# Языки программирования.

**Лектор:** Игорь Геннадьевич Головин

**Выполнили:** Ульянов Алексей, 324 гр.

Лихогруд Николай, 320 гр.

Сергеев Николай, 328 гр.

**Лекция. Ульянов А.В.**

**Определение языка программирования (далее ЯП).**

Существуют два типа определений:

1) эксенсиональное (через объем). Идет перечисление ЯП – C++, Pascal, Фортран, …
2) интенциональное. Определяет понятия по их родовым признакам.

Одно из определений ЯП:

ЯП – это инструмент планирования поведения исполнителя.

Например:

PLANNER – необходим для обмена информации некоторых объектов, планирование поведения. (например поведение людей). Реализовать невозможно! Алгоритмически неразрешен.

APL – компактный язык для записи математических алгоритмов. Один из первых интерактивных ЯП. (в 1964)

Минус – сложность в «читабельности» и понимании.

Основа современных языков - «читабельность».

HTML – есть исполнитель (браузер). Есть поведение, можно планировать. Но не является ЯП, т.к. нет алгоритмической полноты.

Планирование. Что необходимо:

1) прогнозировать
2) контролировать

В дальнейшем под исполнителем будем понимать ВС.

**Основные позиции рассмотрения ЯП.**

**Виды программирования:**

1) Игровое (тянет компьютеры вперед)
2) Научное
3) Индустриальное

Игровое: программирование для себя (как развлечение).

Основное требование к игровому ЯП – легкость в изучении и применении (в основном интерпретируемые).

1964. Basic – был классикой игрового ЯП.

Научное: для научных вычислений и т.п. (В основном на Фортране)

Требование – легкость применения. Фортран в основном используют математики.

Индустриальное: Программирование не для себя. Программа отчуждается, т.е. идет создание программного продукта.

Требования:

1) Эффективность
2) Читаемость
3) модифицируемость

**Кризис в программировании:**

Впервые это понятие появилось при написании ОС для IBM/360.

1961 – начало написания ОС.

1964 – вышла на рынок.

Спрашивается, что же делали разработчики три года?

Оказывается, что все основные задержки были связаны со сложностью программирования – несколько миллионов строк кода. Выяснилось, что один человек такое создать не может. (Книга автора ОС – «Мифический человекописец».)

Срывались сроки и качество продукта. Росла стоимость проекта.

**Распределение расходов:**

Сопровождение – 50% (багги, изменение требований заказчиков/пользователей)

Тестирование – 25%

Кодирование – 10-12%

Остальное – это ТЗ, ТУ и т.п.

**Основные позиции:**

1) Технологическая (как?)
2) Авторская (почему?)
3) Реализаторская
4) Семиотическая
5) Социальная

**Технологическая.**

Необходима возможность командного создания.

ТП -> потребности ЯП.

Паскаль хорош для обучения, читабельный, но менее пригоден для индустриального программирования, чем например С, С++. (хотя бы по скорости написания)

**Авторская.**

Любой язык программирования, по сути – совокупность компромиссов.

Для создания ЯП можно сделать так: взять популярные языки и всунуть что-то дополнительно. (Object Pascal, Object C, C++)

Это не самый лучший подход.

**Реализаторская.**

С точки зрения программиста. Не должна быть главной при разработке приложений.

**Семиотическая.**

Семиотика – наука о знаковых системах.

Любой язык может быть выражен в знаковой системе.

**Социальная.**

Язык нужно не только спроектировать и создать, им должны пользоваться.

Рассмотрим две системы программирования: VB и Delphi. Delphi по многим параметрам лучше VB, но уступает в социальном плане, т.к менее разрекламирован.

**Исторический очерк развития ЯП:**

**1954** – Эмбриональный период.

**1954 – 57** – Фортран (Д. Бекус). Проект добился всех целей, которые были поставлены.

В основном рассчитан для решения уравнений математической физики. Раньше математики рисовали блок схемы, затем подходили к программисту, который переводил программу в двоичный код, переносил на перфокарту, и только потом запускал ее на машине. И вот если происходила ошибка, приходилось все повторять сначала, но уже конечно с исправленной (не всегда верно) программой. Джон Бекус предложил тогда создать программу FT. Программа работала на трехадресной машине. Появился условный оператор if (*) M1,M2,M3 – соответственно переход по больше, меньше, равно нулю.

Три качества программистов, которые отличали, да и отличают программистов от других профессий:

1) Лень
2) Нетерпение
3) Высокомерность

От того с ними было сложно работать. Изобретение данного языка позволило математикам самим писать свои вычислительные программы. Но изначально хотели эту программу «прогнать», т.к. в то время основным ресурсом было время работы компьютера. А работа с Фортраном заняла бы немало этого времени.

+ Эффект мобильности знаний (при переходе на другие компьютеры не нужно переучиваться)

**1958 – 1960** – Algol 58, Algol 60.

Язык для обмена алгоритмами. Синтаксис и структура были описаны на БНФ. Достиг огромной популярности (первая систематизированная попытка создания ЯП)

Algol 60 стал прародителем многих последующих ЯП – Паскаль, Ада…

Но был очень существенный минус – ключевые слова нужно было обязательно подчеркивать. А как дело происходит на перфокартах?

Begin -> ‘Begin’ или \Begin/ и т.п. Отсюда неоднозначность и не совместимость – программа, работающая на одном компьютере могла не работать на другом. Напрашивается вывод, что Algol не подходящий инструмент для программирования.

L1/L2 > 1 – время работы программы, написанной на Алголе/ «ручками».

**1959 – 1961** – Джон Макарти создал язык Lisp.

Информацию легче представлять в виде иерархической структуре.

Основная операция: eval S – вызов функции.

Отсюда появилось понятие функционального программирования.

Первая программа с ИИ была написана на Lisp.

Не самый удачный пример языка для программирования.

**1959** – COBOL. В основном был известен в Европе, в СССР – почти нет.

В банках программное обеспечение было написано на COBOL. Никто не думал, что язык доживет до 2000 года.

**1960** – Бурное развитие ЯП.

«Каждый уважающий себя ВЦ должен иметь свой ЯП». Причем чаще всего на базе других ЯП. Такое бурное развитие – плохая мобильность знаний.

**Заключение**

У любого языка программирования образовалась своя *экологическая ниша*.

Из биологии: Какое место в природе. Два разных вида не могут существовать в одно время в одной нише (как правило).

По теории Дарвина – более совершенный вид выбивает менее совершенный.

Но на практика оказалось иначе – «выбить» старый вид сложнее, чем создать все с нуля.

Языки программирования ведут себя подобным же образом.

Фортран – вычисления.

COBOL – банки.

Lisp – обработка символьной информации.

**Лекция. Лихогруд Н.Н.**

**Очерк развития Языков Программирования**

| | |
|---|---|
| 1954 –начало 60-х | «Эмбриональный» период развития |
| 60-е - начало 70-х | «Экстенсивное» развитие |
| 70-е – начало 80-х | «Зрелость» языков программирования |

**Экстенсивное развитие программирования**

Каждый уважающий себя институт\проект считал своим долгом разработать собственный ЯП)

| | |
|---|---|
| 1967 | «симула 67»:<br>• введение классов<br>• Объекты классов, расположенные в ОП<br>• Оператор «new» |
| 1972 | «Small Talk» - первый ОО(объектно-ориентированный) ЯП |

Языки создавались под конкретные задачи и занимали свои экологические ниши, так же как животные в природе. Как только язык начинали использовать не по назначению (не в своей нише), возникали ошибки

DO 5 I = 1,3 Пробелы в Фортране не рассматриваются

/операторы/ (нет разделителей)

5 continue

DO5I=1.3 Ошибка: Не было надлежащих определений.

(Данная строчка является присваиванием идентификатору значения DO5I 1.3)

Из-за подобной ошибки в своё время погиб американский шаттл

| | |
|---|---|
| 1964 | PL/I (programming language)<br>**Первая** попытка создания универсального языка(компанией IBM)<br>Содержал огромное количество конструкций, которые иногда взаимодействовали самым непредсказуемым образом<br><br>*При разработке языка каждая конструкция должна проверяться на взаимодействие с другими структурами* |
| 1968 | Algol-68<br>**Вторая** попытка создания универсального языка (компанией IFIP) |

***Ортогональность языковых конструкций –*** взаимонезависимость(везде, где допустима переменная, допустимо и выражение и т.д.)

В FORTRAN

DO 5 I=2, N–1 – ошибка.

Правильно будет  N1=N1-1

DO 5 I=1,N1

В Algol-68

exp – выражение

exp; - оператор

Выражение может быть оператором, оператор может стать выражением(V = 5)

Для Algol-68 везде где можно использовать выражение можно использовать оператор. Таким образом, Algol-68 практически полностью ортогональный язык.

В Algol-68 были формально описаны не только синтаксис, но и семантика. В 1979 вышло  пересмотренное сообщение об Algol-68, из-за того что рядовые программисты были не в состояние понять язык. Но и оно не принесло счастья, т.к. использовало двухуровневые W-грамматики, Мета-правила, Прото-правила. Мета-правила генерировали Прото-правила. Язык был сложным, строгим и его не поняли рядовые программисты. Компиляторы были невероятно сложны, и их было создано немного.

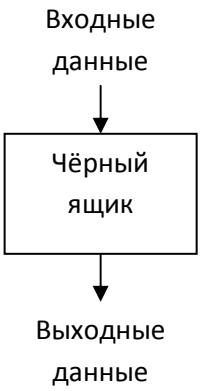
| | |
|---|---|
| 1969 | Денис Ричи создаёт язык «Си» - машинно-независимый ассемблер.<br>На основе языка «Си» Томсон создал UNIX (в противовес MULTICS)<br>Вирт создаёт язык Pascal |

Си и Паскаль являются простыми языками программирования (по сравнению с PL/1 и Algol68). С их появлением завершилась пора экстенсивного развития программирования. Паскаль успешно занял свою нишу языка для обучения студентов программированию, а Си вытеснил ассемблер из ниши системного программирования.

**«Зрелость» языков программирования (70-е – начало 80-х)**

«Языки программирования нужно проектирования нужно проектировать!»

1967 Дейкстра опубликовал статью о "о вреде оператора goto", в которой разбил в пух и прах оператор goto. В этой статье Дейстра затронул тему качества программирования.




Дейкстра предложил подход к программированию как к пошаговой детализации программы. Метод чёрного ящика – известно, что должно передаваться в блок на входе и что получаться на выходе. Внутренние операции и преобразования данных неизвестны (будут детализированы позднее) Например задача обработки информации выглядит как три последовательных чёрных ящика( «Подготовить», «Обработать», «Завершить использование»), которые нужно постепенно детализировать, пока программист не дойдёт до уровня конструкций языка.

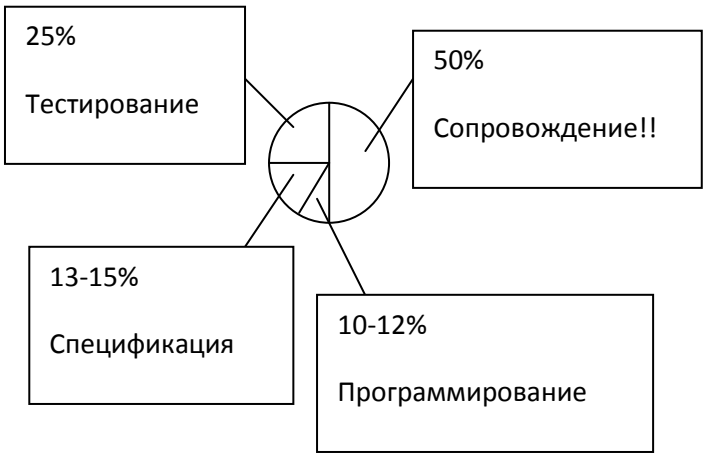
**Абстракция данных**

Программист должен концентрироваться на участках и блоках кода, абстрагируясь, т.е. по-простому «забывая» про другие данные.

По мнению Барбары Лисков, абстракция данных получила наибольшее развитие в языках CLU, ALPARD и Modula. На уровне современного понимания языков программирования считается, что абстракция и инкапсуляция данных являются самыми важными понятиями, даже важнее наследования.

Итак, период зрелости начался с языков программирования Си и Паскаль, которые сразу заняли свои экологические ниши. Следующим важным языком этапа «Зрелости» является **Ада.**

В Пентагоне, который в те времена активно разрабатывал систему ПРО и различные виды компьютеризированных вооружений, провели исследование расходования средств на жизненный цикл программы. Выяснили следующее:




В связи с тем, что Пентагоном и его субподрядчиками использовалось свыше 350 ЯП (из них было около 12 более-менее часто используемых, остальные – редкие => сопровождение ещё дороже, т.к. подходит не любой программист, а только специально подготовленный), отсутствовали стандарты, было принято решение разработать универсальный ЯП(**3-я попытка**), и покупать программы, написанные только на нём.

| | |
|---|---|
| 1976 - 1978 | В 1976 Были выставлены первые требования (их называют «соломенными»), затем выставили «Деревянные», и, наконец, в 1978-м году – «Стальные». |
| 1979 | – Объявлен конкурс и языки-матки. Ими стали PL/I, Algol 68, Pascal.<br>Сквозь все требования прошли 12 языков. Далее было отобрано 4. Все 4 были основаны на Паскале. |
| 1980 | Победивший язык назвали «Ада» в честь первой женщины-программиста (да и вообще программиста), которая в своё время писала программы для создаваемой машины Бэббиджа, которую смогли создать только его потомки в 20-м веке. |

| | |
|---|---|
| 1983 | Стандарт языка Ада. Разработка комплекта аттестации компиляторов |

Примерно раз в два года выходил документ, который разъяснял стандарт, т.к. язык получился очень сложным. Пентагон уделил особое внимание контролю качества компиляторов и придумал целую систему их сертификации.

**Принципы языкового дизайна**

- **Принцип критичных технологических потребностей** («с**ундучка**») - хорошо бы взять с собой все вещи(потребности), которые в принципе могут пригодиться
- **Принцип минимальности языковых конструкций**(«Принцип **чемоданчика**») - чемодан ограничен и тяжёл, много не унесёшь => нужно брать только самое нужное

Как я понял, названия этих принципов подразумевают то, что сундук больше чемодана (на Руси не было шкафов, вещи хранили в больших сундуках) – комментарий студента.

Из вышесказанного видно, что эти принципы являются конкурирующими, т.к. уклон в сторону критичных технологических потребностей ведёт к перегрузке и чрезмерному усложнению языка, что противоречит второму принципу. И наоборот, если реализовать только минимум конструкций, язык может получиться недостаточно мощным.

| | |
|---|---|
| 1988 | Вирт разрабатывает **Оберон.** Язык, который наиболее соответствует принципу минимальности минимальности языковых конструкций. |
| 1993 | **Оберон-2**. Ещё более сжатый и краткий язык. Характерен быстрой скоростью компиляции и эффективностью получаемого кода. Оригинальное описание языка, составленное Виртом, имеет размер около 20 страниц |
| 1995 | Новый стандарт **Ада-95.** В и без того сложную Аду добавили объектно-ориентированную парадигму. |
| 2005 | **Ада-2005** |

GNAT — один из самых популярных компиляторов с языка Ada.

При разработке Ады уделили внимание

- Надёжности – важнейшее требование, т.к. на Аде пишутся программы для военных объектов. И ошибок, падений и прочего тут быть не должно
- Эффективности – все военные системы должны работать быстро
- Читаемости

Основной недостаток Ады – сложность.

Ада и Модула-2 были завершающими языками периода зрелости.

**Объектно-Ориентированное Программирование:**

Первым полностью объектно-ориентированным языком был Small Talk

| | |
|---|---|
| 1980 | Модула-2 |
| 1983 | Си++ |
| 1995 | Java |
| 1999 | C# |

**Лейтмотив** лекции – «Существует ли универсальный язык?». Ответ однозначен – Нет!

**Лекция. Ульянов А.В.**

**4.1. Основные концепции императивных языков программирования.**

**ПП:**

1) Императивное – господствует
2) Функциональное – ML, Lisp, Хаски
3) Логическое

Большинство ЯП – императивные, это связано с наибольшей эффективностью. (Такая у нас сейчас архитектура ЭВМ).

Императивное: основано на принципах Фон Неймана:

Память ← → ЦП (АЛУ, УУ)

Объекты, данные, типы… := операции

:= - абстрактный обмен между памятью и ЦП.

**1978** – Джон Бекус прочитал уникальную лекцию «может ли программирование быть освобождено от Фон Неймановского программирования». Предложил ФП. Такая ситуация не вечна. Просто императивные ЯП заведомо более эффективны. Но от этого отходят.

**Глава 1. Базисные понятия ЯП.**

**П1. Данные, операции, связывания.**

**Данные –** некая сущность, над которыми выполняются операции.

**Операции –** то, что можно делать с данными.

**Дуализм (между данными и операциями).**

Строка – данные. А длина строки?

В **C** - …\0 Length – операция.

В **TP** – длина строки ограничена 255 (в нулевом байте хранится длина строки). Отсюда д.с. – данные.

В **VB** – BSTR – похож на char. Длина строки находится в первых двух байтах => данные.

Такой дуализм вводит понятие Свойства: getters\setters. Которое реализовано, например, в **C#**:

```
int x
{
    get{…}
    set{…}
}
Class c;
c.x;
```

С точки зрения доступа – данные, а вот с точки зрения реализации – x состоит из двух процедур, т.е. – операции.

**Что главное: операции или данные? С точки зрении современной теории:**

В ООП хорошо спроектированный класс должен иметь функциональный интерфейс (данные –> protected) => главные – операции, а данные помогают их реализовывать.

**Связывание:**

**ОД(объект данных)** – переменные, константы – абстрактное место, где могут храниться данные.

**Существуют связи:**

ОД < - > значение

ОД < - > адрес

x.f() – связывание между именем функции и его телом.

Причем время связывания не равно моменту связывания. (с точки зрения интервала)

Существуют 2 основных вида связывания:

a) До начала работы программы (статическое)
b) Во время работы программы (динамическое)

```
i: integer;
```

До начала работы программы идет связывание i и integer, а также integer и набором допустимых операций (+,-,*,div,mod) в момент компиляции.

**Виды связывания:**

1) **Статическое –** в момент трансляции, сборка(линковка) **–** с точки зрения ЯП.
2) **Динамическое –** во время работы программы - между ОД и его значением: V=30;
3) **Квазистатическое** (константы: const C = 20;)

C++: void f( int N ) - КС

{

int K = N; – похоже на статическое связывание, но выполняется динамически, во время работы программы.

}

Ада: N: integer const := 20;

Procedure D( L: integer )

K: integer const := 2; - КС

Begin

…

End D;

**Время связывания будем различать на:**

1) Статическое
2) Динамическое
3) Квазидинамическое

**Атрибуты данных:**

ОД в большинстве ЯП имеет 6 атрибутов данных (не все являются обязательными):

a) имя – необязательно
b) значение – обязательно
c) адрес – не универсален => зависит от компилятора
d) тип данных
e) время жизни
f) область действия

**Имя (идентификатор):** необязательно

int a[10]; a[2] – статическое

int i; a[i] – динамическое (зависит от компилятора)

**Имя <-> значение:**

Для константы – статическое, для переменной – динамическое.

**Адрес:**

X* p= new X();

X & x = *p;

delete(&x);

Нельзя брать адрес от константы. Ссылку на константу можно.

**Типы данных:**

**Чем выделяются (определяются)**

Определяются назначением.

Современная точка зрения:

**Тип данных** = Множество операций + Множество значений.

Примером может служить стек: Pop, Push, IsEmpty

IsEmpty(v) = true;

Push(v) => Not IsEmpty(v)

**Абстрактный тип данных =** множество операций. (основной вид современного программирования)

**Когда происходит связывание: ОД ⇔ ТД**

Большинство императивных ЯП – статическое связывание.

Java Scrypt, php… – динамическое связывание. (в основном такие языки являются интерпретируемыми).

Ада 83: Любой ОД имеет тип.

В традиционных ЯП – любой ОД имеет единственный тип.

ООЯП – любой ОД имеет статический тип. (Base d;)

Однако некоторые ОД могут иметь динамический тип.

C++: указатели и ссылки.(Base* p;)

**Время жизни (время существования): класс память.**

Определяется от момента распределения в памяти, до момента выхода из памяти.

**Все время работы программы: статические объекты (С)**

**Динамические объекты:**

Время жизни определяется программистом – new (delete).

Когда удаляется из памяти – “сборщик мусора”, но отсутствует delete (или от реализации).

**Промежуточные:**

В C++: Автоматический класс памяти (объект)

Блок (является управляющей конструкцией): блок объявлений и блок операторов.

**Объект <-> Память.**

Статический => Статическая.

Динамический => Динамическая.

Квазистатический => Квазистатическая.

**Сохраняемы объекты данных:** могут сохраняться во внешней памяти и восстанавливаться. Могут сохраняться и восстанавливаться между работой различных запусков программы.

**CLR**(спецификация) **–** единая система типов. Набор типов данных, динамический сборщик мусора.

Объявление -> имя; ← связывание → ТД

**Область видимости –** часть программы, где устанавливается объявление.

Область действия (видимости) распространяется на имена.

1. Определяющие вхождения имени (как правило предшествуют использованию) – определяются основные характеристики имени.

2. Использующие вхождения имени.

Область действия – область действия определяющего вхождения.

**Область действия:**

a) статическое
b) динамическое

Это понятие связывается только с ОД.

ОД ⇔ имя

**Делится на 2 класса:**

1) определяющее вхождение
2) все остальное – использующее вхождение.

Область действия имени – часть программы, где действия определяют вхождения.

Область действия определяется синтаксически и часто является блоком.

Pascal:

i: integer;

Правильный ответ – Я не знаю контекста.

1-й контекст: блок begin … end

2-й: var i: integer;

3-й: record … end

Java script:

Var i; - объявление имени.

i = 0; - определяется вхождение.

Если же сделать по-другому, то при a != 0 будет ошибка:

if ( a == 0) i = j;

j = i;

**Перекрытие имен:**

void f();

void f( int );

Перекрытие определяется правилами языка.

Перекрытие имен не является необходимым (Оберон), но иногда это полезно.

В большинстве ЯП области действия не накладываются, а только вкладываются.

В таких языках, как Java, C# локальное имя не должно совпадать с глобальным для данного блока. Иначе будет ошибка.

Классы ведут себя как вложенные области действия, а перекрытие называется скрытым.

Overload - перекрытие, перезагрузка.

Hading – скрытие имен.

Overriding – подмена (для виртуальных методов).

В зависимости от связных областей действия делятся на:

1) статические (определяются во время трансляции)
2) динамические (в момент выполнения)

Определяющее вхождение исключений – это ловушка:

catch(T e) { … }

Происходит динамически (даже в статических ЯП – C++)

**П.2. Понятие виртуальной машины ЯП.**

Само понятие появилось еще в 60-е годы.

**Опр.:** Это гипотетическая машина, машинный язык которой – ЯП.

Стали реализовывать в железе.

В СССР: машина серии МИР(для инженерных расчетов).

Например, в университетах существовала следующая проблема: различные спонсоры дарили компьютеры, а для них было необходимо разное ПО.

Тогда появилась виртуальная машина p-code для языка UCSD-Pascal.

PARC Small Talk: 1972, байт-код.

**Идея Java машины:** есть язык, программы передаются по интернету, на разных машинах работают одинаково.

JVM – виртуальная машина. Язык машины – Java-байт код.

.NET – похожая ситуация. Любой язык, который поддерживает .NET транслируется в CLR

=> не существует виртуальной машины для MSIL.

Вместо этого JIT(трансляция на лету). Система времени выполнения:

1-й раз в бинарный код, затем выполняется. Запоминается в КЭШе и далее работает без компиляции.

+ Понятие виртуальной машины изолирует от архитектуры реальной машины.

**П.3. Схема рассмотрения ЯП.**

Сложность – семантический разрыв (разрыв архитектурных компонентов и реального времени.)

1) Базис. (встроен в ЯП, понимается транслятором)
   Может быть скалярным (не имеет составных частей, простые типы данных, встроенные операции) и структурным (составные типы данных - массив, записи и т.д.). Базисы всех языков похожи.
2) Средства развития. (подпрограммы, создание новых типов данных).
   Самые простой и известный – процедуры и функции.
   Возможно создание специальных библиотек.
   X lib - библиотека языка C, для создания интерфейсов под Unix.
   Xt - Понятие виджетов и т.п.
   Atheng Motif – имеет визуальный интерфейс.

**Лекция. Сергеев Николой.**

## Простые типы данных

Под простыми типами данных мы имеем в виду языковые конструкции, встроенные в язык, то есть типы данных изначально поддерживаемые в языке.

1. Классификация простых типов данных
   - числовые
     - целочисленные
     - вещественные
       - плавающие
       - фиксированные

   Не во всех языках есть такое усложнение класса чисел, например, в таких скриптовых языках как JavaScript и ActionScript есть тип Number, обьекты класса Number могут принимать как целочисленные значения, так и вещественные.

   - логические
   - символьные
   -порядковые
     -перечисления
     -диапазоны
   -ссылки и указатели

   И еще иногда вводят функциональные тип(функции и процедуры)

   Давайте рассмотрим по порядку все типы данных

   Целочисленные
   Перед разработчиком языка при создание целочисленного типа всегда возникают как минимум три проблемы:
   1. Фиксация представления – то есть фиксировать ли размеры типов данных, и набор значений.
   2. Беззнаковые целые числа – нужно ли выделять беззнаковый целый тип
   3. Преобразования одного целочисленного типа в другой

Давайте немного поподробнее остановимся на этих проблемах:

Наиболее известные языки программирования с фиксацией представления типов данных – Java, C#. Но оно и понятно. Ведь код на Java является мобильным, то есть должен работать на различных платформах. А как он будет работать правильно, если на разных платформах диапазон целых чисел будет иметь различную длину. Поэтому Java работает не напрямую, а через виртуальную Java – машину, то есть является интерпретируемым языком. Другое дело с языком C. Именно в языке C решили ввести диапазон чисел, который должен быть покрывать диапазоны, вмещаемые в 1 - 8 байт:

-char

-short int

- int

- long int

- long long

Правда здесь отдельно ничего не ясно про каждый тип(кроме char – 1 байт), например short int на различных платформах может занимать как 1 байт, так и 2 байта. Но зато ясно, что количество байт отводимых под char, меньше либо равно количеству байт, отводимых под short int, количество байт отводимых под short int, меньше либо равно количеству байт, отводимых под int, и т.п. Такая системы была унаследована в языка C++.

Вернемся к C# и перечислим CommonTypeSystem, которые были там реализованы.

| Количество байт | Со знаком | Без знака |
|---|---|---|
| 1 | sbyte | byte |
| 2 | short | ushort |
| 4 | int | uint |
| 8 | long | ulong |

В 2000 году произошло очень важное событие, была реализована компанией Интел 64 – битная архитектура IA-64 для процессора Itanium. Правда 64 – битная архитектура не прижилась. Почему? Не было реализовано популярной операционной системы с поддержкой 64 – битной архитектуры, вследствие этого все программы выполнялись с такой же скоростью как и на 32 – битной архитектуре. Так зачем было платить больше, если за те же самые деньги мы получаем ту же самую производительность? И к тому все 32 – битные приложения на 64 – битной архитектуре выполнялись в режиме эмуляции. Намного умнее поступила компания AMD, которая решила оставить арифметику 32 – битной, а вот адресацию сделала 64 – битной(архитектура x64).

Рассмотрим беззнаковые целые числа. И сразу же возникает проблема как сравнивать между собой знаковый и беззнаковый типы данных. И надо ли допускать неявное приведение беззнакового целого типа данных к знаковому и обратно? Такое допущение может привести к ошибкам, которые появляются только на этапе работы программы, а значит их сложно найти. В 70- ые годы эти вопросы решались однозначно. Беззнаковый и знаковые типы данных не смешивали между собой. Их нельзя было ни сравнивать, ни присваивать друг другу. Так в Модуле-2 существовали два типа INTEGER и CARDINAL. И приведение одного в другой допускалось только явное(существовал специальный оператор для такого присваивания):

I:INTEGER; J:CARDINAL; I := INTEGER(J);

Но в современном мире языков программирования многие языки такими ограничениями не обладают. Например, язык C++. Хотя сам создатель языка, Страуструп, хотел ввести такое ограничение, и лучше того, в первой версии языка было допустимо только явное приведение типов. Но как оказалось, системщики очень любят неявное приведение, и большая часть программ, написанных на C из – за этого ограничения не

компилировалась на первой версии языка C++. Тогда Страуструп принял волевое решение убрать такое ограничение.

А зачем вообще по идее нужен беззнаковый тип данных? Конечно же для адресации памяти(адресной арифметики) и для работы со сдвигами. Так, к примеру в Обероне существует лишь один беззнаковый тип byte(0..255), и Java, позаимствовавшая типы данных именно с Оберона была вынуждена включить дополнительно включить в базовые арифметические операции беззнаковый сдвиг(>>>). То есть базисный набор операции над целыми числами расширен по сравнению с базисным набором операций в С.

Существовали и другие решения данной проблемы. Так в C# решили допускать только расширяющиеся преобразования(так как они безопасные). Обратные преобразования допускались только явно. Про приведение типов в АДА смотреть ниже.

**Подходы к реализации числовых типов данных.**

1. Фиксирование базиса. То есть мы заранее говорим, что вот такие типы данных, как integer,byte … являются целочисленными. Все остальные уже не целочисленные. В некоторым языках фиксируется даже размер каждой переменной данного типа, то есть integer в Java занимает, например, строго 2 байта. Как правило код на таких языках интерпретируется. В Java, например, виртуальная Java – машина. Еще один пример такого языка C#. Программы на таких языках являются мобильными, то есть переносимыми с одной платформы на другую. В других языках, переносимость затруднена, там не фиксированы ни представление чисел, ни семантика числовых операций.

2. Обобщенные числовые типы. Ада создавалось отчасти и для того, чтобы код на ней был кроссплатформенным. Поэтому фиксирование базиса в такой ситуации было затруднено. Перед создателями Ады ставились следующие задачи: Эффективность, Надежность, Читабельность. В следствие этого создатели Ады придумали новую концепцию. Они ввели обобщенные числовые типы данных. То есть типы являющиеся базой для всех других типов( все другие наследовались от них). Объекты разных типов были несовместимы ни по какому множеству операций, но были совместимы объекты подтипов. То есть разные подтипы совместимы между собой и со своим предком. Например:

   *Type Length is new integer;*

   *Type Width is new integer;*

   Length и Width – новые целочисленные типы данных, при этом их нельзя ни присваивать друг другу, ни сравнивать. Однако можно делать преобразования явным. В С или Паскале мы бы таким способом ввели бы понятие эквивалентности.

   Еще один пример:

   *Type Length in new integer range 0..MAXN.*

   Думаю в комментариях не нуждается - диапазон Length ограничен 0..MAXN.

   В чем плюс обобщенных типов, так то что все ошибки обнаруживались на этапе компиляции. Если требовались неявные преобразования вводились под – типы.

   *Sybtype t1 is t2 range 0..N.*

   И тогда преобразования из T1 в T2 допускались. При этом компилятор сам выбирает оптимальное представление для таких чисел. То есть представление таких типов данных зависело не от архитектуры, а от диапазона чисел. Теперь компилятор транслирую присваивание таких типов

данных либо проверяет возможность такого присваивания на этапе трансляции, или же вставляет код по проверке допустимости такого присваивания(квазистатический контроль). В случае ошибки выхода за границу в Ада возбуждается range error.

Сейчас идет возврат к обобщенным типам данных.

## Вещественные типы данных.

При представлении действительных чисел в компьютере, как и целых чисел, используется чаще всего двоичная система счисления (иногда двоично-шестнадцатеричная), следовательно, предварительно десятичное число должно быть переведено в двоичную систему, а затем представлено в нормализованном формате с P = 2 (для двоично-шестнадцатеричной системы P = 16).

Независимо от используемых систем счисления существует два основных типа представления чисел в компьютере, называемые представлениями с фиксированной и с плавающей запятой.

В типах данных, использующих представление чисел с фиксированной запятой, все разряды ячейки, кроме знакового разряда, служат для изображения разрядов чисел. Причем каждому разряду ячейки соответствует всегда один и тот же разряд числа, что и фиксирует место запятой перед определенным разрядом. Такая система упрощает выполнение арифметических действий, но сильно ограничивает диапазон чисел, которые могут быть представлены в таком типе. Чаще всего это бывает диапазон $-1 < x < 1$. Этот случай соответствует соглашению: что при чтении чисел, записанных в машине, запятая ставится непосредственно перед старшим цифровым разрядом.

Наибольшее по абсолютной величине число, которое может быть представлено в машине таким образом, равно 1 – p-n, наименьшее – отличное от нуля – p-n. Для представления чисел, не укладывающихся в этот диапазон, программисту надо вводить масштабные множители, т.е. заменять истинные величины, участвующие в решении задачи, их произведениями на специально подобранные коэффициенты. Это существенно усложняет решение поставленной задачи. Поэтому для представления вещественных чисел в современных компьютерах формат данных с фиксированной запятой не применяется.

Описанные первой главе целые типы данных представляются в компьютере в формате с фиксированной запятой. Положение запятой в них зафиксировано после правого разряда.

Чтобы избавиться от недостатков представления действительных чисел с фиксированной запятой, в современных ЭВМ принят способ их представления с плавающей запятой. Этот способ представления опирается на нормализованную (экспоненциальную) запись чисел:

$+- M * B^p$ ( знак/ мантисса/порядок(B- основание системы счисления).

Представление в виде мантиссы и порядка не единственно. Для арифметических операция используется нормализованное представление, то есть $1/B < -M < 0$. То есть в системе с основанием B все степени B представимы точно.

Еще раз скажу, что при вещественных операциях точность страдает, так мантиссу можно выбрать так, что погрешность будет порядка нескольких сотен(не мало да!)

Использование как в компьютере, так и в калькуляторе представления чисел с плавающей запятой усложняет схему арифметического устройства.

При сложении и вычитании чисел сначала производится подготовительная операция, называемая выравниванием порядков. Она состоит в том, что мантисса числа с меньшим порядком сдвигается в своей ячейке вправо на количество разрядов, равное разности порядков данных чисел. После этой операции одноименные разряды чисел оказываются расположенными в соответствующих (одних и тех же) разрядах обеих ячеек, и теперь сложение или вычитание мантисс выполняется достаточно просто, так же как над числами с фиксированной запятой.

При умножении двух чисел с плавающей запятой их порядки складываются, а мантиссы – перемножаются (предварительное выравнивание не производится).

При делении из порядка делимого вычитается порядок делителя, а мантисса делимого делится на мантиссу делителя.

После операций над порядками и мантиссами получается порядок и мантисса результата, но последняя может не удовлетворять ограничениям, накладываемым на мантиссы нормализованных чисел. Поскольку от результата арифметических операций в машине требуется, чтобы он также; был нормализованным числом, необходимо дополнительное преобразование результата, называемое нормализацией. В зависимости от величины получившейся мантиссы результата, она сдвигается вправо или влево так, чтобы ее первая значащая цифра попала в первый разряд после запятой. Одновременно порядок результата увеличивается или уменьшается на число, равное величине сдвига.

Когда говорят о точности представления вещественных чисел, надо помнить следующее: десятичное число, имеющее даже всего одну значащую цифру после запятой, вообще говоря, невозможно записать точно в любом из вещественных типов. Объясняется это тем, что конечные десятичные дроби часто оказываются бесконечными периодическими двоичными дробями. Так, 0,110 = 0,0(0011)2, а, значит, и в нормализованном виде такое двоичное число будет иметь бесконечную мантиссу и не может быть представлено точно. При записи подобной мантиссы в ячейку компьютера число не усекается, а округляется.

Плавающие числа нужны для представления математических расчетов В других сферах часто требуется несколько иное, например если мы работаем с рублями, то нам важна точность до второго порядка после запятой(копейка). То есть каждое число по идее является целым, но по виду оно вещественное. И если есть только плавающий тип, то при представлении таких чисел неудобно использовать операции с плавающей точкой. Поэтому в некоторых языках ввели delta – типы.

*Ада:  type DATA is delta 1/4096 range –M..M*

При этом вся дальнейшая работа по представлению  ложится опять на компилятор(он сам лучше подберет как хранить это число).

В других языках появляются еще разновидность delta – типов: C# - decimal, currency.

Логический тип данных.

Логические, или булевские, данные предназначены для хранения логических значений "истина"(1) или "ложь"(0). В большинстве языков базис операций над логическим типом сводится к операциям: or, not, and

- *И* (логическое умножение) (AND, &, *),
- *ИЛИ* (логическое сложение) (OR, |, +),
- Отрицание (NOT, ~, !)

Булевый тип данных может быть реализован с использованием только одного бита, но обычно используется минимальная адресуемая ячейка памяти (байт) или машинное слово, как эффективная единица работы с регистрами и оперативной памятью.

Ада: *type Boolean is (False, True); // представление типа Boolean*

```
p : Boolean := True;

...
if p then
  ...
end if;
```

C#, C, Java, Algol: bool p;
Кстати С единственный язык с неявным преобразование int в bool
Delphi, Pascal: var p:boolean

Символьный тип данных.

Символьный тип (Char) — простой тип данных, предназначенный для хранения одного символа в определённой кодировке. Может являться как однобайтовым (для стандартной таблицы символов), так и многобайтовым (к примеру, для Юникода).

Родина большинство языков программирования – Америка и Европа, алфавиты которых не особо отличаются друг от друга и состоят из не очень большего количества различных букв. Поэтому вполне хватало 1 байта(256 различных символов) для кодирования любого символа. Но со временем стало понятно, что 1 байтом не обойтись( например, в программах где приходилось реализовать многоязычность). Поэтому люди стали думать о новых символьных типах данных. И происходило это в 80 – ые годы. Первым путем решения этой проблемы стало введение различных кодировок, как например ASCII-7, которая жива и ныне( первые 128 служебные, потом идут большие и малые английские буквы, цифры, знаки пунктуации…). Многие кодировки содержат первые 128 символов, такие же как в ASCII-7, и что – то свое. Проблемы осложнились когда дело дошло до Китая и Японии(появилось еще множество новых кодировок…) Чтобы хоть как – то уладить проблемы с большим количеством различных кодировок в 91 году появляется стандарт Unicode. На нем остановимся чуть поподробнее. Юнико́д, или Уникóд (англ. *Unicode*) — стандарт кодирования символов, позволяющий представить знаки практически всех письменных языков.

Стандарт предложен в 1991 году некоммерческой организацией «Консорциум Юникода» (англ. *Unicode Consortium*, *Unicode Inc.*). Применение этого стандарта позволяет закодировать очень большое число символов из разных письменностей: в документах Unicode могут соседствовать китайские иероглифы, математические символы, буквы греческого алфавита, латиницы и кириллицы, при этом становятся ненужными кодовые страницы. Рассмотрим наиболее популярный формат UTF-8.

UTF-8 (от англ. *Unicode Transformation Format* — формат преобразования Юникода) — в настоящее время распространённая кодировка, реализующая представление Юникода, совместимое с 8-битным кодированием текста. Нашла широкое применение в операционных системах и веб-пространстве Unicode и его роль в веб-пространстве.

Текст, состоящий только из символов с номером меньше 128, при записи в UTF-8 превращается в обычный текст ASCII. И наоборот, в тексте UTF-8 любой байт со значением меньше 128 изображает символ ASCII с тем же кодом. Остальные символы Юникода изображаются последовательностями длиной от 2 до 6 байтов (реально только до 4 байтов, поскольку использование кодов больше $2^{21}$ не планируется), в которых первый байт всегда имеет вид 11xxxxxx, а остальные — 10xxxxxx.

Проще говоря, в формате UTF-8 символы латинского алфавита, знаки препинания и управляющие символы ASCII записываются кодами US-ASCII, а все остальные символы кодируются при помощи нескольких октетов со старшим битом 1. Это приводит к двум эффектам.

- Даже если программа не распознаёт Юникод, то латинские буквы, арабские цифры и знаки препинания будут отображаться правильно.
- В случае, если латинские буквы и простейшие знаки препинания (включая пробел) занимают существенный объём текста, UTF-8 даёт выигрыш по объёму по сравнению с UTF-16.[1][2]
- На первый взгляд может показаться, что UTF-16 удобнее, так как в ней большинство символов кодируется ровно двумя байтами. Однако это сводится на нет необходимостью поддержки суррогатных пар, о которых часто забывают при использовании UTF-16, реализовывая лишь поддержку символов UCS-2.[1].

Символы UTF-8 получаются из Unicode следующим образом:

| Unicode | UTF-8 |
|---|---|
| 0x00000000 — 0x0000007F | 0xxxxxxx |
| 0x00000080 — 0x000007FF | 110xxxxx 10xxxxxx |
| 0x00000800 — 0x0000FFFF | 1110xxxx 10xxxxxx 10xxxxxx |
| 0x00010000 — 0x001FFFFF | 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx |

Также теоретически возможны, но не включены в стандарты:

| Unicode | UTF-8 |
|---|---|
| 0x00200000 — 0x03FFFFFF | 111110xx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx |
| 0x04000000 — 0x7FFFFFFF | 1111110x 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx |

Замечание: Символы, закодированные в UTF-8, могут быть длиной до шести байт, однако стандарт Unicode не определяет символов выше 0x10ffff, поэтому символы Unicode могут иметь максимальный размер в 4 байта в UTF-8.

Многие программы Windows (включая Блокнот) добавляют байты 0xEF, 0xBB, 0xBF в начале любого документа, сохраняемого как UTF-8. Это *метка порядка байтов* Юникода (англ. *Byte Order Mark*, BOM), также её часто называют *сигнатурой* (соответственно, UTF-8 и UTF-8 with Signature). Чтобы при сохранении избавиться от добавления сигнатуры, используйте, например, Notepad++.

**Лекция. Лихогруд Н.Н.**

**п.2.4 Ограниченные типы данных**

**п.2.4.1 Перечисления (перечислимые типы данных)**

Паскаль:

Type EnumTyp = (va1, …, valN);

Операции:

:=, =, <, >, <>, >=, <= - основаны на функциях succ(x), pred(x)

ord(x) = 0…N-1 (Элементы упорядочены)

Таким образом перечислимый тип – некий способ удобного создания, хранения и использования констант.

Для UNICOD FFFE – «магическая константа», определяет порядок байтов

Совет по программированию – Использование констант делает программу более понятной. В программе не именованными должны быть только 0,1,-1. Все остальные константы нужно именовать.

т.е. I := 54; - плохой стиль

преобразования:

| | |
|---|---|
| EnumType -> Integer | безопасно |
| Integer -> EnumType | Небезопасно. Нужны проверки |

**Статические проверки** – проверки при компиляции программы. Проверяемые данные не меняются в различных вызовах функции (запусках программы)

A:array [1..100] of integer;

…….

a[10] = 10; Статическая проверка

**Квазистатические проверки** - проверки при выполнении. Квазистатическим они называются потому, что проверяемые данные могут меняться в различных вызовах

функции (запусках программы). Поэтому эти проверки делаются именно при выполнении.

Компилятор обнаруживает опасное место в программе и вставляет в это место квазистатический код, который выполнится в процессе работы программы.

a:array [1..100] of integer;

…….

a[n] = 10; n не известна в момент компиляции. Если язык поддерживает квазистатические проверки, то здесь будет вставлен контроль выхода за границу массива

x:0…N-1

x:= expr; Вставится роверка вида «if(expr >= N) or (expr <0) then error()», т.к. значение expr не известно во время компиляции

В машинных языках нет квазистатических проверок. Поэтому нет и в Си

int a[10];

n = 11;

a[n] = 10; В Си проверки не будет

RTTI – динамическая информация о типе. Ради этого в Си++ вставлены квазистатические проверки.

В Си-89 добавили такую конструкцию:

```
enum ET(v0,…,vN);
```

которая эквивалентна последовательности строк

```
#define v0 0;
#define v1 1;
……..
```

Но так как нет квазистатических проверок, то все нижеследующие строки будут корректны:

```
enum ET y;
int I;
x = v1;
I = v2;
x = I;
I = x;
x = -5;
```

Негласная парадигма языка Си – «компактность кода», а квазистатические проверки, естественно, увеличивают генерируемый код.

В своё время перечислимые типы были очень популярны.

Но:

- Языки «Оберон»(1988) и «Оберон-2»(1993) уже не содержали перечислимые типы. В Обероне было «Расширение типов», которому противоречат перечислимые типы данных, которые невозможно расширить.
- uses (pascal, Ада) Вместе с перечислимым типом неявно импортируются все его константы на том же уровне видимости .Таким образом значения констант могут конфликтовать и перекрываться для разных перечислимых типов (в т.ч. из разных модулей)
- Java (1995) – нет перечислимых типов
- C# (1999) – перечислимые типы есть. Одно из назначений - хранение наборов значений параметров компонентов( например влево/вправо/по центру для выравнивания). Т.е. перечислимые типы интегрировали в визуальные средства проектирования.
- Java (2005) - расширение Java, в том числе добавление перечислимых типов, оформленных в виде полноценных классов

**Смещение парадигмы программирования**

| | |
|---|---|
| 1972 | SmallTalk |
| 1979 | Си с классами |
| 1983 | Си++ |
| 1988 | Turbo Pascal 5.5 |
| Начало 90-х | Всеобщая объектно-ориентированность |
| | И т.д. |

Результат смещения – появление «компонентов» - «чёрных ящиков» с «рычагами» (в их роли выступают методы) и «лампочками» (в их роли выступают properties), уход от иерархий классов.

Таким образом, сегодня профессиональный программист пишет sealed(C#, для Java – «final») класс, т.е. класс, от которого нельзя наследовать, если он является его собственным классом или не существует исключительных причин делать иначе.

Возвратимся к перечислимым типам данных («к нашим баранам», как говорит Игорь Геннадивевич):

Вопросы, связанные с реализацией и использованием перечислимых типов данных:

- Проблема представления – реализовывать ли возможность задавать конкретные значения констант (например, для цветов)?
- Проблема эффективности – реализовывать ли возможность управления представлением?
- Проблема преобразований в другие целочисленные типы данных – разрешать или не разрешать?
- Неявный импорт – разрешать или нет?
- Удобство использования – ввод, вывод и т.д.

**Модула-2**

```
type ET = (v1, …, vN);
Преобразования :
   ord(x);          ET -> integer
   val(T,i);        ET -> integer. Либо выдаёт ошибку,  либо выдаёт значение.
                    val(ET,v2) = 1;
```

Неявный импорт возможен
Удобность использования не расширена.

**C++ <-> C-89**

```
enum ET{…};
void f(ET x);
void f(int x);          Будет перегрузка, потому что перечисление – новый тип данных
```

В Си++ typedef задаёт синоним типа, а не создаёт новый тип.

```
typedef int a;

void f(a x)
void f(int x)           Перегрузки не будет, т.к. «a» не новый тип(=> будет ошибка)
```

В Си++ можно задавать значения константам перечисления

```
Enum FileAccept
{
    FileRead = 1;
    FileWrite = 2;
    FileReadWrite = FileRead | FileWrite;
}
```

Удобство использования такое же, как и для int

**Ада**

Типы BOOLEAN и CHARACTER являются перечислимыми типами данных из пакета STANDARD

**Проблема** неявного импорта:

```
type SendColor is (Red,Yellow,Green)
type BasicColor is (Red,Green,Blue)
```

Ввели понятие «литерал» перечисления. Им является либо идентификатор, либо символ(‘символ’)

```
type Latin is (‘A’, ‘B’,’C’,… ,’Z’)
type ASCII Latin(‘A’, ‘B’,’C’,… ,’Z’, ‘a’, ’b’,…)
```

**Литерал перечисления –** функция без параметров, имя которой совпадает с литералом и возвращает нужную константу

```
procedure P(x:SendColor)
procedure P(x:BasicColor)

P(Wellow) – в функцию отправится «1»
P(Red) - ?? ошибка
```

**Решение** проблемы неявного импорта:
**Уточнение типа** - **T’expr** – выражение **expr** трактовать как выражение типа **T**

```
P(BasicColor’Red) – правильно!
```

Представление:

```
for BasicColor use (Red => FF00000x, Green => FF00x, Blue =>FFx)
for BasicColor’Size use 24; - 24- количество битов
```

«’» специальная операция, применимая именами типов, позволяющая получить доступ к некоторым атрибутам

**C#**

Управление реализацией:

```
enum BasicColor
{
    Red = 0xFF0000;
    Green = 0xFF00
    Blue = 0xFF
};
enum SendColor: Byte
{
    Red;
    Yellow;
    Green;
 };
```

Преобразования из **перечислимых типов в целочисленные** в современных языках **только явные**.

Атрибуты – средства сообщать компилятору некоторую информацию о реализации типа.
[FLAGS] – указывает, что перечисление может обрабатываться как битовое поле, которое является набором флагов.

```
[FLAGS]
enum FileAcces
{
    FileRead = 1;
    FileWrite = 2;
    FileReadWrite = FileRead | FileWrite;
}
```

теперь, если вывести на экран FileAccept. FileReadWrite получим «FileRead, FileWrite». Без использования атрибута [FLAGS] получим 3.

**C#, Java**

**Неявный импорт**

```
FileAcces x;
X = FileAcces.Read; //Уточнение перечисления
```

**Удобство использования. Классы-обёртки**

Классификация ТД

1. **Java**
   - Простые Типы Данных
   - Референциальные типы данных
     - Классы
     - Массивы
2. **C#**
   - Типы-значения
   - Классы – размещаются только в динамической памяти

Для каждого типа-значения существует класс-обёртка
**C#** int - Int32,bool – Boolean, long – Int64
**Java** int – Integer,bool – Boolean, long – Long

В C# все обёртки находятся в .Net

Для всех перечислений имеется один класс-обёртка – «Enum»
⇨ всем перечислениям доступны методы класса Enum ToString(), GetValues() и т.д.

```
int [] vals = (int []) Enum.GetValues(typeof(…));
```

**Java v5.0 Tiger 2005**

```
Enum SendColor
{
  Red,Yellow,Green //Статические члены
}
SendColor c = SendColor.Red;
```

В Java v5.0 константы перечисления - статические члены класса, находящиеся в классе и представляющее собой значения класса. Упорядочены. Значения задавать нельзя.

```
 SendColors.valuesof();
                .ordinal();
                .value("Red") // -> 0

enum Apples
{
   ….
   Apple4(10); /// Каждое значение перечислимого типа должно быть со своей ценой
   ….
   int price;
   public Apples(int p)
   {
      price = p;
   }
   public void set_price(int p){…}
   public int get_price(){….}
}
…..

Apples x = Apples.Apple4; // можно писать без «new». Это исключение для перечислений
                          // Apples.Apple4 – значение класса, а не поля класса
```

**Объекты перечислений нельзя копировать.**
**Наследование от перечислений запрещено.**

### п2.4.2 Диапазоны

**Паскаль**

```
var
   x: L..R;
```

**Модула 2**

```
var
    x: [0..N];
    y: CARDINAL[0..N];
    i:INTEGER;
    j:CARDINAL;

x := 0;        Проверки не будет
x := i;        Вставка квазистатической проверки, если нельзя вычислить i
x := y;        Проверки не будет.
```

Напомню, что переменные типа CARDINAL могут принимать значения от 0 до 65535, а INTEGER от -32768 до 32767 (для 16-ти битных компиляторов)

**Ада**

```
Type Diap is range 0..N;
Type Pos is new INTEGER range 0..MAX_INT;    новый тип данных.
 ….
 I:integer;
 Y:Pos;
 Y := I;          компилятор вставит сообщение об ошибке, т.к. это разные типы данных
                  (из-за «new»)
 Y:=Pos(I);  Вставка проверки на знак
```

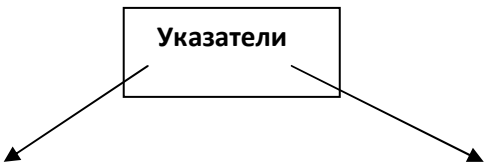
```
Subtype Natural is range 1..MAX_INT;
J:Natural
……
I := J;        Проверки не будет
J := I;        В этой строчке будет проверка
```

Ни в одном из современных языков программирования нет типов-диапазонов, т.к. современных языках чётко определены индексы массив:  0 <= i <= N-1, а основной областью применения диапазонов было именно задание типов индекса массива.

**п.2.5 Указатели и ссылки**

Адрес :

- Указатель
- Имя
- Метка



Проблемы строгих и нестрогих указателей:

| Строгие | Не строгие |
|---|---|
| Стандартный Паскаль, Модула-2, Ада, Оберон (со сборкой)<br><br>Pascal<br>Type PT = ^T;<br>{Modula-2<br>Type PT = pointer to T;}<br>var<br>i :T;<br><br>Инициализировать указатель можно только двумя способами – либо другим указателем, либо выделением новой памяти NEW(p : PT);<br><br>Поэтому все данные чётко разделяются на именованные, либо не именованные.<br><br>Указатель служит для работы с анонимными данными в динамической памяти.<br><br>Смысл – избежание части ошибок | C, C++, Turbo Pascal<br><br>Можно получать адрес любого объекта с помощью операции взятия адреса «&»<br><br>Существует абстрактный указатель «void *»<br><br>T * => void * автоматически<br>void * к T * автоматически не приводится<br><br>T * p;<br>void *pp;<br>pp = p;<br>p = (T *)pp; |

- Удаление памяти (Dispose (p : PT)) не в своё время, что приводит к появлению **«висячих ссылок»** - указателей, которые должны на что-то указывать, но не указывают.
- Накопление **мусора** – памяти, на которую не указывает ни один указатель.

```
P,P1 : PT;

New( P );

New( P2 );

P := P2; {порождение мусора}

Dispose(P2); {P – «висячая ссылка», попытка обращения к памяти, которую она занимает, приведёт к ошибке}
```

Различают системы с динамической сборкой мусора и без таковой.

Строгий язык с динамической сборкой мусора довольно надёжен.

От висячих ссылок защиты нет

**Ада**

В чистой Аде есть только new(p). В модуле STANDARD есть UNCHECKED_DEALLOCATION(p) – подчёркивается небезопасность этой операции

Примеры ошибок:

```
T *p;
```

```
Void f()
{
   T x;
   P = &x; //!!!адрес локальной переменной!!!
 }
void Foo()
{
   f();
   free(p); //!!!попытка освобождения невыделенной памяти!!! – выдастся ошибка
}
```

```
new(p1);
p := p1;
Dispose(p1);  {p «висит»}
```

Для Java, C# - указатели трансформировались в ссылки

Несколько слов о языке Small Talk

Последовательность действий при вычислении значения выражения «2+2»:

1. Посылка сообщения «+» обекту 2 с параметром 3
2. В классе integer ищется по таблице методов доступа обработчик сообщения «+» и вызывается
3. Обработчик отрабатывает и возвращает новый объект «5»

**Лекция. Ульянов А.В.**

**Указатели.**

В чем опасность использования: низкоуровневое программирование.

Если используется динамическая сборка мусора, тогда проще.

Так процедура UNCHECKED_DEALLOCATION(P) является аналогом delete(p), dispose(p) в ЯП, где используется динамическая сборка мусора.

C#, Java - понятие указателя отсутствует (точнее в C# такое понятие есть, но только в небезопасных блоках unsafe)

managed – управляемый код.

Функции .net не дают все возможности по использованию ресурсов ОС, отсюда приходится обращаться к возможностям Win API.

unsafe – код, где появляются новые конструкции, C RTL (как будто внутри языка C).

Обращаться к ним можно тоже только из unsafe. Функции, использующие это, тоже помечены как unsafe.

unsafe {…}

byte []b можно объявить как byte *b и использовать в вышеуказанном блоке.

Как работает динамическая сборка мусора:

Менеджер динамической сборки мусора запрашивает память, ее дают, а как только заканчивается, работает динамический сборщик мусора. Он-то и находит все неиспользуемые куски памяти, сводит к одному блоку.

Именно поэтому ссылка на byte может плавать и нельзя рассчитывать на то, что ее адрес будет постоянным. Потому преобразование byte[] в byte * возможно только блоке (ссылка замораживается):

fixed (byte* pb = b) {…}

C#, Java – понятие указателя исчезло и превратилось в понятие ссылки.

Типы значения,

Референциальные ТД (классы, массивы, интерфейсы) – к ним обращение только по ссылке.

X a; // Если в C#, Java – то объект является ссылкой и пока не существует.

a = new X(); //Вызов конструктора обязателен.

string[] a;

string[] b = new string[N];

a = b; // Присваивание ссылок, а не копирование.

**Понятие указателя в Ада 95(83).**

**Ада 83:**

PT – указатель на тип T.

type PT is access T;

x: PT;

x := new T;

y: T; - Получить адрес y стандартными средствами нельзя.

**Ада 95:**

Если: type PT is access T;

То инициализация возможна только так:

x: PT;

x := new T;

Если же: type PTT is access all T; (на все объекты типа T)

xx: PTT;

xx := new T;

Однако можно ссылаться и на другие переменные:

z: aliased T;

zz: T;

z ’access – операция взятия адреса.

x := z’access; - нельзя, т.к. без aliased.

xx := z’access; -можно.

xx := x; - можно.

x := xx; - нельзя.

В современных ЯП ссылки – это средства доступа к объекту.

В C#, Java, Delphi – имеются референциальные ТД.

“имена” = ссылки.

В C++ добавили отдельный базисный тип - ссылочный.

**С точки зрения операций:**

Чем различаются ссылка и указатель:

*(в C), ^ (M-2, P)

:= только к ссылкам разъименование объекта (и выполняется компилятором).

. – компилятор вставляет разъименование сам.

T is record

A: T1;

B: T2;

end record

type PT is access T;

X: Pt;

X.A; X.B; //разъименование делается компилятором.

X.all; // явное разъименование.

В C++ к ссылкам применяется единственная операция – инициализация (путем присваивания ссылки внешнему объекту, по сути ссылка инициализируется адресом объекта).

X& t = a; // все, что можно делать с a, можно делать и с t.

X* pa = new X();

X& t = *pa;

&t == значению указателя pa.

delete(&t);

Если f(X&t) – тогда инициализируется в момент вызова функции.

**Глава 3. Составные ТД.**

**П.3.1. Массивы.**

Последовательность однотипных элементов.

D x … x Dn

A[i] – операция индексирования. i – индексное выражение.

**Атрибуты массива:**

1) Базовый тип (тип элементов) – D
2) Тип индекса (i)
3) Диапазон индекса (длина)

**В разных ЯП:**

Связывание базового типа статически (везде).

Тип индекса – C/C++/C#/Java/Оберон – всегда тип int (статическое связывание).

Фиксируется нижняя граница значения индекса.

Длина – статическая и динамическая.

Динамическая – чисто-динамическая(можно изменить в любое время) и квазистатическая(значение получено динамически, но изменять нельзя).

Массив всегда непрерывная последовательность байтов. Отсюда возникает проблема распределения памяти. Можно длину сделать статической (жестко), оттого сделали квазистатической.

T[] a = new T[N]; //0..N-1

P,i,L..R(диапазон)

Стандартный Pascal:

Function SCAL( A,B: Arr): real;

Если диапазон для Arr от 1..N, то для массива от 0..N-1 работать будет не корректно(или вовсе не будет работать).

A[i] – компилятор выполняет квазистатический контроль. (не очень-то гибко)

Модула-2:

Объекты данных массива (переменные). Формальные параметры массива (открытые массивы).

**Понятие открытого массива:** зафиксирован базовый тип.

Обычный массив:

TYPE Arr = ARRAY Index of D;

TYPE Index = [1..N];

ARRAY of D;

Index = CARDINALL[0..N]; //В данном случае индекс задается статически.

```
PROCEDURE SUM (VAR A: ARRAY OF REAL): REAL;
VAR S: REAL; I: INTEGER;
    BEGIN
        S:=0.0;
        FOR I:=0 TO HIGH(A) DO
          S:=S+A[I];
        END;
        RETURNS
    END SUM;
```

Открытый массив: к нему применима функция HIGH(A) – максимальная длина без единицы.

Общий синтаксис объявления массива в Обероне

TYPE Arr = ARRAY N OF D;

TYPE NAT IS NEW INTEGER RANGE 1..MAX_INT (NEW – означает новый тип)

X: NAT;

I: INTEGER;

X:=I; и I:=X; - нельзя.

X:=NAT(I);

I:=INTEGER(X);

SUBTYPE POS IS INTEGER RANGE 0..MAX_INT;

X: POS; I: INTEGER;

I:=X; X:=I ~ X:=POS(I);

Неограниченный тип массива.

D,I – зафиксированы.

L..R – не фиксируемые левые и правая границы.

TYPE TARR IS ARRAY INTEGER RANGE 0..N REAL;

Атрибуты данных:

A'LENGTH – длина массива

A'FIRST = L

A'LAST = R

A'RANGE → RANGE A.FIRST..A.LAST

У неограниченных типов данных – атрибуты динамические, у ограниченных – статические.

**Зачем нужны неограниченные типы данных:** они нужны для выведения из них других типов данных.

X1: TARR;

X2: Arr; //Нельзя. Компилятор не может распределять память.

Надо: X2: Arr range 0..N;

```
Function SUM(A: Arr)return real is
S: real := 0.0;
    Begin
        for i in A'RANGE loop S:=S+A(i); end loop;

        return S;
    end SUM.
    a:=SUM(X2);
```

C: POS;

D: INTEGER;

C := D; //ok

C: POS:

D: INTEGER := -1;

C := D; //ошибка.

**Динамический массив (квазистатический).**

procedure P(N: integer) is

A: array(1..N) of real;

C#, Java:

T[] имя;

new имя[len];

int[] a = new int{1,2,3,4,5};

**Многомерные массивы.**

C, C++: int a[N1][N2];

C#: int [,] a = new a[N1,N2];

int [][] a; //ступенчатый массив, он же разрывный.

**Вырезка:** подмножество элементов массива.

Фортран:

A(N1,N2)

A(1,*) – 1-я строчка.

A(*,1) – 1-й столбец.

A(2..5,*), A(1..3,2..4) – прямоугольная вырезка.

Ада поддерживает только одномерные непрерывные вырезки.

**П.3.2. Записи (струтуры).**

struct name {поля}

record последовательность полей end;

Разные типы: тип ⇔ класс обертка (упаковка, распаковка)

C++: класс – обобщение структуры. (у структур в C собственное пространство имен)

Java: отсутствует запись – она не нужна.

C++: структуры – классы.

Отличия:

1) Имена
2) struct – public, class – private.

Delphi:

Новое – класс.

Старое – record.

C#: Типы – значения.

struct c {}

class F{}

C a = new C();

Отличия от класса:

1) память распределена как под типы значений
2) или классы. Есть классы-обертки.

class Point { int x,y; }

Point[] pointArray = new Point[1000]; //Неоправданные затраты памяти и времени. Здесь лучше использовать структуру.

**Лекция. Сергеев Николай.**

## Регулярные комбинировнные типы

-классы

-множества

-файлы

Файлы появились вследствие нужды программистов в средствах ввода и вывода во внешние устройства.

Впервые файлы были реализованы в Паскале, где они стали частью синтаксиса языка.(writeln(i : 8 : 4) - такую запись можно написать если в синтаксисе есть правила задания действительных чисел)

Однако правильно ли это? Оказалось, что нет. Со временем стало понятно, что ввод вывод – часть операционной системы. И правильнее средства ввода и вывода держать в стандартных библиотеках операционных систем, нежели встраивать в синтаксис языка, и при нужде в свой код эти библиотеки просто напросто подключать. Это делает код более переносимым(мобильным). Так поступили в C, C++.

**Множества** также впервые появились в Паскале.

Стандартный базис операций на множествами:

- set of T : T – множество
- a in T: принадлежит ли a множеству T?

- * S1+S2:  объединение множеств
- S1*S2 :  пересечение множеств
- S1-S2 :  разность двух множеств
- [x] + S:  добавить элемент x в множество
- S - [x]:  вычесть элемент из множества

Классический пример применения множества - нахождение простых чисел с помощью решета Эратосфена. Однако удобно ли это? В Паскале множества были реализованы в виде битовой шкалы - структуре данных, реализованной на каком-либо целочисленном типе данных, где каждым бит отвечал за присутствие или отсутствие данного числа в множестве. Вследствие того, что все целочисленные типы данных имеют не более 64 бит, такая структура могла оперировать с множествами небольшой мощности. Поэтому множества в Паскале были маленькими, ущербными.

Как еще можно реализовать множества? Вариантов много : Хэш - таблицы, сбалансированные деревья поиска, битовые шкалы, что ещё придумаете. Но надо понимать, что не существует универсальной реализации для любого случая.

Постепенно множество тоже ушло в стандартную библиотеку(STL C++: MAP, SET, MULTISET)

**Строки:** В стандарте Паскаля строка - упакованный массив символов. В С - строка массив символов, точнее чисел, который заканчивается нулевым символом. Хотя существовали операции для сравнения двух строк, все же строки рассматривались как частный случай массива. И что интересно, по мере развития языков программирования строки не ушли в стандартную библиотеку, а стали частью синтаксиса языка. Давайте рассмотрим причины произошедшего? В чем специфика строк?

Первое: операции. Сама частая операция над строками это их конкатенация, потом идет поиск подстроки в строке, вырезка части строки и тому подобное.

В массиве же самая частая операция - это операция индексации - обращение к определенному элементу массива. Было принято такое решение – сделать строки неизменяемыми с помощью индексации. И теперь единственный вариант изменить часть строки - предварительно её скопировать - механизм CopyOnWrite.

## Управление последовательностью действий.

Любая программа использует циклы, условные выражения, операторы условия и т.д. - всё это операторы, управляющие последовательностью действий. Здесь же рассматриваются такие вопросы, как порядок вычислений в арифметических выражениях, оператор GOTO, структуры ветвления.

До 67 - го года все программисты были наполовину математиками, все рисовали Блок - Схемы, переводили эти блок - схемы в двоичные коды, активно использовали оператор GOTO. Однако в 67 году вышла статься голландского учёного Дейскстры о вредности оператора GOTO, и много после этого поменялось. А именно, это событие зародило начало структурному программированию. Надо сказать, что в 67 - ом году впервые программистов ограничили, их как бы ущемели, сказали, что использовать GOTO - вредно, и надо именно упрощать структуру кода, ведь главное это изобретать, а не сидеть часами над кодом. В 68 году вышла статья

"Заметки о структурном программировании". В 66 году была доказана полнота множества операторов: присваивания + оператор while , то есть любую блок-схему можно было реализовать используя только два этих оператора. Теоретический базис был положен, и нашел свое отражение в таких языках как С и Паскаль, которые являются структурными языками. Структуру можно определить как черный ящик, у которого есть вход и есть выход, а то что происходит с входом заложено внутри черного ящика. У программистов принято в основном делить всю программу на три больших блока: подготовка - считывание данных и тому подобное,обработка, и собственно вывод.

Альтернативы GOTO:

* Ветвление.
* Циклы
* Процедуры
* Переходы - return, break, continue, goto
* Составной оператор(блок)

Напомню блок - это объявление + операторы.

Не во всех языках был реализован составной оператор. Во многих языках(АДА, Модула 2, Оберон) отказались от понятия составного оператора, там группа операторов явно замыкается специальным оператором (например

```
IF (TRUE)
        <OPERATOR1>
        <OPERATOR2>
                ...
        <OPERATORN>
        END)
```

То есть они придумали альтернативное решения для составного оператора - замыкание операторов(завершающий элемент).

1. **Оператор if**.

   Сразу же о проблеме, с которой столкнулись разработчики - вложенные if else. Решение else - прикрепляется к ближайшему if, если мы хотим избежать этого то нужно использоватьсоставной оператор. Надо различать понятие БЛОК и Составной оператор: блок – составляет область видимости, а в составном операторе нет области видимости.

Простое ветвление:

**C,C++:**

```
if (B) then
```

```
        Operator1
    else
        Operator2
```

**АДА:**

```
    if B then
        S1;S2;S3;...
    End if
```

**Оберон:**

```
    IF B THEN
        S1;S2;S3;S4...
    ELSE
        S1;S2;S3;S4...
    END
```

**SHELL**

```
    if
    ...
    fi
```

Иногда применяется многовариантное ветвление(многосложное)

```
    if B1 then S1
    else if B2 then S2
        else if B3 then S3
            else if B4 then S4...
```

Она эстетически не красивая, опытные программисты записывают её в столбик.

```
    if B1 then S1
    else if B2 then S2
    else if B3 then S3
    else if B4 then S4...
```

Теперь попробуете записать эту же самую конструкцию на языке в котором нет понятия составного оператора, там это выглядит ещё ужаснее, появляется много закрывающих операторов в конце.

2. Оператор выбора - дискретный случай.

**в Паскале:**

```
Case Expr of
        список вариантов, Вариант имеет вид const: оператор;
End
```

В чистом Паскале  нет else(default) константы.

**В C, C++, Java, C#:**

```
switch (expr) of {
        case 1: ...  break;
                ...
        case n: ... break;
        default: ... break;
}
```

Java не поддерживает GOTO, однако в данной конструкции она неявно используется. break - указатель на переход на конец структуры, если его нет, то дальше выполнится следующий case. Если в C++, C, Java - break было писать не обязательно после каждого case, то в C# стало обязательным(ошибка компиляции). Если в C# мы хотим после завершения данного case перейти на следующий надо использовать оператор перехода.

**Модула - 2:**

```
CASE EXPR OF
        1: ... |
        2..4 : ... |
        ELSE ...
END
```

**Ада:**

```
Case expr of
   when <список констант иди диапазонов> => оператор1
                ...
   when <список констант иди диапазонов> => операторN
```

when others => операторы

End case;

**3. Операторы циклы.**

Выделяют 4 вида цикла:

1. Пока

   While B loop .. End loop (**ADA**)

   WHILE B DO .. END (**Modula - 2)**

   While (B) S(**C, C++)**

   While B do S(**Pasca**l)

2. До

   REPEAT UNTIL B; (**Modula - 2**)

   do S while (B); (**C, C++)**

3. FOR for v:= r1 to t2 do S(**Pascal)**.

Возможно использование downto

for (подготовка; проверка на выход; действия после каждой итерации)(C++, C)

в Java, C# аналогично C, только там на каждой итерации осуществляется квазистатический контроль, то есть A[i] - должен на самом деле существовать, чтобы не вызвать ошибки.

FOR V:= E1 TO E2[STEP E3(целое значение)] DO END(Модула - 2) можно было задавать шаг.

Он мог быть как отрицательным так и положительным. E1, E2 - типы, к которым применима операция сложения и вычитания. В одно время сложилось тенденция, что цикл for - вообще, как таковой не нужен, можно обойтись другими видами цикла, но в 1993 - вышел Оберон - 2, который по идее является минимальным полным языком для написания любой программы, куда вошел и цикл for.

for v in <диапазон> loop .. end loop

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

for i in A'FIRST..A'LAST loop S:= S + A(i); end loop; (Ада)

Так как квазистатический контроль на каждой итерации цикла считается неэффективным в C# был придуман еще один вариант цикла for - особая форма цикла.

foreach (T o in C) S;

int a[];

foreach (int x in a) S = S + x;

C - произвольная коллекция.

тип T должен наследоваться от IEnumerable, в которой входит такой метод, как получить следующий элемент. В Java сей цикл был реализован в 2005 году.

4. Бесконечный цикл

```
LOOP
  ...  IF B THEN ... EXIT
END (Модула - 2)
```

```
while (1) {... break...}
for(;;) {... break ...}  (C++, C)
```

```
Loop
   ...
   when B => exit
   ...
end loop (Ada)
```

Раньше не было понятно применение бесконечного цикла, сейчас появилось много сервисов, которые работают по 24 часа в сутки, там к примеру для приёма сообщений и их обработки используется бесконечный цикл.

Не обязательно присутствие всех 4 видов цикла в языка, так в АДЕ отсутствует ДО.

Чаще всего работа с циклом строится по следующей схеме:

1. Подготовка к вводу

2. Обработка

3. На каждой итерации проверка на завершение.

В Циклах используются вспомогательные операторы break и continue.

*break* - оператор выхода из цикла

*continue* - переход на следующую итерацию цикла

*goto* - перейти на помеченное место в программе. В Модуле - 2, Java.

Обероне отсутствует. Нельзя по goto выйти за пределы функции или процедуры, в которой он находится.

**Лекция. Лихогруд Н.Н.**

Операторы перехода

goto

break;

continue; (в Модуле-2 EXIT)

return;

В современных языках программирования goto является только локальным

Для организации не локальных переходов:

setjmp, longjmp – В Си++ используются для обработки ошибок.

throw, trace – Обработка исключений

Также существуют специальные операторы для организации параллелеризма

Lock(obj) {блок} – Си#. Поток управления блокируется, если блок кем-то используется

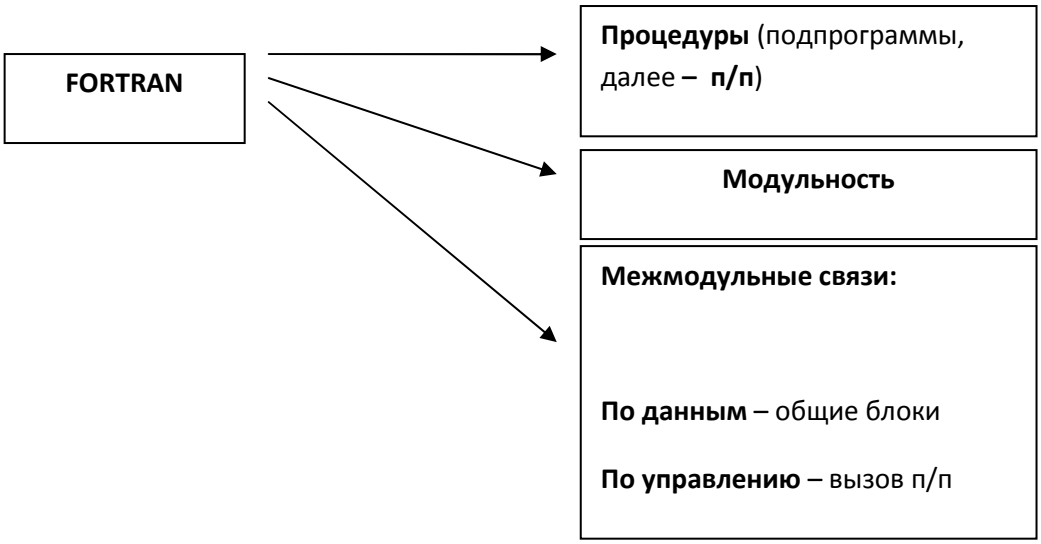
accept, select – Ада

**Базисы:**

Язык Ассемблера <—> Си <—> Си++ < —> Java, C#

Языки программирования в первую очередь различаются за счёт средств развития и их защиты.

Каков минимальный набор средств развития?




Этого уже достаточно для создания больших сложных программ, но без защиты новых абстракций

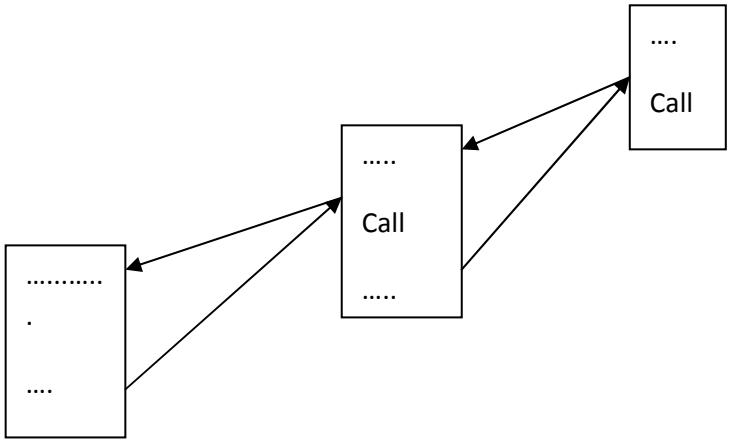
**Глава 5. Подпрограмма**

**п5.1 Потоки управления – подпрограммы и сопрограммы**

Управление входит через заголовок в блоке и возвращается в точку вызове, после выполнения тела.

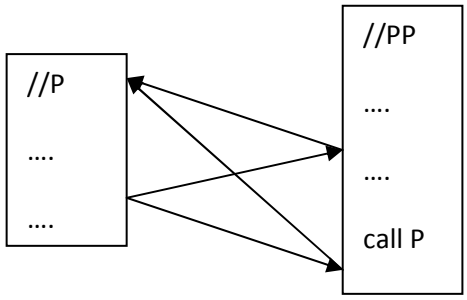
**CALLER** – вызывающий подпрограмму(надпрограмма)

**CALLEE** – вызываемая подпрограмма

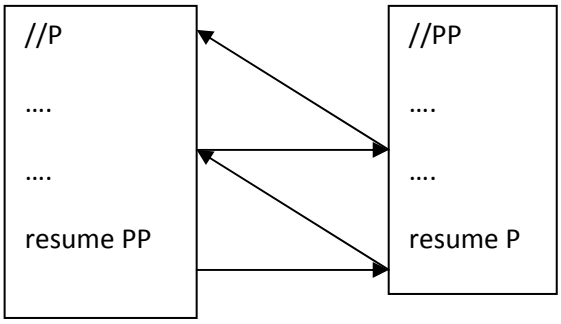



**SUBROUTINE** – подпрограмма

**COROUTINE** – сопрограмма




Неравенство




Равенство

Также нарушение априорного порядка выполнения команд может происходить при генерации исключений.

Впервые механизм сопрограмм был придуман для компилятора COBOL. Вспомните задание по Си++ в 4-м семестре, где нужно было написать транслятор модельного языка:

Лексический анализатор, Синтаксический анализатор, Генератор кода – всё это сопрограммы.

**Сопрограммы –** фактически квазипараллельные процессы.

**Модула-2**

Вызов сопрограммы аналогичен длинному переходу на некоторый абстрактный адрес, по которому находится команда сопрограммы, с которой нужно начать выполнение. Но, помимо этого, нужно ещё как-то запомнить адрес возврата и другую служебную информацию, передать входные параметры, наследуется часть контекста. Для этой цели Вирт в своём языке Модула-2 ввёл тип данных **ADDRESS** (то же самое, что и «void *»).

Этот тип данных является потенциальной дырой в системе безопасности, т.к. любой указатель «T *» автоматически приводится к «void *», и возможно обратное явное преобразование «T *» = (T *)«void *». Для Вирта было неприятной неожиданность то, что программисты часто использовали тип данных ADDRESS.

Строгость типизации зависит от возможностей преобразования.

**Типы** – непересекающиеся области эквивалентности, определяемые операциями на объектами этих областей.

Итак, в Модуле-2 вызов сопрограммы имеет такой вид:

NEWPROCESS(P, C,N);

Где P – процедура без параметров типа PROCEDURE, который является встроенным, C – переменная типа ADDRESS. N - размер области для «запоминания» информации. Область начинается с адреса C

PROCEDURE NEWPROCESS(P : PROCEDURE; VAR C : ADDRESS; N : INTEGER);

Передача управления от одного процесса другому на уровне сопрограмм осуществляется процедурой "Передать управление от процесса P1 процессу P2". В Модуле-2 эта процедура выглядела как

PROCEDURE TRANSFER(VAR P1,P2 : ADDRESS);

При этом в переменную P1 записывается запись реактивации этого процесса, а значение переменной P2 определяет запись активации процесса P2.

RESUME; – оператор языка.

Маленькое замечание:

Изначально Вирт вместо **ADDRESS** использовал тип **COROUTINE**, теперь понятнее? Тип **COROUTINE** был похож не структуру**.**

В современных языках сопрограммы трансформировались в понятие потока.

.Net Thread Квазипараллельный поток

C# 2.0:

```
foreach(T x in C)
```

Тип T должен реализовывать интерфейс IEnumerable. Этот интерфейс содержит метод GetEnumerator(), который возвращает объект некоторого типа, который должен реализовывать интерфейс IEnumerator со свойствами Current,методом Reset и методом bool MoveNext(). Любой класс, поддерживающий интерфейс IEnumerable должен содержать класс, поддерживающий IEnumerator.

yield-операторы в C# 2.0:

```
yield return <expression>;
yield break;
```

Итератор – процесс(сопрограмма), выдающий последовательно очередные элементы коллекции тогда, когда они понадобятся. yield-оператор используется в блоке итератора для предоставления значения объекта перечислителя или для сообщения о конце итерации. Т.е. это не простой «return» или «break», а оператор, совмещающий в себе дополнительно работу по переходу между сопрограммами (от процесса-итератора в основной процесс). Выражение expression вычисляется и возвращается в виде значения объекту перечислителя; выражение expression должно неявно преобразовываться в тип результата итератора.

```
ublic class List
```

```
{
    //using System.Collections;
    public static IEnumerable Power(int number, int exponent)
    {
        int counter = 0;
        int result = 1;
        while (counter++ < exponent)
        {
            result = result * number;
            yield return result;
        }
    }

    static void Main()
    {
      // Display powers of 2 up to the exponent 8:
        foreach (int i in Power(2, 8))
        {
          Console.Write("{0} ", i);
        }
    }
}
/* Output:
 2 4 8 16 32 64 128 256 */
```

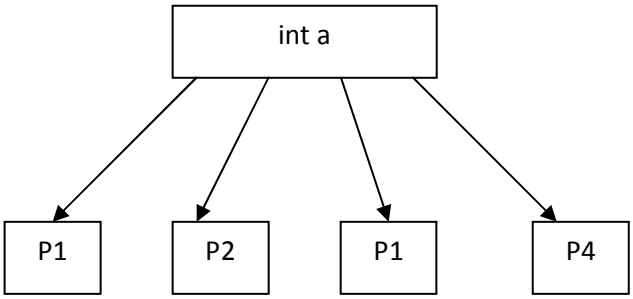
Генеральная линия развития C# - добавление элементов функционального программирования

**п.5.2 Потоки данных в подпрограммах**

- Через глобальные данные
- Через параметры

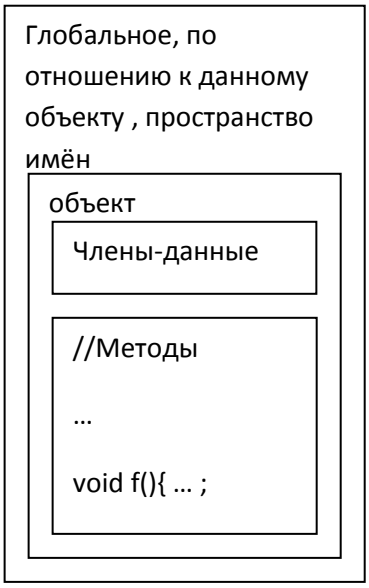
**Побочный эффект действия процедур и функций** – изменение значений глобальных переменных и данных, а так же модификация данных, глобальных по отношению к самой процедуре\функции.

**Глобальная переменная** – переменная, которая видна везде




Возможен конфликт связей

В объектно-ориентированной парадигме:




Сначала «i» ищется в теле функции, потом в членах-даннных, и только после этого вне объекта

**Виды формальных параметров(семантика)**:

- Входные (in) – должны быть определены до входа
- Выходные (out) – должны быть определены к моменту выхода
- Вх/Вых(InOut) – и то и другое

**Способы передачи**

**Способ передачи –** способ связывания фактических и формальных параметров:

- По значению (семантика - in)
- По результату (семантика – out)
- По значению/результату (семантика – InOut)
- По адресу(по ссылке) (семантика - любая)
- По имени

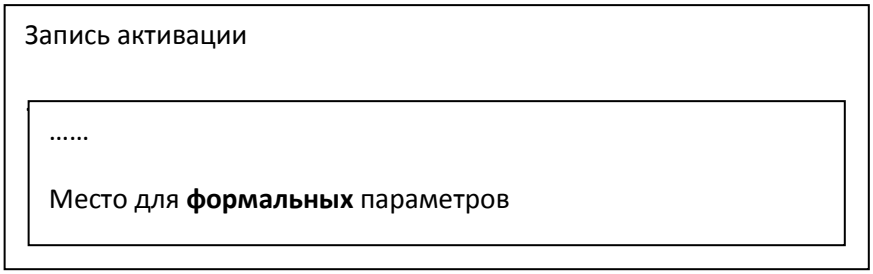
**Ада83:**

Квалификаторы: in, out, inout

```
Procedure P(int X:T;inout Y:T;out Z:T);
```

X может быть выражением типа T. Компилятор может вставлять квазистатические проверки. Эффект процедуры – модификация Y и Z. Каков **способ передачи** определяет компилятор( что не есть хорошо, т.к. различные компиляторы в одной и той же ситуации могут выбрать разные способы передачи, что приведёт к различной работе программ).

Пользователь определяет **семантику** использования.




Формальные параметры – те, которые объявлены в заголовке подпрограммы и используются в теле. Большинство ЯП переменные, которые объявлены в заголовке, считают частью тела.

Фактические параметры – те, которые передаются в подпрограмму при её вызове.

При вызове подпрограммы фактические параметры, указанные в команде вызова, становятся значениями соответствующих формальных параметров, чем и обеспечивается передача данных в подпрограмму.

| Способ передачи | Семантика | Что делается |
|---|---|---|
| По значению | in | При вызове подпрограммы фактические параметры копируются в Запись Активациии |
| По результату | Out | При выходе из подпрограммы из записи активации формальные параметры копируются в фактические |
| По значению и результату | inout | При вызове подпрограммы фактические параметры |

| | | |
|---|---|---|
| | | копируются в Запись Активацаии<br><br>При выходе из подпрограммы из записи активации формальные параметры обратно копируются в фактические |
| По Адресу | Любая | При передаче по Адресу в Запись активации копируется адрес фактического параметры. Именование и разыменование происходят автоматически |

- В **Фортране** обычно параметры передаются по адресу, но когда передаётся простой объект данных, чтобы не происходило лишних операций разыменования, можно передавать «по значению и результату»( /<параметр>/)
- В **Аде-83** способ передачи зависел от компилятора, т.е. компилятор сам выбирал способ передачи в зависимости от ситуации. Пример программы, в которой это важно:

```
Procedure P(inout X : T; inout Y : T)
   X := <newvalueX>;
   <возбуждение исключения>
   Y := <newvalueY>;
End P;
….
P(a,a);
```

Предположим, что оно не обрабатывается. Тогда запись активации пропадает. Значит, если была передача по ссылке, то значение «а» изменится, а если по значению и результату, то не изменится, т.к. копировать в «а» будет нечего.

**Энтропия** – явление, при котором программа может выдавать различные результаты при одних и тех же исходных данных. Если в программе есть энтропия, то это очень плохо. Очевидно, что при программировании на Ада риск энтропии значительно повышается, т.к. не известно какой способ передачи выберет в этот раз компилятор.

- В **Ада-95** – по значению, по ссылке
- В **Си** – не определяется семантика использования. Способ передачи только по значению
- В **Си++** – по значению, по адресу (ссылке). Контроль семантики: in – ссылка константная, out, inout – не константная

```
void f( T &); //Компилятор считает, что f будет менять значение => константный объект
//передавать нельзя
```

Это указание для компилятора, чтобы он следил за соблюдением семантики

```
class X
{
   void f(); //неконстантная функция;
}
……..
const X a;//     X * const this; внутри методов
a.f();// ошибка!!
```

```
class X
{
   void f() const; //константная функция;
}
……..
const X a;//     X * const this; внутри методов
a.f();// Правильно!!
```

- **C#, Java**

```
void f( T x) {….;}
……
T a; // a – ссылка, если T – объект
f(a); // передаётся ссылка
```

Оба языка поддерживают идею примитивных типов (которые в C# являются подмножеством типов-значений — value types), и оба для трансляции примитивных типов в объектные обеспечивают их автоматическое «заворачивание» в объекты (boxing) и «разворачивание» (unboxing) (в Java — начиная с версии 1.5).

**object** – предок всех классов. => Объект любого класса неявно приводится к типу object.

```
object o;
```

```
int i;

o = i;

i = o; // ошибка!
```

Автоупаковка, Автораспаковка:

```
o = i; //  o = new Integer(i) – Java

       //   o = new Int32(i)    – C#
```

Так что если функция объявляется как void f(object o), то в неё можно передавать любой объект( для примитивных типов будет производится автоупраковка\распаковка)

- **Java:**

В Java параметры метода передаются только по значению, но поскольку для экземпляров классов передаётся ссылка, ничто не мешает изменить в методе экземпляр, переданный через параметр. Структур в Java нет.

Передача объектов примитивных типов в методы «как по ссылке» выполняется через классы-обёртки:

```
void f(Integer X){…X = ….; }

…..

int i = 1;

Integer px = new Integer(i);

f(px);

i = px;
```

Integer – **класс-обёртка** для примитивного типа «int». Суть способа – преобразовать объект примитивного типа в объект класса и работать внутри функции с объектом класса.

- **C#**

для C# создана более развитая терминология**:**

**Тип-значение(value type)** – тип, объекты которого передаются по значению. Если где-то нужен объект такого типа, то отводится место под сам объект. типами-значениями являются простые(примитивные) типы данных и структуры

**Референциальный тип** – тип, объекты которого передаются по ссылке. Если где-то нужен объект такого типа, то отводится место под адрес. Референциальными типами являются  классы(любой массив – наследник класса Array, строка это объект класса String, и т.д.)

В принципе, в C# можно передавать объекты простых типов в функции с помощью классов-обрёток, но C# также поддерживает явное описание передачи параметров по ссылке – ключевые слова ref и out. «ref» реализует семантику inout, а «out» реализует семантику out. При использовании out компилятор вставляет квазистатический контроль на наличие в методе присваивания значения, зато не требует инициализированность фактического параметра перед вызовом метода.

```
void f(ref int x){x = -3;}

….

int a = 0;

f( ref a); // a будет передано по ссылке, если бы объект «a» был структурой, то он так же передался бы по ссылке //
```

**Разрешать или не разрешать функции с переменным количеством параметров?**

**Си**

В Си можно было создавать функции с переменным количеством параметров при помощи библиотеки «stdargs»: va_list, va_start, va_end, va_next и т.д

**Си#**

```
void f(param object [] args){…. args [0] = …; }

void g(object [] args){…. args [0] = …; }


….

f(a , b);

f();
```

```
f(1);
f(new X[]{new X(). new X()});// Ошибка!!
f(new X(), new X()); // Правильно!
g(new X[]{new X(). new X()});// Правильно!!
```

**java**

```
void func(Object a[])
{
    for(int i = 0;i < a.length; i++)
    System.out.println(a[i]);
}
```

Для **java 1.5** – func(1,2,3, new Object(), "word");

Для **java 1.4** – func(new Object[] {1, 2, "some string"});

**Передача параметров по имени**

**Алгол-60**

Передаём сам объект «как он есть» . Фактически передаётся его идентификатор как строчка. Далее в теле процедуры идентификатор формального параметра заменяется на идентификатор фактического

Обоснование невозможности написание процедруры **swap** на Algol-60:

```
procedure Swap(a,b); //a, b передаются по имени
Begin
    T tmp;
     tmp := a; a:= b; b := tmp;
End;
```

….

```
swap(i, A[i]);
```

```
T tmp;
tmp := i;
i := A[i];
A[i] := tmp;// A [ A[i] ] := i; неправильно!!!
```

```
swap(A[i], i);
```

```
T tmp;
tmp := A[i];
A[i] := i;
i:= tmp;//  i := A[i] – всё правильно
```

Решение проблемы: С каждым параметром передавалась процедура **thunk**, которая выполнялась при каждом обращении к фактическому параметру.

**Параметры по умолчанию**

В C++ можно задавать параметры по умолчанию: void f(int a = 4){ …;}

В C# эту возможность убрали. Вместо этого предлагается использовать перегрузку:

```
f(){ f(1, 2); }
f(int i ){ f(i , 2) ;}
f(int i,  int j ) { … ; }
```

**Лекция. Ульянов А.В.**

**Подпрограммы. Типы данных.**

Ада 83, Java – нет.

**П.1. Передача подпрограмм, как параметров.**

Присваивание [:=]

Вычисление [()]

Ада 83: generic не только для ТД, но и для процедур, функций.

//generic – параметризация.

c/c++: typedef void (*f)(int);

Отсюда, процедурный тип – указатель.

Проблема в том, что  в Ада 83 и Java отказались от указателей, т.е. и от П/П ТД.

//В Java целиком, в Ада частично.

Ада 95:

type Func_Pointer is access function (L,R: float) return Boolean;

function Compare (X,Y: float) return Boolean … end Compare;

F: Func_Pointer

F:=Compare'access

Модула-2, Оберон:

TYPE FUNC_INT = PROCEDURE (L,R: REAL): BOOLEAN;

PROCEDURE COMPARE (X,Y: REAL): BOOLEAN;

VAR F: FUNC_INT;

F:=COMPARE;

**П.2. Функции обратного вызова.**

Для реализации ООП(динамическое связывание), например, в xlib → xt → Motif;

События → Реакция.

Чистые ООЯП: Любой ОД принадлежит классу.

//Автоупаковка, автораспаковка => Простой ТД ⇔ объект.

Java:

Существуют класс (интерфейс) Interface, метод Integrate, Virtual подынтегральная функция. Наследует и заменяет подынтегральную функцию.

C#: Делегаты(расширение П/П ТД)

C++: class X

```
{
    void f(int);
    int i;
}
X::*int p; //Указатель на член
```

Смещение относительно начала.

```
Typedef void (*f)(int);
Class X
{
    static void p(int);
}
f Fptr;
Fptr = x::p;
```

Если virtual: this, таблица виртуальных функций, смещений,

иначе: this, адрес.

```
class X
{
    public delegate void delf(int);
    delf g;
}
```

=; +=; -=; ()

//присваивание, добавление, удаление, выполнение.

this хранится вместе с указателем на функцию. Делегатом может быть и статический и нестатический член класса.

1. Параметр функции – делегат.
2. Подписка – рассылка.

EventProducer, EventConsumer.

Почта:

```
public delegate void OnNewMail (object o);
OnNewMail onNewMail;
EventProducer ep = new EventProducer();
```

EventConsumer ec = new EventConsumer();

ec.onNewMail += new OnNewMail(…); //подписка

OnNewMail(MailMessage);//рассылка.

Незащищено. =>

public delegate void OnNewMail(object o);

public event OnNewMail onNewMail;

- ⇨ Только += или -=.

**Глава 6. Логические модули.**

ТД = (множество операций = набор подпрограмм) + (множество значений = набор структур данных)

Модуль = контейнер

**Класс:**

1) ТД
2) Контейнер

**Модуль –** набор взаимосвязанных ресурсов, которые служат для использования других модулях.

Модуль-ресурсы = ОД, ТД, П/П.

Интерфейс = определение ресурсов + реализация.

**Межмодульные связи.**

**Пространство имен** в ООЯП заменена на модули.

М-2:

1) Главный модуль //Один
2) Библиотечный модуль
3) Локальный модуль //параллельное программирование.

**Клиент ← Библиотечные модули → Сервис**

**Клиент ⇔ Сервис.**

Глобальное пространство имен => видимость для всех

Непосредственная видимость: имя использует ASIS.

Потенциальная видимость: имя с уточнением.

DEFINITION MODULE имя;

Определение ресурсов.

END имя;

IMPLEMENTATION MODULE имя;

Реализация всех процедур/функций из DEF + дополнительные ресурсы.

END имя;

TP, DELPHI:

init имя;

interface

…

implementation

End имя;

Все имена экспортируются в глобальное пространство имен.

ПОТЕНЦИАЛЬНО.

IMPORT M; //Первые в модуле => клиент.

IMPORT InOut; //Видимы потенциально.

InOut.WriteString(“counter”);

InOut.WriteInt(out);

InOut.Writeln;

FROM InOut Import Writeln, WriteInt.

Видны непосредственно!

TP, Delphi: uses список имен; //видимы непосредственно.

Оберон: оставлен только библиотечный модуль.

MODULE M;

…

ENDM;

Принцип РОРИ: Зазделение Определение Реализация Использование.

* - экспорт имени.

//имя *

MODULE ST

TYPE STACK* = …

PROCEDUR PUSH* (VAR S: STACK); … PROCEDURE P…

END. => псевдомодуль.

DEFINITION ST;

TYPE STACK = …

PROCEDURE PUSH = …

D st.

=> IMPORT список_имен_модулей.//В Обероне только так.

=> ST.STACK //интерфейс.

Или FROM ST IMPORT R; => R//реализация.

Реальное программирование: работа с древообразным модульным проектом:

1) Сверху вниз
2) Снизу вверх

Ада: библиотечный модуль ⇔ пакет(спецификация, тело)

Спецификация:

package M is

Определение типов, переменных, констант, заголовков процедур

end M;

Тело:

package body M is

Реализация всех процедур и функций

end M;

package STANDART; //пакет стандартных имен.

Пользовательские пакеты встраиваются в STANDART.

Любой пакет можно вложить в другой.

STANDART

package M1 is

package M1.2 is

…

end M1.2; end M1;

package M2 is

package M2.1 is

package M2.2 is

…

end M2.2; end M2.1; end M2;

Тела вкладываются также, как и спецификации!

Более близкое описание скрывает менее близкое описание одинаковых имен.

Неявный импорт: вместе с одним именем неявно импортируется другое.

C++: T operator+(T x1, T x2);

Ада: function “+” (x1, x2: T) return T;

Импорт: use M;//все имена становятся явно видимыми.

Java: import имя_пакета.имя_класса; или имя_пакета.*;

C#: using …

Переименование: a renames b

M-2:

```
DEFINITION MODULE STACKS;
TYPE STACK*=RECORD
B:ARRAY[0..N] of T;
TOP: [0..N];
END;
PROCEDURE PUSH*(VAR S: STACK; X: T);
PROCEDURE POP*(VAR S:STACK; VAR X: T);
…
isEMPTY*
isFULL*
INIT*
PEEK*
…
VAR DONE*: BOOLEAN; //
END STACKS;
```

**Глава 7. Инкапсуляция и абстрактный тип данных.**

ТД = МнОП + МнЗн;

РОРИ: алгоритмы инкапсуляции.

АТД = МнОП;

**Инкапсуляция:**

Единица инкапсуляции – тип.

Атом инкапсуляции – отдельные поля, члены или весь тип.

М-2: скрытые ТД

```
DEFINITION MODULE STACKS;

FROM MYTYPES IMPORT T;

TYPE STACK; (*скрытый ТД*) //компилятор не знает, что это.

PROCEDURE PUSH

        INIT

        DESTROY

END STACKS.
```

DEF → транслируется в SYM(таблица символов) и OBJ(реализация).

- ⇨ STACK ~ INTEGER или POINTER
- ⇨ TYPE STACK = POINTER TO STACKREC

STACKREC = RECORD … END

- ⇨ := (shallow, copy), = (равно), # (не равно)

Ада 83:

приватный ТД (~скрытый ТД)

ограниченно приватный ТД

- ⇨ package stacks is type stack is private;
  … - описание всех заголовков.
  private
  … - описание всех приватных структур данных.
  :=, =, /=
- ⇨ ограниченно приватный:
  type T is limited privaty;
  … - оперции.
  private type T is … ;

**Лекция. Сергеев Николай.**

**Межмодульные связи**

**Определения:**

***РОРИ*** – метод, когда реализация скрыта от пользователя. Пользователю доступен лишь интерфейс , а реализация инкапсулирована.

***Тип данных*** = множество значений + множество операций

***Абстрактный тип данных*** = множество операций

***Контейнер*** - средство создания новых типов данных. В некоторых языках контейнером выступают модули (Модула – 2, Ада, C#(namespace), Java(package)), в других - классы, структуры(C++, C). Delphi – гибрид, содержащий как модули, так и классы.

***Атомы инкапсуляции*** – минимально возможные данные, которые можно скрыть от пользователя. Для всех класс – ориентированных языков атомы инкапсуляции – это поля класса, а таких языках как Ада, Модула – 2 – минимальным атомом является весь класс, то есть хороший программист на таких языках всегда использует абстрактные типы данных.

**Модуль** - независимая единица трансляции. Подключить модуль чаще всего означает заимствовать написанный кем – то код, сохраненный в модуль. Множество языков имеют кучу встроенных библиотечных модулей, реализация которых опирается на РОРИ. Структура модулей как правило древовидная.

***Видимость подразумевает*** *возможность обращения с объектом, то есть , к примеру, знаем его имя и тип.* *непосредственно видимы в глобальном пространстве имен только модули Имена полей из модуля видимы только потенциально.*

## 3 подхода организации проектирования модулей

Без сомнения, главнейшее условие успешного создания крупных программ заключается в применении надежных методов проектирования. Широкое распространение при написании программ получили следующие три метода: нисходящий (сверху вниз), восходящий (снизу вверх) и специальный (на данный конкретный случай). В случае *нисходящего метода* вы начинаете созидательный процесс с программы высокого уровня и спускаетесь до подпрограмм низкого уровня. *Восходящий метод* работает в обратном направлении: вы начинаете с отдельных специальных подпрограмм, постепенно строите на их основе более сложные конструкции и заканчиваете самым верхним уровнем программы. *Специальный подход* не имеет заранее установленного метода, так сказать комбинация восходящего и нисходящего проектирования( часто сводится к реализации базисной части программы, а потом к постепенному расширению её функциональности , сопровождается использованием «заглушек» в местах нереализованных частей)

Поскольку С является структурированным языком программирования, то лучше всего он сочетается с нисходящим методом проектирования. Подход сверху вниз позволяет производить ясный, легко читаемый программный код, который в дальнейшем не вызовет трудностей и при сопровождении. К тому же данный подход помогает прояснить и сделать понятной всю структуру программы в целом до кодирования функций более низкого уровня. Такой подход позволяет уменьшить потери времени, обусловленные неудачными или ошибочными начинаниями.

RAD подход ( rapid проектирование ) – быстрое написание прототипа программы, для того чтобы заказчик мог оценить свой будущий проект на начальной стадии проектирования, и внести соответствующие изменения в случае надобности.

Модульная структура языка АДА

Библиотечный модуль в АДЕ это пакет, видна аналогия с Модулой – 2.
Пакет - это средство, которое позволяет сгруппировать логически связанные вычислительные ресурсы и выделить их в единый самостоятельный программный модуль. Под вычислительными ресурсами в этом случае подразумеваются данные (типы данных, переменные, константы...) и подпрограммы которые манипулируют этими данными. Характерной особенностью данного подхода является разделение самого пакета на две части: спецификацию пакета и тело пакета. Причем, спецификацию имеет каждый пакет, а тело могут иметь не все пакеты.

Спецификация определяет интерфейс к вычислительным ресурсам (сервисам) пакета доступным для использования во внешней, по отношению к пакету, среде. Другими словами - спецификация показывает "что" доступно при использовании этого пакета.

Тело является приватной частью пакета и скрывает в себе все детали реализации предоставляемых для внешней среды ресурсов, то есть, тело хранит информацию о том "как" эти ресурсы устроены.

Необходимо заметить, что разбиение пакета на спецификацию и тело не случайно, и имеет очень важное значение. Это дает возможность по-разному взглянуть на пакет. Действительно, для использования ресурсов пакета достаточно знать только его спецификацию, в ней содержится вся необходимая информация о том как использовать ресурсы пакета. Необходимость в теле пакета возникает только тогда, когда нужно узнать или изменить реализацию чего-либо внутри самого пакета.

Средства построения такой конструкции как пакет дают программисту мощный и удобный инструмент абстракции данных, который позволяет объединить и выделить в логически законченное единое целое данные и код который манипулирует этими данными. При этом, пакет позволяет программисту скрыть все детали реализации сервисов за развитым функциональным интерфейсом. В результате, структурное представление программного комплекса в виде набора взаимодействующих между собой компонентов облегчает понимание работы комплекса в целом и, следовательно, позволяет облегчить его разработку и сопровождение.

Необходимо также заметить, что на этапе начального проектирования системы можно предоставлять компилятору только спецификации, обеспечивая детали реализации только самых необходимых элементов. Таким образом проверка корректности общей структуры проекта осуществляется на ранней стадии, когда не потрачено много усилий на разработку реализации отдельных элементов, которые позже придется переделывать (что, к великому сожалению, в реальной жизни происходит достаточно часто).

Общий вид:

*Package M is*
*  //определение(типов переменных констант заголовки процедур)*
*End M*
*Package body M is*
*  //реализация процедур и функций*
*End M*

*Рассмотрим поподробнее отдельно спецификацию и реализацию*

*Спецификация:*

```
package M is

    type A_String is array (Positive range <>) of Character;

    Pi  : constant Float := 3.14;

    X   : Integer;

    type A_Record is
        record
          Left  : Boolean;
          Right : Boolean;
        end record;

    -- примечательно, что дальше, для двух подпрограмм представлены только
    -- их спецификации, тела этих подпрограмм будут находиться в теле пакета
```

```
    procedure Insert(Item : in Integer; Success : out Boolean);
    function Is_Present(Item : in Integer) return Boolean;

end M;
```

Для подключения пакета используется оператор with: with <Имя пакета>. В случаях, когда использование полной точечной нотации для доступа к ресурсам пакета обременительно, можно использовать инструкцию спецификатора использования контекста **use**. Это позволяет обращаться к ресурсам которые предоставляет данный пакет без использования полной точечной нотации, так, как будто они описаны непосредственно в этом же коде. Для обращения к функции из пакета используется оператор «.»: <имя пакета>.<имя функции>

Пример:

```
with M;

procedure P is

    My_Name : M.A_String;
    Radius  : Float;
    Success : Boolean;

begin
    Radius := 3.0 * M.Pi;
    M.Insert(4, Success);
    if M.Is_Present(34) then ...
        . . .
end P;
```

Реализация(тело пакета)

Тело пакета содержит все детали реализации сервисов, указаных в спецификации пакета. Схематическим примером тела пакета, для показанной выше спецификации, может служить:

```
package body M is

    type List is array (1..10) of Integer;
    Storage_List : List;
    Upto         : Integer;

    procedure Insert(Item     : in   Integer;
                     Success  : out  Boolean) is
    begin
        . . .
    end Insert;

    function Is_Present(Item : in Integer) return Boolean is

    begin
        . . .
    end Is_Present;


begin                   -- действия по инициализации пакета
                        -- это выполняется до запуска основной программы!
    for I in Storage_List'Range loop
        Storage_List(I) := 0;
    end loop;
```

```
        Upto := 0;
    end M;
```

Все модули линейные в МОДУЛЕ - 2, но В аде могут вкладываться друг в друга, пример может быть не в тему( Головин ), но чтобы получше понять что  такое вложенность рассмотрим вложенные процедуры в Паскале:

```
procedure p1
var x2, x3;
  procedure P2
    var x1
  end
end
```

Рассказ про вложенные области видимости мы отбросим.

Таким образом ещё раз повторяюсь в АДЕ пакеты могут быть вложены друг в друга. В других лекциях рассказывалось про существование в некоторых языках особого класса – базового абсолютно для всех классов, то есть все классы как бы непроизвольно являются порожденными от него. В Аде аналогичная ситуация с пакетами.  в АДЕ есть пакет

*packet STANDART*

все другие пакеты вкладываются в СТАНДАРТ

Пример вложенности в языке АДА:

```
Standard
   package m1 is
      package m2 is
     end m2
   end m1
  package m3 is
      package m4 is
     end m4
 end m3
```

Подробный рассказ об области видимости пакета оставим на лучшие времена в виду интуитивной понятности(область действия имени пакета начинается с  объявления пакета, например, в конце Standart видно m1, m3, m1.m2, m3.m4 точка обязательна). В виду вложенности пакетов, для вложенных пакетов существует также понятие вложенных областей действия. Здесь все аналогично с ситуаций вложенных областей видимости в С++.

перегрузка функций, перегрузка операций:

```
 T operator + (T x, T y);
```

перегрузка операций нам знакома с языка С++, важной особенностью перегрузки операций является, то что количество операндов должно совпадать с количеством операндов у соответствующей  операции, то есть для “+” – количество операндов всегда равно 2.

***в Аде:***

```
    function "+" (x2 , x3:T) return T;
```

В Аде возникает проблема перегруженных операторов и функций, которые описаны во внутренних пакетах, так как они не видны в внешних пакетах. Вследствие этого ввели новый оператор

```
    use M;
```

После применения этой конструкции все имена из спецификации пакета M становятся видимыми.

Может возникнуть конфликт имен при описании одинаковых переменных в различных модулях. При конфликте имен «виноваты оба», поэтому в точке начала перекрытия перекрываемые переменные с одинаковым именем перестают быть видимыми.

*Переименование:*

Ада, Модула - 2 предоставляет программисту возможность осуществлять переименования. Следует заметить, что переименование иногда вызывает споры в организациях программирующих на Аде. Некоторым людям переименование нравится, а другим - нет. Существует несколько важных вещей, которые необходимо понять:

- Переименование не создает нового пространства для данных. Оно просто создает новое имя для уже присутствующей сущности.
- Не следует постоянно переименовывать одно и то же. Этим можно запутать всех, включая самого себя.
- Переименование необходимо использовать для упрощения кода. Введение нового имени, в некоторых случаях, делает код более легко читаемым.

Пример:

```
with  Ada.Text_IO;
with  Ada.Integer_Text_IO;

procedure Gun_Aydin is

    package TIO renames  Ada.Text_IO;
    package IIO renames  Ada.Integer_Text_IO;


with  Graphics.Common_Display_Types;
package CDT renames Graphics.Common_Display_Types;
```

полезно при переименовании перекрывающихся имен в модулях.

**Лекция. Лихогруд Н.Н.**

**Классы**

Принципиальное отличие класса от модуля заключается в том, что класс – это тип данных, а модуль нет. Но во многих вещах они похожи.

**Тип Данных = Структура Данных + Множество Операций над этими данными**

В C#, Java всё является классами или находится в классах в качестве статических челнов. Даже математические функции находятся в классе – System.Math. И для вызова функции cos(x) требуется написать Math.Cos(x);

Наибольшее сходство между классом и модулем достигается если класс содержит только статические методы и поля. При этом такой класс, как правило, реализуется в виде модуля.

**п1. Понятие членов класса**

синтаксис в **C++.Java,C#:**

```
class Name
{
    ….
    Определение членов класса
    …..
}
```

В Си++ допускается вынесение определений, т.е. В Си++ можно члены класса лишь объявлять. В Java, C# все определения должны быть внутри класса

синтаксис в **Delphi:**

```
type T =
class (наследование)
   обявление членов класса
end;
```

Члены класса:

- Члены-данные
- Члены-типв
- Члены-функции(методы)

**Чем члены-функции отличаются от обычных функций?**

Такой функции при вызове всегда передаётся указатель «**this**»(в Delphi «**self**») на объект в памяти, от имени которого вызывается функция.

| **Java,C#,** | **C++.** | **Dephpi** |
|---|---|---|
| T x;<br>x = new T(«параметры конструктора»);<br><br>В первой строчке определяется ссылка на объект (выделяется память для хранения ссылки), место в динамической памяти под объект не отводится. Во второй непосредственно отводится место в динамической памяти («куче») для объекта и адрес присваивается ссылке на объект. | T x;<br>T x(«параметры конструктора»);<br>T x = T(«параметры конструктора»);<br><br>В этих определениях выделяется место не под ссылку на объект, а под сам объект (не в динамической памяти). Чтобы выделить место в динамической памяти, нужно использовать операцию «new» | x : T<br><br>x – ссылка, её ещё надо проинициализировать. |

Ещё раз о структура в Си#:

```
struct Name // не имеет ссылочной семантики
{
    ….
    <определения членов>
    ….
}
…….
 Name x;// память отводится здесь же
 x = new Name(<параметры>); // Явное выделение динамической памяти
 Name [] arr = new Name[50];// В памяти отведётся место под 50 объектов типа «Name»,
//а не под 50 указателей
```

От структур нельзя наследовать. Сами структуры неявно наследуются от класс System.Struct

**Обращение**

«имя_объекта».«имя_члена»

Для членов-данных по определению выполнение операции обращения к элементу класса «.»(class member acces operator) является просто смещением относительно адреса «this»(«self»)

```
class X
{
    ….
     объявления \ определения для Си++, определения для C#, Java
    …..
 }
```

Внутри всех функций-членов члены класса видимы непосредственно.

Однако формальные параметры метода класса относятся к блоку метода и могут закрывать члены класса. Тогда доступ к членам классам можно получить через указатель this при помощи операции обращение к элементу класса «.»: this.«имя члена».

В **Delphi** формальные параметры функций-членов находятся в той же области видимости, что и все остальные члены класса и, следовательно, не могут с ними совпадать.

**Члены-типы**

STL – набор соглашений. Одно из соглашений – контейнеры должны сами себя описывать. В Delphi членов-типов нет.

**Статические члены**

SmallTank **class variable**

**instance variable**

**class variable** – члены-данные класса, которые принадлежат всем экземплярам класса.

**instance variable –** принадлежат экземплярам класса, у экземпляра своя **instance variable.**

С точки зрении Си++ статические члены классов отличаются от глобальных только областью видимости.

```
class T
{
```

```
    ….
    static int x;
    static void f();
    ……
}
…
T t;
 t.x;//операция доступа
 //или, что тоже самое
 T::x//операция разрешения видимости

 t.f();//операция доступа
 //или, что тоже самое
 T::f() //операция разрешения видимости
```

В C#,Java,Delphi обращение к статическим членам происходит только через тип класса.

В статических функциях нет ths/self => в них нельзя использовать нестатические члены класса, т.к. по умолчанию все обращения к нестатическим членам идут через указатель ths/self

В C#, Java статические члены используются намного чаще, чем в Си++, Delphi, т.к. в C#, Java нет глобальных функций и переменных:

```
public class Name
{
   ….
   public static void Main(string [] args)
   ….
}
```

**Паттерн Singleton (Одиночка)**

Если установка соединения между клиентом и сервером слишком трудна, требует больших затрат ресурсов и времени, то неэффективно каждый раз устанавливать его заново. Нужно иметь единственный экземпляр соединения и запретить произвольное создание экземпляров (для этого можно, например, сделать конструктор приватным).

```
class Singleton
{
   static private Singleton *Instance;//объявление, не путать с определением
   private:
      Singleton();
      Singleton(const Singleton &);
   public:
      static getInstance()
       {
          if(Instance == NULL)
             Instance = new Singleton(); //здесь нужен конструктор
          return instance; //здесь нужен конструктор копирования
       }
   }
```

Определение объекта подразумевает размещение его в памяти.

Т.к. внутри класса имеется только объявление static private Singleton *Instance, то где-то вне класса нужно произвести определение этого статического члена:

```
Singleton * Singleton::Instance = NULL; //отличается от определения (размещения) глобальной переменной только тем, что кроме этого места Singleton::Instance нигде нельзя будет далее использовать, т.к. это скрытый член класса.
```

Это единственный случай, когда можно инициализировать скрытые члены класса

**Вложенные типы данных (классы)**

```
class Outer
{
   ….
    class Inner                  //Inner – обычный класс, но с ограниченной областью
    {                            //видимости
        …..
    };
    ….
};
```

Все классы являются статическими членами. Ко всем именам правила прав доступа применяются одинаково, т.е. специфика имени не участвует в разрешении прав доступа.

**C#**

В **C#** имеется понятие «статического класса»

```
static class Mod
{
   public static void f () { ….;}
   public static int i;
}
```

**Статический класс** – служит контейнером для статических членов. От статических классов нельзя наследовать. Нельзя создавать объекты статических классов. Статические классы подчёркивают их чисто модульную природу.

Без «static» - обычный класс в понятии Си++. На вложенность классов не влияет.

**Статические члены** – набор ресурсов, независимых от какого-либо экземпляра.

**Java**

Статический импорт – импорт всех статических членов класса. Часто применяется к математическим функциям.

**Статические классы в Java:**

```
public  class Outer
{
   ….
    public static class Inner                    //Тоже самое, что и в C#, C++ без «static»
    {
        …..
    };
    ….
};
```

Это сделано для того, что доступ к Inner был через Outer

Декларируются внутри основного класса и обозначаются ключевым словом static. Не имеют доступа к членам внешнего класса за исключением статических. Может содержать статические поля, методы и классы, в отличие от других типов внутренних классов в языке Java.

**Внутренние классы в Java:** без «static»

```
public  class Outer                              // «Outer» является владельцем «Inner»
{
   ….
    static class Inner          //Тоже самое, что и в C#, C++ без «static»
    {
        …..
    };
    ….
};
```

Декларируются внутри основного класса. В отличие от статических внутренних классов, имеют доступ к членам внешнего класса, например «Outer.this». Не могут содержать определение (но могут наследовать) статических полей, методов и классов (кроме констант).

**Инкапсуляция**

Единица защиты – тип класса или экземпляр класса

Атом защиты – член класса.

В современных языках единицей защиты является тип класса. Правила защиты во всех языках определяются для всех экземпляров одинаково. Нельзя один экземпляр защитить больше чем другой.

Управление инкапсуляцией:

- Управление доступом – C++, C#, D
- Управление видимостью – Java

Управление видимостью – «private»-членов как бы просто нет для других классов, они «невидимы».

Управление доступом – все не скрытые (не переопределённые) члены видны, т.е. компилятор постоянно «знает» об их существовании, но при обращении проверяются права на доступ. При попытке обращения к недоступному члену выдаётся ошибка.

```
class X
{
   public:
      virtual void f();
      void h();
};
class Y: public X
{
    private:
       virtual void f();
       void h();
```

```
};

class Z: public Y

{

    public:

        virtual void f();              //В Java заместится функция, видимая в данный момент – X::f

        void g(){ … h();…. } //В Си++ для этой строчки будет выдана ошибка – с точки зрения управления доступом попытка вызова функции Y::h() незаконна, т.к. она приватна(к ней нет доступа вне класса Y).

                                В Java спокойно вызовется функция X::h() и никакой ошибки не будет, т.к. функция Y::h() просто вычеркнута из рассмотрения (невидима)

}
```

**Три уровня инкапсуляции:**

1. public
2. private
3. protected

«свой» - член данного класса

«чужой» - все внешние классы

«свои» - члены наследованных классов

**public** разрешает доступ всем

**private** разрешает доступ только «своему»

**protected** разрешает доступ «своим» и «своему»

```
class X

{

    ….
    protected: void f();

    ……

};

class Z:public X

{

   ….
```

```
    …..
};
class Y:public X
{
    void g()
   {
      f(); //так можно
      Y another;
      another.f(); // так тоже можно
      Z getanother;
      getanother.f(); // В C# и Java так нельзя (у класса Y и класса Z независимые контракты // с классом X). В C++ так можно
     (X)getanother.f(); // В C# и Java так будет работать (в C++ и подавно)
};
```

**Перегрузка операций**

```
a += b;// ~a.operator+=(b)

a + b;
```

Есть два пути вычисления этого выражения:

```
   a.operator+(b);        или
   operator+(a,b);

class X
{
   public:
       X operator+(X & a);
       X(T);
```

};

//либо

X operator +(X & a, X & b);

T t;

X a,b;

a = a + b;

a = t + b; // ищет T.operator+(X), operator+(T,X) и т.д.

Правильным является **operator+(X(T),T)**

Если требуется, чтобы доступ к приватным членам был не только у «своего», можно для этой этого объявить нужную дружественную конструкцию в теле класса:

friend «объявление друга»;// Можно писать сразу определение. Другом может быть функция или целый класс

friend «прототип глобальной функции»

friend «прототип функции-члена другого класса»

friend class «имя класса-друга»;// Все методы этого класса становятся дружественными

В Delphi, C#, Java друзей нет

В них реализованы этот механизм реализован немного по-другому:

| | |
|---|---|
| Delphi | **UNIT** |
| Java | **package** |
| C# | **assembly** |

В **Java** по умолчанию пакетный доступ. Это значит, что использовать класс может каждый класс из этого пакета. Если класс объявить как «public class …», то он будет доступен и вне пакета. Использовать класс – наследовать, создавать объекты.

**C#:**

**Сборка** – надъязыковое понятие в .NerFramework. Сборка представляет собой совокупность файлов + манифест сборки, Любая сборка, статическая или динамическая, содержит коллекцию данных с описанием того, как ее элементы связаны друг с другом. Эти метаданные содержатся в манифесте сборки. Манифест сборки содержит все метаданные, необходимые для задания требований сборки к версиям и удостоверения безопасности, а также все метаданные, необходимые для определения области действия сборки и разрешения ссылок на ресурсы и классы.

Внутри сборки идёт разделение на пространства имён, которые содержат описания классов.

Для использования какого-либо пространства имён нужно сначала подключить сборку, содержащую его. Пространство имён может быть «размазана» по нескольким сборкам.

В C# для членов классов имеются следующие квалификаторы доступа:

- public
- private // по умолчанию
- protected
- internal – член доступен только в классах из сборки
- internal protected – член доступен только в классах-наследниках, находящихся в сборке

Для самих классов:

- public – класс можно использовать в любых классах
- internal – класс можно использовать только в классах из его сборки (по умолчанию)

```
public class X //Этот класс может унаследовать любой класс
{
    ….
     internal int a; // Это переменная доступна только внутри сборки
     ….
}
internal class Y //Этот класс может унаследовать только класс из сборки
{
    ….
}
```

Delphi

```
type T = class

   …. // здесь объявляются члены, видимые везде их данного модуля и не видимые
        //  других

   public
      ….
   protected
      ….
   private
      …..
end;
```

UNIT – единица дистрибуции

**Специальные функции**

Функции-члены, обладающие семантикой обычных функций-членов, о которых компилятор имеет дополнительную информацию.

**Конструктор** – порождение, инициализация

**Деструктор** – уничтожение (В Java и C# не деструкторов, вместо это можно сделать собственный метод Destroy() )

**Управление жизненным циклом объекта**

- создание, инициализация
- использование
- уничтожение

У конструктора нет возвращаемого значения.

Т.к. все объекты в C# и Java и Delphi размещаются в динамической памяти, то в этих языках обязательна операция явного размещения объектов:

X = new X(); // В Си++ X * a = new X;

Синтаксис конструкторов и деструкторов:

C++. C#, Java, D

```
class X
{
    X(«параметры»);// В C# и Java обязательно определение тела
}
```

Delphi

```
type X = class
   constructor Create;   // Имя конструктора произвольное
   destructor Destroy;   // имя деструктора произвольное
end;
…..
a:X;
….
a := X.Create;
```

В C++, C#, Java конструкторы не наследуются, но могут автоматически генерироваться компилятором по определённым правилам.

**Классификация конструкторов:**

1. Конструктор умолчания X();
2. Конструктор копирования X(X &); X(const X & );
3. Конструктор преобразования X(T); X(T &); X(const T &);

В классах из простанства имён System платформы .NetFramework не определены конструкторы копирования. Вместо этого, если это предусмотрено проектировщиками, имеется метод clone();

**Лекция. Ульянов А.В.**

**П.3. Специальные функции.**

**Конструкторы. Классификация.**

Объект ⇔ ссылка.

**Классификация (C++):**

1) КУ
2) КК
3) КП
4) Все остальное.

Почему выделяем КУ в особый класс:

X(){…} – выделяется.

X(int) – нет. (Java, C#)

Конструкторы либо создаются, либо вызываются.

Java, C#, D – в базовом типе object есть конструктор Create и деструктор Destroy. Соответственно здесь ничего создавать не надо, они уже есть и наследуются (если).

X a; - подразумевается вызов конструктора по умолчанию (неявно).

X a(); - нельзя, т.к. это прототип функции.

X* px = new X; - нельзя в Java и C#, в C++ - можно.

X* px = new X();

```
class X
{
    X();
}
Class Y:public X
{
    Y(); //Y():X(){} – писать так явно, нельзя, но можно (если определено) Y():X(i){}
}
```

В C++: если нет конструктора, то будет сгенерирован.

Как: вызов базового конструктора X, затем под объект и только потом сам конструктор.

В общем случае вид конструктора: заголовок [инициализация]

Java, C#: Понятие КУ остается.

Есть понятие инициализация объектов:

```
class X
{
    Z z = new Z(); // Z z; - значение неопределенно.
    Int I = 0;
}
```

Для базового должен быть конструктор.

C#:

Y(): base(0) {…} //base – конструктор базового типа.

super – ссылка на базовый класс (Java)

Вызов: super(…);

Вызов конструктора базового класса только у первого оператора. Если нет, то автоматически подставляется конструктор базового класса.

КУ – не определяется явно.

M-2, Ада:

Init(); – явная инициализация.

Destroy();

**КК:** Параметр передается по значению.

```
void f(X x);
X f() {return X();}
X a = b; //Вызов КК. ~ X a(b);
```

**Deep (глубокое) и Shallow (поверхностное) copy (копирование).**

```
int []a;
int []b;
a = new int[N];
```

b = a; - поверхностное копирование.

Побитовый КК:

```
class X
```

{

… //В общем случае КК работает рекурсивно.

}

class Y: public X

{

Y Y(&y) {…} //Если этого нет, то вызывается базовый КК.

}

Общее правило: Если программист не обращается к базовому, то вызывается КК.

Y Y(&y):X(y) {…} – КК от базового класса.

C#: object.

Protected ObjectMembeWizeClone();

- ⇨ По умолчанию копировать нельзя, но в произвольном классе можно самим переопределить.

ICLoneable → Clone();

Java: Интерфейс-маркер.

Cloneable; //Семантика зашита в компилятор.

protected object Clone();

**Уровни поддержек:**

1) Полная поддержка копирования.
   Class X: Cloneable{ public object Clone() throws {…}}
   Допускается public X Clone() {…}
2) Полный запрет состоит в том, что не реализован Cloneable.
   class X {public Object Clone() { throw CloneNotSupportedExeption; } }
3) Условная поддержка:
   Для массива: сам массив может поддерживать, а его элементы:
   class X: Cloneable { public X Clone() throws CloneNotSupportedExeption {…} }
4) Не наслед. интерфейс Cloneable
   protected object Clone() {…} //Поддерживает несколько операц. конструкт., не выбрасывает исключений.

Метод копирования нельзя переопределять.

D: inherited Create; //inherited – вызов соответствующего конструктора.

C#: X Z::a(…);

static X(…){…}

static конструкторы нужны, когда не хватает инициализаторов.

**Деструктор –** функция, которая вызывается автоматически, когда уничтожается объект.

C++, D, C# - OO модель.

C#, Java – Автоматическая сборка мусора.

D: object

Free, Destroy

X := ICreate(…);

x.Free;

delete p;

RAII:

X(){RA}

~X(){освобождение захваченных ресурсов}

{…} освобождение захваченных ресурсов при выходе из блока.

При динамической сборке мусора сложно определить, когда объект больше не используется.

Image.FromFile(fname);

…

Image.SaveToFile(fname);

Освободить файл можно, когда мы указываем, что с ним не буде работать.

Т.е. сборка мусора здесь не поможет.

=> C#, Java:

try

{

…

} finally {…}

D:

try

{

…

} finally

…

End

То, что в блоке finally выполняется всегда, даже если было исключение.

IDispose  - общий интерфейс освобождения памяти.

Dispose();

try { im = … } finally {im.Dispose;}

имя (инициализатор) блок

T x = expr;

X = expr;

~

try {инициализатор} finally {x.Dispose;}

C#, Java: object

protected void finalize();

public void Close();

Есть методики, которые позволяют возродить уничтоженный объект. Но finalize – полностью его удаляет (нельзя вызывать дважды).

Close() – ресурсы освобождены.

C#: ~X(){…} – нельзя вызывать явно.

System.Object.finalize – можно вызвать явно.

**Сбощик мусора:**

mark_and_sweep

Живые и мертвые объекты (ссылки).

Есть стек, в нем ссылки на объекты. Если живой, то помечаем и заносим в таблицу живых объектов, остальные – мертвые, они-то и уничтожаются.

Можно построить КЭШ объектов. Если объект мертвый, то нм нужен он. Но он еще не утилизирован (не успели).

Strong reference – на живой объект.

Weak reference – объект готовится к уничтожение, не пока еще не нуничтожен.

Java:

Reference (Object o)

get – выдает сильную ссылку.

clear() – делает ссылку несильной.

class DataFromFIle

{

```
    Public void read() {… d = idData.get(); if (d!=null) return idData;
                        //реальное чтение
                        idData = new …}
    Weak Reference idData; //в любой момент может быть уничтожен. Если пока нет,
    //то get дает ссылку на объект и делает ее сильной, иначе – null.
}
```

Soft Reference(мягкая ссылка) – все ссылки считаются равноправными.

Является разновидностью слабой, но удаляются из очереди с самой ранним временем загрузки (дольше всех не использовалась).

**Преобразования.**

**Неявные** (автоматически вставленные компилятором).

Int → long

А может ли их задавать пользователь.

C#, Java – неявные преобразования, задаваемые пользователем разрешены.

Ф: v = expr – можно считывать различные типы данных.

Действительные + комплексные:

Ада: ToComplex(A) + x*i*ToComplex(f)*d + …

* - можно перегрузить. Тогда в чем проблема?

6 видов бинарных операций и много стандартных типов. Только для умножения ~20. Итого ~120.

А можно C*I -> d -> Complex.

(d,0) – можно делать автоматически.

C++: Страуструп ввел неявные преобразования:

char* => string => …

operator +(op) {…}

C#:

static operator+(T t1, T t2){…}

static operator*(T t1, T t2){…}

Недостаток C++:

```
Class Vector
{
    T* body;
```

```
    int size;
public:
    Vector(int sz) {body = new[size = sz];}
}
Vector v(20); //ok
Vector t(10);
v = t;
v =1; ~ v = Vector(1); // Вдруг описались, получилась чушь, но работать будет КП.
```

expliced – снимает семантику КП (т.е. конструктор вызывается явно).

V = Vecto(1);…

C#: иначе,

implicit

explicit – по умолчанию.

**П.4. Дополнительные возможности.**

**Свойства (properties).**

С точки зрения операций – данные,

С точки зрения реализации – get, set.

D:

```
T Get X() {return _x}
void Set X (T value){_x = value;}
```

C#:

```
class X
{
    T prop {get{…} set{…}} //value является зарезервированным в set.
}
…
X a = new X();
a.prop = t1;
T t2 = a.prop;
```

D:

property prop: T

read – заголовок_get

write – заголовок_set

private _x: T;

public → published//все опубликованные свойства появляются в интегрированной среде разработки.

property X: T read_x; write_x;

C#: Нельзя переопределять [] => понятие индексатор:

```
class X
{
    T this(index){…}
}
class Outer{…}
class Inner{…}
```

Нестатический блок-класс имеет ссылку на Outer.this

Inner in = this.newInner();

Если public class Inner{…}

Outer invoice;

Inner = invoice.newInner();

```
Iterable f(object[] objs)
{
    class Local: Iterable {int i; Local() {i = 0;}… if (i>objs) …}
     return new Local(C);
}
```

Имеет мест доступ к локальным переменным функции, если в данном блоке не изменяются.

delegate int Processor(int i);

Prcessor p = new Processor(func);

Анонимные делегаты:

P = new delegate(int i){…return k+ i;}

Анонимные делегаты могут захватывать память.

k =0;

```
int j = P(3); //3

k = -1;

j = P(3); //2
```

**Лекция. Сергеев Николай.**

> «Объект обладает *состоянием*, *поведением* и *индивидуальностью*».
>
> **А.Буч**

**Примечание:** В лекции использовались дополнительные материалы: обе книги Страуструпа(“язык С++” и “Эволюция и дизайн языка С++”), Wikipedia.org , google.ru

Доброго времени суток, студенты ВМиК. Данная лекция является предпоследней в цикле лекций по Языкам Программирования, и сегодня мы поговорим об Объектно-ориентированном программировании. Надеюсь вы уже сталкивались с этим понятием, изучая материалы 4 семестра, и хорошо знаете что программировании на С++ подразумевает достаточное хорошее понимание основ объектно-ориентированного программирования. Ну если нет, то уверен, что после прочтения этой лекции, вы уже достаточно хорошо будете владеть основами. Ну что же приступим…

Начнем с определения ООП (далее будем использовать такое сокращение для Объектно-ориентированного программирования):

**Объектно-ориентированное программирование** (ООП) — парадигма программирования, в которой основными концепциями являются понятия объектов и классов. Если для вас эти слова ничего не значат, то рекомендуется вернутся на пару лекций назад и прочесть их, так как незнание такие фундаментальных понятий осложнит дальнейшее восприятие материала.

Истоки ООП восходят к далеким 70-ым, во времена когда шло бурное развитие языков программирования. Каждый день появлялся хотя бы один язык, и каждый день хотя бы один язык умирал. В то смутное время группкой ученой была выдвинута идея ООП, которая была отчасти реализована в языке СИМУЛА-67. Однако реализация та была далеко неполной, и понадобилось целых 13 лет, чтобы придти к первой полной реализации ООП в языке Smaltalk

На сегодняшний день концепция ООП реализована в таких языках как С++, Java,C#,Ada,Object Pascal,Turbo Pascal, Delphi, Oberon и в многих других. Очевидно, что эта черта присуща популярным языкам.

Ключевые черты(требовния) ООП хорошо известны:

1. Первая — **инкапсуляция** — это определение классов — пользовательских типов данных, объединяющих своё содержимое в единый тип и реализующих некоторые операции или методы над ним. Классы обычно являются основой модульности, инкапсуляции и абстракции данных в языках ООП.
2. Вторая ключевая черта, — **наследование** — есть способ определения нового типа, наследуя элементы (содержание и методы) существующего и модифицируя или расширяя их. Это способствует выражению специализации и генерализации.

3. Третья черта, известная как **полиморфизм**, позволяет единообразно ссылаться на объекты различных классов (обычно внутри некоторой иерархии). Это делает классы ещё более удобными и делает программы, основанные на них, легче для расширения и поддержки.

Инкапсуляция, наследование и полиморфизм — фундаментальные свойства, требуемые от языка, претендующего называться объектно-ориентированным (языки, не имеющие наследования и полиморфизма, но имеющие только классы, обычно называются объектными языками). Таким образом мы остановимся более подробно на последних двух свойствах: наследование и полиморфизм.

Головин вел понятие наследования, как отношения между классами: при наследовании всегда есть базовый класс и класс, который наследуется от базового, то есть вы определяете новый тип, расширяя или модифицируя существующий, другими словами, производный класс обладает всеми данными и методами базового класса, новыми данными и методами и, возможно, модифицирует некоторые из существующих методов. Различные ООП языки используют различные жаргоны для описания этого механизма (derivation, inheritance, sub-classing), для класса, от которого вы наследуете (базовый класс, родительский класс, суперкласс) и для нового класса (производный класс, дочерний класс, подкласс).

При наследовании дополнительно к тому что было сказано выше должны выполняться следующие свойства: совместимость унаследованного класса с базовым при передаче параметра и присваивании, то есть унаследованный класс должен приводиться к базовому. Тоже самое должно выполняться и для ссылок и указателей: ссылке (указателю) на базовый класс должно быть возможно присвоить ссылку(указатель) на производный класс. Эта возможность работы со ссылками приводит нас к понятию **динамического типа**: ссылка или указатель одного класса не обязательно указывает(ссылается) на объект того же класса, она может ссылаться на объект производного класса, таким образом динамический тип – это тип объекта, на который ссылается (указывает) ссылка(указатель)

Рассмотрим синтаксис наследования в различных языках программирования, и убедимся, что в них много общего.

**C++:** *class Derived : [модификатор] Base {*
*// обьявление новых членов* *};*

*[модификатор] ::= {private, public, protected}*

C++ использует слова public, protected, и private для определения типа наследования и чтобы спрятать наследуемые методы или данные, делая их приватными или защищёнными. Хотя публичное наследование наиболее часто используется, по умолчанию берётся приватное. Чаще всего приватное наследование используется в написании интерфейсов. Public – не меняет модификатор доступа свойств наследуемого класса в производном, protected – делает все публичные свойства наследуемого класса защищенными, а private – все свойства наследуемого класса делаем закрытыми(модификатор доступа private). Для более подробного осознания материала автор рекомендует обратиться к известной книге Страуструпа.

Не будем отходить далеко и рассмотрим синтаксис

**C#:** *class Derived : Base {*
*//определение новых членов* *}*

**Java:** *class Derived extends Base {*
*// определение новых членов* *}*

Java использует слово extends для выражения единственного типа наследования, соответствующего публичному наследованию в C++. Java не поддерживает множественное наследование. Классы Java тоже происходят от общего базового класса.

**Oberon, Object Pascal, Turbo Pascal:**

```
TYPE Derived = RECORD (Base)
```

```
        // определение новых членов
    END;
```

**Delphi:**

```
    TYPE Derived    =   class (Base)
        // определение новых членов
    END;
```

**В** этих языках при наследовании используется не ключевое слово, а специальный синтаксис - добавление в скобках имени базового класса. Эти языки поддерживают только один тип наследования, который в C++ называется публичным.

**Ada:**

```
type Base is tagged record
// члены                                                    end;

Type Derived is new Base with record
// определение новых членов                                 end;

// новые члены не определяются, используется для добавления новых методов к // базовому классу

Type Derived is new Base with null record;
```

**Замечания:**

1.В некоторых ООП языках(Java, C#, Delphi) каждый класс происходит по крайней мере от некоторого базового класса по умолчанию. Этот класс, часто называемый Object, или подобно этому, обладает некоторыми основными способностями, доступными всем классам. Фактически, все другие классы в обязательном порядке его наследуют. Этот подход является общим ещё и потому, что так первоначально делалось в Smalltalk. Хотя язык C++ и не поддерживает такое свойство, многие структуры приложений базируются на нём, вводя идею общего базового класса. Пример тому — MFC с его классом CObject. Также любой класс может стать первым в иерархии классов в таких языках, как Оберон, Ада - 95

2.Единственный язык, поддерживающий множественное наследование из языков, проходимых в курсе ООП – язык C++.

3. Единственный язык, поддерживающий модификацию прав доступа свойств базового класса в производном – язык C++.

Теперь давайте поговорим о реализации наследования. Во многих языках (всех, рассматриваемых в курсе кроме Smaltalk) свойства объектов распределены линейно, то есть если у нас есть класс

```
class Base{
int a;    // 2 байта                                          char
b;  // 1 байт                                                int * c;  // 4 байта
                                                          } Object ;
```

то адрес памяти для Object.b равен (*Object + 2), а для Object.a - (*Object).

**Замечание:**

Методы класса в этом распределении не участвуют, так как для всех экземпляров класса методы абсолютно одинаковые, в отличии от свойств класса.

В производном классе, смещения адресов свойств базового класса относительно начала сохраняется, а новые свойства как бы дописываются в конец класса. Такая реализация позволяет относительно просто реализовать присваивание обьектам базового класса обьектов производных, (программисты в таком случае говорят, что происходит «срезка») и также просто реализовать при передаче параметров обьекта производного класса вместо базового. Но обратное присваивание: производному классу присвоить базовый не допускается, так как в этом случае останутся не инициализированные поля(такое ограничение действует также и для ссылок и указателей).

Рассмотрим пример:

```
class Base {
        void f();
} ;

class Derived: public Base {
        int f;
};
```

Как мы видим и класс Base, и производный от него класс имеют общее имя f(в такой ситуации говорят, что f в производном классе скрывает f в базовом, и если бы в производном классе вместо *int f;* было бы *void f(int);* – то здесь происходило бы тоже скрытие(не перегрузка!)) Такая ситуация вполне возможна, поэтому давайте поговорим о:

1) перегрузке(overloading) (в одной области действия)
2) скрытии(hiding)
3) переопределении(overriding) (динамическое связывание - рассмотрим чуть – чуть позднее)

Все эти понятия вы должны были изучить на примере C++ в 4 семестре, поэтому обратиися к материалам 4 семестра:

**Перегрузка функций(! Именно функций, так как не существует перегрузки свойств)**

Статический полиморфизм(мы не работаем с указателями, где есть базовый класс и производный) позволяет давать одно имя нескольким функциям. Как правило, эти функции имеют схожую семантику, но отличаются списком формальных параметров. Какая функция будет вызвана, определяется на этапе трансляции. ***О перегрузке функций можно говорить только в пределах одной области видимости.*** Кстати, когда мы об'являем несколько конструкторов одного класса – это тоже перегрузка функций.

Проблема поиска подходящей перегруженной функции (**best matching**) – нетривиальная задача. Для начала опишем этот алгоритм для функции одного аргумента.

1. Поиск функции, точно совпадающей по типу параметра (**точное отождествление**). Если функция вызывается от параметра типа `T`, то может быть вызвано описание с прототипом от `T`, `T&`, **`const`** `T`, **`const`** `T&`, переопределения этих типов с помощью **`typedef`**, `T[]` эквивалентно `T*`, функция эквивалентна указателю на функцию.
2. Если не найдено точное соответствие, то пробуем применить стандартные преобразования. На втором шаге могут сработать безопасные преобразования – целочисленное или вещественное расширение (integral/floating promotion). Тут **`bool`**, **`char`**, **`short`**, **`enum`** (знаковые или беззнаковые) преобразуются к **`int`**(если возможно) или **`unsigned`**, **`float`** преобразуется к **`double`**.
3. Если не получилось выполнить шаг 2, пробуем все остальные стандартные преобразования: оставшиеся арифметические преобразования и преобразования указателей и ссылок (указатель на производный класс приводится к указателю на однозначный доступный базовый класс, любой указатель приводится к **`void*`**, 0 приводится к NULL).
4. Пользовательские преобразования - рассматриваются конструкторы, которые могут быть вызваны с одним параметром. Также рассматриваются специальные функции преобразования типов. При

выполнении пользовательского преобразования можно сделать еще одно (!) преобразование, но только с шага 2 или 3.

5. Если ничего не помогло, придётся вызывать функцию с '…'.

Если функция имеет один параметр, то алгоритм действует следующим образом: если на некотором шаге найдена одна функция – отлично, ее и будем вызывать. Если две и более – ошибка (неоднозначный вызов). К следующим шагам переходим тогда и только тогда, когда ни одного соответствия нет.

Рассмотрим ряд примеров.

*Пример на 2-й шаг:*

```
void f(int);
void f(double);
void g() {
      short a=1;
      float ff=1.0;
      f(a);  // f(int)      // 2-й шаг
      f(ff); // f(double) // 2-й шаг
}
```

*Пример на 3-й шаг:*

```
void f(char);
void f(double);
void g() {
      f(1); // ошибка: неоднозначность
                  (две возможности на 3-м шаге)
}
```

*Пример на 4-й шаг:*

```
struct S{
      S(long);              // long -> S
      operator int(); // S -> int
};
void f(long);
void f(char *);
void g(S);
void g(char *);
void ex(S &a){
      f(a);      // f((long)(a.operator int()))
      g(1);      // g(S((long)(1))
      g(0); // g((char *)0) - 3-й шаг!
}
```

Особенности четвёртого шага:

1. Отсутствие транзитивности пользовательских преобразований. То есть, за один раз не может выполниться более одного преобразования типа.

```
class X { public: operator int(); ... };
class Y { public: operator X(); ... };
void f(){  Y a;  int b; ...
     b = a;  // нельзя
}
```

Можно явно указать `b = a.operator X().operator int();`.

2. Пользовательские преобразования могут применяться неявно, только если они однозначны.

```
class B {
public: B (int i);
     operator int();
     B operator+ (int B);
};
void f(){
     B l(1);  ... l+1 ...
}
```

Возникает неоднозначность: то ли `l` стоит преобразовать к **int** с помощью определённого преобразования и складывать числа, то ли вызвать конструктор от **int** и складывать об'екты типа `B`.

3. Конструктор должен быть описан так, чтобы он допускал неявный вызов. То есть, конструктор не может быть описан как **explicit**.

```
class X { public: X(int); };
X a(1);  X b  = 2;  // так можно
```

Теперь изменим об'явление:

```
class X { public: explicit X(int); };
X a(1);         // так можно
X b = 2;       // так нельзя!
X c = X(2);     // так можно
```

Зачем же нужна такая конструкция? Вспомним наш класс String.

```
class String { String (int n); ... };
String s1 = 10;   // выделится память под строку из 10 символов
String s2 = 'a';  // неужели мы хотим выделить память
                  // под строку из код('a') символов?!
```

Никто нам не запрещает так делать. Но, если мы допишем **explicit** к конструктору, то такая нелогичная запись не прокатит и придётся вызывать через скобочки.

Алгоритм поиска наилучшего соответствия для вызова функции с произвольным числом параметров N:

I. По каждому из параметров ищется best matching по пятишаговому алгоритму за тем исключением, что если на каком-то шаге несколько кандидатов, способных обслужить вызов, запоминаем все их. В итоге получаем N множеств возможных функций.

II. Ищем пересечение этих множеств. Если оно пусто, то нет подходящей функции. Если пересечение содержит 2 или более элемента, то неоднозначность. Но если там одна функция, она и обслужит вызов.

*Пример.*

```
class X { public: X (int); ... };
class Y { ... };
void f (X, int);  /* 1 */
void f (X, double);  /* 2 */
void f (Y, double);  /* 3 */
```

Пусть мы вызываем `f(1,5);`. По первому параметру мы оставляем 1 и 2 (пользовательские преобразования), по второму – 1 (точное соответствие). Пересечение даёт первый вариант.

Теперь попробуем вызвать `f(1,5.0);`. По первому параметру мы оставляем 1 и 2 (пользовательские преобразования), по второму – 2 и 3 (точное соответствие). Пересечение даёт второй вариант.

*Пример на *пятый шаг**.

```
class R { public: R (double); ... };
void f (int, R);
void f (int, ...);
void g () {
      f (1, 1);                 // первый обработчик
      f (1, “preved!”);         // второй обработчик
}
```

Нужно сделать еще одно замечание, касающееся описанного алгоритма. Как определить, какие именно функции попадают в множество “испытуемых”? Ответ таков: все функции с таким именем, которые могут быть вызваны с таким набором фактических параметров. Это правило кажется тривиальным, однако оно может помочь в ситуации, когда у некоторых функций есть значения по умолчанию, а у некоторых - '`...`' в списке параметров.

**Скрытие:** скрытие происходит при наследовании, в унаследованном классе, если в производном классе есть такое же имя как в базовом. Этот случай надо отличать от последнего, так как в последнем тоже совпадают имена, но там за дело берётся виртуальность.

**Переопределение:** в переопределении главная роль уделяется виртуальным функциям, собственно в переопределении и заключен весь механизм работы виртуальных функций.

**Механизм виртуальных функций.**

Чтобы включился механизм виртуализации, должны быть выполнены следующие требования:

- присутствует иерархия классов;
- в базовом классе функция объявлена как виртуальная;
- в производном классе описана функция с таким же профилем (совпадают имя, список параметров с точностью до имён и тип результата);
- используется вызов функции через указатель (без уточнения с помощью оператора разрешения контекста `::`).

Однако к этим четырём заповедям нужно прибавить ещё некоторые замечания:

- Если описывать функцию вне класса, то в её профиле **`virtual`** не повторяют. В описании функции в производном классе также можно указать **`virtual`**, но это ни на что не повлияет, разве что на наследство производного.
- Тип результата может слегка отличаться. Если в базовом классе виртуальная функция возвращает указатель на базовый класс, то в производном она имеет право возвращать указатель на производный класс (то же со ссылками).
- Если списки параметров совпадают, а типы результата существенно отличаются, мы получим синтаксическую ошибку.
- Если списки параметров различаются, то функция из производного класса скрывает функцию из базового, виртуальность не работает, и при вызове через указатель получаем статическое определение по типу указателя.
- Если в производном классе вообще нет функции с таким именем, то метод полностью наследуется, то есть мы сможем использовать виртуальность во внуке, а сын (производный класс) просто получит в свое распоряжение эту функцию.
- Если вызываем метод через объект (`h.print()`), то виртуальность не работает, статическое определение по типу объекта, от которого его вызываем.
- При явном указании класса виртуальность не работает, даже если функция вызвана через указатель, то есть `pp->Person::print();` – статическое определение.

**Замечание:** В Java, Delphi также имеется синтаксическая конструкция virtual

**Замечание:** Головин сказал, что можно считать, что вызов через ссылку или указатель всегда виртуальный

**Замечание:** Если виртуальная функция возвращает один из базовых типов языка, то тип возвращаемого значения у замещающей функции должен совпадать с типом замещаемой функции. В случае если она возвращает объект класса, то при переопределении может возвращаться производный от него тип.

**Замечание:** Если открытая функция переопределена как приватная, то она может быть вызвана из интерфейса базового класса

**Замечание:** В C++ нет никаких ограничений с точки зрения наследования, то есть нет никаких языковых методов запретить наследования данного класса. В Java можно запретить наследование с помощью ключевого слова final. Если это слово стоит перед именем метода, то этот метод не может переопределяться в производных классах. Если же оно стоит перед определением класса, то класс нельзя наследовать. В C# такую роль играет sealed.

Рассмотрим ситуацию наследования в языках C#, Delphi. Как и в C++, там есть виртуальные методы. Если в классе метод объявлен как виртуальный, то в классе наследнике возможна ситуация – есть функция, у которой совпадает профиль, имя, параметры, но она не замещает функцию из базового класса. Разработчики ввели специальный модификатор override – для того чтобы функция действительна замещала.

Поговорим немножко о реализации виртуальности на примере C++. Разобраться в этом нам поможет отрывок из книжки «Дизайн и эволюция языка C++»:

### 3.5.1. Модель размещения объекта в памяти

Ключевая идея реализации состояла в том, что множество виртуальных функций класса определяет массив указателей на функции, поэтому вызов виртуальной функции – это просто косвенный вызов функции через данный массив. Для каждого класса с виртуальными функциями имеется ровно один такой массив, который обычно называют таблицей виртуальных функций или `vtbl`. Любой объект данного класса содержит скрытый указатель (часто его называют `vptr`) на таблицу виртуальных функций своего класса. Если имеются определения

```
class A {
   int a;
public:
   virtual void f();
   virtual void g(int);
   virtual void h(double);
};

class B : public A {
public:
   int b;
   void g(int);  // замещает A::g()
   virtual void m(B*);
};
class C : public B {
public:
   int c;
   void h(double);  // замещает A::h()
   virtual void n(C*);
};
```

то объект класса C выглядит примерно так, как показано на рис. 3.2.

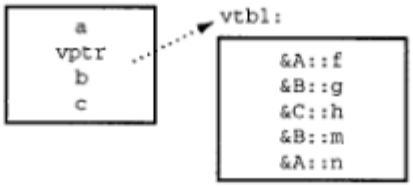



*Рис. 3.2*

Вызов виртуальной функции преобразуется компилятором в вызов по указателю. Например,

```
void f(C* p)
{
   p->g(2);
}
```

превращается в

```
(*(p->vptr[1]))(p,2);   /* сгенерированный код */
```

Эта реализация не единственно возможная. К ее достоинствам можно отнести простоту и эффективность, к недостаткам – необходимость перекомпиляции при каждом изменении множества виртуальных функций класса.

Таким образом, объект перестает быть просто конгломератом данных-членов. В языке C++ объект класса с виртуальными функциями кардинально отличается от простой структуры в C. Так почему же было решено разделить понятия класса и структуры?

Дело в том, что я хотел иметь единую концепцию: один набор правил размещения в памяти, один набор правил поиска, простые правила разрешения имен и т.д. Единая концепция обеспечивает лучшую интеграцию языковых средств и упрощает реализацию. Я был убежден, что если бы `struct` воспринималось пользователями как «C и совместимость», а `class` – как «C++ и более развитые возможности», то все сообщество раскололось бы на два лагеря, общение между которыми вскоре прекратилось бы. Для меня было очень важно, чтобы при проектировании класса число используемых языковых средств сократилось до минимума. Лишь единая концепция соответствовала моей идее о плавном и постепенном переходе от «традиционного стиля программирования на C» через понятие абстракции данных к объектно-ориентированному программированию.

Оглядываясь назад, я думаю, что такой подход в немалой степени способствовал успеху C++ как инструмента для решения практических задач. На протяжении ряда лет выдвигались самые разные дорогостоящие идеи, которые могли быть реализованы «только для классов», а для структур предлагалось оставить низкие накладные расходы и облегченную функциональность. Но, по-моему, приверженность единой концепции `struct` и `class` избавила нас от включения в классы дополнительных полей и т.п., радикально отличающихся от того, что мы сейчас имеем. Другими словами, принцип «`struct` – это `class`» уберег C++ от превращения в гораздо более высокоуровневый язык с независимым подмножеством низкого уровня, хотя некоторые, возможно, и желали такого развития событий.

### 3.5.2. Замещение и поиск подходящей виртуальной функции

Виртуальная функция может быть замещена только в производном классе и только функцией с такими же типами аргументов, возвращаемого значения и именем. Это позволило избежать проверки типов во время исполнения и не хранить в выполняемой программе обширную информацию о типах. Например:

```
class Base {
public:
   virtual void f();
   virtual void g(int);
};

class Derived : public Base {
public:
   void f();     // замещает Base::f()
   void g(char); // не замещает Base::g()
};
```

Здесь скрыта очевидная ловушка. Невиртуальная функция `Derived::g()` никак не связана с виртуальной функцией `Base::g()` и скрывает ее. Если вы работаете с компилятором, не выдающим предупреждений по этому поводу, то здесь может возникнуть проблема. Однако обнаружить такую ситуацию компилятору не составляет труда. Cfront 1.0 не давал таких предупреждений, чем вызвал немало нареканий. Уже в версии 2.0 это было исправлено.

Правило точного соответствия типов для замещающих функций позже было ослаблено в отношении типа возвращаемого значения (см. раздел 13.7).

### 3.5.3. Сокрытие членов базового класса

Имя в производном классе скрывает любой объект или функцию с тем же именем в базовом классе. Несколько лет продолжалась полемика на тему, разумно ли такое решение. Данное правило впервые введено в C with Classes. Я считал, что это естественное следствие обычных правил областей действия. Отстаивая свою точку зрения, я говорил, что противоположная позиция – поместить имена из базового и производного классов в единую область действия – приводит к не менее многочисленным трудностям. В частности, по ошибке можно вызвать из подобъектов функции, изменяющие состояние:

```
class X {
   int x;
public:
   virtual void copy(X* p) { x = p->x; }
};

class XX : public X {
   int xx;
public:
   virtual void copy(XX* p) { xx = p->xx; X::copy(p); }
};
```

```
void f(X a, XX b)
{
   a.copy(&b);   // правильно: скопировать часть b, принадлежащую X
   b.copy(&a);   // ошибка: copy(X*) скрыто за copy(XX*)
}
```

Если разрешить вторую операцию копирования – а так и случилось бы в случае объединения областей действия, – то состояние объекта `b` изменилось бы лишь частично. В большинстве реальных ситуаций это привело бы к очень странному поведению объектов класса `XX`. Я видел, как люди попадались в эту ловушку при использовании компилятора GNU C++ (см. раздел 7.1.4), разрешающего такую перегрузку.

Если функция `copy()` виртуальна, то можно считать, что `XX::copy()` замещает `X::copy()`, но тогда для нахождения ошибки с `b.copy(&a)` потребовалась бы проверка типов во время исполнения, и программистам пришлось бы страховаться от таких неприятностей (см. раздел 13.7.1). Это я понимал уже тогда и боялся, что есть и другие, не осознанные мной проблемы. Вот почему пришлось остановиться на таких правилах, которые казались мне самыми строгими и одновременно наиболее простыми и эффективными в реализации.

Теперь уже я подозреваю, что с помощью правил перегрузки, введенных в версии 2.0 (см. раздел 11.2.2), можно было справиться с этой ситуацией. Рассмотрим вызов `b.copy(&a)`. Переменная `b` по типу точно соответствует неявному аргументу `XX::copy`, но требует стандартного преобразования для соответствия `X::copy`. С другой стороны, переменная `a` точно соответствует явному аргументу `X::copy`, но требует стандартного преобразования для соответствия `XX::copy`. Следовательно, при разрешенной перегрузке такой вызов был бы признан ошибочным из-за неоднозначности.

Теперь пришло время рассмотреть как обстоят дела в модульных языках:

**Ада, Оберон:**

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

Существуют только лишь либо публичные члены, либо пакетные, а как таковых защищенных и приватных нет.

```
Package M is type base is tagged private;
…
Private type Base is tagged record
                        …
        end record
end M;

Package M1 is use M;
Type Derived is new Base with record
                        K:integer;
                End record
Procedure B(x:inout Derived) is…
```

В Аде появляется понятие дочерних пакетов.

```
Package M.M1 is …
Type Derived is new Base with recprd
```

*K:integer*
*end record*
*procedure P(x : inout Derived)*

**Множественное наследование.**

**Множественное наследование** - ситуация, когда производный класс создаётся на базе нескольких базовых классов. Как было сказано выше из всех проходимых нами языков, множественное наследование реализовано только в C++, поэтому примеры ниже будут написаны на C++

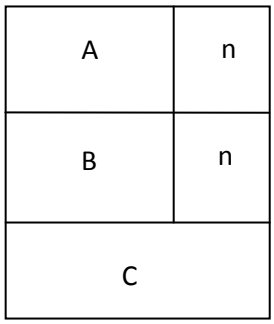
Чтобы понять, зачем это нужно, можно привести пример от Бьярна Страуструпа. При программировании некого GUI у нас есть класс ‘окно’. От него есть производные классы: ‘окно с рамкой’ и ‘окно с меню’. А что если мы хотим создать окно с рамкой и с меню? Нам на помощь приходит множественное наследование.

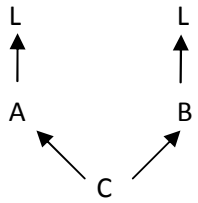
*Пример.* Далее приведём чисто технический пример, который не несёт никакой смысловой нагрузки, но на котором можно показать синтаксис.

```
class A { ... };
class B { ... };
class C: public A, public B { ... };
```

Базовый класс не может появиться несколько раз в этом списке явно (ситуация ‘одна база – два раза’). Однако может возникнуть такая ситуация:

```
class L { public: int n; ... };
class A: public L { ... };
class B: public L { ... };
class C: public A, public B { ... };
```




решётка смежности

Например, мы напишем: `C.c; c.n = 0;` Возникает неоднозначность: какую с нам вызвать, ту которая пришла к нам через A, или же ту, которая наследовалась через B. Здесь мы можем уточнить: `c.A::n = 5;` или `c.B::n = 7;`. Перед оператором разрешения контекста мы указываем точку, от которой начинается поиск переменной.

**Замечание:** в других языках тоже существует оператор уточнения (x::N ~ super.N ~ base.N ~ inherited.N)

**Лекция. Лихогруд Н.Н.**

**Абстрактные классы и интерфейсы**

Сами по себе иерархии классов бесполезны, если в них нету динамически связанных методов.

- Если существуют признаки, общие для всех объектов в класса иерархии, то эти признаки целесообразно поместить в базовый класс
- У каждого объекта класса имеется состояние (текущие значения параметров) и поведение (методы класса)
- Некоторая функциональность (особенности поведения), общая для всех классов в иерархии, может быть реализована только при дальнейшей детализации в производных классах.

=> приходим к понятию абстрактного базового класса

В языках Delphi, Оберон-2, TP 5.5, C++, Java, C# имеется языковая поддержка абстрактных классов.

Пример:

**Figure** – абстрактная фигура

Общие данные – (x,y) – координаты фигуры

Общие методы – Draw(), Move() – не могут быть реализованы для абстрактной фигуры

Что будет если просто не определить витуальную функцию в базовом классе:

Полиморфный класс – класс, в котором есть виртуальные функции (и. следовательно, таблица виртуальных функций)

```
class X
{
   public:
      virtual void f(); //Объявлена, но не определена
      void g();//Объявлена, но не определена
}
class Y: public X
{
    public
       virtual void f() {…;}//Замещена
}
….
```

X * py = new Y(); // уже здесь компилятор выдаст ошибку из-за того, т.к. непонятно что записывать в таблицу виртуальных функций

py -> f();

X a; // здесь компилятор также выдаст ошибку из-за того, из-за того, что не сможет заполнить таблицу виртуальных функций. Если бы функция X:: f() не была виртуальной, то ошибки здесь не было бы.

a.g();// ошибка, т.к X::g() не определена

Решение проблемы – языковая поддержка

В **C#, Java**:

**abstract** перед классом и перед функцией, для которой не существует реализации на данном уровне детализации. Такие функции и классы, их содержащие, называют аббстрактными

```
abstract class Figure //Абстрактный класс
{
   abstract public void Draw(); // Функция без реализации
   abstract public void Move();// Функция без реализации
}
```

**В C++:**

Чисто виртуальная функция (абстрактная) – **virtual «прототип» = 0;**

**В Аде:**

```
procedure P( X : T ) is abstract;
```

где T – тегированный тип.

**Объекты абстрактных классов нельзя создавать.**

При вызове виртуальной функции в конструкторе виртуальность вызова снимается:

class X{…}

class Y: public X{…}

class Z: public Y {…}

При создании объекта класса Z сначала вызовется конструктор класса X, потом класса Y и в самом конце класса Z. Нельзя вызвать в конструкторе класса Y замещённую функцию из Z, потому что объект Z ещё не до конца создан и для него ещё даже нет таблицы виртуальных функций.

Существует метод, который не должен быть чисто виртуальным – деструктор. Он всегда должен быть виртуальным и реализованным.

Base * px = new Derived;

«использование»

delete(px); // уничтожиться должен объект класса Derived

**Различие между абстрактными классами и абстрактными типами данных:**

В абстрактных классах абстрагируются от реализаций некоторых методов. В абстрактных типах данных абстрагируются от всей структуры.

Например множество – каждое множество должно поддерживать операции

- include(const T &)
- exclude( const T &)

При этом, естественно, реализация этих методов зависит от типа элементов, способа хранения и т.д.

class Iset

{

virtual void include(const T &) = 0;

virtual exclude(const T &) = 0;

«статические члены»

}

Такой класс называется **класс-интерфейс.** В таких классах не имеет смысл объявлять нестатические поля, т.к. не реализовано ни одного методы для работы с ними.

```
class Slist{…}

class Slist_Set: public Iset, private Slist {…; } //  Iset – интерфейс


Iset  *  Move()

{

  ….

   return new Slist_set(«параметры»);

}
```

В C# и Java, в отличие от C++, существует языкового понятия интерфейса:

```
interface «имя»

{

    «объявления членов»

}
```

Внутри интерфейса могут быть статические поля. Поля без «static» будут восприняты как статические. Также членами интерфейса могут быть методы и свойства, которые считаются чисто виртуальными и публичными. Т.е. интерфейс – чистый контракт, не реализующий структуру.

Если класс наследует интерфейс и не определяет все методы интерфейса, то он становится абстрактным.

**Множественное наследование**

Только в C++ поддерживается множественное наследование. В C# и Java множественное наследование поддерживается только для интерфейсов.

C#: class D: [«класс»]{, «интерфейс»}

Java: class D extends Base implements «интерфейс» {, «интерфейс»}

Проблемы, связанные с множественным наследованием:

- **Конфликт имён**

**Java:**

```
interface Icard
{
  void Draw(){ …; }// Раздать карты
}
interface IGUIControl
{
{
   void Draw(){ …; } //нарисовать колоду
}
class Sample implements ICard, IGUIControl {…; }// Ошибка!!
```

**C++**

```
class D public I1, public I2
{
   virtual void I1::f(){…;} // Операция разрешения видимости
   virtual void I2::f(){…;} // Операция разрешения видимости
}
…
D * px = new D;
px-> f(); // ошибка
px->I1::f(); // ошибка
((I1 *)px) -> f(); //Работает! Явное приведение
```

**C#:** Неявная и явная реализация интерфейсов

```
interface ISample
{
  void f();
}
class CoClass: ISample
{
   public void f(){…} // неявная реализация, «public» перед void f() обязателен
}
class CoClass2: ISample
{
   void ISample.f(){…} // явная реализация. Запрещено указывать public, private,  protected.
}

CoClass x = new CoClass();
x.f();// работает!

CoClass2 x = new CoClass2()’
x.f();/// ошибка! Цитата компилятора: «CoClass2' does not contain a definition for  'f'
((ISample)x).f(); // работает! Явное приведение

В платформе .Net имеется класс
class FileStream:IDisposable
{
   ….
   void IDisposable.Dispose(){…;}
   void Close(){ ((IDisposable)this).Dispose();}
   ….
}
```

FileStream реализует интерфейс IDisposable явным образом, т.е. через объект FileStream нельзя вызвать метод Dispose. Вместо этого имеется не виртуальный метод Close, который явно вызывается Dispose внутри себя.

```
class IControl
{
   void Paint();
}
interface IEdit:IControl
{
   …..
}
interface IDropList:Icontrol
{
   …..
}
class ComboBox : IDropList, IEdit // Элемент управления ComboBox реализует оба
                                  //интерфейса. И для каждого из них должен быть свой                                  //Paint;
{
   void IDropList.Paint(){…; } //Явная реализация
   void IEdit.Paint(){….;}//Явная реализация
   public void Paint() //Этот метод будет вызываться по умолчанию
   {
       ….
       ((IEdit)this).Paint(); // Явный вызов
       …..
    }
}
```

В C# реализованные методы интерфейсов считаются по умолчанию «sealed» - запечатанными. В наследниках они перекрываются.

```
class ExampleClass : IControl
```

{
   public void Paint(){  … ;}

}

class FancyComBox: ExampleClass

{

   public void Paint(){….; } // Компилятор выдаст предупреждение для этой строчки, в котором сообщит, что «FancyComBox.Paint() скрывает унаследованный метод ExampleClass.Paint(). Используйте ключевое слово «new», если это сделано целенаправленно.» Т.е. если поставить «new» перед определением, то предупреждение исчезнет.

    public void override Paint (){… ; } // Для этой строчки компилятор выдаст ошибку, т.к. нельзя замещать(переопределять) методы, не указанные в базовых классах как virtual, override или abstact

}

В целом множественное наследование можно заменить включением (агрегацией).

Как было показано выше, конфликт имён решается через явное приведение и уточнение класса.

Но существует ещё одна проблема – эффективность динамического полиморфизма (виртуальных функций)

Эта проблема возникает только при наследовании по данным.

Рассмотрим одиночное наследование:

class A

{

public:

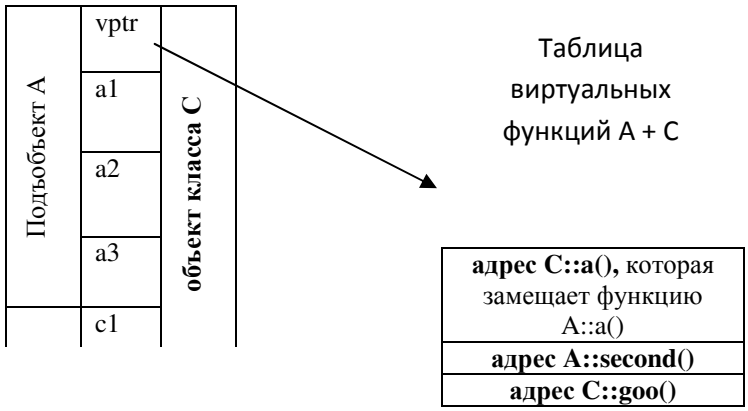
   A(){};

   virtual void a(){ a1 = 1;};

   virtual void second(){..;}

   int a1, a2, a3;

};



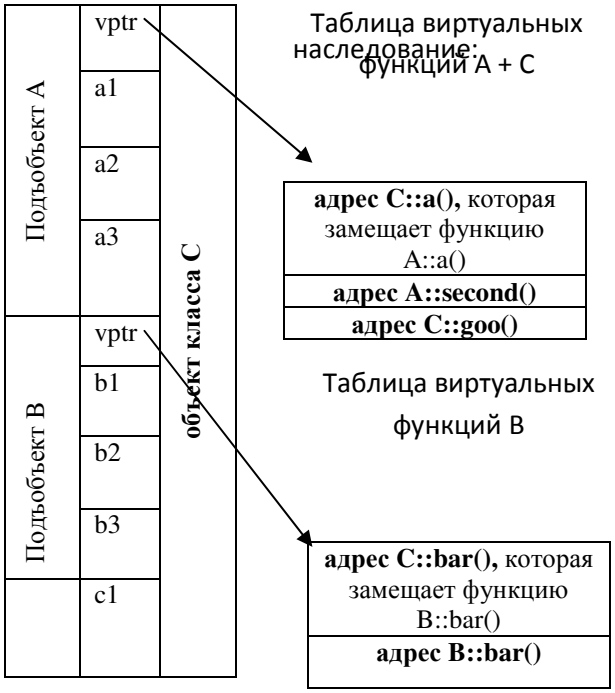

class C : public A

```
{
public:
    C() : A(){};
    virtual void goo(){};
    void a(){}; // переопределение
    int c1;
};
….
C c;
```

Если в классе C переопределить метод, то в соответствующую ячейку в таблице виртуальных функций будет записан указатель на новый метод. Если же в классе C добавляются новые функции – они дописываются в конец таблицы. При вызове методов никаких лишних действий не происходит.

А теперь рассмотрим множественное наследование:

```
class A
{
public:
    A(){};
    virtual void a(){ a1 = 1;};
    virtual void second(){..;}
    int a1, a2, a3;
};
```




```
class B
{
public:
   B(){};
   virtual void bar(){};
   virtual void bbar(){};
```

```
    int b1, b2, b3;
};
class C : public A
{
public:
    C() : A(){};
    virtual void goo(){};// Собственная новая виртуальная функция
    void a(){}; // переопределение
    void bar();// переопределение
    int c1;
};
….
C c;
```

Тут надо обратить внимание на следующее:

- Таблица виртуальных методов самого нижнего класса в иерархии доступна через первый указатель vptr.
- Каждый подобъект, который содержит виртуальные методы, имеет свою таблицу виртуальных функций.

Если в классе C переопределить метод, то в соответствующую ячейку в таблице родительского объекта будет записан указатель на новый метод. Если же в классе C добавляются новые функции – они дописываются в конец первой таблицы.

Такой алгоритм становится понятен, если рассмотреть возможные преобразования типов:

- C -> A. Через указатель на класс A можно вызывать только методы, которые прописаны в этом классе.
- C -> B. Ситуация аналогична, только мы можем вызывать виртуальные методы, определенные в классе B.

Новые виртуальные методы (которых нет в родительских классах) можно использовать только через указатель на класс C. В этом случае всегда используется первая таблица виртуальных функций.

**Сложность реализации  заключается в следующем:**

Во время преобразования типов меняется адрес указателя:

C c;

B *p = &c;

Указатель p будет содержать **адрес объекта c + смещение подобъекта B**. Т.е. все вызовы методов через такой указатель будут использовать вторую таблицу виртуальных методов объекта C. Но ведь в такой ситуации при вызове переопределённой в **C** функции через указатель на **B** в эту функцию передастся неправильный указатель this! Он будет указывать не на C, как это нужно, а на B.

Приходится **расширять таблицу виртуальных функций** добавлением в неё смещения от указателя на объект класса до таблицы виртуальных функций для каждой функции. Если виртуальная функция из B переопределена в C, то для неё такое смещение будет равно (-**смещение подобъекта B).** Если же не была переопределена, то оно будет равно нулю. Для всех виртуальных функций из класса A это смещение будет нулевым, т.к. указатель на подобъект A совпадает с указателем на весь объект C(объект A находится в начале объекта C). Теперь в функцию можно передать правильный указатель:

**this = current_this + offset**

где **current_this –** на подобъект, через который вызывается функция. **offset** – значение, которое берётся из **расширенной таблицы виртуальных функций**.

Без наследования по данным таких проблем не возникает, т.к. указатель на таблицу виртуальных функций всегда один.

**Ромбовидное и не ромбовидное наследование**

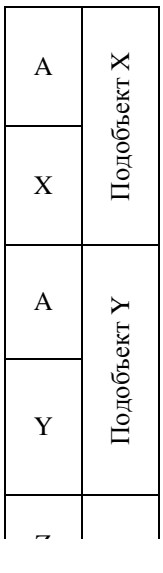
Не ромбовидное: : В объекте Z будет два экземпляра объекта A с разными реализациями таблицы виртуальных функций

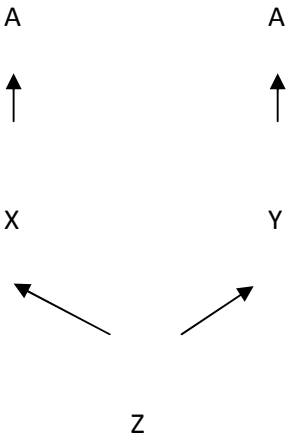
class A{ .. ;}

class X:public A{ …; }

class Y:public A{… ; }

class Z: public X, public Y {…;}






Ромбовидное: В объекте Z будет только один экземпляр объекта A

class A{ .. ;}

class X: public virtual A{ …; }

class Y: public virtual A{… ; }

class Z: public X, public Y {…;}