Лекции по Языкам Программирования 2015-го года

Авторы:
Костя Софиюк, Игорь Коняхин, Денис Купляков, Валя Макарова, Андрей Старостин, Лев Высоцкий, Саша Громов, Алик Гайбулаев, Стас Долганов, Илья Говорков (автор рукописных лекций)

Содержание

Костя ([834-856], завершено)

Виды программирования

- развлекательно-игровое
- научное (Fortran, Pascal, Python)
- индустриальное (C++, C, Ада, Java, C#, Delphi)

Языки, появившиеся для использования в web-приложениях: Python, JS, PHP, Ruby. Все эти языки интерпретируемы.

Парадигма программирования - совокупность идей и понятий, определяющих стиль программирования.
Основные парадигмы (их определения см. в файле с термином): императивная, объектно-ориентированная, обобщенное программирование, функциональная, логическая.

Объект - сущность, обладающая состоянием и поведением. Состояние моделируется членами класса. Поведение моделируется сообщениями и их посылкой.
Object C = C + Smalltalk.
Абстрактный тип данных - множество операций, без состояния.

Упоминаются термины **методы доступа, accessors и mutators (getters и setters), свойство объекта**, но нет их определений.

Далее идет сравнение языков программирования на примере задачи реверса входной строки. Примеры программ приведены для языков **C, C#, Java, Python, C++** (с использованием шаблонов в рамках парадигмы обобщенного программирования).

C++ и STL. STL включает в себя контейнеры, алгоритмы и итераторы (думаю, что стоит найти определения итераторов и контейнеров). Рассматриваются итераторы ввода/вывода для cin, cout, istream_iterator<char>.

Реализация реверса на C++ чисто через STL:

```
#include <algorithm>
#include <iterator>
…
int main() {
      vector<char> v;
      copy(istream_iterator<char>(cin), istream_iterator<char>(),
      back_inserter<char>(v));
      copy(v.rbegin(), v.rend(), ostream_iterator<char>(cout));
}
```

Функциональная парадигма на примере LISP (List Processing, 1959). В ФП данные представляются в виде списков.
Пример для LISP: `(print (reverse (read)))`

Основные точки зрения рассмотрения ЯП

- *Технологическая позиция*, отражающая взгляд человека, желающего или вынужденного пользоваться языком программирования как технологическим инструментом на каком-либо из этапов жизненного цикла программного обеспечения. Одна из основных позиций.
- *Авторская позиция*. Говорит о том как это могло быть и почему это не реализовано, и почему приняты именно такие решения. ЯП — система компромиссов. PL/1 — бескомпромиссный язык, поэтому и провалился. Часто компромиссное решение хуже, чем любое из предложенных, поэтому компромиссы не должны составлять весь язык. Авторская позиция рассматривает различные за и против включения тех или иных сущностей в язык.
- *Семиотическая позиция*, отражающая особенности языка как знаковой системы.
- *Реализаторская позиция,* отражающая взгляд на язык реализаторов транслятора с этого языка и авторов документации. Позиция человека, который собирается создать язык.
- *Социальная позиция.* Иногда нельзя не учитывать те или иные аспекты развития языка программирования в отрыве от общества. Например, Delphi лучше VB по всем четырём предыдущим позициям, но в результате популярнее стал VB. Так что важно учесть кем, в каком контексте и для кого разрабатывался язык.

В СССР основным учебным языком с 80х годов был Pascal, поэтому у нас много людей, программирующих на Delphi. В Америке же учились на Basic'е, поэтому там широко распространен Visual Basic.

Типы данных, операции и связывание, константы времени компиляции и времени выполнения. Базис парадигмы

Основные понятия языков программирования: данные, операции и связывание.

В любом ЯП есть данные, над которыми выполняются операции. В функциональной парадигме функции тоже могут быть рассмотрены как данные .

Тип данных характеризуется множеством принимаемых значений и множеством операций над ним.
Составные типы данных – это типы, которые состоят из подобъектов, то есть имеют внутреннюю структуру.
Связывание – это процесс установления связи между элементом программы и конкретным атрибутом или характеристикой. **Время связывания** – это момент установления этой связи. Связывание бывает статическим и динамическим (см. термин).

Есть два типа констант:

- **константа времени компиляции** - задаются в коде в виде конкретных значений
- **константы времени выполнения** - константные переменные, которые нельзя модифицировать

Объектно-процедурная парадигма включает в себя

1) Базис - то, на чем строятся все остальные конструкции. Базис бывает скалярным (то, что не имеет внутренней структуры, простые типы данных) и структурным (составные типы данных, выражения, операторы).
2) Средство построения абстракции
3) Средство защиты построенных абстракций

Основным расширением скалярного типа данных в C++ относительно C есть ссылочный тип данных.

Побочный эффект - изменение состояния выполняемой программы.

Различие между readonly и const в C#

Ключевое слово **readonly** отличается от ключевого слова **const**. Поле с модификатором **const** может быть инициализировано только при объявлении поля. Поле с модификатором **readonly** может быть инициализировано при объявлении или в конструкторе. Следовательно, поля с модификатором **readonly** могут иметь различные значения в зависимости от использованного конструктора. Кроме того, поле **const** является константой во время компиляции, а поле **readonly** можно использовать для констант времени выполнения.

Имя – это строка символов, служащая для обозначения некоторой сущности в программе.
Область видимости (scope) - обозначает область программы, в пределах которой идентификатор (имя) некоторой переменной продолжает быть связанным с этой переменной и возвращать её значение. (из википедии)
Область видимости (действия) начинается от языковой конструкции, где вводится объект данных. С каждым объектом связываются тип, область действия, время жизни.

Виды объектов (статические, квазистатические…)

- статические — существуют всё время работы программы
- квазистатические — существуют только во время работы некоторой синтаксически выделенной конструкции (блока)
- динамические — существуют до тех пор, пока нужны: до сборки мусора, удаления
- хранимые (persistent)

Объектно-референциальная модель

Объекты - принципиально анонимные, именования являются ссылками на объекты. В таких моделях все параметры передаются в функции всегда по ссылкам. Связывание с другими объектами чисто динамическое.

В языках C# и Java ссылки - это единственный способ обращения к объектам. В этих языках принята референциальная объектная модель.
Примеры:

Java	Алгол 68

`int[] a; // объявили именованную ссылку, но пока пустую`
`a = new int [10]; // связали именованную ссылку с объектом`
`X x;`
`x = new X();`
	`int i;`
	`ref int a = i; // ссылка на int`
	`ref ref int b = a; // ссылка на ссылку на int`

Примитивные ТД. Объявление переменных в Ада

- арифметические
- логический boolean
- символьный char
- указатель
- порядковые ТД (определяются на уже описанных). Например, перечисления.

Описываются какие-то особенности языка Ада: в основе всех типов лежит какой-то канонический тип (дальше не получилось разобрать, что написано). Описание синтаксиса объявления в Аде. В Аде нет неявного приведения типов.

```
x: positive;
i: interger;
n: natural;
x := positive(i);
i := integer(x);
```

Особенности копирования объектов в Java, в C#

Использование оператора присваивания не создает нового объекта, а лишь копирует ссылку на объект (т.н. **поверхностное копирование**). Таким образом, две ссылки указывают на одну и ту же область памяти, на один и тот же объект.
Для создания нового объекта с таким же состоянием используется **клонирование объекта**:

- Клонирование объекта можно реализовать, имплементировав интерфейс Cloneable и реализовав копирование состояний полей и агрегированных объектов. Такая реализация клонирования требует большого внимания и может стать источником ошибок.
- Более безопасным способом является использование конструктора копирования, которое позволяет избежать многих проблем.
- Другим безопасным вариантом является фабричный метод (Factory method), который представляет собой статический метод, возвращающий экземпляр своего класса. Фабричный метод имеет следующие преимущества перед конструктором копирования:
  - Имеет имя (чаще всего getInstance или valueOf), что делает код более понятным.
  - Необязательно создавать новый объект в результате вызова: Объекты могут быть кэшированы и реиспользованы.
  - Могут возвращать подтип своего возвращаемого типа. В частности, могут возвращать объект, у которого неизвестен класс реализации.

В C# по умолчанию тоже используется поверхностное копирование. Для глубокого копирования можно использовать несколько подходов:

- Глубокое копирования посредством сериализации (самый удачный подход). С использованием класса BinaryFormatter и методов Serialize, Deserialize. Класс должен быть помечен как “[Serializable]”.

Сборка мусора

Автоматическая сборка мусора. Если на объект больше нет ссылок, то память выделенная под объект освобождается сборщиком мусора.
Висячие ссылки – это адреса уже уничтоженных объектов.
Мусор – это неуничтоженные объекты из динамической памяти, на которые отсутствуют ссылки.

RAD (Rapid Application Development)

Концепция создания средств разработки программных продуктов, уделяющая особое внимание быстроте и удобству программирования, созданию технологического процесса, позволяющего программисту максимально быстро создавать компьютерные программы.
RAD предполагает, что разработка ПО осуществляется небольшой командой разработчиков за срок порядка трёх-четырёх месяцев путём использования инкрементного прототипирования с применением инструментальных средств визуального моделирования и разработки. Технология

RAD предусматривает активное привлечение заказчика уже на ранних стадиях — обследование организации, выработка требований к системе. (из Википедии)

JIT, MSIL

Языки бывают **компилируемыми** и **интерпретируемыми** (некомпилируемыми).
JIT-компиляция (Just in Time) - технология увеличения производительности программных систем, использующих байт-код, путём компиляции байт-кода в машинный код или в другой формат непосредственно во время работы программы. Таким образом достигается высокая скорость выполнения по сравнению с интерпретируемым байт-кодом (сравнимая с компилируемыми языками) за счёт увеличения потребления памяти (для хранения результатов компиляции) и затрат времени на компиляцию (from Википедия)

MSIL (Microsoft Intermediate Language) - промежуточный язык, разработанный фирмой «Microsoft» для платформы .NET Framework. JIT-компилятор MSIL является частью CLR (англ. common language runtime) — общей среды выполнения программ, написанных на языках .NET. В настоящий момент он переименован в **CIL** (**Common Intermediate Language).**

Java = ядро Java + JVM + окружение.
Окружения Java:
- Java SE (Standart Environment)
- Java ME (Mobile Environment)
- Jave EE (Enterprise Edition)

Вся стандартная библиотека Java представлена в виде совокупности пакетов.
`java.<имя пакета>`

В C# и в Java есть некоторые части, которые выглядят как классы, но на самом деле вшиты в компилятор:
- String в Java
- System в C#

Игорь ([857-871], завершено)

Скалярные базовые типы данных. Арифметические типы данных

- Целочисленные типы
- Вещественные
    - С плавающей точкой. Определён стандартом IEEE-754. Принят и реализован производителями процессоров, а следовательно и вошёл в языки программирования.
    - Фиксированные - мало прижились в индустриальных языках

Целочисленные типы

В Modula 2 Вирт ввёл integer - знаковый тип и cardinal - беззнаковый тип, при этом неявных преобразований между ними быть не может.

Страуструп при создании C++ решил, что квалифицированные программисты понимают особенности неявных преобразований типов (знаковые <-> беззнаковые) и сами могут решить, что нужно написать, поэтому не нужно запрещать такие преобразования.

Вычисления с char и short производятся только через int, поэтому для производительности важно, чтобы int был родным. В таком случае потерь производительности почти нет.

В C# есть uint-беззнаковые условия и можно заставить, например, дополнительно проверять выход результата за границы допустимых значений:

```
uint i;
cheched {
      for (i = 0; i >= 0; i--) { }
}
```

checked будет дополнительно проверять выход из беззнакового типа.

C# использует **CTS (Common Type System)**

1	sbyte	byte
2	short	ushort
4	int	uint
8	long	ulong

Размеры типов строго зафиксированны, а неявные преобразования возможны только в случае, если гарантированно не происходит потери значений, т.е. если тип, в который происходит преобразование может хранить любое значение исходного типа.

Возможные неявные преобразования:

```
byte -> short -> int -> long
ushort -> int -> long
uint -> long
byte -> ushort -> uint -> ulong
```

(возможно любое преобразование в пределах строчки слева-направо, например, byte -> long)

При появлении новой архитектуры factum64 встала проблема переноса программ с 32-битной архитектуры на 64-битную. Это оказалось очень сложно и дорого. Ничего не получалось до появления архитектуры x86-64, оказавшейся крайне энергонеэффективной, но полностью совместимой с x86 по командам и с 64-битным размером виртуального адресного пространства.
Размер int остался равным 32 битам.
Реальной, чистой 64-битности нет.

Битовый сдвиг может быть как влево, так и вправо, как арифметический, так и логический, но реально есть только три кода операций, так как арифметический и логический сдвиг влево эквивалентны:

sal ⇔ shl

sar - арифметический сдвиг вправо. Старшие биты результата заполняются старшим битом аргумента

shr - логический сдвиг вправо. Старшие биты результата заполняются нулём

В C, C++, C# какой сдвиг выполняется зависит от типа целого числа. Если целое число знаковое, то выполняется арифметический сдвиг, если беззнаковое - логический. **В Java беззнаковых чисел нет** (за исключением 1-байтового типа ‘byte’). Поэтому чтобы отличить арифметический сдвиг вправо от логического в Java пришлось ввести два разных оператора:
i >> N - арифметический сдвиг

i >>> N - логический

Вещественные типы

По стандарту любое вычисление с float происходит через расширение до double. В printf всегда дадут double, хотя выдаст он точность float.

IEEE-754 унифицировал представления и арифметику чисел с плавающей точкой для 32 и 64 бит. В целом произошло упрощение, вещественных чисел за счёт принятия IEEE-754 и целых чисел за счёт унификации арифметических оперций в x86, x86-64 и нескольких RISC архитектур вроде ARM.

Пример определения вещественных чисел с фиксированной точкой в Ada:

```
type Def is delta H range L..R
```
где
H - шаг, по сути - точность
L, R - можно предположить, минимальное и максимальное число

Компилятор может сам выбрать удобное ему представление. Например, если L и R не слишком велики, то для представления данного типа, он может выбрать целое число. Всего требуется хранить (L - R)/H возможных чисел. Это определяет минимально необходимую разрядность регистра. Стараемся выбрать поменьше, чтобы экономить память.

Впервые вещественные числа с фиксированной точкой появились в Cobol, затем - в IBM-PC x86. Кроме того, в IBM-PC x86 есть двоично десятичные числа. Каждое число кодируется своими десятичными цифрами, каждая из которых в свою очередь кодируется 4 битами. Например 947 будет закодировано как 1001.0011.0111. Не эффективно по памяти, но обеспечивает быстрый ввод/вывод.
Также есть десятичный тип данных. Decimal в SQL и decimal в C#.

В Python есть только плавающий и целый (базисный\) тип. Длинных вещественных чисел нет, а длинные целые - есть. Число, если не умещается в 32 бита, становится 64-битным, а если не влезает и туда, то уже будет моделироваться несколькими числами, хотя и медленнее.

Javascript, Lua

Вообще только один тип - Number - 64 битное вещественное число, но там нет точной точности.

Почти любой язык программирования поддерживает логический тип данных как отдельный, не совместимый ни с каким другим, тип данных. Но не стоит забывать про C++ в котором ‘bool’ неявно конвертируется в ‘int’.

Перечислимые типы

Вирт, как создал его в Pascal, так и убил его в Oberon, по двум причинам:

1) При импорте из другого модуля перечислимого типа неявно импортируются и все имена констант
   `type col = ( red, green, blue );` в одном модуле
   `type Col = ( red, green, … );` в другом модуле
   возникнет конфликт

2) Перечислимый тип - не расширяемый, а в Oberon была концепция расширения типа, а такой (перечислимый\) тип просто противоречит ООП.

Однако перечислимый тип очень подходит для визуально компонентного программирования:
- Сборочное программирование - не надо ничего расширять
- Например, визуальное проектирование интерфейсов. Можем выбирать свойства объектов из списка перечисления. Становятся доступными подсказки к коду.

Заметим, что 1-ая “проблема” легко обходится разграничением пространства имён как в C++:

```
enum Color { red, green, blue }; // глобальные константы, обращение прямое: red
enum class Colour { red, green, blue }; // обращение: Colour::red
```

В некоторых случаях для флагов используются целочисленные значения, которые можно комбинировать с помощью битовых операций &, |, таким образом избегается необходимость использовать массив флагов. Например,

```
int Maximized = 1 << 0;
int Minimized = 1 << 1;
int Centered = 1 << 2;
int TopMost = 1 << 3;
int BorderLess = 1 << 4;
// Для создания вируса можно скомбинировать флаги Maximized, TopMost и
BorderLess
CreateWindow(“Virus”, Maximized | TopMost | BorderLess);
```

Повторим проблемы перечислимого типа:
- Неявный экспорт констант
- Нерасширяемость

Но он все равно вошёл в компилируемые языки, хотя и с изменениями.
В C#: enum States { U, A, … }
+ Можно выбрать базовый тип перечисления из встроенных числовых типов (byte, short, int), таким образом сэкономив память для перечислений с небольшим количеством констант (т.е. для большинства)
+ выделяется в отдельное пространство имён

Но в целом дано разработать иерархию для наследования очень сильно. Поэтому нерасширяемость не так уж проблематична.

В Java перечисления появились не сразу. В первой версии языка в 1995 году их не было. Появились они лишь в 2005 году, но сделали их классовым ссылочным типом данных. enum States { A, B, C } по умолчанию наследуется от Object. A, B, C - самоопределённые константные имена без типов по умолчанию static, public, const члены. Кароч небольшое расширение синтаксиса. Но это класс, можно писать методы, конструкторы, его можно наследовать.

Диапазоны массивов из Pascal не выжили нигде, хотя были в Ada: range L..K
Везде прижился C-подход к индексции массива, при котором массив длины N, имеет индексы 0..N-1, а сами индексы имеют тип int.

Символы и кодировки

Символьный тип данных char был ещё в Pascal. Но сейчас перешли с 8бит на Unicode.

1991 год - появился стандарт Unicode. Он состоит из двух основных разделов: универсальный набор символов (англ. *UCS, universal character set*, бывает UCS-2 и UCS-4) и семейство кодировок (англ. *UTF, Unicode transformation format*). Универсальный набор символов задаёт однозначное соответствие символов кодам — элементам кодового пространства, представляющим неотрицательные целые числа. Семейство кодировок определяет машинное представление последовательности кодов UCS.
Проще говоря, UCS сосоставляет иероглифу код 26218, а UTF-8 (например) сопоставляет коду 26218 представление на компьютере в виде последовательности бит F0 A6 88 98.

i18n = internationalization (18 - количество букв между ‘i’ и ‘n’) = когда пишем программы сразу с расчётом на несколько языков и возможностью локализации проложения.

globalization - меню выводится на Английском языке, но вводить можно на любом языке.

0..127 <разбивка символов как в ASCII и совпадает во всех кодировках>
128..255 <ISO-latin1> - первое расширение

KOI-8R - особая кодировка для русских букв, которая позволяет прочитать русские буквы даже с обнулённым правым битом.

В UTF-8 символ может занимать от 1 до 6 байт. Количество байт и кодирование числа определяется в соответствии и минимально необходимым количеством бит для представления данного кода в двоичном виде. В приведённой таблице количество ‘x’-ов в строчке соответствует количеству значащах бит - на место этих ‘x’-ов и подставляется стандартная двоичная форма данного кода.

Количество байт	Значащих бит	Шаблон полностью
1	7	0xxxxxxx
2	11	110xxxxx 10xxxxxx
3	16	1110xxxx 10xxxxxx 10xxxxxx
4	21	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
5	26	111110xx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx
6	31	1111110x 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx

В UTF-8 все коды от 0 до 127 кодируются также как в ASCII. Поэтому если текст закодирован в ASCII, а просматриваем мы его как UTF-8-текст, то текст на Английском языке будет оторажаться правильно, а тексты на дургих языках будут отображаться неправильно.
Конец строки:
UTF-8: 0x00
wchar_t: 0x0000 - чистый Unicode

В UTF-16 коды из UCS-2 кодируются 2-х байтовым числом. Коды, которые не влезают в UCS-2 кодируются парой двухзначных чисел.

Но появляется проблема порядка:

FF.F**E** - big endian - порядок, в котором числа записывает человек
F**E**.FF - little endian - порядок, в котором младшие байты идут вначале (т.е. логичный)

Процессор использует little endian, а сеть - big endian, поэтому для определения того, какой порядок используется, строка в Unicode должна начинаться с BOM - Byte Order Mark, специальных двух байт = 0xFFF**E**, по которым определяется используемый порядок байт.

Комментарий! Вообще, на мой взгляд проблема высосана из пальца. Сеть должна передавать поток байтов в точности как ей говорят. Причин для изменения порядка байтов не видно. Проблемы не возникает в UTF-8, где кодирование идёт побайтово. При реализации UTF-16, по всей видимости используются, функции работающие с блоками по два байта и на разных архитектурах сохранение этих байт идёт в разном порядке.

В языках программирования, появившихся после стандарта Unicode, строковым или символьным типом является Unicode, и он неявно не преобразуется к числу. К таким языкам относится C#, Java, JavaScript. В C# и Java есть Char, обычно в UTF-16.
Проблемы появились, у языков, появившихся до 90-х. В C и C++ всё решили простым расширением. В C++ появились новые типы: wchar_t, wstring, соответствующие функции добавили определения Unicode символов и строк:
L'c' => Unicode символ
L"stss bla bla bla" => Unicode строка

Больше проблем возникло в Python. В версиях 2.* были как Unicode, так и не-Unicode строки: ' ', " ", w" " (Unicode строка), а в версии 3.* оставили только Unicode строки обозначаемые теперь просто ' '. Таким образом поменялась семантика программ.

Итого структура простых типов данных:
- Арифметические
- Перечислимые в компилируемых языках
- char -> string
- bool

Указатели и ссылки

К указателям в разных языках разные подходы:
- C/C++:
  Указатель - абстракция адреса, в самом общем виде - void*
- Pascal/Ada:
  Указатель используется для моделирования динамических структур только для обращения к выделенным объектам в динамической памяти
  Ключевое отличие от C/C++ - **нет адресной арифметики**

Ada 83
```
type R is record
     A: T1
     B: T2
end record;
type PR is access R; // Указатель
X:PR;
X = new R;
X.A ; X.B ;
X.all - ссылаемся на весь объект (зачем?\)
Y:R;
Y.A, Y.B; //
```
Т.е. что получается, синтаксис для доступа к полям через указатель и объект одинаковы? \

Благодаря тому, что указатель указывает только на динамическую память должно быть проще создать сборщик мусора. Но в первом стандарте Ada 83 сборщика мусора не было, так как язык был разработан для задач реального времени. Главная проблема сборщика мусора заключается в том, что свободная память должна быть в едином пуле.
Создатели Ada исходили из трёх критериев:

1) Надёжность (так как разрабатывался для задач обороны)
2) Эффективность (для задач реального времени)
3) Читаемость (для сопровождения)

но добавили модуль unchecked_deallocation - часть стандартной библиотеки, но отдали на откуп компилятору. Компилятор может сам сделать сборку мусора и проигнорировать unchecked_deallocation.

Ada 95
Ввели другой тип указателя, так как вся API было из окружения и написано на C, то надо было вызывать API на другом языке, а там (на C) главное - указатель, а значит нужно ввести указатель и у себя:

```
type PI is access all integer;
x:PI;
i:integer;
x := new integer;
x := i'access;
```

При этом нововведённому указателю можно присваивать старый, но не наоборот. Такой указатель стал указателем по типу языка C. Он может указывать на любой объект. Но в самом языке это было не нужно. Добавили только для взаимодействия с внешним миром.

Заметим, что для C и C++ нельзя создать сборщик мусора, так как указатель может указывать на любое место, в том числе и в статической памяти.
CLR - особая реализация, в которой ввели сборщик мусора. Для этого появился особый тип указателя, который может указывать только на динамическую память, и к которому не применима адресная арифметика. Только к таким указателям и применима сборка.

Все типы в C#

- Типы значения (скалярные типы данных)
  - char
  - bool
  - int, float, decimal, …
  - struct (в C# не тоже самое что class!)

  Где объявлены, так и выделяется под них место

- Референциальные типы
  - массивы
  - интерфейсы
  - класс
  - делегаты: функциональный тип данных, по сути частный случай ссылки

  int[] a; // Объявляется только ссылка на массив, память под сам массив не выделяется
  a = new int[50]; // Выделяется память на сам массив в динамической памяти

Любой сборщик мусора может перенести адрес. После выделения большого количества объектов и освобождения части из них, может получиться так, что свободной памяти имеется много, но

большого последовательного куска памяти нет. Возникает необходимость произвести переразмещение объектов с целью уплотнения данных в памяти. Сборщик мусора делает это автоматически. Это одна из причин возможных подвисаний программ с автоматической сборкой мусора. Также объекты могут изменить свои адреса и традиционные указатели на данные объекты стали бы недействительными, но ссылки являются более высокоуровневым типом данных, они автоматически обновляются и продолжат работать правильно.

C# -> .Net работает в VES (Virtual Execution System) над определённой ОС. При необходимости использовать родные для данной ОС API, нужно использовать систему Platform/Invoke.\

Java -> JNI (Java Native Interface). Особый интерфейс для обращения в родным для данной ОС сервисам. При использовании приводит к абсолютной непереносимости, но зато эффективно.

Блок fixed в C#

В C# ввели блок fixed, который фиксирует адреса всех объектов, используемых внутри блока, не применяя к ним стандартный сборщик мусора. Адреса фиксированы, а значит можно их получить:

```
byte[] b = new byte[1024];
fixed { byte* pbffer = b; … }
```

Просходит преобразование ссылки в традиционный C-шный указатель без всякого контроля. Внутри fixed можно использовать любые функции из библиотеки языка C. Но требуется указать атрибут unsafe, таким образом мы “подписываемся” под тем, что вся ответственность лежит на нас. И сборку тоже обязаны делать с атрибутом unsafe. При этом на unsafe проекты накладывается ограничение, их нельзя запускать без ЭЦП (гарантия неизменности bin-файла).

Ссылочный тип в C++

В C++ ссылка всего лишь “альтернативное имя” объекта

```
T& a;
T& a = *new T();
delete(&a); // автоматического освобождения не произойдёт
```

Единственная операция со ссылкой - инициализация объектом, всё остальное применяется к объекту.

```
T x;
T& a(x);
T& a = x;
```

Если ссылка является частью класса, то инициализировать её можно только в конструкторе в списке инициализации:

```
class T {
public:
      T(): <вот здесь> { }
}
```

Время жизни ссылки не может быть больше времени жизни объекта.

Похоже на другие языки программирвания то, что сам выделенный объект не может изменить свою длину.

В плане скалярного базиса типов данных языки программирования различаются не слишком сильно

а) Большой набор типов под архитектуру машины (Java, C#)
б) Немного типов, не эффективно (Python, Ruby)

Структурный базис

Совершенный хеш - это хеш без коллизий и с абсолютно плотным отображением. Его можно построить для любого конечного множества.

Паскаль:
- Массивы
- Записи (то, что в С называется структурой)
- Строки

Эти три есть по сути в любых языках программирования, а также:
- Множества
- Файлы

Структурный базис объектно ориентированных языков программирования
В любом статичном (компилируемом) языке (C, C++, C#, Java), последовательности:
- массивы
- строки
- записи/структуры => класс

Из базиса ушли файлы, множества и перешли в стандартную библиотеку.

Массивы

Остановимся на понятии массива. Тут есть три операции:
- A := B поэлементное копирование элементов из B в A
- A[i] индексирование, возвращает элемент с номером ‘i’ массива A. Срабатывает при обращении к элементу в массиве (printf(“%d”, A[i]))
- A[i] -> ref T возвращает ссылку на элемент массива ‘A’ с номером ‘i’. Срабатывает при присваивании элемента массива (A[i] = 50)

При этом время доступа константное, в отличие от словаря, где время доступа обычно O(logN).
Также заметим, что суть массива отражает принципы фон Неймана:
- Однородность (все хранимые данные имеют один и тот же тип)
- Линейность (время доступа к элементу не зависит от ‘i’)

С массивами иногда может появиться неэффективность. Например, когда массив выделяется в динамической паамяти, на момент трансляции адрес массива не известен, а известна ссылка. Возникают расходы на разыменование и сборку мусора.

В Паскале все адреса массива известны в момент трансляции. Память используется эффективно, но и длины массивов статичны. Некоторые же языки позволяли выделять массив в стековой памяти. Например, Ada:

```
procedure P(N:integer) is
      X: array range 1..N of integer;
      Y: array range 1..2*N of real;
begin
…
end
```

Андрей ([872-879], завершено)

Выделяется память в момент входа в процедуру и освобождается в момент выхода (квазистатические массивы - не могут меняться после создания). Но мы все-равно не знаем, где именно в стеке лягут переменные, но зато мы точно всегда знаем offset.
Из-за такого вида динамических массивов нужно дополнительно иметь смещение относительно кадра стека, но они все-равно делигированные..
Постоянная борьба эффективность/гибкость.

Неограниченный массив Ада

В Аде есть неограниченный тип данных.
Общее объявление типа массива:
`type LARR is array INDEX range L..R of T;`
INDEX - тип индекса, L,R - константные выражения типа индекса, T - тип значения
Неограниченный массив:
`type ULARR is array Integer range <> T;`
На самом деле все фуфло, потому что при объявлении переменной размер все-равно надо указать:

`y : ULARR;` ← нельзя

`z : ULARR range 1..N;` ← можно, если N - const

`w : ULARR range 0..N-1;`

Единственная плюшка - переменные z и w будут формально одного типа, и поэтому с ними можно делать присваивания, если их длины совпадают `z := w`.
Неограниченный тип удобен для определения процедур, где аргументом является массив, так как по определению адрес статически мы не знаем.
Во всех операциях с неограниченными типами включается квазистатический контроль (либо сам компилятор, если знает все параметры, проверяет операцию, либо вставит проверку)

Работа с массивами Ада

```
function Sum(A: ULARR) return T;
      count : T := 0.0;
begin
      for i in A'RANGE
            Sum := Sum + A(i) ← Индексация круглыми скобками
      end;
      return count;
end Sum;
```

Атрибутные функции массивов Ада: A'LENGTH, A'UPPER, A'LOWER, A'RANGE

Массивы в Модуле-2

Открытый массив:
`TYPE A: ARRAY of T;`
Обычный массив:
`TYPE B: ARRAY[L..R] of T;`
При этом переменные типа A могут быть только формальными параметрами процедур и функций (их нельзя создавать)

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

Модула-2 упоротый язык, и в нем нельзя получить нижнюю границу массива, только верхнюю (HIGH(A)).

Массивы в Обероне

Есть многомерные открытые массивы. Они могут использоваться только в качестве формальных параметров функций.
Нет беззнакового типа. Нет диапазонов.
Массив задается:

B : ARRAY N of real; ← N - длина (индексы будут от 0 до N-1)

Массивы в JAVA и C#

Современные языки имеют референциальную модель (все объекты составных типов хранятся в динамической памяти)

T[] a = new a[N]; ← Длина нужна только для инициализации

Поэтому обращение `a[i]` всегда поразумевает два разыменования.
Это квазастатические массивы: длина постоянна, но задается в момент инициализации.
(Это нужно для эффективности. Потому что алгоритм расширения/сужения массива неуниверсален и поэтому не входит в базис)
В плюсах есть vector, у которого операция erase, удаляющая все элементы массива, не производит реаллокацию, т.е. зарезервированное место сохраняется.

Массивы в динамических языках (JS, Python)

JavaScript
Массив - это объект, прототипом которого является Array. По сравнению с обычными объектами, массив имеет дополнительное поле length, содержащее длину, а также набор специфичных функций вроде push, map и т.п. При обработке массива учитываются только поля с целочисленными ключами.
Про объекты в JS ниже по тексту

```
var a = [1,2,3];
console.log( a.length ); // 3
a.length = 4;
console.log( a ); // [1,2,3,undefined]
```

Полностью динамический массив, как в питоне.

Python

Есть обобщенное понятие последовательности. Последовательность - это такая штука, которую можно индексировать.
Есть списки, которые на самом деле массивы, но называются списками.
`a = [1,"str",3]` // непрерывная область памяти, константная по времени индексация
Списки полностью динамические. Можно удалять/вставлять элементы. Реаллокация происходит автоматически. Можно смешивать разные типы данных

Есть кортежи (tuple)
`a = (1,2,"str")` // неизменяемая длина, может быть ключом в словаре

Есть строки
`a = "str"` // неизменяемая длина, при модификации строк просто копируется в новое место, можно использовать как ключ в словаре
Строка - это не просто массив чаров, а отдельный объект со своими функциями.

Есть словари
`a = {'a': 1}`
Ассоциативный массив, реализован в виде хеша. При этом все ключи должны быть одного типа

Есть генераторы. Генератор - это такая штука, которая создает последовательность на ходу по необходимости. (Как потоки в Лиспе)
```
def natural():
      i = 0
      while True:
            yield i
            i += 1
```
`natural()` - создает генератор, который создает бесконечную последовательность натуральных чисел. Как все остальные последовательности, можно использовать в циклах.

Со всеми последовательностями можно делать следующее:

s.index(obj) ← Поиск первого входждения элемента в последовательность

s.count(obj) ← Подсчет количества вхождений элемента в последовательность

s[2] ← Индексация

s[2:3:1] ← Вырезка (slice) со 2го по 3й элемент с шагом 1.

В Python `range(right), range(left, right), range(left, right, step)` - это генераторы соответствующих последовательностей (В Python 2 - функции. создающие списки)

В Python нет цикла for, есть только foreach, который называется for. Цикл обязательно принимает на вход последовательность. Через foreach можно имитировать обычный for:

`for x in range(left, right, step)` в Питоне
Эквивалентно
`for(int x = left; step > 0 ? x < right : x > right; x += step)` в Сях

Последовательности в Java и C#

В Java и C# есть интерфейсы для создания последовательностей: Iterable и IEnumerable соответственно, а также специальные циклы для перебора этих элементов.

C#

Цикл

```
foreach (elem in collection) { … }
```

Для работы с итераторами в .NET есть специальный язык LINQ (Language Integrated Query), который позволяет в декларативном стиле описывать преобразования данных. LINQ использует ленивое порождение объекта, то есть генераторы.

Записи (структуры) в C++, C#, JS (объекты)

Запись - это как структура в Сях. Или массив из разнородных элементов, но таких, чтобы все смещения можно было вычислить на этапе компиляции.
В динамических языках такая штука отсутствует за ненадобностью, так как там есть ассоциативные массивы (хеши).
Даже в Java ее нет.

C++

```
struct Record { int x; double y; }
Record t;
```

t.x, t.y; ← Обращение к полю записи

```
Record* t = …
```

t->x, t->y; ← Обращение к полю записи через указатель на запись

Оператор `->` можно перегрузить: `T* operator ->() { … }`
Это обычно используется для оберток над ссылками (врапперов)

C#

Есть структуры. В отличие от C++, в C# структура отличается от класса поведением. Классы в C# являются ссылочными типами. Структуры - типами-значениями. Это значит, что в отличие от ссылок, доступ к полю структуры требует только одно разыменование. Кроме того, структуры хранятся на стеке и присваиваются полным копированием.
Это нужно для эффективности, особенно если приходится постоянно создавать/удалять объекты, что на стеке делается быстрее, чем в куче.
Структуры нельзя наследовать, переопределять в них конструктор по умолчанию.

Объекты в JS

Это ассоциативные массивы с улучшением в виде прототипного наследования.

```
var a = {}; // пустой объект имеет прототип Object и набор стандартных методов
var b = { cat: 5 };
a[“dog”] = b.cat;
b[“cat”] = a.dog + 1;
// a = { dog: 5 }   b = { cat: 6 }
```

Обращения a.cat и a[“cat”] эквивалентны, как и a[1] и a[“1”]. Все ключи объектов на самом деле строки. Если ключ не найден, то возвращается undefined.
Про прототипное наследование как нибудь потом.

Денис ([880-903], завершено)

Управление последовательностью вычислений

Два типа сущностей:

1. вычисления (выражения, неявное управление)
2. последовательность операторов (явное управление)

Выражения

В рамках одного выражения последовательность вычислений зависит от семантики, при этом если семантика допускает неоднозначность, то порядок не определен.

В функциональных языках есть “Ленивые вычисления” (операнды вычисляются *только* при реальной необходимости).

Логические операции `and`, `or` – ленивые (в `and` если левая часть false, то правая не вычисляется).

Первый вариант **Ada** предлагал: порядок вычислений не должен иметь таких особенностей, предложили специально добавить `andthen` и `orelse`. Окончательно (83), оставили ленивые `and` и `or`.

В **JavaScript** если операнды не `bool`, то возвращаются сами значения операндов (Денис: если написать `'foo' or 'bar'` то получится `'foo'`, а не `true` или `false`)

Циклы, переходы

Fortran: единственный цикл – `DO`, с фиксированным числом повторений. Все остальное переходы; к концу — 3 типа `goto` (+ еще появились).

1967: Дейкстра написал статью «Goto statement considired harmful» -- говорит, нужно выбрать 1 форму управляющих структур, чтобы ясна была структура алгоритма.

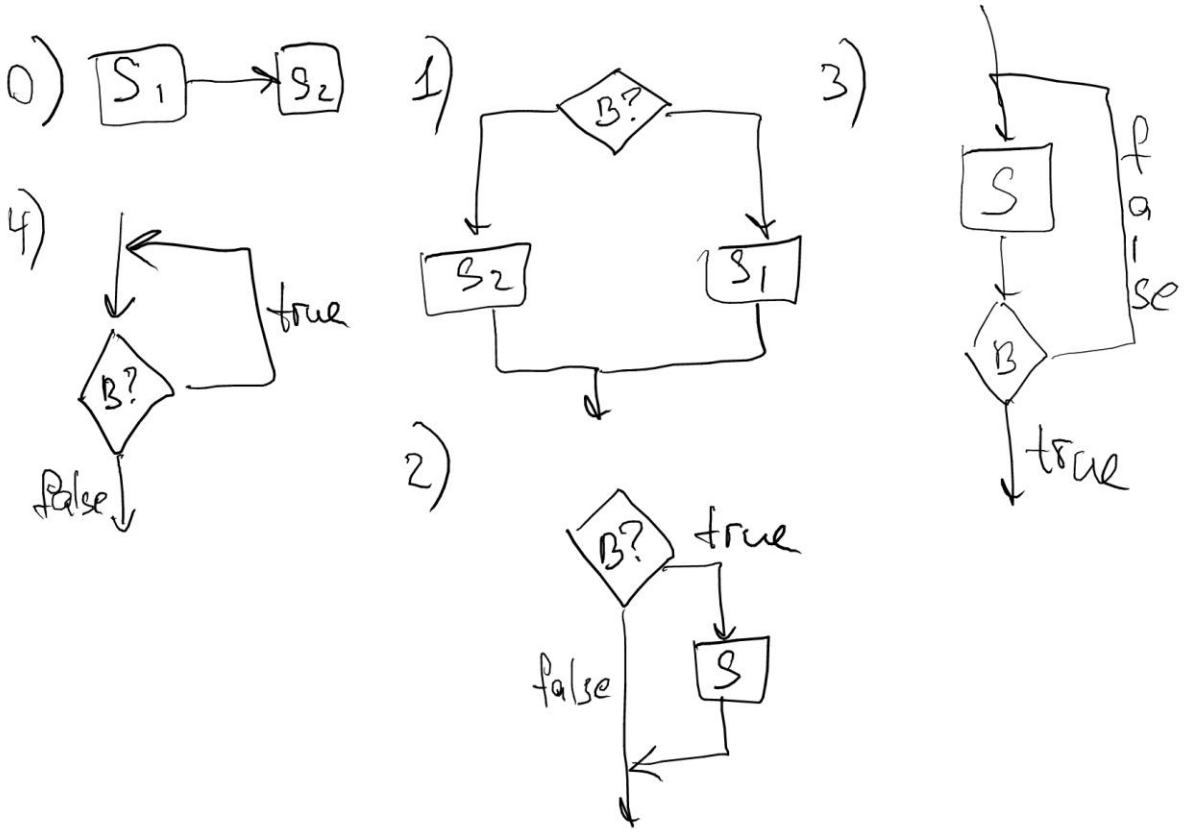
Программа — последовательность преобразований предикатов (формальное программирование – дает доказательство правильности программы).

P1 (k) P2, где *k* — управляющая структура с определенной семантикой – четко определено, какой именно предикат *P2* получится из *P1*.

P -- инвариант цикла, если	P – верно для любого состояния при вычислении S `while B do S` P and not S – верно если цикл закончится

Идея “доказательного программирования” – очень сложна, т. к. любое доказательство сложное → более современная идея «программа – надежная совокупность ненадежных компонент» → assert, тестирование и т. д.

1967 – «Notes on Structural Programming» 3 статьи → появление структурного программирования. Управляющих структур, которые надо использовать достаточно мало – программист тем лучше, чем меньше видов упр. структур он использует



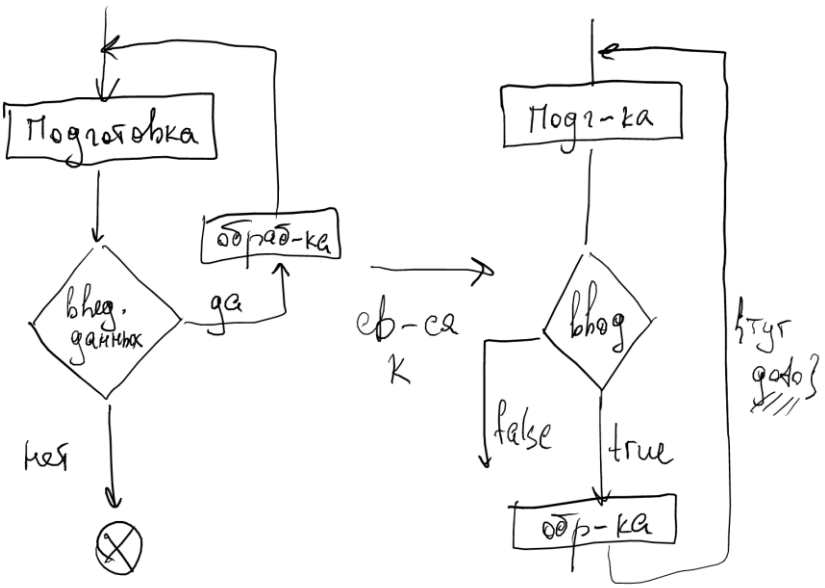
1965 – доказали, что любую простую программу можно реализовать только с использованием 0), 1), 4).

3 статьи:

1. Дейкстры о `goto` (против **DS-programs** -- Dish of Spagetti, где используется только goto)
2. Хоара(на её основе Вирт, добавив `for i := e1 to/downto e2 do S`, создал **Pascal**)
3. ???

1974 -- Д. Кнут, “Structural Programming with goto statements”, пишет, что идея Дейкстры в разделении программы на уровне абстракции и дальнейшей детализации пока не дойдем до уровня операторов ЯП.

Пишет, что указанных управляющих структур не хватает, т.к. многие программы имеют вид:



`prepare;`
`while B do`
`begin`
`  process;`
`  prepare;`
`end`
	• 2 раза повторяется `prepare`
	• пришлось подгоняться под существующую структуру
	• лучше сделать `goto`, если такой структуры нету в ЯП

Итого:
1) условные переходы -- `if-then-else` или многовариантный выбор (цепочка `if` или `case`)
2) циклы (`while-do; do-while; for`)
3) "Заменители `goto`" (`break; continue; return`, дополнительные `goto, throw`)

В 70-м появились языки без `goto`: **Java**, **Modula-2**

В **Java** `goto` зарезервировано, но не используется, и до сих пор не появился. Но зато есть расширенный `break M`; где `M` -- метка начала конструкции, из которой надо выйти (Денис: в случае вложенных `for`-ов например, можно сделать `break` для внешнего из внутреннего)

RAII -- В **C++** *Resource Acquisition Is Initialization*, любой ресурс надо завернуть в класс, инициализация в конструкторе, а уничтожение в деструкторе, при этом из-за свертки стека все уничтожится. Если нет свертки, то возникнет куча проблем (`try-catch` позволяет не использовать `goto` для чистки ресурсов в конце программы).

Составные структуры

Заметим, что простые управляющие структуры составные:
`if then S1 else S2` -- S1 и S2 операторы, которые надо различать

(Учебник: во всех языках: `if A if B else C <==> if A { if B else C }`)

Есть два варианта: без явных терминаторов и с явными терминаторами.
Без явных терминаторов -- вводится составной оператор (обычно что-то типа блока в C -- { операторы } из него можно делать `goto`)

Но есть объемлющий блок-тело функции -- из него `goto` наружу сделать нельзя

Pascal	**C**
while B do S repeat S1...S2 until B	while (B) S do S while (B)

Везде -- ровно 1 оператор

Явный терминатор -- явно составной оператор

`begin ... end` -- связан с понятием блока, а не составного оператора

Появился особый оператор многовариантного выбора:

Ada: <Modula-2, Fortran, Oberon>		**Python** (явный терминатор -- "отсутствие отступа")

`if B1 then`
`  S1`
`elsif B2 then`
`  S2`
`elsif B3 then`
	условный оператор состоит из 3 частей
	
	`elseif` не считается `if` (иначе
		`if B:`
		`  pass - явный пустой опер.`
		`elif B2:`
		`  S2`
		`else:`

S3
else
S4
endif
	нужно было бы столько же `endif` писать).	S3

4 типа циклов в **Modula-2**:			

`while B do`
`loop`
`S`
`endloop`
	`do`
	`loop`
	`S`
	`end loop`
	`until B`
		`for …`
		`loop`
		`S`
		`end loop`
			`loop`
			`S`
			`end loop`

Oberon	**Ada**	**Ruby** придуман японцем и сильно отличается от европ. языков

`loop`
`S`
`end`
	Решили, что цикл Кнута (выход по ?? цикла) выглядит не удобно, поэтому добавили:
	`loop`
	`when B ==> exit;`
	`end loop;`
		можно и отступом и begin-end
		`if B`
		`S1`
		`else begin`
		`S2`
		`end`

(Денис: далее идет описание наворотов, которые относятся к **Ruby**)

`X = B? e1 : e2` ↔ `X = if B then e1 else e2 end;`
`if B then S1 end` ↔ `S1 if B` (для вложенных ассоциативн. правая)
`S1 unless B` ↔ `S1 if not B`

Цикл:

```
while B
    S
end
```

(Андрей: Навороты с блоками в Ruby тут вообще не в тему, потому что скорее относятся к замыканиям и лямбда-функциям, чем к управляющим конструкциям.)

Есть возможность присоединения блока к некоторому выражения (можно блок применить к итератору).

```
f.readlines do | line |
    …
end
    ← →
f.readlines { | line| … }
```

На самом деле эта страшная штука аналогична вызову функции с лямбдой-замыканием
Например, как это могло выглядеть на JavaScript:
`f.readlines(function(line) { … });`

Преимущество блоков над лямбдами в чистоте - нет слова function
`3.times { … }` -- блок выполнится 3 раза

Многовариантный выбор case

Можно сделать много `if`, а можно эффективно реализовать таблицей переходов.

В ЯП высокого уровня есть:

```
case e of
   C1 : S1;
   C2 : S2;
   …
end
```

ADA	**Ruby**

`case e of`
`when 1, 2, 3, 5, 9, 11 ⇒ seq f`
`when list 2 ⇒ seq ∩`
`when other s == seq Nel(??)`
`end case`
	`case e`
	`when val1 then S`
	`when val2, val3, … valN then S2`
	`when Class then S3`
	`when range then S4`
	`when predicate then S5`
	`end`
	
	здесь добавили специальный сложный оператор сравнения ===, который используется только внутри case и делает сложные штуки, позволяя писать умопомрачительные case-ы
	
	`case a`
	`when 1..5`
	`  "It's between 1 and 5"`
	`when 6`
	`  "It's 6"`
	`when String`
	`  "You passed a string"`
	`else`
	`  "You gave me #{a} -- I have no idea what to do with that."`
	`end`

C, Java (незавуалированный `goto`)			

`switch (e)`
`{`
`    case N:`
`    default:`
`}`
	case и default - просто специальные метки	<==>	`if (e == i)`
			`    goto case i`
			`else`
			`    goto default`

Учебник: в **C#** наличие `break` требуется после всех `case` и `default`.

В **Python** нет `case`, но его можно сомнительно промоделировать:

```
def switch(K):
     {f1: f1, f2 : f2, … fn : fn}[k]()
```

foreach

Циклы наиболее общего вида:

C	Ada

`for (e1; e2; e3) S`	`for f1 := e1 do e2 step e3 do`
	`  …`
	`end`

Сейчас появляются циклы вида:

`foreach (var t in C) S`, где `C` -- коллекция (последовательность) элементов

C# : `IEnumerable` имеет метод `GetEnumerator()`, который возвращает интерфейс `IEnumerator`:

свойство `Current` -- текущий элемент
`bool MoveNext()` -- следующий, `false` если его нет
`Reset()` -- переводит в начало

`foreach` берет ссылку на `C`, получает `IEnumerator`, и делает `Next()` и какие-то действия с `Current`.

Вид: `foreach (T v in C) S`

C++11: `for (T el : C) S`
Java: `for (T v: C) S`

Средства развития ЯП

Нужно уметь

1. вводить новые абстракции
2. защищать эти абстракции

Другие способы передачи управления, не входящие в базис:

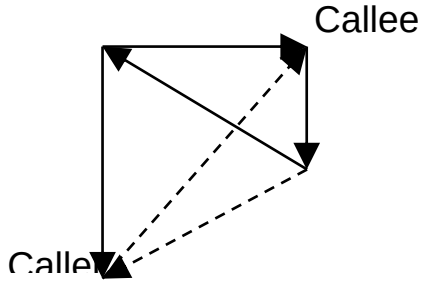
- Подпрограммы
- Исключительные ситуации

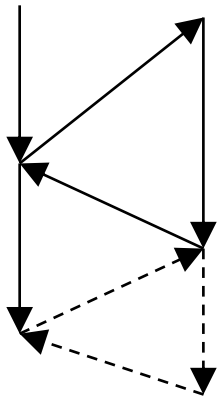
Понятие подпрограммы, как абстракции алгоритма появилось еще в **Fortran**.

Связанные понятия:

- передача потоков управления и их управление
- потоки данных
- подпрограммные ТД

Подпрограммы и сопрограммы в ЯП

	*Subroutine* в **Fortran** всегда ведет себя одинаково, когда бы ее не вызвали. Выполнение Subroutine каждый раз начинается сначала.

	*Coroutine* (*сопрограмма*) - вызов происходит туда, где она потеряла управление, вместо `call` и `return` повявляется resume (1 команда вместо двух, но изначально 1 `call`)

Modula-2

Введен ТД `Coroutine` (для адреса сопрограммы).

```
PROCEDURE NEWPROCESS (VAR C : COROUTINE, P : PROC, N : CARDINAL);
```

Строка задает: инициализацию программы, процесс без параметров, размер рабочей области → `C` будет работать по процедуре `P`

```
PROCEDURE TRANSFER (VAR C1, C2: COROUTINE);
```
→ передает управление от `C1` к `C2`.

Архитектура стека появилась для реализации подпрограмм (*frame*, *activation record* для хранение данных о вызове).

Но у каждой сопрограммы должен быть свой стек, что и подразумевается в передаче параметра `N` -- размера области для стека реализации и т.п.)

Python

Сопрограммы используются в генераторах (генераторы являются частным случаем итераторов).

Пример сопрограммы (возвращает элемент, и далее не выполняется, пока не попросят еще).

```
def fib(): # генерируем числа Фибоначчи
      n1 = 1
      yield n1
      n2 = 1
      yield n2
      while True:
            yield n1 + n2
            n1, n2 = n2, n1 + n2
```

Теперь можно писать:

```
      for f in fib():
            print(f)
```

Еще можно писать:

```
      f = fib()
      n1 = next(f) // 1
      n2 = next(f) // 2
      n3 = next(f) // 3
```

(Денис: далее описывается механизм `generator.send(value)`, вставлю лучше сюда выжимку из документации и пример со stack overflow, т.к. из лекций я ничего, оказалось, не понял)

В конце версий 2.x `yield` стал полностью выражением → возвращает определенное значение и повявляется функция:

`generator.send(value)`

Resumes the execution and “sends” a value into the generator function. The value argument becomes the result of the current yield expression. The `send()` method returns the next value yielded by the generator, or raises `StopIteration` if the generator exits without yielding another value.

То есть механизм, позволяет передавать в генератор информацию для обработки. Информация приходит, как результат вызова `yield`.

Пример:

```
>>> def double_inputs():
...     while True:
...         x = yield
...         yield x * 2
...
>>> gen = double_inputs()
>>> gen.next() #run up to the first yield
>>> gen.send(10) #goes into 'x' variable
20
>>> gen.next() #run up to the next yield
>>> gen.send(6) #goes into 'x' again
12
>>> gen.next() #run up to the next yield
>>> gen.send(94.3) #goes into 'x' again
188.5999999999999
```

В **Python 3.5** возможен реально параллельный запуск сопрограмм, путем использования ключевого слова `async`.

```
async def readl_db():
    …
```

`data = await(db.fetch("Select …"))` -- ожидает завершения

`yield from` -- конкретно указываем от кого хотим получить данные

Накладные расходы в **Python** на создаение процесса - 8 МБ, для сопрограммы - 1 КБ.

C#

Для итерирования нужно реализовывать `IEnumerable`, возвращающий `IEnumerator`:

```
// Collection of Person objects. This class
// implements IEnumerable so that it can be used
// with ForEach syntax.
public class People : IEnumerable
```

```
{
    private Person[] _people;
    public People(Person[] pArray) {...}

    IEnumerator IEnumerable.GetEnumerator() {return (IEnumerator)
GetEnumerator();}

    public PeopleEnum GetEnumerator() {return new PeopleEnum(_people);}
}

// When you implement IEnumerable, you must also implement IEnumerator.
public class PeopleEnum : IEnumerator
{
    public Person[] _people;

    // Enumerators are positioned before the first element
    // until the first MoveNext() call.
    int position = -1;

    public PeopleEnum(Person[] list) { … )

    public bool MoveNext() { position++; return (position < _people.Length); }

    public void Reset() { position = -1; }

    object IEnumerator.Current { get { return Current; } }

    public Person Current { get {
            try
            {
                return _people[position];
            }
            catch (IndexOutOfRangeException)
            {
                throw new InvalidOperationException();
            }
    } }
}
```

Получается примерно страница кода.

В **C# второй версии** появились ключевые слова `yield break` и `yield return`:

```
public class PowersOf2
{
    static void Main() {
        // Display powers of 2 up to the exponent of 8:
        foreach (int i in Power(2, 8)) {
            Console.Write("{0} ", i);
        }
    }
```

```
    public static System.Collections.Generic.IEnumerable<int> Power(int number,
int exponent) {
        int result = 1;
        for (int i = 0; i < exponent; i++) {
            result = result * number;
            yield return result;
        }
    }

    // Output: 2 4 8 16 32 64 128 256
}
```

Все остальное компилятор делает сам, создавая анонимную реализация класса, реализующего `IEnumerator`.

В **C# начиная с пятой версии** появилось ключевое слово `async`, позволяющее выполнять сопрограммы реально параллельно.

Fortran 77(???)

Появилось `Entry` -- особое обозначение для разных точек входа в подпрограмму:

```
Subroutine P
      Entry P1 ---------- альтернативные точки входа
      …                   (кроме еще и точек выхода)
      Entry P2
      …
```

Ruby

Итератор - метод → можно прикрутить к нему блок, в которой по `yield` передается введенные значения, т.е. описывается тело функции, которая вызывается внутри итератора при выполнении `yield`.

Примеры:

```
Fib(){|f| print f} ← в | f | кладутся значения из yield
3.times do { |X| … }
```

collection.each do { |X| … } ← перебор элементов коллекции

Общие слова

По сути -- сопрограммы -- квазипараллельные процессы. (Квази -- потому что, реально программа в целом выполняется последовательно).

Но некоторые языки используют понятие потока для сопрограмм (**erlang**, в частности).

По сути сопрограммы очень похожи на thread (но потоки ОС реально параллельные, а сопрограммы квазипараллельные).

В Windows есть *fiber* (волокно) -- легковесные поток, но на одном процессоре → они квазипараллельные.

Сопрограммы -- очень похожи на потоки, но затраты гораздо меньше.

Передача параметров в подпрограммах

1) семантика передачи (сематика входа/выхода)
2) способ передачи (механизм)

Все хотят, чтобы можно было специфицировать (1), а (2) чтобы компилятор выбрал сам.

`P(C_1, ..., C_n)` - вызов `P(Arg_1, …, Arg_n)`
-- способ передачи и есть как именно `C_i` связано с `Arg_i`

Формальные параметры бывают трех типов:

1. `in` -- функция либо не влияет на данный параметр, либо он не должен изменяться (должны сохранить на входе).
2. `out` -- должен изменять свое значение, что обязательно приводит к изменению значения фактического параметра.
3. `inout` -- требуют определенности и на входе и на выходе.

В языке **Ada**: ввели 3 ключевых слова `in, out, inout.` Указываем передачу параметром функции → по ходу программы не должны нарушать этих требований.

Конкретный механизм передачи определяет компилятор (способы передачи):

1) по значению (копируем перед вводом, кладем на стек)
2) по результату (копируем на стек на выходе)
3) по значению и результату (два копирования)
    a) плюс -- минимальные накладные расходы для маленьких параметров
    b) минус -- максимальные для больших данных
4) по ссылке/адресу (тоже что по значению, но передается адрес) ← наиболее общий способ
5) по имени (указатель параметра становится тем, что написано в формальном параметре).

Algol60

Описываем параметры и их типы; по умолчанию по имени (макроподстановка с переименованием):

```
procedure P(A, N);
integer array A[N, N]; value integer N;
```

thunk -- крохотная подпрограмма, создается для каждого формального параметра, в зависимости от вида фактического параметра; при каждом обращении к себе дает актуальный адрес объекта (и передается по сути вместо параметра) → на каждое обращение к формальному параметру вызывается целая подпрограмма *thunk*.

Рассмотрим следующий пример:

procedure swap(x, y); (нет value → по имени)

```
integer x, y;
begin integer tmp;
      tmp := x; x := y; y := tmp
end
```

```
integer array A[1:10]
integer i; i := 1; A[1] := 5;
swap(A[i], i);
swap(i, A[i]); (i -- изменилось сначала, а только потом A[i] будет
                                                    вычисляться)
```

При обращении к `A[i]`, будет вызываться *thunk* по значению `i` и вычислять текущий адрес.

```
tmp := i; i := A[1]; // A[5] = 1, i == 5
```

Выходит, что никак универсальный `swap` не написать.

Итого в **Ada** (Денис: из лекций не понял, вроде бы речь шла про **Algol60**): укажем `in` → компилятор выберет 4 способ, но если передаем массив то он будет отслеживать, чтобы его не меняли.

C - передача параметров осуществляется *только по значению*.

C++

`T&` -- компилятор выборет `out` или `inout` семантику;

`const T&` -- будет выбрана только `in` семантика.

Однако часто только семантику задавать просто невозможно. Всегда надо явно специфицировать способ передачи, так в **Ada95** -- уже отказались, т.к. обнаружили, что программы, написанные полностью по стандарту, на разных компиляторах дают разный результат.

```
procedure P(inout A, B : T) is
 …
begin
      A := expr1;
      raise range_error;
      B := expr2
end P;
```

- если по значению, то формальный параметр не изменяет занчения
- если по ссылке, то `A` изменит, а `B` нет

Следовательно, `P(X, X)` -- даст разный результат

Итого: надо всегда явно указывать тип передачи параметра, но надо еще указать, поддерживаем или нет семантику `in, out, inout`.

Возможны следующие варианты:

- вся тройка - по значению; все референциальные -- по ссылке (в **Java** так, кроме того семантики - игнорируются)
- **C++**

  `T a` -- по значению

  `T &a` -- по ссылке; по умолчанию `inout`

  `const T &a` -- по ссылке; `in`
- Только одни указатели
- **Ada95**

  `out` и `inout` по ссылке, `in` по значению (если маленький) и по ссылке если большой объем

C#

Можно указать модификаторы `ref` и `out` (при вызове их необходимо дублировать перед параметрами):

```
void f(ref int i)
void f(out int i)

int i; // неопр. значение особо ицициализируется
g(ref i); // нельзя → ошибка
g2(out i) // можно
```

`ref` ⇔ `inout` Чистая `out` семантика (не требуется копир. значения на входе, использовать до установки значения нельзя (см. пример выше))

`ref` и `out` можно указывать и для ссылок:

```
Class X .. // ссылка передается по ссылке
void f (out X a) { … a = new X(); … }
```

Валя ([903-914], завершено)

Глобальные переменные

- поток данных минует формальные и фактические параметры. Глобальная переменная меняет свое значение неконтролируеммым для программиста способом
- идеи о том, что надо убрать глобальные переменные и передавать всё через список параметров функции
- невозможно проконтролировать обращения к глобальным переменным
- запрет глоб.перем -> нет глоб. функций

В Java только метод объекта `Math.sin(x)`

В C# ввели static: `static class X {}`. Ввели, но нельзя создавать объект, если сделать using, то можно методом без спецификации обращаться.

В Java static тоже есть, но означает другое

В C++

- в непосредственной области видимости: прямо тут список
- в потенциальной области видимости: через уточнение “::” или “.”

имея `namespace NS { class T {...}; void f(T); }`

`T x;` // так нельзя

`NS::T x`; // можно

`f(x);` // можно. если нет в своей области, ищет в области, где определен аргумент.

В Аде обязательно писать use <имя_пакета>

Вызовы

1) Позиционный вызов //как в C++
2) Ключевой Вызов //как в python

1) Позиционный. Соответствие формальных и фактических параметов устанавливается по позициям:

- вполне естественен, появился изначально
- попытка уйти от глобальных переменных -> список параметров в функции сильно разростается.
- для упрощения появились параметры по умолчанию

2) Ключевой. вызов через имя формальных параметров
Ада:
```
procedure P(x: T; y: T1);
P( a, b)  -- a->x, b->y
P(x:= a, y:=b)  ⇔ P(y:=b, x:=a)
procedure  P2(q: T = default, y: T1)
P(y:= b)
```
- только в ключевом вызове можно опускать переменные по умолчанию
В Oberon этого нет. В модуле InOut опередедены функции для каждого типа:
```
InOut.writeString("hello, world"); InOut.writeln;
```
В C - “дырка в ?типах” - var.args - > компилятор ничего не контролирует относительно типов
В ООЯП f(...), где “...” - массив объектов
C# `void f(..., params int []args)`
`f()` - > массив из нуля элементов, `f(1,2)` -> массив из 2 эл. `f(1, "string")` -> ошибка, несоответствие типов

Подпрограммный тип данных

Необходимость именно в переменных (по событию должны дать значение) -из событийно-ориентированных систем(когд динамически нужно ставить callback)
Ада83: нет подпрограммного типа, используют делегаты(обобщенное программирование) (положить на параметр задачу ?перепрыгивать командную? функцию в зависимости от параметров, в частности от данных и подпрограмм)
Ада95: есть объектные указатели: `Type PT is access TM` - на что угодно можно указывать; для работы с API из C + появился подпрограммный тип данных (тоже указатели) из тех же соображений. По большей части как в C делали, только иногда типизировали

Oberon, Modula2	Ада

`TYPE PRC =PROCEDURE(INTEGER)`
`PROCEDURE RUN(INTEGER)`
`X: PRC;`
`X:= RUN;`
`X(0)`
	`Type PRC is procedure(in integer)'`
	`access;`
	`procedure FUN(p: in integer)`
	`x: PRC;`
	`x:=FUN'access;`

Операции: := = != () {/= в Аде}
Java: нет указателей, нет подпрограммного типа данных, только ссылки -> всё это через динамическое связывание методов класса
C#: аналогично. Базовый класс - в нём переопределяем особый вызов родительского base.fm() + делегатский тип данных
Сначала не было ничего и надо ?непонятный набор букв похожий на “поре платив”? писать, но потом появились анонимные классы
```
Button b1, b2,b3;
b1 = new Button {
      protected void onClick() {...
            super.onClick();
      }
}
```
`b2 = …; //` тут b2 заменяем без необходимости реально создавать много классов. Тут появился термин closure - замыкание
{если переменные видны где-то в области; используя внутри нек. области, то она захватывается и при необходимости видна и вне}

Делегаты в C#. Ключевое слово event

Обычный процедурный ТД. Крайне ограниченная функц-ть, обычно имеет вид “подписка” - ”рассылка”. Язык C# тем не менее включает в себя понятие «делегаты», аналогичное функциональному типу, но не основанное непосредственно на концепции указателя и являющееся более надежным.

Делегат – это тип, который представляет собой ссылки на методы с определенным списком параметров и возвращаемым типом. Делегаты используются для передачи методов в качестве аргументов к другим методам.

Делегаты обладают достоинствами функционального типа данных, и в то же время не имеют его недостатков. Внешне объявление делегата очень похоже на объявление функционального типа.

Объявление делегата начинается с ключевого слова **delegate**, за которым идет спецификация функции:

delegate - только внутри какого-то класса

```
class X {delegate void f(int); /* в System есть Delegate и цепочки
делегатов;теперь можно писать f d; операции:*/
```

1) :=

```
 class y{
  void m(int p){...}
  static void R (int i){} //оба полностью удовлетв. виду f
 }
}
y = new Y();
d = y.m; //ссылка на метод
d = y.R; //просто функция
{//надо было писать
 d = new f(y.m);
 d = new f(y.R);
}
```

2) К экземпляру делегата можно применять следующие операции

Изначатьно значение делегата пустое

- = - присваивание
- += добавление функции
- -= удаление функции
- вызов

Присваивание делегату какой-то функции - стирает всё содержимое делегата и создает в нём список из одной функции, += добавляет в список ещё функции(удовлетворяющие виду делегата)Теперь операция вызова делегата будет вызывать по очереди все функции из списка, передавая каждой функции аргументы из вызова

3) () d(0) - каждый метод вызываетя с параметром 0, порядок не определен.

“+=” - “подписка” “()” - “рассылка” //в лекциях 2012 года () называют уведомлением

`f d;` – каждый может сделать +=, -=, (), =(т.е. удалить всех), хочется, чтобы был владелец и мог что-то делать, но все ТД единообразны -> !делегаты не являются типом данных

event - специальное новое ключевое слово, появилось в 99 году. реализует “подписку-рассылку”, является модификатором переменной делегатского типа. + появилась возможность захвата (closure) И если область делегата в области видимость в области видимости local, то local захватывается. События это особый тип многоадресных делегатов, которые можно вызвать только из класса или структуры, в которой они объявлены (класс издателя).Если на событие подписаны другие классы или структуры, их методы обработчиков событий будут вызваны когда класс издателя инициирует событие.

```
{delegate (int x) {return 2*x;}}
```

```
delegate event Run(int x);
{ Run f; T local;
  f = delegate (int x) {return 2*x + local;}
}
```

Лямбда-функции

- В **Python** всё просто

```
f = lambda x: 2*x
print(f(1)) // 2
```

Можно использовать замыкания (захват переменных):

```
y = 5
f = lambda x : x + y
print(f(3)) // 8
```

Лямбды в питоне могут содержать только одно выражение, результат которого является результатом функции.

Таким образом лямбды являются чистыми функциями (без побочных эффектов). Нельзя менять переменные из лямбд-замыканий

- **JavaScript**

```
function f(x) { return 2*x; } //определяет функцию
function g() {
      ...
      y = 1;
      function f(x){
            return x+y;
      }
      return f;
}  // захват y
```

функции в `js` - это объекты первого порядка (их можно передавать как аргументы в функции, присваивать переменным и т.п.).

Вообще это не лямбда, а замыкание (Closure, захват переменных).

Лямбда функция в JS выглядит так:

```
var lambdaSum = function(a, b) { return a + b; };
```

Основная суть лямбд - анонимность (имя функции не обязательно):

```
var sum = [1,2,3].reduce(function(a, b) { return a + b; }); // sum = 6
(сумма элементов)
```

Можно еще так писать:

```
var sum = [1,2,3].reduce(lambdaSum);
```

Или даже так:

```
function sum(a,b) { return a + b; }
var sum = [1,2,3].reduce(sum);
```

В стандарте языка JS под названием ES6 (он же ES2015) можно писать так:

```
var sum = [1,2,3].reduce( (a,b) => a + b );
var sum = [1,2,3].reduce( (a,b) => { return a + b; } );
```

Использование Лямбда + Замыкание:

```
var delta = 10;
```

```
var oldArray = [1,2,3];
var newArray = oldArray.map(function(n) {
      return n + delta;
});
// newArray = [11,12,13]
```

В отличие от Java/Python, замыкания могут изменять переменные:

```
var value = 10;
function subtract(n) {
      value -= n;
      return value;
}
var result = subtract(4); // result = value = 6
```

- В компилируемых языках программирования надо добавить делегатов.

  C#. (x) => 2*x ⇔ x => 2*x //генератор делегатов, это выражение

  delegate int Run(int)
  Run f;
  f = x => 2*x // тут происходит генерация делегата,
  // идет вывод типа из определения делегата (инт)
  {//есть generic -> можно написать
  Run <int, int>
  Libor<int> ⇔ void Libor(int)
  }

  Заметим, что лямбда-выражения не могли появиться без обобщений(generic)
- **Java**. захват, в 2005 появились обобщения, в 2014 в Java8 окончательно появились лямбда-функции

  Общий вид: (список параметров) -> выражение. // на фото 914 есть пара примеров использования.

Андрей ([915-924], завершено)

Функторы и лямбда-фукнции в C++

Если в классе перегружен `operator()`, то такой класс называется функтором. Это как альтернатива лямбдам.

Например можно делать свертки:
Есть стандартная функция в <algorithm>:
`count_if(f1, f2, predicate)`
где f1 и f2 - итераторы, predicate - функтор от одного аргумента
она может принимать как указатель на функцию типа `bool (*)(int)`, либо объект класса с перегруженным `operator()`.

В Си++11 появились лямбда-функции:
`count_if(f1, f2, [](int n) { return n%2 == 0; });`
Семантика как в лиспе, даже замыкания есть.
Про них подробнее тут: http://en.cppreference.com/w/cpp/language/lambda
Если внутри блока идет обращение к глобальным переменным, или переменным функции, внутри которой создается лямбда, то будет ошибка, т.к. захват контекста (замыкание) по-умолчанию

выключен. Чтобы включить, надо писать `[&]` или `[=]` или `[varname]` или `[&varname]` (смотри по ссылке выше).
Можно даже писать `[=, &x, &y]`. (Захват x и y по ссылке, остальных по значению)

А еще появился обобщенный класс `function` в библиотеке `<functional>`. Его можно использовать для объявления функций высшего порядка.

В Java есть лямбды, но там захват переменных можно делать только read-only.

Модульность (C++, Модула-2, Oberon)

C++
`typedef` не определяет нового типа, а только создает алиас.
Потому что при создании нового типа можно перегружать функции, но
```
typedef int newtype;
void f(int);
void f(newtype);
```
выдает ошибку.
Единственное, что в плюсах создает новый тип - это директива class.

Физический модуль - это единица компиляции (трансляции)
Часто логический и физический модули совпадают (как в плюсах)

Модула-2
Модули бывают локальные и глобальные (библиотечные). Весь код по умолчанию пишется в главном локальном модуле. Локальный модуль содержит и объявления и реализацию.
Структура локального модуля:
```
MODULE ModuleName
      … definitions …
BEGIN
      … module code …
END ModuleNmae;
```
module code будет выполнен сразу при инициализации модуля. Поэтому он и используется как точка входа в программу.

Глобальные модули нужны для написания библиотек. В них код разделяется на определения (заголовки) и реализацию. Определение и реализация пишутся в разных файлах. Имя файла как правило совпадает с именем модуля.
Файл HELLO.DEF
```
DEFINITION MODULE Hello
      EXPORT Hello;
      PROCEDURE Hello;
END Hello;
```

Файл HELLO.MOD
```
IMPLEMENTATION MODULE Hello;
      FROM Terminal2 IMPORT WriteString, WriteLn;
      PROCEDURE Hello;
      BEGIN
            WriteString("Hello");
            WriteLn;
      END Hello;
```

```
END Hello;


TYPE T;
PROCEDURE INIT_T(VAR P:T);
DESTROY(VAR P:T);
```
Это реализация абстрактного типа данных, внутренней стркутуры которого мы не знаем.

У модулей в Модуле-2 можно задать специальную маску устойчивости:
```
MODULE M[4];
```
4 воспринимается как битовая маска. Суть этой маски в том, что она запрещает некоторые прерывания при выполнении модуля M.

Еще в Модуле-2 есть сопрограммы, но это уже совсем другая история

Oberon
Нет разделения на определение/реализацию, нет главного модуля
Все модули могут экспортировать процедуры типа PROC.
```
MODULE M
      TYPE Obj* = RECORD
            X : INTEGER;
            Y* : INTEGER; ← Экспорт отдельных полей структуры
      END;
      PROCEDURE P*; ← * - значит экспорт
      VAR Done*- : Boolean; ← *- - значит только на чтение
END M
```

За пределами модуля нельзя обращаться к X:

```
VAR V : M.Obj
V.Y = 1; ← Можно
V.X = 1; ← Нельзя
```

В этом языке к содержимому импортированного модуля можно обращаться только через точку, нельзя как в Модуле-2, импортировать конкретный объект. (см. ниже)

Допускается создать псевдомодуль только с выделенными именами (?)

Логические модули (Modula-2, C)

Есть в языках Ада, Оберон, Модула-2, Эйфель, … - все основаны на понятии модуля, как на понятии конейнера языковых ресурсов

Можно определить интерфейс модуля (перечисление того, что надо экспортировать и реализацию. Тогда реализация экспортирует все в глобальное пространство имен
(wtf?)

Вообще видимость и доступность - это разные понятия.
Видимость бывает двух типов:

а) непосредственная (прямая, доступ к имени модуля/функции/переменной без точки, например `WriteString`)
б) потенциальная (доступ к имени только через уточнение модуля/объекта с помощью точки, например `Terminal2.WriteString`)

Modula-2
В модуле-2 доступно все что видно (нет модели доступа - все что видим, можно брать)
Непосредственно видимы только стандартные имена и имена библиотечных модулей (определений модулей)

Импорт позволяет подгрузить дополнительные модули:
```
IMPORT <список имен модулей>
IMPORT M;
```

Доступ после импорта все-равно через точку: `M.P`
Хотя можно делать и непосредственный импорт
```
FROM M IMPORT P;
```

При компиляции файл с определением модуля M.DEF переходит в M.SYM
А файл с реализацией модуля M.MOD переходит в M.OBJ

C
```
#ifndef FILENAME
#define FILENAME
...
#endif
```
Стандартная защита от повторного включения заголовочных файлов

extern нужно писать только в заголовочных файлах, но переопределять чужим typedef (wtf?)

Модульные языки (Delphi, Object Pascal, Ада)

Turbo Pascal, Ada, Modula-2, Oberon
В простом случае Модуль - это набор ресурсов, которые можно реиспользовать, а не создавать заново.
Любой класс - это модуль.

Delphi
Модульный подход как в Модуле-2, но ОО (объектно-ориентированный?)
Общее глобальное пространство имен
Видимость: потенциальная, непосредственная
Порождение имен - определяющее (должно быть одно, если без перегрузки), использующие
Непосредственно видимы только именя библиотечных модулей

Импорт: `uses M;`
Все имена непосредственно импортируются (без точки можно юзать)

Object Pascal
`unit:`

interface ← переходит таблицу символов

implementation ← переходит в obj-файл

Итого получаем, что программа - это набор модулей. Экспортирующие -> Библиотечные.
Удовлетворение зависимостей импорта

Нисходящий подход - сначала проектируется верхний модуль в иерархии импортов
Восходящий подход - сначала проектируется нижний модуль в иерархии импортов

Ада

```
package P is
      ← Описание экспортируемых ресурсов
      Type T is record …
      end T;
      procedure init (X: out T);
      procedure destroy(X: inout T);
      ...
end P;

package body P
      procedure init(X: inout T)
      …
      begin
            ....
      end init;
      …
end P;
```

Очень похоже на Модулу-2
Все имена испортируются последовательно

Вложенные модули (Ада, Модула-2)

Ада
Пакеты могут быть вложены

```
package Outer is
      procedure P;
      package Inner is
            procedure Y;
      end Inner
end Outer
```

Во внутреннем пакете доступно все содержимое внешнего пакета, но не наоборот.
Еще можно писать `Outer.Inner.Y`

Можно даже вкладывать определение пакета в реализацию другого пакета:

```
package body Outer is
      package P is
            ….
      end P;
```

```
      package body P is
            ….
      end P;
end Outer;
```

Это типа нисходящий подход. (см. выше)
Современный термин - тесно связанные модули (Closely Coupled Modules)
Их реиспользование ограничено

Модула-2

```
IMPLEMENTATION MOD:
      MODULE M[4]
            EXPORT P1, P2;
      END M;
END MOD;
```

Лев ([925-946], завершено)

Абстрактный тип данных (АТД)

Ранее (годов до 70-х) считалось, что тип данных (ТД) – это множество объектов.
Начиная с 70-х появилось понятие Абстрактного Типа Данных – это просто интерфейс – перечисление операций, которые можно делать с объектами этого типа.
Во многих стареньких языках это реализуется с помощью модулей:

Модула-2 – скрытый тип данных (opaque). В модуле определений пишется:

```
DEFINITION      M;
TYPE  T; <---- Скрытый (Opaque) тип данных
CONST N; <---- Просто            CONST, без типа!
PROCEDURE INIT(VAR X: T) <---- Обратите          внимание на тип!
      . . . . .
END M.
```

Объекты скрытого типа можно передавать в функции, делать операции :=, =, # (это значит “не равно”). Обычно это реализуется с помощью указателей. Поэтому оператор := может иметь две семантики: глубокое и поверхностное копирование.

В **Обероне** похожее дело, тоже структуры могут быть скрыты. Однако присваивание уже побитовое, т.е. есть единственная семантика.

В **Модуле-2** для компиляции модуля **C** с импортом модуля **M** достаточно модуля объявлений, т.е. компилятор может всё необходимое вычислить по нему. Это минимизирует затраты на перекомпиляцию, т.к. при изменении модуля реализации требуется скомпилировать лишь его и перелинковать.

В **C** и **C++** есть трюк под названием .pch – Precompiled Headers – просто сериализация таблиц компилятора, для экономии времени на компиляцию заголовочных файлов.

Вот пример стека на **Аде**:

```
package Stacks is
type Stack is private;
  procedure Push(S:inout Stack; X:T);
  procedure Pop(S:inout Stack; X:out T);
  procedure IsEmpty(S: in Stack) return Boolean;
```

```
  private:
  <структура приватного типа>
   Type Stack is record
     Body:array(1..128) of T;
     top:integer := 1;
   end record;
end Stacks;
```

По-прежнему есть операции присваивания и сравнения. Опять появляются две семантики копирования. Можно сделать совсем настоящий АТД:
`type Stack is limited private;` - целиком закрыть всю структуру данных.
Тогда все операции нужно реализовывать самостоятельно.

Эквивалентность типов данных

В Аде и Си – именная эквивалентность (совместимы только типы с совпадающими именами).
Ада:

```
type Newint is new Integer; ----> Newint != Integer
```

С:

```
struct C {int a;} != struct S {int s;}
```

C++: всё переводит в типы Си:

```
class X { struct X {…};
      void f(); ---> void f_X( X *this);
}
```

Для генерации имени функции используется mangling (генерация уникальных имён), поэтому достаточно именной эквивалентности.

Виды трансляции

1. Цельная – модульность отсутствует. (стандартный Паскаль)
2. Пошаговая или инкрементальная – транслятору предъявляется лишь очередное исправление или дополнение (REPL – read – eval – print – loop)
3. Независимая – транслятору предъявляют отдельные модули, а связывание оттранслированных модулей выполняет редактор связей или загрузчик (Фортран)

Раздельная трансляция

1. Зависимая – при трансляции используется контекст. Контекст используется для контроля межмодульных связей. Нужно, если есть двусторонние связи между модулями.
2. Независимая – никакого контекста нету (например, ассемблер и Си – Си++)

Зависимая трансляция

Ада. Модули бывают

- Первичные – спецификация пакетами
- Вторичные – реализация (тело)

Стандартная двусторонняя связь устанавливается между модулем спецификации и модулем реализации.

Когда же сами спецификации представляют собой внутренние компоненты других модулей, то по-прежнему можно оформлять соответствующие таким спецификациям тела пакетов, процедур и задач как вторичные модули, но для этого нужно явно указать уже двустороннюю связь. Именно: в том модуле, где находится спецификация, применяют так называемую заглушку, указывающую на вторичный модуль, а в заголовке вторичного модуля явно указывают имя того модуля, где стоит заглушка. Признаком двусторонней связи служит ключевое слово separate.

Например, можно оформить как вторичный модуль тело любой процедуры из пакета управление_сетями. Выберем процедуру "вставить" и функцию "перечень_связей". Тогда в теле пакета вместо объявления этих подпрограммнужно написать заголовки вида

```
function перечень_связей (узел: имя_узла)
return BOOLEAN is separate;
procedure вставить (узел: in имя_узла) is separate;
-- перед нами две ссылки на вторичные модули,
-- две заглушки.
```

Соответствующие вторичные модули нужно оформить так:
`separate (управление_сетью)` -- указано местонахождение заглушки
`function перечень_связей (узел: имя_узла) return BOOLEAN is`
`. . .` -- тело как обычно
`end перечень_связей;`

```
separate (управление_сетью)
procedure вставить (узел: in имя_узла) is
. . . -- тело как обычно
end вставить;
```

Теперь вторичные модули можно транслировать отдельно. В Аде предписан определенный (частичный) порядок раздельной трансляции. В частности, все вторичные (secondary) модули следует транслировать после модулей с соответствующими заглушками (т.е. после "старших" модулей, которые в свою очередь могут быть как первичными, так и вторичными)

Пакеты JAVA

Пакет – совокупность типов, предоставляющая защиту доступа и пространство имён. Если хотите объединить несколько типов, нужно написать в начале файла:
`package <имя пакета>;`

Есть переменная окружения CLASSPATH. От неё отсчитывается иерархия пакетов (сами пакеты не иерархичны, это всего лишь имена, содержащие точку, т.е. если есть пакеты X.Y и X.Z, то import X.* не подключит их). Когда есть запись вида `import X.Y.Z`, происходит поиск файла CLASSPATH/X/Y/Z.jar

После этого надо писать X.Y.Z.Class… Можно избавиться от этого, если написать
`import X.Y.Z.*` или `X.Y.Z.Class`

C++. namespace. Пространства имён.

```
namespace X {
      namespace Y {…} <===> namespace X::Y {…}
}
```

C#. Аналогично.

```
namespace X.Y {…}
X.Y.Myclass <=> using X.Y;
```

Единица компиляции и единица дистрибуции --
Понятие единицы дистрибуции характерно для промышленного программирования – единица распространения.

C++. Файл .src – единица компиляции, .obj – дистрибуции.
C#. ЕД – *assembly (сборка)*. – .exe или .dll

Модули в интерпретируемых языках

Единицы дистрибуции – тексты программ (на крайняк – промежуточное представление).

Python. Модуль – файл.py. У каждого – глобальное пространство имён и стандартный контекст (встроенные функции).

```
L = [1,2,3] ---> L поместили в глобальное пр-во
def foo(): ---> foo тоже
...
```
По сути – инкрементальная трансляция (пошаговая).

Экспортируется вся глобальная таблица.

```
import <имя модуля>
```

подгружает в текущую таблицу всю таблицу <имя модуля> (кроме начинающихся на _), т.е. имена будут видны как <имя модуля>.<имя>

```
from <имя модуля> import * ---> имена копируются в тек. таблицу
(кроме конфликтов)
```

Есть и пакеты – просто директория + файл __init__.py В нём прописано, что делать при загрузке пакета. Можно определить переменную __all__ – список подпакетов. Автоматически загрузятся модули из данной директории + подпакеты из __all__

Javascript. Изначально не было модулей. Поэтому появилась сторонняя спецификация модулей под названием CommonJS. Используется в node.js

mod.js
```
module.exports.x = 0;
module.exports.foo = function(x) {return x + 1;}
```

b.js
```
m = require('mod.js') ---> возвращает объект module.exports
m.x == 0
m.foo(1) == 2
```

В новом стандарте JavaScript под названием ES6 (ES2015) появилась официальная спецификация модулей:

mod.js
```
export function sayHello(t) { console.log("Hello world - " + t); }
export var value = 5;
```

b.js
```
import { sayHello, value } from 'mod.js'
sayHello(value);
```

Обработка ошибок(аварийных ситуаций)

Ошибки есть всегда!
Вопросы:

1. Объявление исключительных ситуаций (далее ИС).
2. Возникновение ИС
3. Распространение ИС
4. Реакция на ИС

[Кстати! Накладные расходы на обработку ИС очень велики – в нормальной жизни не используем.]

Ада. Первый язык, в котором появились исключения (попытки были в PL/1).
Вводится псевдотип exception. Вводится 6 стандартных исключений (напр., RANGE_ERROR).
Правила видимости обычные. Одна операция – возбуждение.
raise <имя исключения>.

C++. На основе Ады. В стандартной библиотеке появился std::exception. Генерируются часто самим компилятором (например, для `dynamic_cast<...>(...) ===> bad_cast`).

Visual Basic.
`ON <имя> <реакция>` (обычно GOTO метка) – оператор установки обработчика (локально в процедуре).

`Error.raise( номер ИС, сообщение)` – бросить ошибку. Если ON не установлен, то распространяется вверх по программе (т.к. интерпретатор), если не нашли ==> отладчик.
Всего 4 варианта обработки:

1. resume – повторить оператор.
2. resume <метка> – начать с метки
3. resume next - начать со следующего оператора.
4. exit sub

Саша & Стас ([947-968], завершено)

Исключения. Принцип динамической ловушки. Свертка стека

ИС - исключительная ситуация.

Семантика завершения
Выдавший? блок обязательно завершается (составной оператор)
// т.е. любой блок может иметь исключения

```
Ada:
begin
      операторы
      (блок реакций)
      when ис1|ис2|ис3|...|исN => операторы реакции 1
      when исN[1,1]|..|исN[1,k] => операторы 2
      when others
end
```

но при этом **обязательно выходим из блока**, (можем вызвать заново, но это не возобновление, а перезапуск)

- Распространение -- для и-проуры? + выдаем полный стек вызовов, если вышли из прист.?
- Обработка: собственно выполняются операторы
  raise; -- частичной обработки -> перевозбуждение (вроде throw без юрзов?)

при этом никакой ???, кроме имени с ИС не связывается -> недостаток (т.к. просто реагируется: ??? в списке целочисленного идентификатора)

C++

ИС ⇔ с типом данных связывается (int, char * и т.д.)

(const char * what()) -- class std::exception в STL появился

throw **expression**; -- объект ИС

Выделяется **особая** память для **ровно 1 объекта ИС** (не динамическая, но особая) + есть **явное** понятие **“свертка стека”** (т.к. происходит вызов деструкторов всех созданных? объектов)

Принцип динамической ловушки заключается в том, что выбирается реакция на ИС, ближайшая по динамической цепочки вызовов.

Свертка стека - процесс уничтожения переменных в аварийном блоке (там где произошла ИС). Для локальных переменных или объектов работают деструкторы.

Синтаксис:

```
try {
      операторы;
      } // дальше идет список “ловушек”
catch (Тип [имя]) {блок реакции}
….
catch (Тип [имя]) {блок реакции}
catch (...) { … }
```

Тип -- **строгое** отождествелние с типом, без преобразований, кроме производный -> базовый.
имя -- если хотим обращаться к инфуги? вызя? объекта
Важно! В момент свертки стека не должно быть исключений (места для него уже нет). Программа завершится аварийно.

Если нет больше try блоков выше по стеку, то для? в данной точке можно завершиться -авост? -> можно самостоятельно вызвать библиотечную функцию terminate();
+ установить свой обработчик set_terminate(новый); // отдать? старый обработчик

C#, Java

ИС - специальный класс ошибок.

+ есть член-данные?, Exception InnerException внутри класса Exception чтобы “разрасталось” исключение => можно выдать стек

ИС делятся на **пользовательские** и **системные**. Возбуждение пользовательских ИС происходит в коде пользователя, а системных ИС - в коде, генерируемым компилятором.
Пользовательские ИС в C++ наследуются от класса exception, в C# - от ApplicationException, в Java - от Exception.

Уничтожение объектов носит недетерминированный характер в языках с автоматической сборкой мусора.
В C# и Java используется конструкция - try{} - ловушки - finally {код обязательно выполнится}

C#	Java

Класс **Exception** (непосредственный потомок Object, от него наследуются **все** ошибки throw new Err(); catch (Exception ex) - **все** поймает catch { операторы } - аналог catch (...) {}, хотя и не нужен	Класс Throwable {свертки стека вообще нет, деструктор - именно при разрушении объекта(finalize), они вообще могут не выполняться, вызовятся? только при сборке мусора} catch(Exception ex) finally {...} -- вызывается всегда в конце try блока в независимости от способа выхода из try блока и обработки исключений throw new Err();

В Java тоже **RAII** реализуется {Resource Acquisition Is Initialization}
Res x = new Res();
try {
    ....
  }
finally {освобождение X}
То есть нет явного вызова финализаторов, но есть возможность задать явное уничтожение (нужно, чтобы класс **реализовывал** ???)

```
IDisposable {
      void Dispose();
}
```

тогда:

```
var img = Image.FromImage("some.png")
try {
      // работа с img
}
finally {
      // нужно явно привести к интерфейсу, если явно реализовали интерфейс
      // "явная реализация интерфейса"
      ((IDisposable)img).Dispose(); //освобождение картинки
}
```

Альтеративa -- использование **using**(имя) // имя переменной, которая реализует IDisposable

```
using (Image img = Image.FromImage("some.png"))
{
      //work with image
}
//img автоматически освобожден
```

Компилятор сам подставит try/finally + проверит, что класс действительно реализует IDisposable

Python:
Error -> либо throw Error, либо throw <class, наследующий от Error>
// вроде как в Python **raise** (точно raise).

```
try:
      операторы
<ловушки>
except имя1:
```

```
      …
except:
finally:
else: -- когда вышли нормально + сработал finally в конце
```

JS:
```
try {
      … throw new Error(...);
}
catch (e) { … }
```

e -- имя, без типов, т.к. изначально подразумевается, что всегда выбрасывается объект **Error**.
!Нет множественных ловушек, т.к. только 1 Объект - Error
Комментарий: Семантика возобновления работает в основном только в маленьких проектах, поэтому в большинстве языков реализованна именно семантика завершения

Спецификация исключений: (только в C++ и Java)

C++:
```
void f() throw (список типов)); {function body}
// спецификация опциональна
// список типов может быть любым
```
Это означает, что это -- **контракт**, что кроме указанных исключений никаких других не появится.
Это гарантирует, что все ИС не из списка либо не возникают, либо обрабатываются внутри функции.

Пример:
```
void f() throw (X, Y, Z);
void suppress() throw()
{
      try {
            f();
      }
      catch (X x) {}
      catch(Y y) {}
      catch(Z z) {}
}
```

В случае, если через функцию проходит не специфицированное исключение -> unexpected() + set_unexpected() { по сути terminate }, при этом в этой же точке более специфицированно место падения,
-> подразумевается, что любой деструктор как будто специфицирован throw() {т.к. не должен выбрасывать исключений}

```
Java:
class X {
      void f() throws список типов ИС (все наследники Throwable) { … }
      void g() [пустой список по умолчанию -> по умолчанию ничего не может
бросать] { … }
```

-> throws обязателен -> уже на этапе компиляции легко проверять -> компилятор гаранитирует

- однако есть JNI -- а это по сути L-? там нет исключений, а ошибки возникать могут }

- существуют ошибки, на которые нельзя реагировать (ошибки процессора, неожиданно, редко, но в любом месте), но на? них не надо реагировать

Виды ошибок в Java

- class Error extends Throwable{...} - ошибки фунционирования Java-машины и JRTE
- class Exception extends Throwable{...} - базовый класс для пользовательских ИС, их нужно указывать в **throws**
- class RuntimeException extends Exception {...} - базовый класс для ошибок выхода за границу массива, нехватки памяти и т. п., на них не обязательно реагировать, но можно, как следствие, можно опускать в **throws**

ADA:
Создание исключения:

```
New_Exception: exception;
```

Поднятие/возбуждение исключения:

```
raise New_Exception;
```

Отлов исключения:

```
begin
      Rise_Exception;
exception
      when New_Exception =>
            Do_Smth;
end;
```

Исключения

+

1. Код обработки симализован? - легче читать и писать
2. [Проброс инерции]? - за счет одного исключения, не надо делать кострукции
   `if (...) {return -1;}`

-

1. Асинхронность - в любом месте неожиданно может возникнуть любое количество ошибок
2. Писать хорошие отказоустойчивые программы тоже? сложно

Наследование

- расположение в памяти (layout)
- **множественная** теория типов

C++: производный T1 < T2 базовый, где < -- отношение наследования между **типами**
Smalltalk:
переменная -> класса (static в C++)
-> экземпляра -- все инкапсулированные -> надо ввести get и set

при этом все операции -- **обмен сообщениями**:

{ 5 + 3 - числу “5” посылается сообщение “+” с аргументом “3” ⇔ 5.+(3) }

-> при наследовании наследуются все свойства и атрибуты(+ операции)
можно добавить новые; при этом в стандарте нигде не сказано, что нельзя реализовывать непрерывной памятью
[-----память под базовый-----][---память под производный---]

this в C++: X * **const** this (const -- нельзя (кроме C#) сделать this = new X();)

mutable поля - их могут менять даже const методы.

В большинстве языков -- линейное расположение при единичном наследовании -- когда новый член в D дописывается в конец B (возможно с выравниванием) -> все члены-данные имеют фиксированное смещение

Smalltalk: (вставить картинку 953 фото)

Комментарии от Саши, в рукаписных лекциях отсутствуют:
/*
В C++ не наследуются конструкторы, деструктор и операция присваивания.

Java - class name extends name-base-class {...}
C# - class name: name-base-class {...}
В C# и Java нет модификаторов доступа. Все классы в отличии от C++ имеют общего предка Object (в нем есть много методов, но нет членов-данных).
*/

С другой стороны множество типов -- есть множество объектов
Простой компилируемый язык -- все ТД **не пересекаются** => для любого объекта данных существует единтсвенный ТД. Но в C, например, разрешаются некоторые приведения.

Объединения

Иногда нужно объединение:
C:
```
      union  -> T1
             -> T2
             -> Tk
```

```
union y {
      T1 v1;
      T2 v2;
      T3 v3;
} y; -- не размеченная, все просто имеют общее расположение в памяти
```
Pascal:
```
type vr = record
      f1: T1;
      …
      fn: Tn;
      case <имя>: тип of
            value1: (структура 1);
            value2: (структура 2);
            valuem: (структура m);
      end;
-- вариантная запись, "дискриминированная, размеченная" (можно знать тип структур)

R: vr; f1|f2|...|fn|<имя>|структура I
```

Ada:

```
record
      …
      case имя:тип of
      when value1 => структура 1
      when others => структура m;
      end;
```

Синтаксис New: New(p, t1, …, tk) -> тип p указывает на вариантную запись

```
type UPR = ↑VR;
p: VR;      -> new(p, value), value -- значение <типа> и память выделяется по
размеру, при этом значение в <имя> не присваивается
```

нужно **P↑.<имя> = value;** -> возможно неверно поставить

В C -- не размеченные объединения -> часто вручную вводили, чтобы объединения начинались с short -- идентификатора типа, “диспетчеризация по полю типа” -- работаем по switch, согласно тому, какого типа сообщение к нам пришло

Ada:
разрешаются только **размеченные** записи с дискриминированием

```
case имя_дискриминанта: типа of //
```
не может отсутствовать?, как на Pascal

Pascal?:

```
type word =
      record
            case WORD_TYPE of
            WD_BITS: (bits: parted? array [1..48] of boolean);
            WD_INT: (i: integer);
            WD_REAL: (d: real);
            …
      end;
X: word;
X.i := 65536; -> [48] - знаковый бит
x.bits[17] := 0; -> можно с битами отдельно работать
```

Ada:

```
record word (D: WORD_TYPE) // обязательно как параметр указываем
      record
      case D:WORD_TYPE of
      when …
      end;
-> X:WORD(WD_BITS); -- и только так -> ??? того, что D установлен? корректно и
не изменится
```

Но никто не гарантирует, что программист правильно напишет обработку через case
=> в ООЯП больших case/switch нет, так как должен быть полиморфизм
=> не нужна диспетчеризация по типу
=> объединений тоже нет -- все уходит в наследование
[постоянная часть -- base_class, вариантивная часть -- в наследников]

Анопя? теория
тип -- некоторое множество объетов (потенциально бесконечное)

D < B: ∀x∈D => x∈B, D ⊆ B -> в Smalltalk: D -- подкласс, B -- суперкласс

Если T1 ⊆ T2, то T1 **ковариантен** T2.

Совместимость типов

`x ∈ T1, y ∈ T2 -> x = y, f(T x) => f(y)`

если T2 ⊆ T1, то можно полагать x=y

```
T1 *p1; T2 *p2; // p2 можно приводить к p1, но не наоборот

void f(B &x) {...} // внутри по сути 2 типа -- статичный?
D y => f(y) // можно, так как D ковариантен B
void g(B x) // это B, и динамический -- D
g(y) // при этом тип поменяется; x = y по сути
```

Класс как область видимости

```
class X {...}
class Y: X {...}
class Y:X {
      void f() { id = … // используется ??? }
}
/* как минимум 2 области видимости -- локальная и ??? пространство имен класса
*/
/* сначала ищем в локальном, потом в теле класса -> базовом и т.д. до
глобального пространства (если оно есть) */
```

- в C++ для функций: если дошли до верхнего пространства ???, то [да но, может дальше]? поищем в пространстве имен, в которых вписаны аргументы
- в C++ нет управления **видимости**, только доступа
- в Java -- управления видимостью именно?
  (private -> видим только внутри класса)
  ([также]? -- в Ада и Оберон, где нет классаов)
- в C++ видимость нарушается только определением одноименности? объекта (при чем именно **по имени**, а не по сигнатуре)
- в C# почти также, с исключениями

```
class X {
      public: void f() {...}
}
class Y: X {
      public: void f(int i) {...}
      public: void g() { … f(); …}
}
// произойдет ошибка отсутствия аргумента
```

Преобразование типов (неявное)

Если T1 ковариантен T2 (T1 = T2 или T1 наследник T2) - преобразование допустимо.
T1 контравариантен классу T2, если T2 ковариантен классу T1.

Если T1 и T2 инварианты (не ковариантны и не контрварианты), можно придумать пользовательское преобразование

1. T2(T1) - Ада, модуло, оберон
2. (T2)<выражение типа T1> // C
3. C++
    a. static_cast<T>(expr)
    b. dynamic_cast<T>(expr)
    c. const_cast<T>(expr)
    d. reinterpret_cast<T>(expr)

Классы-оболочки. Автораспаковка в Java, C#

Кроме базовых типов данных, в языке Java широко используются соответст-вующие **классы-оболочки** (wrapper-классы) из пакета java.lang: Boolean, Character, Integer, Byte, Short, Long, Float, Double. Объекты этих классов могут хранить те же значения, что и соответствующие им базовые типы.

Компилятор умеет преобразовывать значения простых типов в экземпляры класса-оболочки (упаковка) и обратно (распаковка).

C#	Java

int i = 123;
// The following line boxes i.
object o = i;
 	
o = 123;
i = (**int**)o; // unboxing
	// Автоупаковка
	Integer integer = 9;
	
	// Автораспаковка
	int in = 0;
	in = new **Integer**(9);

Запрет наследования (sealed в C#, final в Java, final в C++11)

sealed в C#
При применении к классу, модификатор **sealed** запрещает другим классам наследовать от этого класса. В приведенном примере класс B наследует от класса A, но никакие классы не могут наследовать от класса B.

Модификатор **sealed** можно использовать для метода или свойства, которое переопределяет виртуальный метод или свойство в базовом классе. Это позволяет классам наследовать от вашего класса, запрещая им при этом переопределять определенные виртуальные методы или свойства.

```
class A {}
sealed class B : A {}
```
- Никакие классы не могут наследовать от B.

При применении **sealed** модификатора к методу или свойству его необходимо всегда использовать с override.
Поскольку структуры неявно запечатаны, их нельзя наследовать.

final в Java
Ключевое слово final может применяться к классам, методам или полям. В применении к классу оно означает, что данный класс не может подразделяться на подклассы. В применении к методу оно означает, что данный метод не может быть заменен каким-либо подклассом. В применении к полю, оно означает, что в каждом конструкторе значение поля должно задаваться только один раз и после этого никогда не может изменяться.

final метод не может быть перегружен или скрыт производными классами.

Для классов	Для методов

public final class MyFinalClass {...}
 	
public class ThisIsWrong **extends** MyFinalClass {...} // forbidden
	public class Base {
	  public void m1() {...}
	  public final void m2() {...}
	
	  public static void m3() {...}
	  public static final void m4() {...}
	}
	
	public class Derived **extends** Base {
	  public void m1() {...} // Ok, overriding Base#m1()
	  public void m2() {...} // forbidden
	
	  public static void m3() {...} // OK, hiding Base#m3()
	  public static void m4() {...} // forbidden
	}

final в C++11
Указывается только у виртуальной функции, обозначая что она не может быть переопределена в производных классах.

```
struct A {
  virtual void foo() final; // A::foo is final
  void bar() final; // Error: non-virtual function cannot be final
};
struct B final : A { // struct B is final
  void foo(); // Error: foo cannot be overridden as it's final in A
};
struct C : B { // Error: B is final
};
```

Основные аргументы

1. Проектирование иерархии наследования очень сложно -> надо запечатать, если реально не наследуем
2. Эффективность - вызов через ?? таблицу дорого => если даже из X вызывается виртуальная функция, то так как она точно не будет замещена, то сложно снять виртуальность вызова
3. Безопасность

Динамическое связывание методов. Полиморфизм. ТВМ

Динамическое связывание состоит в том, что метод, который нужно вызвать, определяется в процессе выполнения, а не на этапе компиляции.

C++, C# -> virtual
Java - все методы являются виртуальными

В C# обязательно запись override перед заместителем.

ТВМ - таблица вирутальных методов.
В каждый полиморфный объект компилятор неявно помещает указательно на соответствующую ТВМ, хранящую адреса вирутальных методов.
Строки ТВМ содержат адреса заместителей виртуальных методов: если метод замещен в классе, то записывается его адрес, если нет - заимствуется адрес заместителя из таблицы для базового класса.
Пример:

```
class A {
      public:
            virtual void f() {cout << "A::f";}
            virtual void u() {cout << "A::u";}
            void g() {cout << "A::g";}
};
class B:public A {
      public:
            void f() {cout << "B::f";}
            void g() {cout << "B::g";}
            virtual void h() {cout << "B::h";}
};
class C:public B {
      public:
            void f() {cout << "C::f ";}
            void u() {cout << "C::u";}
            void h() {cout << "C::h";}
            virtual void w() {cout << "C::w";}
};
void P(A *pa, B & b) {
      pa->f(); pa->u(); pa->g();
      b.f(); b.u(); b.g(); b.h();
      delete pa;
}
int main() {
      B b;
      P(new A, b);
      cout << "-------------------------" << endl;
      C c;
      P(new B, c);
      return 0;
}
```

Результат работы следующий

```
A::f A::u A::g B::f A::u B::g B::h
-----------------------------------------
```

`B::f A::u A::g C::f C::u B::g C::h`

Устройство ТВМ таблиц для данного примера.

<table>
<tr><td>Объект типа класс *</td><td>ТВМ для класса *</td></tr>
<tr><td>A</td><td><table><tr><td>A::</td><td>&f()</td></tr><tr><td>A::</td><td>&u()</td></tr></table></td></tr>
<tr><td>B</td><td><table><tr><td>B::</td><td>&f()</td></tr><tr><td>A::</td><td>&u()</td></tr><tr><td>B::</td><td>&h()</td></tr></table></td></tr>
<tr><td>C</td><td><table><tr><td>C::</td><td>&f()</td></tr><tr><td>A::</td><td>&u()</td></tr><tr><td>B::</td><td>&h()</td></tr><tr><td>C::</td><td>&w()</td></tr></table></td></tr>
</table>

Пример для C# (для Java нужно убрать override и virtual)

```
class Program {
      class A {
            public virtual void f(int x) {
                  Console.WriteLine("A::f");
            }
      }
      class B : A {
            public override void f(int x) {
                  Console.WriteLine("B::f");
            }
      }
      static void Main(string[] args) {
            A a;
            B b = new B ();
            b.f(l); // B ::f
            a = b;
            a.f (1); // B::f
            a = new A ();
            a.f (1); // A::f
      }
}
```

Издержки:

1. Дополнительные команды по извлечению адреса метода при виртуальном вызове.
2. Дополнительная память для ТВМ

3. Дополнительная память для указателя на ТВМ.

Важно! Деструктор полиморфного объекта должен быть виртуальным.

Алик ([969-981], done)

Можно в наследованном классе написать new virtual f, тогда мы создадим виртуальный метод f, который начнет новую иерархию замещения (скрывает f из базового).

В Java тоже появился override - специальный, стоит после функции, как подсказка.
В C++ - final - нельзя замещать этот метод. Как final на Java.

Инкапсуляция. Уровни доступа в C#, Java. Управление видимостью

В C# роль библиотеки играет понятие сборки (assembly), в Java - понятие пакета (package)

Уровни доступа в C#

1. public - члены доступны везде.
2. internal - внутренний, доступный методам всех классов из этой же сборки.
3. protected internal - доступен своим методам и методам производных классов (из любых сборок), методам всех классов этой сборки.
4. protected - защищенный, доступный только собственным методам и методам производных классов(из любых сборок)
5. private - закрытый, доступ можно получить только из кода того же класса или структуры

Уровни доступа в Java

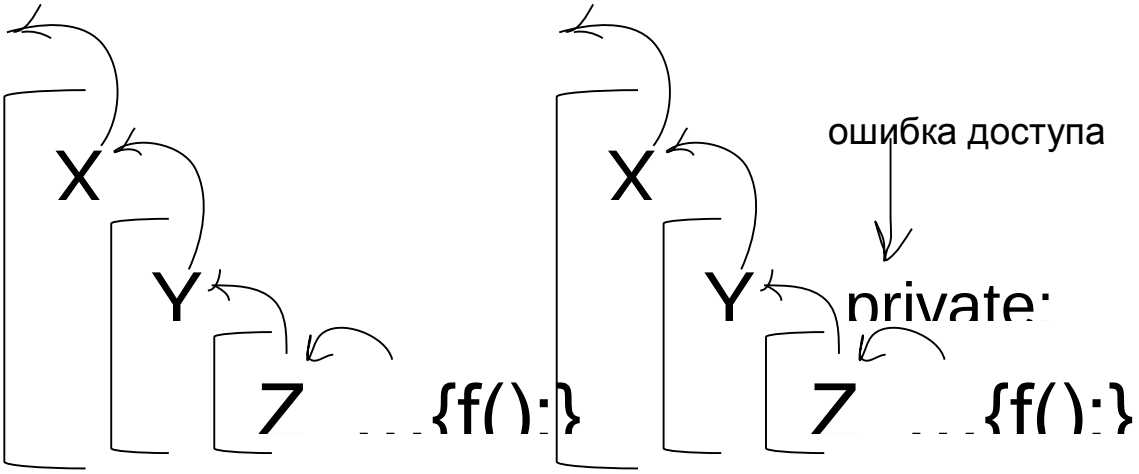
1. public - доступен методом любых классов, любых пакетов
2. без названия (умолчательный), доступный методам всех классов из этого же пакета
3. protected - защищенный, доступный только собственным методам, методам производных классов из любых пакетов и методам всех классов из этого же пакета.
4. private - доступен только собственным методам

В C# и Java есть возможность управлять доступностью из сборки(пакета).

Уровни доступа в C++

1. public
2. protected
3. private

C++, C#



В Java дело обстоит иначе.

X: public void f()
Y: private void f()
Z: {f();} ← вызов X::f

→ Нужно поиск проводить по правилам для связи

(→ private не является заместителем, т. к. нельзя замещать то, что не видно)

→ внутри Y вызов f() не будет виртуальным → можно виртуальность вызова снять.

Заместитель не может ослаблять доступ.

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

Если создать класс Z, унаследованный от Y и определить там функцию f, то вызов g приведет к вызову f из Z, что и должно быть.

в C# - если private, то virtual быть не может.

C++

→ Реализация событийной обработки

```
class X{
     protected: virtual void OnRvent(){ … }
     private: void Work(){ OnRvent();}
}
```

```
class Y: public X{
     protected: void OnRvent();
}

void Y::OnRvent(){
     // своя обработка
     // хотим вызвать функцию из базового класса
     this->X::OnRvent(); // снятие виртуальности вызова
}
```

Заметим:

```
     X *px; px->X::f(); //снятие виртуальности тоже
```

В некоторых языках появляются дополнительные слова:

Delphi: **inherited**.имя_метода
C#: **base**.f() //как ссылка
Java: **super**.f()

Java

```
     class Y extends X{
          Y() {super(...) //вызов конструктора базового }
     }
```

C#

```
     class Y: X{
          Y(): base(...) {...}
     }
```

Вложенные (внутренние) классы, их особенности в Java, C#, C++

Это класс целиком определенный внутри другого класса. Такие классы поддерживаются в C#, C++, Java.

В Java существует 4 типа вложенных классов:

- **Статические вложенные классы**. 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.
- **Внутренние классы**. Декларируются внутри основного класса. В отличие от статических вложенных классов, имеют доступ к членам внешнего класса. Не могут содержать (но могут наследовать) определение статических полей, методов и классов (кроме констант).
- **Локальные классы.** Декларируются внутри методов основного класса. Могут быть использованы только внутри этих методов. Имеют доступ к членам внешнего класса. Имеют доступ как к локальным переменным, так и к параметрам метода при одном условии - переменные и параметры, используемые локальным классом, должны быть задекларированы final. Не могут содержать определение (но могут наследовать) статических полей, методов и классов (кроме констант).
- **Анонимные классы.** Декларируются внутри методов основного класса. Могут быть использованы только внутри этих методов. В отличие от локальных классов, анонимные классы не имеют названия. Главное требование к анонимному классу - он должен наследовать существующий класс или реализовывать существующий интерфейс. Не могут

содержать определение (но могут наследовать) статических полей, методов и классов (кроме констант).

Вложенные классы в C++, C#, Java имеют доступ ко всем защищенным и приватным полям класса, который их содержит. В Java (в C#, C++ - не знаю) из объемлющего класса можно получить доступ к приватным полям вложенного класса.

Абстрактный класс - как интерфейс для наследования

Тело виртуальной функции обязательно нужно задавать, т. к. ссылка на нее все равно есть в VTBL → ошибка.

Таблица виртуальных методов инициализируется в конструкторе. В самом конструкторе еще нет замещений. Кроме того, тут можно вызвать абстрактный метод, если ему задали реализацию.

То есть можно задавать тела абстрактных методов.
(по умолчанию - выдает диагностику и рушит программу)

C#, Java

```
abstract class X{     // ключевое слово abstract обязательно
     abstract void f();  // и здесь тоже
}
```

→ Абстрактный класс - если перед ним стоит abstract (но может не содержать abstract методов) (просто требуем от него унаследовать).

Любой производный, где еще нет переопределения - обязательно с модификатором abstract.

АТД (Абстрактный Тип Данных) - абстрагируется от реализации класса (только public методы)
АТД (определение из учебника) – это тип, в котором внутренняя структура данных полностью инкапсулирована.
АК (Абстрактный Класс) - абстракция от реализации некоторых методов.
Абстрактный класс – это класс, который предназначен исключительно для того, чтобы быть базовым классам. Экземпляры абстрактного класса нельзя создавать, но можно (и нужно) использовать ссылки на абстрактный класс.
Интерфейс класса – совокупность открытых членов класса.

АК + АТД → “интерфейс”

(если абстракция ото всех реализаций всех методов)

{только метод-деструктор - виртуальный, но нельзя чисто, т. к. компилятор сам вставляет **невиртуальный** вызов деструктора без класса в производном}

C++:

```
class Y: <модификатор наследования> X {...}
class Y: private X {} // часто бывает
```

Пример. Множество.

```
class ISet   // <-- абстрактный класс
{
      virtual void Ird (const T& x) = 0;
      virtual void Rxd (const T& x) = 0;

      virtual bool IsInt (const T& x) = 0;
      virtual ~ISet() {}
}

class BitSet: public ISet, private BitScale {...}
      // соответствующие методы ISet вызывают методы BitScale
      // реализация делегирования

class TreeSet: public ISet, private RBTree {...}
```

+ дополнительно сводим затраты на перекомпиляцию в min, т. к. достаточно перекомпилировать реализацию без клиентов
+ private наследование позволяет инкапсулировать (скрыть) внутреннюю структуру класса.
+ public наследование интерфейса необходимо.

Различия между абстрактным классом и интерфейсом в Java (на англ)

- All methods in an interface are implicitly abstract. On the other hand, an abstract class may contain both abstract and non-abstract methods.
- A class may implement a number of Interfaces, but can extend only one abstract class.
- In order for a class to implement an interface, it must implement all its declared methods. However, a class may not implement all declared methods of an abstract class. Though, in this case, the sub-class must also be declared as abstract.
- Abstract classes can implement interfaces without even providing the implementation of interface methods.
- Variables declared in a Java interface is by default final. An abstract class may contain non-final variables.
- Members of a Java interface are public by default. A member of an abstract class can either be private, protected or public.
- An interface is absolutely abstract and cannot be instantiated. An abstract class also cannot be instantiated, but can be invoked if it contains a main method.

В **C#** и **Java множественное** наследование может быть **только по интерфейсам**!

```
      interface имя {сигнатуры методов} //языковое средство.
```

Примеры интерфейсов

C#	**Java**

`// стандартный интерфейс C#`
`interface IEnumerable { //`
`стандартный интерфейс C#`
`  IEnumerator GetEnumerator();`
`}`
	`// стандартный интерфейс Java`
	`interface Iterable{`
	`  Iterator iterator();`
	`}`
	`interface Iterator {`

```
interface IEnumerator {
      object Current { get; }
      bool MoveNext();
      void Reset() ;
}
// реализация интерфейса
class MyCollection : IEnumerable {
      public IEnumerator
GetEnumerator()
      { /* реализация метода */ }
}
```

```
      boolean hasNext();
      Object next();
      void remove();
}
// реализация интерфейса
class MyCollection implements
Iterable {
      public Iterator iterator ()
      { /* реализация метода */ }
}
```

Java	`class X extends Base implements I1, I2, …, IN {...}`
C#	`class X: Base, I1, I2, …, IN {}`
[!]	`Base - класс, от которого производится наследование, может быть только один!` `I1, I2, ... , IN - интерфейсы, может быть несколько.`

Конфликты имен:

Java	Необходимо реализовывать только один метод, public реализация должна быть ровно одной. (и неизвестно откуда он именно, адрес в VTBL будет совпадать).

C#	понятие “явная реализация интерфейса”
	`class X: Base, I1, I2 {`
	`  void I1.f(){...}`
	`  void I2.f(){...}`
	`}`

Явная и неявная реализация интерфейса в C#

Реализация интерфейса может быть как явной, так и неявной.
interface ISum{ … void foo(); }

Неявная реализация интерфейса	**Явная** реализация интерфейса

`class D: ISum {`
`  public void foo(){...}`
`}`
 	
`D d = new D();`
`d.foo();`
	`class D: ISum {`
	`  void ISum::foo(){...}`
	`  //^ public не пишется`
	`}`
	
	`D d = new D();`
	`d.foo(); // так нельзя, ошибка.`
	`((ISum) d).foo(); // так правильно`

Сам Microsoft в .Net часто используют явную реализацию.

```
class P:I1,I2 {
      void I1::foo() {...} // обязательно всегда явно реализовать
      void I2::foo() {...} // если у нас есть конфликт.

      //возможна умолчательная (default) реализация
      public void foo() {... I2::foo();}

}
```

Сравнение явной и неявной реализации интерфейсов

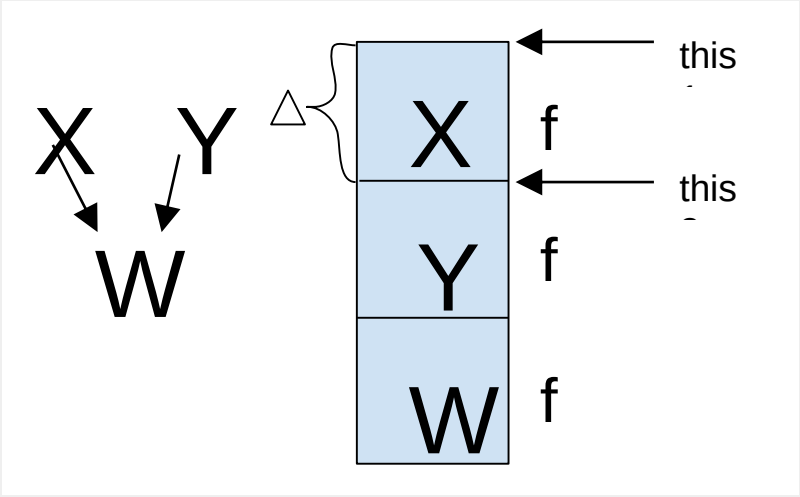
Видимость	Неявная имплементация всегда являелся открытой (public), поэтому к методам и свойствам можно обращаться напрямую.	Явная имплементация всегда закрыта (private). Чтобы получить доступ к имплементации необходимо кастовать инстанцию класса к интерфейсу (upcast to interface).
Полиморфизм	Неявная имплементация интерфейса может быть виртуальной (virtual), что позволяет переписывать эту имплементацию в классах-потомках.	Явная имплементация всегда статична. Она не может быть переписана (override) или перекрыта (new) в классах-потомках.
Абстрактный класс и реализация	Неявная реализация может быть абстрактной и реализовываться только в классе-потомке.	Явная реализация не может быть абстрактной, но сам класс может иметь другие абстрактные методы и сам быть абстрактным.

Множественное наследование

- конфликты имен
- реализация

Чистое множественное наследование - только в C++
Но в других множественное наследование - по интерфейсам.

Проблема множественного наследования:




```
W::f(){
    X::f(); // this1
    Y::f(); // this2
}
```

необходимо VTBL делать из 2х частей, где во второй части хранить эту Δ, которую надо прибавлять к this1.

Нас обязывают разобраться с конфликтом имен, в отличие от Java.

Python
В любой момент времени есть как минимум **3** области видимости.

1) глобальная
2) module
3) def

Класс - еще одна область видимости

`class C(Base):`
`  i = 0`
`  def foo(self, a, b):`
`  self.i = 1`
`  def __init__(self):`
`  self.i = 0`
	`O = C()`
	`O.foo(1,2) //пошли в C, потом в Base`

`class C(Base1, Base2, ...):` `  ...`	`//пошли в C, потом вглубь в Base1, потом в Base2 и т. д.`

Нет конфликта имен, но есть накладные расходы на поиск методов (как и в любом модуле).

Ада83 - чисто статический язык, четкая типизация, без объектов.
Ада95 - добавлены объекты.

```
package Sample is
     type Base is tagged record;
     …
     end record

type Derived is new Base with tagged record
```

```
     новые члены
end record.
```

Есть обычные записи, а наследоваться могут только tagged записи.

```
procedure P(S:Base);
procedure P(D:Derived);     чистое перекрытие
```

Но есть инкапсуляция:

```
package File_System is
    type File is tagged private;
    procedure View(F : File);

    type Ada_File is new File with private;
    procedure View(F : Ada_File);

  private
    type File is tagged
     record
       Name : String(1..20);
     end record;

    type Ada_File is new File with
     record
       Compiled : Boolean := False;
     end record;
  end File_System;
```

Также в Ada95 ввели child package (наследование пакетов)
package Sample.DerivedSample //подгружаем весь Sample сверху
type Derived is new Base with tagged record
        procedure P(D:Derived)...
        // можем использовать все из тела Base

Теряем полную инкапсуляцию → любой может так присоединиться к любому пакету и любой класс наследовать.

Мультиметод - метод, который привязан динамически к **нескольким** объектам.

class Figure {
public: virtual double getArea();
}
	Figure *pF = …;
	pF->getArea();

Figure* Intersection(Figure* pf1, Figure* pf2);
// в общем, надо описать для любой пары производных классов
Circle C;
Line L;

Figure *pp = Intersection(&C, &L);
// если для данных аргументов есть перегрузка, то выолнится то, что необходимо.

Figure *p1 = &C, *p2 = &L;
Intersection(p1, p2); // вызовет именно от двух Figure*.

2009г. - в **Python** появились **мультиметоды.**

```
def f(a, b): # привязки к типам нет
      …


from multimethods import *

@multimethod(int, int)
def foo(a, b):
      …

@multimethod(int, double)
def foo(a, b):
      …
```

вызов foo(a, b) будет адресован к тому multimethod, в котором типы аргументов будут совпадать.

foo(2.0, 2.0) → ошибка

Теперь есть специальный модуль **multimethods** {есть этот декоратор}.

JavaScript

```
O = {}  <---> O = new Object()
```

MyClass(...) { //⇐ это функция

```
      this.x = 0;   // this имеет ссылку на этот объект.
      this.y = 0;   // x и y - приобретенные свойства, заводятся тут.
}

O = new MyClass() // вызов функции-конструктора
O.prototype //  есть такое свойство, имеет значение прототипа функции-конструктора.


Figure() {
      this.x = 0;
      this.y = 0;
      this.prototype.area = function(){ return this.x * this.y; }
}

O = new Figure();
O.x = 10;
```

```
O.y = 10;
O.area();
```

Наследование:

Как-то определить прототип Figure как функции (но до первого вызова) ⇒ имеем наследование. Например, так:

```
function Circle(){
     this.radius = 1;
}

Circle.prototype = Object.create(Figure.prototype);
Circle.prototype.area = function() {
     return 3.14 * this.radius * this.radius;
};
```

Костя ([982-990], завершено)

Параметрический полиморфизм

Отличие параметрического полиморфизма от статического и динамического состоит в том, что при связывании может порождаться новая сущность — класс или функция (метод). Методы реализации параметрического полиморфизма достаточно сильно различаются для разных языков.

При параметрическом полиморфизме полиморфными сущностями являются параметризованные типы (классы) и подпрограммы (методы).
Два основных понятия параметрического полиморфизма — это **объявление параметризованной абстракции** (шаблона или обобщения) и **конкретизация этой абстракции** (т. е. ее использование). Связывание конкретизации и объявления всегда происходит статически (во время трансляции).

- В C++ при конкретизации шаблона происходит создание новых непараметризованных сущностей — классов и функций.
- В C# при конкретизации обобщения проверяется корректность (соответствие типов). Генерация машинного кода для создаваемых непараметризованных сущностей происходит при запуске сборки (JIT-компилятором среды .NET) на основе промежуточного кода, созданного при трансляции объявления обобщения, и информации о типах — аргументах конкретизации.
- В языке Java при конкретизации обобщения только проверяется корректность конкретизации (соответствие типов). Можно рассматривать конкретизацию как процесс создания новых непараметризованных сущностей (классов и методов), однако вся информация об этих сущностях существует только во время трансляции и стирается при генерации объектного кода для JVM.

Механизмы параметрического полиморфизма во всех языка позволяют обеспечить:

- повышение надежности создаваемых абстракций за счет статического контроля соответствия типов и повысить эффективность
- уменьшить объем разрабатываемого кода (меньше усилий)

generic: Ada83 - “родовые”

generic: C# (2003), Java (2005)
template: C++ (90ые)

Шаблоны в C++. Явная и частичная специализация шаблонов

Общий синтаксис:

```
template <список_параметров> объявление_функции_или_класса
```

Шаблонная функция	Шаблонный класс

`template<class T>`
`void g(T x, T y)`
`{`
`  // ...`
`}`
`....`
`int a, b = 0;`
`long c = 1;`
`g(a, b);`
`g<int, int>(a, b);`
`g(a, c);`
	`template<typename T>`
	`class Foo {`
	`public:`
	`  Foo();`
	`  void someMethod(T x);`
	`private:`
	`  T x;`
	`};`
	
	`template<typename T>`
	`Foo<T>::Foo()`
	`{`
	`  // ...`
	`}`
	`template<typename T>`
	`void Foo<T>::someMethod(T x)`
	`{`
	`  // ...`
	`}`

`template <typename T, int size> T`
`operator * (VectorCT,size>&vl,`
`Vector<T,size>&v2)`
`{`
`  T sum = 0;`
`  for (int i=0; i < size; i++)`
`  sum+=vl.getAt(i)*v2.getAt(i);`
`  return sum;`
`}`	

Ключевые слова **typename** и **class при описании прототипа** шаблонной функции не различаются.
А вот в этом случае внутри класса при typedef нужно использовать только **typename**.

```
template<typename param_t>
class Foo
{
    typedef typename param_t::baz sub_t;
};
```

Конкретизация шаблона приводит к порождению нового типа. Его именем является идентификатор шаблона. Идентификатор шаблона полностью эквивалентен имени типа. При порождении нового типа может происходить генерация кода для функций — членов нового типа.

Порожденные функции можно рассматривать как семейство перегруженных функций (считая, что имя порожденной функции совпадает с именем шаблона функции). Тогда возникает возможность вызова порожденной функции без явного указания параметров шаблона. Поскольку компилятор знает профиль вызова (например, (int)), то он может сопоставить его с профилем шаблонной функции ((T)) и отождествить (int<->T). Это процесс называется **выводом типа**.

Важно понимать, что для конкретизации необходимо иметь полный текст шаблона (как функции, так и класса, включая определения всех его членов). Порождение новой сущности происходит во время конкретизации с учетом контекста конкретизации и полного текста шаблона. Именно поэтому библиотеки шаблонов распространяются в исходных текстах (иначе их использовать нельзя).

Доступность полной информации как о шаблоне, так и о контексте конкретизации позволяет реализовать такие понятия, как перегрузка шаблонов функций, **явная специализация** шаблонов и **частичная специализация** шаблонов классов.

Явная специализация
Явная специализация шаблона порождает вариант шаблона для конкретного набора его аргументов. Явную специализацию иногда путают с конкретизацией, ведь и при конкретизации тоже указываются аргументы. Конкретизация указывает компилятору следующее: возьми шаблон, примени к нему аргументы и породи новый класс (функцию), а явная специализация — это указание взять новое описание старого шаблона, набор аргументов и считать, что это новый шаблон, который применим только к этому набору и порождает только один вариант.
вариант.

Явная специализация шаблона функции — это конкретная функция с именем шаблонной функции, перед объявлением которой стоит **template <>**

При явной специализации функций template <> можно опускать, тогда специализация будет иметь вид обычной функции. Для классов template<> опускать нельзя.

Когда компилятор видит вызов функции с именем шаблона, то он пытается сначала найти явную специализацию, и только если ее нет, ищет шаблон.

```
// некоторая шаблонная функция
template<class T> void f(T t) {
};
// явная специализация 1
template<> void f<char>(char c) {
}
// явная специализация 2
template<> void f(double d) {
}
```

```
template <typename T> class
Container {
// реализация
} ;

template <> class Container<const
char *> {
// оптимизированная реализация
} ;
```

Частичная специализация
Частичная специализация порождает новый вариант параметризованного шаблона, который является частным случаем общего шаблона. Набор аргументов, подходящих для частичной специализации, подходит и для общего шаблона, но является «более точным».

Например, общий шаблон использует просто имя любого типа, а частичная специализация — тип указателя, тип массива, функциональный тип и т. п. Другим вариантом специализации является меньшее число параметров, чем в общем случае, и т.д.

Пример:
```
template<class T1, class T2, int I>
class A {}; // общий шаблон

// частичная специализация, где T2 - указатель на T1
template<class T, int I>
class A<T, T*, I> {};

// частичная специализация, где T1 - указатель
template<class T, class T2, int I>
```

```
class A<T*, T2, I> {};

// частичная специализация, где T1 - int, I - число 5, T2 - указатель
template<class T>
class A<int, T*, 5> {};
```

Обобщения в C#. Ключевое слово where

На уровне синтаксиса универсальные шаблоны языка C# являются более простым подходом (по сравнению с C++) к параметризованным типам, исключающим сложность шаблонов языка C++. Кроме того, в языке C# не делается попытки обеспечить все функциональные возможности, обеспечиваемые шаблонами языка C++.На уровне реализации основное отличие заключается в том, что **замена универсальных типов языка C# выполняется во время выполнения** и информация универсальных типов, таким образом, сохраняется для экземпляров объектов.

Обобщения C# не поддерживают ни явной, ни частичной специализации, не позволяет параметрам типов иметь значения по умолчанию.

При определении универсального типа можно ограничить виды типов, которые могут использоваться клиентским кодом в качестве аргументов типа при инициализации соответствующего класса. При попытке клиентского кода создать экземпляр класса с помощью типа, который не допускается ограничением, в результате возникает ошибка компиляции.Это называется **ограничениями**. Ограничения определяются с помощью контекстно-зависимого ключевого слова **where**.В следующей таблице приведены шесть типов ограничений:

where T: struct	Аргумент типа должен иметь тип значения.
where T : class	Аргумент типа должен иметь ссылочный тип; это также распространяется на тип любого класса, интерфейса, делегата или массива.
where T : new()	Аргумент типа должен иметь открытый конструктор без параметров. При использовании с другими ограничениями ограничение new() должно устанавливаться последним.
where T : <base class name>	Аргумент типа должен являться или быть производным от указанного базового класса.
where T : <interface name>	Аргумент типа должен являться или реализовывать указанный интерфейс. Можно установить несколько ограничений интерфейса.Ограничивающий интерфейс также может быть универсальным.
where T : U	Аргумент типа, предоставляемый в качестве T, должен совпадать с аргументом, предоставляемым в качестве U, или быть производным от него.

```
 // Общий шаблон
public class GenericList1<T> {
    void Add(T input) { }
}
```

```
// Общий шаблон с ограничениями
public class GenericList2<T> where T : class, new() {
    void Add(T input) { }
}
class TestGenericList {
    private class ExampleClass { }
    static void Main() {
        // Declare a list of type int.
        GenericList<int> list1 = new GenericList1<int>();

        // Declare a list of type string.
        GenericList<string> list2 = new GenericList1<string>();
    }
}
```

Обобщения в Java. Ограничения с помощью extends, super и wildcards

Обобщения в Java по своей природе аналогичны обощениям в C#. Синтаксис определения шаблонных методов и классов является фактически одинаковым за исключением синтаксиса ограничений.

- <T **extends** X> - тип T должен либо совпадать с X, либо являтся производным от него классом.
- <T **super** X> - тип T должен либо совпадать с X, либо являтся одним из его суперклассов (предков). Можно использовать данное ограничение только с шаблонными методами, но не с классами.
- <**?** extends X>, <**?** super X> - если в теле шаблонного класса или функции нам не нужно работать непосредственно с подставляемым типом, то можно использовать **wildcard**-нотацию.

Пример:

```
public class Garage<X extends Vehicle> { }

class Car extends Vehicle { }
class Motorcycle extends Vehicle { }
class Fruit extends Object { }

class Vehicle { }
interface PassengerVehicle { }
interface MotorVehicle { }
class ParkingGarage<X extends Vehicle & MotorVehicle & PassengerVehicle>

int totalFuel(List<? extends Vehicle> list) {
    int total = 0;
    for(Vehicle v : list) {
        total += v.getFuel();
    }
    return total;
}
```

Шаблоны в Ада-83

В Аде используется понятие дженериков (generic units). По своей природе они очень близки к шаблонам C++.
В Ада объявление шаблонной функции просходит в два действия: объявления дженерика и объявление самой функции.

Примеры:

Обмен двух чисел

```
generic
  type Element_T is private;
procedure Swap (X, Y : in out
Element_T);

procedure Swap (X, Y : in out
Element_T) is
  Temporary : constant Element_T := X;
begin
  X := Y;
  Y := Temporary;
end Swap;
```

Каркас шаблонного стека

```
generic
  Max: Positive;
  type Element_T is private;
package Generic_Stack is
  procedure Push (E: Element_T);
  function Pop return Element_T;
end Generic_Stack;

package body Generic_Stack is
  Stack: array (1 .. Max) of
Element_T;
  Top  : Integer range 0 .. Max := 0;
  -- ...
end Generic_Stack;
```