```

Ошибка времени трансляции.

Однако, следующее описание юнита m2 будет уже корректно, т.к. нужно оттранслировать интерфейс m2, затем интерфейс m1, затем implementation m2 и m1 (d в произвольном порядке):

```
unit m2;
```

```
interface
…
implementation
uses m1;
```

Стандартные библиотеки многих языков используют одностороннее связывание. Недостаток – большие списки импорта.

Двусторонняя связь.

Она реализована только в ЯП Ада (односторонняя связь в ней тоже реализована).

При трансляции программы на ЯП Ада программист имеет две опции: либо собрать весь проект в один файл и подать этот файл на трансляцию, либо разбить проект на отдельные ЕК и подавать на трансляцию по кускам, а затем собрать эти куски воедино. В ЯП **Ада** есть понятия первичных ЕК и вторичных ЕК.

Первичные ЕК – те, которые нужны для трансляции других модулей. Пример: спецификация пакета или процедура. Они формируют трансляционную библиотеку.

Вторичные ЕК – нужные сами по себе (тело пакета). Они формируют программную библиотеку.

Односторонняя связь, указание контекста:

```
WITH список_первичных_ЕК;
USE список_2; – может отсутствовать
```

текст ЕК

список 2 – собственное подмножество список_первичных_ЕК. Имена из пакетов из списка 2 становятся видимыми непосредственно (для имен из список_первичных_ЕК\список_2 нужно указывать имя модуля).

```
WITH – аналог import в M2.
WITH Stacks; – подключает пакет Staks к нашему пространству имен.
USE Stacks; – имена из Staks становятся видимыми непосредственно.
```

Двусторонняя связь.
Отличие Ады от других рассматриваемых ЯП – логические модули в этом ЯП могут быть вложеными.

ЕК1:

```
package Outer is ...;
   package Inner is ...;
procedure P(x:t1, y:t2);
...
end Outer;
```

Вложенность тел должна соответствовать вложенности спецификаций.

ЕК2:

```
package body Outer is
...
...
   package body Inner is separate; –separate означает, что определение в другом
месте
procedure P(x:t1, y:t2) is separate; – определение в другом месте
end Outer;
ЕК3: (неправильный вариант)
WITH Outer;
USE Outer;
package body Inner is
end Inner;
```

Этот вариант неправильный, т.к. чтобы откомпилировать ЕК3, нужно загрузить имена из implementation-части Outer, а в этом варианте загружаются только имена из определения Outer. Здесь нужна двусторонняя связь. Двусторонней связью может обладать только вторичная вложенная ЕК.

Правильно ЕК3 нужно писать так:

```
separate(Outer.Inner)
package body Inner is ...
end Inner;
ЕК4:
separate (Inner)
procedure P(...)
...
end P;
```

При изменении вторичных модулей надо перетранслировать только эти вторичные модули. При изменении первичных модулей надо перетранслировать не только эти модули, но и все модули, зависящие от них. Цель механизма раздельной трансляции – минимизировать время перекомпиляции. При независимой трансляции сам программист при сборке должен указывать зависимости. При зависимой трансляции зависимости отслеживает компилятор, определяя их по тексту программы.

III) Управление пространствами имен

Проблемы:

1. Указание контекста трансляции. Как управлять контекстом трансляции?
2. Дистрибуция (распространение). Что является единицей дистрибуции?

(В C/C++ ЕК является файл)

Проще всего в **Java** – там есть понятие пакета. Там ЕК является тоже файл, но он должен содержать

```
package имя_пакета;
```

Любой класс, который может использоваться вне пакета, должен быть помечет public.

Пример: любой пакет, транслирующийся в исполняемую программу, должен иметь публичный класс со статической функцией main определенной сигнатуры.

```
package mine;
public class Entry
{
public static int main(String Args[])
{
}
}
```

Пакет собирает ЕК воедино. Он служит для указания контекста трансляции и как единица дистрибуции. Имена пакетов иерархические.

```
package Proj1.Root.com.Foo;
```

Изначально идея была в том, чтобы имя пакета отражало его положение на некотором сервере.

```
import имя_пакета;
import имя_пакета.*; – импорт всех имен из пакета в наше пространство имен
имя_пакета.имя класса – для того, чтобы так писать, не нужно предложения import.
Proj1.Proj11.Proj111
Proj1.Proj12
```

Это не имеет отношения к зависимости пакетов друг от друга, а служит только для указания места в иерархической файловой системе. В **C++** и **C#** есть понятие пространства имен, которое уже подразумевает именно зависимость вложенных пространств имен от внешних.

[lect29]

Пространство имен C#, C++

Единица компиляции – файл
Единица контекста – пространство имен
Единица дистрибьюции – сборка

Часть 3. ООП – языки

Глава 1. Наследование

Base => Derived (базовый и производный классы), можно также использовать терминологию **Smalltalk**: Superclass => Subclass.

C#, Java, Delphi – Object (TObject) – есть некий базовый класс для всех.
C++, Оберон, Ада95 – нет общего класса.

C++:

```
Class Derived: public Base { //множественное наследование
   ...
};
```

C#:

```
Class Derived: Base ,Base1 … { //только один из base может быть классом
   ...
}
```

Smalltalk:

```
Class Subclass extends Superclass implements I1, I2 … {}
//здесь тоже только один суперкласс и множество интерфейсов
```

Delphi:

```
type Derived = class(Base) begin ... end;
```

Оберон:

```
TYPE BASE = RECORD ... END;
TYPE DERIVED = RECORD(BASE) ... END;
```

Ада – *тегированные ТД*:

```
type Base is tagged record ... end;
type Derived is new Base with record ... end; //or with null record
```

Все языки позволяют линейное распределение памяти – дань эффективности реализации. В языке Smalltalk ситуация другая. Subclass отдельно от Superclass. В Subclass есть некое поле, содержащее ссылку на объект суперкласса. При этом поиск переменных или методов производится рекурсивно динамически. Есть возможность динамически добавлять переменные или методы. Проигрываем скорость, получаем гибкость. Поэтому **Smalltalk** неэффективен как язык индустриального программирования.

Все функции существуют в одном экземпляре, с этой точки зрения накладных расходов(расширения памяти) не происходит.

При наследовании каждый класс – это своя область действия.

```
Class X{
   Int f;
};

Class Y: public X{
   Новые члены – своя область действия
   Void f();
};
Y y;
y.f – Вопрос: что это!? Ответ: Из класса Y!(скрытие)
y.f() – недопустимо
```

Область действия Y вложена в область действия X. Т.о. допустимы совпадающие имена. С точки зрения старых областей действия существуют понятия скрытие(новое имя) и перегрузка(только имена функций и процедур исключение - Ада – литералы перечисления). Т.е. есть разница между именами функций и именами остальных объектов.

```
Class X{
   void f(int); – хоть виртуал хоть не виртуал – все равно скрытие.
};
Class Y: public X{
   Новые члены – своя область действия
   Void f();
};
```

y.f(); - скрытие или перегрузка? Скрытие! Т.е. y.f(1); допустимо, а y.f(); нет
чтобы обратиться к ф-ии из X y.X::f(1); или this->X::f(1)

Разные области действия – перегрузки не может быть. Есть еще переопределение – только для функций, которые обладают динамическим связыванием - если ф-я объявлена в базовом классе как виртуальная и имеет тот же прототип

Модульные ЯП.

Ада95, Оберон:

```
MODULE M;
   TYPE BASE* = RECORD
         I: INREGER;
         J: REAL
   END;
ENDM.
```

Допустим, мы хотим сделать так:

```
MODULE M1;
   IMPORT M;
   TYPE DERIVED* =
         RECORD(BASE)
               K: INTEGER;
         END;
   PROCEDURE P(VAR X: DERIVED);
```

Тут появляется понятие дочерних пакетов. Дочерние пакеты в Ада:

```
package M.M1 is ... //пакет m1 – расширяет m, он как-бы дописывается в него
```

В этом случае у нас есть доступ даже к private-членам базового пакета, так как это дочерний пакет. По настоящему защищенных методов в модульных языках **нет**.

Объект базового класса всегда является объектом производного класса, но не наоборот.

Множественное наследование влечет такую проблему:

```
Base *pb = new Base;
Derived *pd = new Derived;
pb = pd;
class X: public Y, public Z {
   x py;
   x *px = new X;
   py = px;
   pz = px;
};
```

В общем случае, множественное наследование влечет некоторые проблемы, поэтому его не используют во многих ЯП.

[lect30]

Досрочный экзамен – 20.12 – П8а, первая пара
Экзамен – 15.01 (с 12 часов)

Множественное наследование

Только C++ допускает полное и неограниченное наследование. Ограниченное наследование (один суперкласс, но есть много интерфейсов) допускают все современные языки. Если допускается реальное множественное наследование, то допускаются разнообразные схемы, например «бриллиантовое» наследование или наследование от разных экземпляров одного класса. Тут возникают некоторые проблемы. Так как поиск неразрешенных имен производится параллельно в обоих предках, поэтому мы должны обязательно уточнять, из какого предка используется члены. Где может понадобиться такое наследование? Например, в обобщенных списках. В результате в этих контейнерах можно хранить произвольные ТД. Конечно, позже появились контейнеры-шаблоны, которые более эффективны. Такие сложные схемы наследования изучает специальный раздел математики – теория решеток (или теория категорий). А как можно использовать «бриллиантовое» наследования? Например, в библиотеке **<iostream.h>**. Есть базовый класс **ios** – это обертка над файловыми дескрипторами, из него наследуются классы **istream** и **ostream**, а уже из них наследуется **iostream**. Как нужно реализовать такое наследование? Использовать виртуальное наследование:

```
class A { ... };
class X: public virtual A { ... };
class Y: public virtual A { ... };
class W: public X, public Y { ... };
```

Идеология наследования такова, что когда мы проектируем структуру классов, мы заранее знаем, что и от кого будет наследоваться, и пишем классы именно такими. Хотя сперва, кажется, что писать виртуальное наследование класса, не зная, кто, будет наследовать его нельзя. Если перед классом написано final (в **Java**), то он «финальный», «запечатанный» и от него нельзя наследоваться. Такие классы имеют большую производительность. Есть еще причины использования их вместо наследуемых.

Глава 2. Динамическое связывание методов

Использование субкласса вместо суперкласса не должно вызывать потери надежности. Ничего не должно меняться при передаче указателя на субкласс в параметрах вместо указателя на суперкласс. Но все меняется, когда мы передаем параметры по ссылке.

В **C#** мы можем писать так:

```
var x = new System.Windows.Forms.Form(...);
```

По идее, компилятор всегда сам может определить тип выражения. Динамический тип указатели или ссылки всегда меньше или равен статическому типу. Объект данных – это инициализированный кусок памяти. Если уж объект base размещен в памяти, то он не может вдруг расшириться до derived. Тип этого куска памяти мы уже не изменим. А вот ссылки или указатели – можем. Они в свою очередь указывают на вполне определенный тип.

Нестатический метод статически привязан. Он всегда вызывается через ссылку. Его вызов осуществляется исходя из статической ссылки. Динамический метод – из динамической. Ковариантность – если тип X < Y, он может совпадать с Y или быть наследованным (или приводимым) от него.

```
class Y {
   public void f() {...}
}
Class X extends Y {
   public int f() {...}
}
```

В языке Java все методы связаны динамически. В C#, Delphi – существует специальный модификатор **virtual**, то метод становится динамически связанным. Это свойство вызова, но мы привязываем его к методу. Пример:

```
//наследование такое: A => B => C => D => ...
B *pb;
pb->f(); //компилятор ищет функцию в B с таким профилем, потом идет в объемлющий уровень, пока не найдет нужный метод
void B::g() { //в этом случае применяются те же самые правила
   f():
}
```

Если f – статический метод, то ищется функция до загрузки программы в память. Если же динамический метод - то в момент выполнения, ищем ковариантные переопределения (именно в момент выполнения). Если переопределена приватная функция, то она вызвана не будет. Компилятор **Java** даже не видит приватные функции, **C++** видит, но предупреждает, что доступа нет.

```
class A {
   private:
         virtual void f() {...}
};
class B: public A {
   private:
         virtual void f() {...} //переопределили, но не можем сделать ее public
}
```

Здесь мы даем базовому классу вызывать нашу приватную функцию. Другие классы по-прежнему не могут вызывать эту функцию.

В ряде языков (в том числе C++) есть возможность **снятия** механизма виртуальности вызова. Например, когда нам не надо рекурсивно зацикливаться. Снятие происходит путем уточнения имени класса.

```
Void PatternRect::Draw()
{
   Fill(pattern, ...);
   Rect::Draw();            //сняли виртуальности
}
```

В отличие от override, overload означает, что надо либо целиком замещать методы, либо использовать их без изменения.