Московский Государственный Университет
имени М.В. Ломоносова

Факультет вычислительной математики и кибернетики

Конспект лекций
по курсу
Языки программирования

Автор лекций:
Головин Игорь Геннадьевич, доцент, к.ф.-м.н.

Лекции записали:
Борисова Татьяна
Николаев Евгений
Сукманова Дарья

Москва, 2012

Оглавление

Литература по курсу:

- Пратт, Зелковиц «Языки программирования. Разработка и реализация».

Введение

Начнем с определения языка программирования.

Язык программирования – это инструмент для планирования поведения электронного или механического исполнителя.

Таким образом, понятие исполнителя неразрывно связано с поведением этого исполнителя.

Первым исполнителем можно считать человека, но в определении идет речь об искусственном исполнителе.

Первым механическим исполнителем можно считать Жаккардов ткацкий станок, предназначенный для выработки крупноузорчатых тканей и позволявший программно управлять процессом создания ткани при помощи перфокарт.

Известным механическим исполнителем была машина Бэббиджа. Она была предназначена для автоматизации вычислений.

Компания IBM первоначально выпускала машины для автоматизации переписи населения (подразделение Tabulating Machine Company) – они тоже являлись механическими исполнителями.

В данном курсе под исполнителем мы понимаем некоторую вычислительную систему. Однако, могут быть и виртуальные исполнители.

Возникает вопрос, можно ли язык разметки страниц HTML назвать языком программирования. Для этого языка существует исполнитель – браузер. Однако в нем нет алгоритмической полноты.

Заметим, что язык включает в себя символьную нотацию и удовлетворяет законам символьных систем. Изучением символьных систем занимается наука семиотика.

Парадигма программирования

Определение (Парадигма программирования)

Парадигма программирования – это набор способов, методов, нотаций, использующийся при данном стиле программирования.

Парадигма программирования – это неформальное понятие.

Пример (Smalltalk)

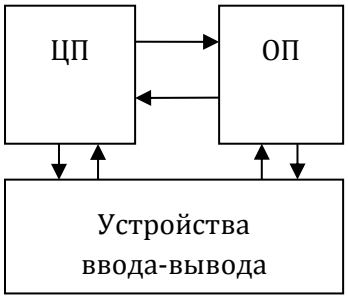
Язык Smalltalk воплощает объектно-ориентированную парадигму программирования.

Таким образом, парадигма программирования связана с языком программирования.

Императивная парадигма

Императивная парадигма является основной парадигмой программирования, так как на императивной парадигме основаны все вычислительные системы.

Рассмотрим несколько упрощенный вариант архитектуры фон Неймана.



Императивная парадигма, базирующаяся на архитектуре фон Неймана, включает в себя следующие понятия:

1. Переменная – абстракция ячейки памяти
2. Оператор – действие, которое меняет состояние вычислительной машины
   Основные операторы:
   - оператор присваивания (он напрямую меняет состояние ЭВМ);
   - оператор ввода-вывода;
   - оператор управления.

   Остальные операторы являются группировкой
   Операторы могут обладать побочными эффектами.

Пример (FORTRAN)

```
IF (E) M1, M2, M3

DO

CALL P
```

Остальные операторы – операторы перехода.

Заметим, что IBM 709, на которой первоначально работал FORTRAN, была трехадресной.

---

3. Выражение
   Операция – служит для вычисления значения.
   Пример: операция сложения.
   Выражение – конструкция, в которой можно группировать несколько операций.

Пример (C)

Классический процедурный язык.

Будем сопровождать лекцию примерами программ, написанных на различных языках программирования. Для наглядности будем рассматривать одну и ту же программу, которая считывает последовательность символов со стандартного ввода и выводит их на стандартный вывод в обратном порядке. Будем называть этот пример «стандартным примером».

Пример (C)

```
#define MAX_INP 4096
char Buffer[MAX_INP];
int main() {
  int index = 0;
  int curch;
  int i;
  while((curch = getchar()) != EOF) {
        if(index == MAX_INP) {
              fprintf(stderr, "Input is too long\n");
              exit(1);
        }
        Buffer[index++] = curch;
  }
  for(i = index - 1; i >= 0; --i)
        putchar(Buffer[i]);
  return 0;
}
```

Характерное свойство процедурных языков: при изменении входных данных задачи значительно изменяется программа. Однако, для каждой конкретной программы можно написать оптимальное решение.

Объектно-ориентированная парадигма

Как следует из названия, ключевым понятием для этой парадигмы является понятие объекта.

Объект можно определить как некую сущность, которая обладает определенными поведением и состоянием.

В объектно-ориентированных языках понятию объекта отвечает конструкция языка, называемая класс.

Пример (JavaScript)

В языке JavaScript нет классов. Он является прототипным языком. В прототипных языках отсутствует понятие класса, а повторное использование (наследование) производится путём клонирования существующего экземпляра объекта — прототипа. Его также относят к объектно-ориентированным языкам.

Рассмотрим наш стандартный пример на объектно-ориентированном языке C#.

Пример (C#)

```
public class Program {
  public static void Main(string[] args) {
        var s = System.Console.In.ReadToEnd();
        for(int i = s.Length - 1; i >= 0; --i)
              System.Console.Write(s[i]);
        /* другой вариант (вместо цикла)
        char[] arr = s.ToCharArray();
        System.Console.Write(arr.Reverse());
        */
  }
}
```

В ООП применяются объекты, а также методы объектов.

Одновременно с объектно-ориентированными языками появились RAD – среды быстрого программирования. Самая популярная среда разработки, использующая RAD – Visual Basic, особенно она популярна в Америке. Другой популярной средой разработки, использующей RAD, является Delphi.

Самой популярной архитектурой современных компьютеров является Intel. Однако, для них характерно большое энергопотребление.

Для мобильных устройств применяется архитектура ARM. Она отличается меньшим энергопотреблением, но машинный язык не совместим с архитектурой Intel.

Первые Macintosh работали на архитектуре Motorola.

Motorola была лучшей архитектурой, но со временем Intel вытеснили ее с рынка.

Парадигма обобщенного программирования является разновидностью объектно-ориентированного программирования. Она включает в себя статическую параметризацию.

В чистой объектно-ориентированной парадигме существуют абстрактные классы и механизм наследования. Они позволяют уточнять поведение производных классов.

Пример (C#)

sealed – ключевое слово, означает запечатанный класс: класс, от которого нельзя наследовать.

Определение (Абстрактные иерархии классов)

Абстрактные иерархии классов – иерархии классов, которые служат лишь для возможности наследования от них.

В обобщенном программировании существуют обобщения (generic).

Для иллюстрации различий между этими подходами рассмотри пример вызова функции. В процедурной и чистой объектно-ориентированной парадигме связывание формальных и фактических параметров при вызове функции осуществляется на этапе компиляции. Обобщения связываются до начала выполнения, на этапе компиляции.

Пример (C++)

STL – «бескомпромиссное стремление к эффективности».

Пример кода на STL:
```
copy(from, to, where)
```
где from, to – входные итераторы, where – output-итератор.

STL включает в себя множество контейнеров – специальных классов, предназначенных для хранения объектов и работы с ними.

Контейнер-список:
```
list<char>
```

Стандартные потоки ввода-вывода – тоже контейнеры:
```
ostream_iterator<char>
```

Синтаксис STL во многом схож с указателями C:
```
*where++ =*from; // копирование
```

Итератор inserter вставляет в конец, back_inserter – в начало (возможно, со сдвигом)

Рассмотрим наш стандартный пример на языке C++.
```
int main() {
  list<char> c;
  copy(isteam_iterator<char>(cin),
       istream_iterator<char>(),
       back_inserter(c));
```

```
  copy(c.begin(),c.end(),ostream_iterator(cout));
}
```

Сопровождение грамотно спроектированных программ становится легким.

С точки зрения парадигмы не существует чистых объектно-ориентированных языков программирования.

Первым объектно-ориентированным языком программирования был язык Simula 67 (расширение языка Алгол 60).

Автор объектно-ориентированного языка C++ – Бьярн Страуструп. Он его разрабатывал для имитации моделирования телефонных систем.

Существует несколько других модификаций языка C.

- Язык Objective-C – тоже настройка над C, объектно-ориентированное расширение языка C. Используется в операционных системах Apple.
- C++/CLI – .NET версия языка C++ от компании Microsoft. CLI расшифровывается как Common Language Interface.
- C# – тоже .NET язык, отдельные его части взяты из C++, другие из .NET.

Основная проблема программирования на C++ – сложность языка. Объясняется она тем, что язык C++ развивался достаточно долго, и все это время он расширялся. Необходимо было поддерживать обратную совместимость со старыми программами. В результате язык оказался переполнен возможностями и стал слишком сложным. Полностью новый язык мог бы быть более прост и эффективен.

ФУНКЦИОНАЛЬНАЯ ПАРАДИГМА

Рассмотрим основные особенности функциональной парадигмы на нескольких примерах.

В качестве первого примера рассмотрим наш стандартный пример на языке Lisp.

Пример (Lisp)

Рассматриваемый нами стандартный пример на Lisp выглядит особенно кратко:
(`print(reverse(read))`)

На примере Lisp можно понять многие особенности функциональных языков программирования.

Пример (Lisp)

Lisp является мультипарадигменным языком, но чистый Lisp воплощает функциональную парадигму.

Любая сущность в языке Lisp является списком.

Существуют так называемые «атомы»: целое (`123`*), символ (*`+`*,* `Коля`*), пустой список(* `()`*,* `nil`*).*

Списки могут быть вложенными:
```
(a (b c ) x)
```
Глубина вложенности не ограничена.

Lisp поддерживает иерархические списки.

Основной операцией в Lisp является применение функций к списку. Если список рассматривается как функция, то первый символ в списке – это имя функции. Пример:
```
(+ 2 3)
```
Результат:
```
(5)
```

Разберем подробнее наш стандартный пример.
`read` *– встроенная функция, считывает символ из входного потока. Результат выполнения функции – список атомов из входного потока.*
```
(print(reverse(read)))
```

В языке Lisp есть конструкция eval:
```
(eval s)
```
– вычисление функции S.
Это означает, что в Lisp встроен интерпретатор.

Стандартный Lisp получил название Common Lisp.
В него входит множество стандартных функций, а стандарт языка занимает 15000 страниц.

Рассмотрим еще один пример: функцию, прибавляющую единицу к ее аргументу.
Синтаксис функции в Lisp задается следующим образом:
```
(defun имя (список переменных-параметров) тело)
```
Тогда функция, прибавляющая единицу к ее аргументу, будет выглядеть так:
```
(defun  plus1(x) (+ x 1))
```
В общем случае она ничего не знает о типе `x`*. Подразумевается лишь, что к* `x` *применима операция сложения.*
Переменную также называют place holder.

Рассмотрим «наивную» функцию reverse на Lisp
```
(defun reverse1(x)
  (if  (NULL x) ()
  true (append (reverse1) cdr x)
       cons(car x) ()))
```

```
(defun reverse1 (x)
  (cond
    ( (null x) () )
    ( T (append (reverse1 (cdr x))
                (cons (car x) ()) )
    )
  )
)
```

```
    )
)
```

Её сложность $O(n^2)$.

Рассмотрим еще несколько важных функций работы со списками.

Функция `append` *создает новый список.*
`(append s1 s2)` (append (1 2 3) (4 5)) = (1 2 3 4 5)

Функции `car` *и* `cdr` *очень важны для работы с списками.* `car l` не создает список, *создает список, состоящий из первого элемента списка* `l`*. cdr* `l` просто первый элем. *создает список, состоящий из всех элементов списка* `l`*, кроме первого.*

Пример:
```
l = (a b c)
car l = (a)  = a
cdr l = (b c)
```
Заметим, что функции `car` *и* `cdr` *не изменяют тот список, к которому применяются.*

Функция `cons` *формирует список*
`(cons s1 s2)= (s1 s2)` (cons (1 2 3) (4 5)) = ((1 2 3) 4 5)
Для того, чтобы сформировать список из атома, нужно добавить `()`
`(cons 5 ()) = (5)`

Заметим, что большинство стандартных функций Lisp можно написать на Lisp, однако функцию `cons` *– нельзя.*

Раньше Lisp был одним из основных языков для создания AI (Artificial Intelligence, искусственный интеллект).

У языка Lisp существует множество диалектов, один из которых – язык PLANNER.

Пример (PLANNER)
Язык Planner – это расширение языка Lisp. служил для передачи информации от человека к человеку.
На этом языке можно было описывать невычислимые вещи.
Пленэр – это урезанная версия PLANNER.

В 1978 году Джон Бэкус, руководивший разработкой FORTRAN, получил премию Тьюринга. На вручении этой премии он прочитал свою знаменитую лекцию, которую озаглавил «Можно ли освободить программиста от фон-неймановского стиля? Функциональный стиль и определяемая им алгебра программ».

В этой лекции Джон Бэкус говорил о том, что стиль императивных программ «одна операция в один момент времени» примитивен. Он назвал языки программирования,

базирующиеся на фон-неймановской модели вычислителя, «машинами с операцией присваивания» и указал, что в операции присваивания кроется несовершенство из-за разделения вычислений на математически чистую абстрактную часть (функции) и "приземленную" часть присваиваний, которая необходима из-за особенностей аппаратных средств. Бэкус доказал, что такое разделение кода является источником очень высокой сложности языков программирования. В качестве альтернативы он предложил функциональный стиль программирования – вообще без переменных, то есть без операций присваивания и без возможности присвоения данным имен.

В качестве возражения тезисам Джона Бэкуса можно указать, что неимперативные программы «более естественны» с аппаратной точки зрения.

В последнее время функциональная парадигма возвращается в индустриальное программирование. Объясняется это тем, что функциональная парадигма позволяет эффективно распараллелить вычисления.

В 2006 году мощным процессором считался Intel Pentium 4 с тактовой частотой 4.06 ГГц. Сейчас, в 2012 году, частота мощных процессоров остается примерно той же, и повышение производительности достигается за счет проведения вычислений параллельно на нескольких ядрах.

Замечание

Суперкомпьютер – это такой компьютер, который всего на одно поколение отстает от потребностей физиков.

Однако, следует помнить, что у распараллеливания существуют ограничения.

С параллельными вычислениями тесно связано понятие ленивых вычислений в функциональных языках программирования.

Пример (Lisp)

```
(fun 1 arg1 … argN)
```

Суть ленивые вычислений: arg вычисляется только тогда, когда он необходим. Таким образом, можно гарантировать, что аргументы независимы, а значит, их можно вычислить параллельно. В императивных же программах такие гарантии сложнее обеспечить из-за возникающих зависимостей между частями кода.

Императивное программирование практически достигло предела возможного распараллеливания.

Заметим, что технологические аспекты часто не играют решающей роли при распараллеливании чего-либо.

Следующий язык, который мы рассмотрим в качестве примера – язык REFAL.

Пример (REFAL)

Это единственный язык, который широко известен во всем мире и был придуман в СССР.

Первая версия REFAL была создана в качестве метаязыка для описания семантики других языков. Впоследствии он стал находить практическое использование в качестве языка программирования.

Семантика языка Рефал описывается в терминах виртуальной машины, называемой «рефал-машина». Машина имеет поле зрения, в котором может находиться произвольное рефал-выражение, не содержащее рефал-переменных.

Функции в базисном REFAL:

```
имя {
  правило1
  правило2
  …
  правилоN
}
```

Правило состоит из образца и шаблона.

На вход функции подаётся некоторое выражение; вычисление функции состоит в сопоставлении выражения поочерёдно образцам, взятым из `правило1`, `правило2`, *…,* `правилоN`*. Если очередное сопоставление проходит успешно, то на основании шаблона, взятого из того же предложения, формируется новое Рефал-выражение, которое и будет результатом функции.*

Образец состоит из скобок и переменных.

Переменные, описываемые в образцах, описываются следующим образом:

`s.1` – *один символ;*
`v.1` – *произвольное непустое выражение;*
`e.1` – *произвольное выражение.*

Язык REFAL можно использовать для XML-преобразований.

Рассмотрим теперь наш стандартный пример на языке REFAL.

Пример (REFAL)

```
reverse1 {
s1 e1 = reverse1<e.1> s.1;
      =;
}
```

Вызов этой функции осуществляется так:

```
$ENTRY GO {
  = Prout <reverse1<card>>;
}
```

Логическая парадигма

Логическую парадигму рассмотрим на примере, пожалуй, самого известного логического языка – языка Prolog.

Пример (Prolog)

Prolog был изобретен в 1971 году.

Prolog основана исчислении предикатов – исчислении высказываний для функций.

Программы на Prolog оперируют с понятием факта.

`man(SOKRAT).`

`SOKRAT` *является* `man`. `man` – *предикат.*

$$\forall X\ p(X) \in p_1(x) \wedge p_2(x) \wedge \dots \wedge p_n(x)$$

Правила в Prolog формулируются следующим образом:

$p(x) :- p_1(x), p_2(x), \dots, p_n(x).$

и читаются так: $p(x)$ *истина, если* $p_1(x), p_2(x), \dots, p_n(x)$ *истина.*

Такие правила называются Хорновскии клаузами или дизъюнктами Хорна.

?: - pp(x)

Непоянтно, как доказать истинность этого предложения.

Сложность вычислений линейно зависит от размера входных данных. Это означает, что существует линейный алгоритм определения истинности.

MAN(SOKRAT).

MORTAL(X):-MAN(X)

?:-MORTAL(SOKRAT)
Эта программа определяет, верно ли MORTAL(SOKRAT).

В Prolog есть понятие списка.
```
[a,b,c]
[a|b, c] ~ [a, b | c] ~ [a, b, c | [] ]
```
| – символ для разделения головы и хвоста списка.

append(l_1, l_2,l_3)

```
append([a,b],[c],[a,b,c])
```
ответ: yes

```
?:-append([a,b],[a,b|x], [a,b,c])
```
ответ: no

```
?:-append([a,b],[x|c], [a,b,c])
```
ответ: no

```
?:-append([a,b],[c|x], [a,b,c])
```
ответ: []

В процессе вычисления происходят отождествления, которые и выводятся.

Рассмотрим теперь наш пример на языке Prolog.
```
reverse1([],[]).
reverse1([x|y],z) :-
  reverse1(y,w),append(w, x, z)
?:-reverse1([a,b,c],x)
```

Результат: [c, b, a]

Существует множество подпарадигм в рамках вышеназванных Например, многие ЯП являются мультипарадигменными.

Пример (Lisp)

SET, SETQ – *процедурная составляющая языка*

CLOS – *объектно-ориенттированное расширение).*

Будем рассматривать и говорить в основном об императивной и объектно-ориентированной парадигме, рассматривая языки C++, Java, C#.

Параллельная парадигма

CSP – описание взаимодействия между параллельными процессами.

Почему не говорим про параллельные парадигмы?

1. Их будем рассматривать в других курсах
2. При обсуждении параллельных парадигм возникает вопрос: что является фундаментальным знанием? CSP реализует одну парадигму, Ocaml реализует другую парадигму, другие понятия. В параллельном программировании ещё не устоялись управляющие структуры. Фундаментальное знание выделить нельзя.

Существует два подхода к параллельному программированию

1. Программист не задумывается о том, как распараллеливать программу.
   Здесь возникает вопрос: а должны ли мы учитывать особенности кэш-памяти и другие особенности машинной архитектуры? Например, есть мнение, что списки вообще не стоит использовать в программах, так как «наивная» реализация списков не адаптирована под кэш-архитектуру современных процессоров (часто возникают кэш-промахи).
2. Создание специальных библиотек и средств, чтобы сразу программировать на параллельность и явно использовать эти средства

С распараллеливанием связан критический прорыв в эффективности и производительности программ.

В программировании мы всё дальше и дальше уходим от свойств архитектуры, поэтому логичнее первый подход: программист должен иметь возможность не работать с параллелизмом.

Существует множество неразрешимых проблем при распараллеливании процедурных программ. Вообще, параллельное программирование для машин фон-неймановской архитектуры является чрезвычайно сложным.

Одна из важных задач – распараллеливание программ на языке FORTRAN для физиков.

Другая важная задача – автоматическое распараллеливание, которое сильно усложняет программу (процессоры INTEL распараллеливают вычисления на уровне выполнения программы).

Заметим, что если проводить вычисления на n процессорах, то выполняться не в n раз быстрее, а несколько медленнее, потому что существуют издержки на синхронизацию процессов.

Возникает вопрос: какими должны быть средства, которые позволяют не задумываться о параллельном программировании? Для этого хорошо подходят функциональные языки.

`F(expr1, expr2, …, exprN)` – можно вычислять параллельно, так как отсутствует понятие состояния

`F(c++, a[i])` – результат зависит от того, в каком порядке выполнять.

В функциональной парадигме отсутствует понятие состояния

В языке Java есть замыкание.

В замыкание могут входить только переменные которые потом не изменяют свое значение – это сделано для распараллеливания.

Существует понятие ленивые вычисления. Ленивые вычисления вычисляются только тогда, когда есть необходимость в их.

С помощью ленивых вычислений можно моделировать бесконечную последовательность (Python).

В рамках процедурной парадигмы существует объектно-ориентированная парадигма.

Чаще всего будем говорить о языках объектно-императивной парадигмы.

Понятие «оператор» – характерно только для императивной парадигмы, понятия «данные», «операция» – ко всем.

Проблемные области

Будем сужать область рассмотрения по степени отчуждаемости результата программирования.

По-другому это можно сформулировать так: кому нужен результат программирования, кто будет его использовать?

Игровое программирование

Заметим, что игровое программирование не следует путать с программированием в игровой индустрии.

Игровое программирование другими словами – это программирование для себя.

Частным случаем является программирование студентов – нет дальнейшего использования программы. Реальные программы будут читаться другими программистами. Студенческие программы просматривает преподаватель.

Самым популярным языком для игрового программирования был BASIC. Его особая прелесть для игрового программирования – строчка вводилась, непосредственно интерпретировалась и выполнялась, таким образом позволяя интерактивно исследовать язык и обучаться ему.

Пример (BASIC)

Интерпретатор BASIC позволяет работать в интерактивном режиме. Это интерпретируемый язык с инкрементной трансляцией.

```
LET x = 5
PRINT X
FOR I=1, N
---
---
NEXT I
```

Научное программирование

Это программы, которые пишутся в научных, исследовательских целях. Программирование с целью получения результата – после получения результата сама программа обычно становится не нужна.

Степень отчуждаемости больше, чем при игровом программировании. Отчуждается не столько программа, сколько результат. Программа может отчуждаться, но обычно она работает вместе с автором.

Многие средства, используемые современными исследователями, (например, библиотека NLTK) написано на Питоне.

Часто игровое программирование превращается в научное программирование. Например, человек писал проект для себя, но проект может начать использоваться для получения результата.

Индустриальное программирование

Индустриальное программирование – это производство программных продуктов – специальных программных систем, которые выполняются на заказ.

Важным свойством программных продуктов является то, что они отчуждаемы от автора.

Инструменты индустриального программирования очень сильно отличаются от инструментов научного и игрового.

Далеко не для всех задач нужны инструменты индустриального программирования. Для физиков, например, важна простота, распространенность языка и огромные запасы ПО в соответствующих областях – поэтому они пишут на Фортране. Им не нужны средства индустриального программирования. Фортран простой, распространенный, на нём уже есть огромные запасы ПО, а также он достаточно быстро работает. И не так важно, какой язык программирования используется, если он позволяет быстро решить задачу.

В нашем курсе мы будем заниматься языками для индустриального программирования.

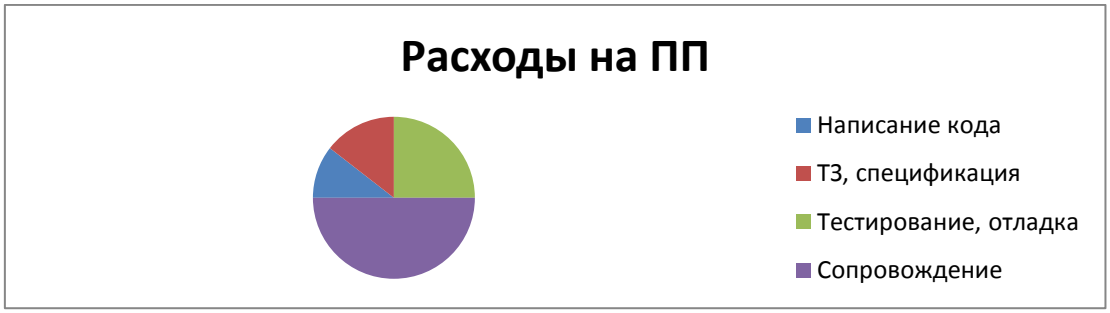
Средства разработки для коммерчески-ориентированных систем должны отличаться от научных.

Жизненный цикл программного продукта

В начале 1970-х годов в Министерстве Обороны США решили посчитать, куда идут деньги, выделяемые на компьютерную индустрию. Расходы на оборудование легко посчитать и проконтролировать – можно видеть физический объект, устройство. Расходы на программистов не так легко проконтролировать. По магнитной ленте непонятно, сколько было затрачено работы.

Исследование постановило:

- 10-11% средств тратится на написание собственно кода;
- 15% тратятся на спецификацию, постановку целей и прочее;
- 25% тратятся на тестирование и отладку;
- 50% на сопровождение (maintenance).



Предметными областями Министерства Обороны были бухгалтерские системы (для которых характерны тривиальные вычисления над очень большими объемами данных, записанные в понятной человеку форме), системы ПРО, системы ПВО, системы вооружений.

Из этого следует, что в первую очередь нужно сокращать расходы на сопровождение программ. Именно эту задачу призваны решать индустриальные средства программирования.

Один из способ уменьшения расходов на программное обеспечение – не нанимать программистов самим, а заказывать ПП на стороне. Это называется аутсорсинг (outsource). Например, типичной outsource-фирмой является Luxoft.

Существует технология OTS (On the Shelf) – дословно, «программа с полки». Идея технологии в том, чтобы подогнать готовую программу, а не писать новую (Word, Excel).

В рамках технологии OTS была разработана концепция ERP (Enterprise Resource Planning – планирование ресурсов предприятия). Это огромные программные комплексы, которые нужно настраивать на месте для каждого конкретного предприятия. Примеры таких комплексов: SAP R/3, Baan. Для настройки используются собственные скриптовые языки. Они тривиальны, а вся сложная логика зашита в библиотеках.

Самые дешевые системы, выполняющие аутсорсинговые работы, находятся в Индии. Однако, для outsource необходимо чрезвычайно много писать спецификации и документацию.

Вернемся к вопросу сопровождения программных продуктов. Зачем оно нужно?

1. Меняется аппаратура.
2. Поиск ранее не обнаруженных ошибок (например, такое может быть актуально для распараллеленных программ, где может быть много сочетаний данных). Заметим, что исправление ошибки дешевле всего на этапе спецификации.
3. Повышаются потребности. Если производитель не реагирует на изменяющиеся потребности, то клиент уходит.

Для сопровождения и нужны читабельные программы, это очень важный аспект. Грамотно написанную объектно-ориентированную программу легко изменить. Любой программный продукт обязан модифицироваться (причем это не зависит от предметной области).

Другое важное требование к программным продуктам – это надежность. Это важно для штучных изделий (ракеты, спутники), для firmware (того, что зашивается в ПЗУ), поэтому там нельзя допускать ошибок.

Точки зрения на языки программирования

Технологическая (Что? Как?)

Технологическая точка зрения отвечает на вопросы «Что? Как?».

Заметим, что речь здесь идет только об индустрии программирования.

Существуют некоторые технологические потребности и критические технологические потребности (например,возможность раздельной трансляции модулей).

Примеры технологических точек зрения на языки программирования:

- Pascal не является языком технологического программирования.
- Керниган в языке AWK – ориентирован на обработку config файла – текстовый.
- Страуструп «Язык программирования C++»

Авторская (Почему?)

Авторская точка зрения отвечает на вопрос: почему было принято то или иное решение?

Эта точка зрения не менее важна, чем технологическая

Примеры, когда язык программирования обсуждается в ключе: а как можно было бы сделать по-другому

- Страуструп «Дизайн и проектирование C++»
- Вирт в статье о языке Оберон объясняет, почему он создал Оберон именно таким.
- Создатели языка ADA 2: LRM (Language Reference Manual) – технологическая точка зрения, Rationale – обоснование.

Реализаторская

В 1961 году вышла статья под названием «Язык программирования – это то, что определяется компилятором».

Ключевой документ для языка программирования – это его стандарт. Отклонения от стандарта не приветствуются, хотя допускаются диалекты. Например, C89 был совместим со стандартом C++, а C99 – уже нет.

Компиляторы gnu gcc, gnu g++ хорошо соответствуют стандарту, хотя и в них встречаются диалектизмы. Например, в gnu gcc есть вложенные функции, которых в стандарте нет.

В нашем курсе мы будем прибегать к реализаторской точке зрения тогда, когда это проясняет основную концепцию.

Семиотическая

Семиотика – наука, которая изучает общие законы знаковых систем.

Графематика – уровень лексики, анализ представления

Синтаксика– структура ЯП.

Семантика – смысл, который вкладывается в предложения ЯП.

Прагматика – отвечает на вопрос «зачем?».

Эти четыре понятия важны с точки зрения естественных языков.

Для объяснения этих понятий можно привести следующую аналогию: пусть есть отправитель и получатель, отправитель шлет получателю письмо. Графематика отвечает на вопрос, как изображено письмо (буквы, шрифт, лексика). Синтаксика отвечает на вопрос о структуре письма. Семантика отвечает на вопрос о смысле письма: о чем пишет оправитель? Прагматика отвечает на вопросы – зачем получатель принимает сообщение, а отправитель отправляет его? Как получатель воспринимает сообщение?

Знаковые системы нужны для механизмов передачи сообщения. Сообщение с точки зрения семиотики – текст программы.

Синтаксис ЯП удалось хорошо описать (БНФ, грамматики класса 2). Однако средства описания семантики оказались чересчур сложными.

На каждом из четырех уровней существуют неоднозначности и противоречия.

Примеры синтаксической неоднозначности:

«Привет освободителям востока *от Феликса Дзержинского*»

«Толстая торговка *вяленой воблой* стояла за прилавком»

Другой известный пример из языков программирования (позже мы остановимся на нем подробнее):

```
If B then

    If B1 then S1

    Else S2
```

Здесь неоднозначно, к чему относится else – к какому из двух условий.

В языке программирования Python эта неоднозначность разрешается просто: обязательно писать с отступами: отступы говорят, где конец.

Прагматическая неоднозначность – нетривиальные примеры в ЯП.

В естественных языках Кауфман приводит пример: лектор объявляет о коллоквиуме. Лектор после этого рассуждает так: «Сейчас все будут меня слушать и записывать».

Студенты рассуждают иначе: «А что его слушать? Пора уходить,читать пособия!». В итоге лектор добился противоположного.

Другой пример:

Вирт ввел в язык МОДУЛА2 встроенный тип данных `address` (аналог `void*`):

```
POINTER TO T
```

Он был нужен для реализации менеджера памяти. Также он используется для создания коллекций типов (generic collection) – таких коллекций, в которых элементы могут быть любого типа. Однако с помощью него можно обойти любое ограничение системы типов. Это прагматическая неоднозначность.

В языках C#, Java, Delphi ранних версий проблема generic collection решалась при помощи общего корня иерархии – тип Object. Коллекции типов, у которых есть общий предок, называются гомогенными, а коллекции типов без ограничений – гетерогенными.

Преобразование от Object к чему-то контролируемое.

```
(T) e
```

– e-object ссылается ли на T? Если нет – исключение

Плюс такого подхода в том, что это надежно Минус накладные расходы(особенно C++), диагностика возможна только в Runtime, необходимость поддержки множественного наследования.

В 1985 Страуструп, автор языка C++, выбирал между множественным наследованием и шаблонами. Выбрал первое, что было ошибкой. Позднее появились ObjectiveC (похож на SmallTalk), C++ – расширения языка C. Некоторые говорят, что если возможности какого-то языка нет в Smalltalk, то это не объектно-ориентированный язык. В нем нет множественного наследования. Утверждение: множественное наследование нельзя реализовать эффективно на подобных языках. В итоге из-за того, что Страуструп выбрал множественное наследование, пять первых лет языка C++ прошло без стандартной библиотеки, и все ведущие компиляторы имели свои реализации библиотеки. От стандарта 1998 года прежде всего ждали STL.

Социальная

Самой лучшей системой RAD (Rapid Application Development) в середине 1990-х была Delphi. В ней была мощная библиотека VCL. Однако Visual Basic 4.0, хотя и был хуже, стал популярнее, во-первых, благодаря мощной поддержке Microsoft и, во-вторых, потому, что Pascal не было принято изучать в качестве первого языка программирования.

Схема рассмотрения ЯП

I. Базис языка программирования

Базис языка программирования – те составляющие ЯП, которые встроены в компилятор и определяются синтаксисом.

Базис может быть скалярным и структурным.

Скалярный базис – простые типы данных и операции.

Структурный – составные типы данных операции над ними, операторы.

Современные императивные ЯП обладают следующими свойствами:

1. Скалярный базис очень похож
2. Структурный базис деградирует

II. Средства развития

Средства развития – новые типы данных ЯП.

Пример: в C++ вся стандартная библиотека реализована средствами C++.

Заметим, что C++ – это единственный язык, в котором строковый тип (std::string) реализован средствами языка. В других языках зашито в компилятор или .NET

Существует два основных средства развития ЯП:

a. Процедура
b. Класс

III. Средства защиты.

Средства защиты говорят о том: как язык защищает программиста от неправильного использования средств ЯП.

Другими словами – средства поддержки необходимых уровней абстракций.

Простейший пример: const.

Заметим, что при изучении классов возникает вопрос, зачем же они нужны, если все то же самое можно написать на структурах? Например, существовала оконная библиотека XWindow. Библиотека клиента Xlib была написана на C в процедурном стиле. Существовал также XToolkit – framework для объектно-ориентированного программирования на C. В нем был базовый набор «классов», которые выглядели примерно так:

Пример (C)

```
struct Window {
    Core _core_part;
    Window _window_part;
};
```

Главным оператором в коде был switch. Казалось бы, чего не хватает, вот вам объектно-ориентированное программирование на C. А не хватает как раз средств защиты. Полноценный класс позволяет инкапсулировать и таким образом защищать.

Базис языков программирования

В курсе мы будем рассматривать индустриальные языки программирования. Основная проблема индустриального программирования – это сложность. Самые сложные программы – это операционные системы. Что такое миллион строк? Несколько шкафов, заполненных бумагой. Осознать одному человеку такой объем кода просто не под силу. Порядок сложности какой? Например, X Server, сложность 100 тысяч строк на языке Керниган Ритчи C, а также реализация протокола tcp/ip. В общем, не разобраться в такой программе. Однако люди это делают. Помогают уровни абстракции. Заметим, что если руководитель не может изложить суть проекта на одной странице, значит он ничего не понимает, и его надо уволить.

Для уменьшения сложности необходимы средства защиты – средства поддержки необходимого уровня абстракции. Нас будут интересовать именно средства построения уровней абстракций.

Исторический очерк развития языков программирования

1. **Эмбриональный** (**зарождение**) с 54 года до начала 60-х годов
2. **Бурный рост** – до начала 80-х годов
3. **Эволюционный** – до настоящего времени

Заметно, что длительность периодов увеличивается. В названных периодах можно выделить какие-то подпериоды.

Зарождение

Пример (Fortran)

Язык программирования Фортран 1954 – 1957гг – эпохальный язык программирования. FORTRAN – formula translator (транслятор формул).

Всегда служил примером – как не надо делать языки программирования. Но он используется до сих пор. Наиболее популярные версии Фортран77 и Фортран90 (иногда Фортран9x).

Фортран добился ровно тех целей, которые перед ним были поставлены. Ключевые цели: область научно-технических расчетов, область бизнес-приложений (коммерческих расчетов, бухгалтерия), как раз то, чем занимались люди в 50-х годах. Отличие – много входных данных, мало расчетов, а у бизнес-приложений наоборот, они занимались вводом входных данных, в основном.

Писать программы мог тот, кто непосредственно знаком с архитектурой. А вот модели писать могли и математики. Математик не понимает машинные коды, поэтому он объяснял, что ему нужно программисту с помощью блок-схем. Блок-схема состояла

из блоков (последовательных вычислений, разветвлений). Фортран переводил эти блок-схемы на машинный язык. До языка Фортран вопрос о мобильности (переносимости, portability) программ вообще не стоял. Как только появлялась новая машина, для нее сразу же появлялся компилятор языка Фортран. Появилась мобильность знаний, человек стал уже не привязан к конкретному компьютеру. А также компилятор Фортран наиболее оптимален по сравнению с другими языками.

В чем суть блок-схемы? Какие основные операторы Фортран?

`V = e`*; тип* `REAL` *(с плавающей точкой);*

`DOUBLE PRECISION` *(для особенно длинных чисел);*

понятие функции и процедуры.

Например:

```
RES = F(X)*CEXP(-k*x/z).
```

Попробуем записать на языке Си:
```
Plus(RES,Mult(F(X),CEXP(ToComplexInt(k),..
```

И.Г. – Я уже запутался, что дальше. Можете закончить эту запись до конца, и получить море удовольствия. Особенно при расстановке закрывающих скобок.

*Придется придумать несколько десятков функций умножения, деления, которые будут делать одно и то же – делать из целого – комплексное, и т.д. Это уже языковые ограничения. На языке C++ эти проблемы очень адекватно решаются с помощью шаблонов и перегрузки операций +, -, *, /. Добавляем конструктор преобразования – и все. Это достижение. Этим C++ хорош. В языке C эти проблемы не решаются. В языке C99 они решены с помощью способов расширения базиса.*

`IF(l)M1,M2,M3` *– единственный условный оператор.*
Еще был цикл, с известным числом повторений, для прохода по сетке.

```
DO 5 I=1,3
<операторы тела>

5 CONTINUE
```

40% операторов в языке Фортран имеет вид `I = I + 1.`
Математиков это не смущало. Зато объяснить этот язык было

«проще пареной репы». Еще были 4 формы оператора `goto`*, но их, с вашего позволения, я не буду писать (И. Г.).*

Формы ввода/вывода:

```
PRINT 100,RES
```

```
READ …
100 FORMAT …
```

Все, что есть в Фортране, мы до сих пор используем. Фортран – научное программирование. Главная его критика связана с тем, что из своей ниши он перелез в нишу языков индустриального программирования. Фортран очень интересный язык. Нужно ли описывать переменные? Как вы думаете? (Ответ, конечно же – нет). Массивы начинаются с 1, как всегда у математиков. Переменные `I, J, K, L, M, N` *– и все. Решили очень просто. Первые 6 пунктов перфокарты – метка. Потом специальное место для строки продолжения. А дальше соответственно запись. Все пробелы просто игнорировались везде, кроме строковой константы. Тем, кто делал сканер, так было проще. И поэтому в Fortran значимы первые шесть символов имени.*

В чем минусы языка Fortran?

Давайте поставим вместо пробела точку.

`DO5I=1.3` *– появилась новая переменная, цикл выполнился 1 раз. Ошибка при тестировании не выявилась, а стоила несколько миллионов долларов.*

Фортран сумел создать и занять особую языковую нишу. Языковую нишу занимает тот, кто делает это первым, по аналогии с экологической нишей: каждую экологическую нишу идеально занимает один вид. Побеждает тот, кто пришел туда первым. И в этом смысле программные системы ведут себя как динозавры. Программные системы не внедряются, они выживают.

После Фортрана появилось много языков, но они не вытесняют его. Все знают, что Фортран несовершенный, но Фортран подходит к нише, поэтому не вытесняется. Фортран имел оглушительный успех, трудно найти еще один такой успешный язык.

Затем появился Алгол-60, но он был уже вторым.

Пример (Алгол 60)

Алгол-60 был создан группой ученых. Алгол интересен тем, что был правильно спроектирован. Также впервые появились понятия блока,

области видимости, рекурсия, LIFO (Last In First Out) модель памяти и понятие стека в архитектуре для передачи параметров в процедуре.

Алгол 60 тоже был создан для численных расчетов, однако нишу свою не занял.

Бекус и Наур придумали форму, которая в точности соответствовала грамматике типа 2 по Хомскому – БНФ (нормальная форма Бекуса - Наура). Сейчас синтаксис всех языков описывается с их помощью.

В Алголе 60 существовали ключевые слова, но не было зафиксировано, как их выделять. 'BEGIN' или \BEGIN/ или begin. Это зависело от реализации. Ученые не задумывались о технологическом аспекте переносимости программ. Они заботились о переносимости знаний. О вводе-выводе они также не задумывались (его не было в стандарте языка). Язык оказался тяжело и неэффективно реализуем.

Быстродействие ЯП – усредненное соотношение времени выполнения программ, написанных на ЯП, ко времени выполнения программ, написанных на ассемблере машины. Например, для языка Фортран этот показатель равен 1,05. Для языка Алгол этот показатель находится в пределах 7-10. Заметим, что производительность программиста зависит от его таланта, и практически не зависит от конкретного языка.

Язык Алгол-60 – это язык для людей. Влияние Алгола в том, что это прежде всего язык обучения алгоритмизации, умению решать задачи. Сейчас существует много кодеров, которые учатся на Си. На основе Алгола созданы все базисные языки индустриального программирования.

Затем программисты начали заниматься теорией грамматик, формальных языков. Вирт, Кнут исследовали LR(1), нисходящий, восходящий анализ. Все ученые отметились в этой области. В начале 70-х годов изобрели Look – Ahead грамматики. Произошел скачок в этой области.

Следующий язык, принадлежащий этому периоду – Cobol.

Пример (COBOL)

COBOL (Common Business-Oriented Language).

Второй по распространенности язык программирования в США.

Особое внимание было уделено вводу/выводу по маске:

```
#.##
```

Типы данных языка: строка, целое число, дата. DDMMYY – Здесь возникло множество проблем: никто и не мог предположить, что

программы на COBOL'е доживут до 2000-х годов (и поэтому не заботились о решении «проблемы 2000»).. А они до сих пор существуют!

Вообще в 60-е годы создавалось множество языков программирования. Каждый уважающий себя ВЦ должен был иметь собственный ЯП. Военные, чтобы создать систему ракетной обороны создали свой язык. На самом деле ничего военного в языке не было. Он был основан на Алголе-68.

Для решения задач искусственного интеллекта (моделирования человеческого разума) был создан язык LISP в MIT – «язык обработки списков». Для этих задач язык Фортран был абсолютно не нужен. Для поиска не нужна арифметика. Основа LISP – символьные выражения (s-expression).

Бурный рост

Итак, первый язык программирования появился в 1957г., спустя 10 лет в США был проведен сравнительный анализ существующих ЯП. Там их было уже 400. В настоящее время количество ЯП точно никто не назовет, многие языки еще и вымирают. Очень примерно их около 10000, но может быть, число приближается к 20000. Далее мы будем изучать только наиболее значимые.

Люди начали придумывать собственные ЯП. А когда создается много ЯП – естественно, нет одного универсального. Придумывались свои языки программирования по двум причинам: нет хорошего языка программирования для некоторой предметной области, и не все языки были переносимы. Программисты стали пытаться придумать универсальный язык. Всего было 3 неудачные попытки создания идеального универсального ЯП. Почему неудачные? А посмотрите – вы программировали хотя бы на одном из них:

1. PL\I – 1964г.
2. Algol-68 – 1968г.
3. Ада – 1980 - 1983гг

Программисты брали разные элементы из разных ЯП, просто потому что понравились, комбинировали их, таким образом получались новые ЯП, более сложные. Первой над таким проектом начала работать IBM (проект IBM 360).

Пример (PL\1)

Язык PL\1 комбинировал языки Fortran (передача параметров по значению, списки ввода-вывода), Алгол 60 (блочная структура), Cobol. Язык создавался комитетом.

Существовало 2 компилятора PL/I – быстрая и отладочная версии. Это настолько сложный язык, что даже корректная программа могла на отладочной версии работать, а на быстрой – нет. Если выкинуть около 80% возможностей языка, то получился бы очень неплохой (Мейерс).

С появлением мини ЭВМ вымер язык PL/I. IDB вложила около одного миллиарда долларов в язык PL\1.

Причина провала языка PL\1 заключалась в несогласованных компромиссов, а «несогласованный компромисс всегда хуже любой из крайних точек зрения» (И.Г.).

Другой пример – Алгол 68.

Пример (Алгол 68)

Другая попытка – Algol 68. Рабочая группа создателей языка сменилась.

Впервые была описана семантика ЯП.

W – грамматика – совокупность двух правил класса 2:

$A \to \alpha$

A – могут быть составными. Протоправила используются для генерации частей метаправил. С помощью бесконечного числа правил можно было выразить произвольную семантику.

Алгол-68 не «пошел» из-за сложности описания, конструкций.

Основная идея – полная ортогональность (независимость) языковых конструкций.

Пусть даны конструкции K1, K2, K3, тогда в контекстах

K1 K2

K1 K3

Смысл K1 должен быть одним и тем же.

К примеру, Фортран был самым неортогональным ЯП.
`DO 5 I=1,~~N-1~~` *может стоять только* `N`*. Переменная ≠ выражение.*

К примеру, на Паскале: `for I := v1 to v2 do S`
`v1, v2` *– интегрального типа,* `S` *– произвольное.*

Заметим, что, вообще говоря, архитектура фон Неймана не ортогональна. В этой архитектуре есть понятие переменная, которая отражает состояние, выражение, и оператор, который имеет побочный эффект – отсюда следует ортогональность.

В языке Алгол 68 выражение без точки с запятой является выражением, а если поставить точку с запятой, то это уже оператор.

expr
expr;

Раз любое выражение может быть оператором, то и любой оператор может быть выражением. Это означает, что любой оператор должен иметь значение.

Примеры:

`v = e` – *имеет значение v*
v_1, v_2, v_3 – *имеет значение* v_3
`print(`v_1`)` – *имеет значение* v_1
`while {…}` – *неопределенное значение, если не выполняется.*

Любое выражение должно быть корректно как правостороннее. То, что должно быть слева, имеет на один уровень ссылок больше.

`const` – *ссылка уровня 0*
`ref int` – *ссылка уровня 1, и так далее*

В результате язык получился сложным и для восприятия, и по существу.

В 1969г. появился Паскаль. Вирт просил не ассоциировать свою фамилию с названием языка. Можно провести такую аналогию: Алгол-60 (Неандерталец) – Паскаль (Homo sapiens). Паскаль – язык для обучения программированию. В нем был использован принцип «Алгоритм + структура данных = программа». Принципы языка – простота, логичность, ортогональность (там, где это возможно).

Третья попытка создания «идеального языка программирования» – язык Ада.

Пример (Ада)

Ада – последняя попытка создать универсальный ЯП.

Как уже упоминалось в курсе, согласно исследованию Минобороны США, 50% затрат на создание программного продукта уходит на сопровождение. Это связано в том числе с многообразием языков. С точки зрения отдельного человека, менять компании – нормально, однако вновь пришедшему в компанию человеку бывает сложно разобраться.

В 1975г. только подрядчиками Пентагона использовалось 350 языков. Выделили 10 наиболее используемых. Затем установили соломенные, деревянные, стальные требования по надежности и эффективности. Критические задачи должны были решаться за гарантированное время.

Анализ требований установил, что будет достаточно одного языка. Был объявлен конкурс. Уже существовавшие языки PL\1, Алгол 68, Паскаль не подходили. Было 12 кандидатов на участие, в финал вышли 4 претендента, среди которых было два лидера: «красный» и «зеленый» языки программирования. В итоге победил «зеленый», и его назвали «Ада» в честь первого программиста Ады Лавлейс.

Все ПО создавалось только на ЯП Ада. С тех пор запретили создание подмножеств и надмножеств этого языка. Использовались только сертифицированные компиляторы. Ушло 3 года на создание компиляторов и переиздание стандарта. Первый советский компилятор языка Ада был создан в Ленинграде (конечно, он был несертифицированным). Скорость компиляции составляла примерно 3 строки в минуту.

Язык получился слишком сложным. В 1984 году у выпускников колледжа зарплата могла повыситься на 10 тысяч долларов в год, если они изучали Ада.

Ада – учебник технологий программирования.

Задачи, поставленные перед языком, решены не были.

История стандартов: Ада 83 – Ада 95 – Ада 2005.
В язык был встроен механизм внутренней многозадачности.

Что нового было в Ада? Почти ничего.

Рандеву, платформонезависимый механизм внутренней многозадачности (оказался неэффективным, аналогично многопоточности JVM).

У каждого программиста была своя библиотека интерфейсов.

Также в это время появились другие ЯП.

Модула2 – был создан Виртом в 1980г. Был лучше Паскаля, однако не «пошел».

PL/360 – язык Ассемблера, не зависящий от архитектуры.

BCPL – предок Си. АСТРА – создан в Советском Союзе. Эта языковая ниша уже была занята языком С. В языке С, по сравнению с его предшественником В, были введены типы данных.

Однако в С тип данных – это всего лишь шаблон размещения в памяти (и не более того). Язык С на большинстве архитектур обогнал Fortran.

Prolog был создан в 1971 году, Smalltalk в 1972.

Во время бурного роста получили развитие функциональная, логическая и объектная парадигмы языков программирования.

Функциональная парадигма

В 1978 году Тьюринговскую премию по программированию проучил Джон Бэкус. Он – автор языка программирования Algol, придумал БНФ – Бэкусову нормальную форму. По традиции, на церемонии вручения Тьюринговской премии награжденный должен прочитать лекцию. Джон Бэкус прочитал лекцию, которая называлась «Может ли программирование быть свободно от фон Неймановского стиля?».

В программировании в то время можно было наблюдать вялотекущий кризис, который был связан с борьбой с все нарастающей сложностью программного обеспечения. Джон Бэкус предложил ориентироваться на функциональное программирование. Он изобрел язык FP, который подходил для формального доказательства программ.

Формальное доказательство программы ставит целью показать, что спецификация программы тождественна самой программе. Были разработаны доказательства для программ, использующих Фон-Неймановскую парадигму.Однако, такие доказательства оказались очень сложными. Доказательство программы оказалось сложнее самого написания программы.

Для FP есть более простые программы.

В языке FP можно повысить эффективность программ с помощью эквивалентных преобразований.

Примеры других функциональных языков программирования:

1. ML
2. Miranda
3. Erland
4. Haskell

Логическая парадигма

В 1971 году появился язык программирования Prolog. Он был разработан для проекта автоматического перевода.

Идея языка: DCG-грамматика, грамматика определенных выражений.

Предложения грамматики строятся следующим образом:

Pronoun(case, A) => NP (case, S)

Этот пример означает утверждение: «Местоимение – это существительное». Здесь Pronoun – местоимение, NP – существительное, case – падеж, S – строка.

В языке используются дизъюнкты Хорна:

$P_1(S_1)$ ^$P_2(S_2)$ ^ $P_3(S_3)$ => $P_0(S_1+…+S_3)$

P_0 :- $P_1(S_1)$ ^ … ^ $P_n(S_n)$

ALPAC (Automatic Language Processing Advisory Committee): большую часть денег на перевод ушло на русско-английский и русско-французский перевод

FRAP

В 1985-1990 в Японии начались разработки нового поколения компьютеров с ИИ. Предполагалось вести общение с человеком на естественном языке. В большой степени это был маркетологический проект по привлечению инвестиций в соответствующую отрасль. В качестве языка был выбран язык Prolog как наиболее подходящий для обработки естественных языков. Проект обратил внимание на логическое программирование, как на одну из основных парадигм программирования.

Объектная парадигма

1972 Smalltalk

Внимание ученых было сосредоточено на абстрактных типах данных. Понятие абстрактного типа данных (АТД) – «надпарадигменное»: оно появляется не только в объектной парадигме. В то же время начинается математизация разработки программ.

1975 Object Pascal

Объектная парадигма изначально была рассматриваема как разновидность императивной парадигмы.

После появились Turbo Pascal, Delphi.

В 1979-1983 годах появились C++, Objective C.

Появились языки CLOS (расширение Lisp), Oz, Nemerle.

Эволюционный период развития языков программирования

Этот период характеризуется тем, что люди перестали изобретать универсальные языки программирования.

Эволюционный период продолжается с середины 90-х по настоящее время.

Для эволюционного периода наиболее характерна именно объектно-ориентированная парадигма.

В 1995 появился язык программирования Java, в 1999 появился язык C#. Эти языки во многом очень схожи, в каком-то смысле их можно назвать двумя реинкарнациями одного языка.

Пример (Java)

Java WORA (Write Once Run Anywhere) – эта технология имела целью создать переносимую реализацию языка программирования.

Ранее уже существовали аналогичные технологии.

UCSD – Калифорнийский университет в Сан-Диего. Там хотели сделать переносную реализацию для Pascal. Для этого придумали P-code, нотацию, аналогичную ПОЛИЗ. Транслятор переводил из Pascal в P-code, и для каждой архитектуры был написан интерпретатор с P-code (это проще сделать).

Аналогичная идея использовалась в виртуальной машине для языка Java. Её идея заключалась в том, что язык транслировался в промежуточный язык, называемый байт-кодом, а исполнялся он затем на JVM (Java Virtual Machine). JRE (Java runtime environment) включает в себя JVM и реализацию стандартной библиотеки. Существуют различные реализации JRE: Java SE (Java Standard Edition) – для индивидуального использования и малых предприятий, Java ME (Java Micro Edition) – для устройств, ограниченных в ресурсах, Java EE (Java Enterprise Edition) – для серверных платформ и задач средних и крупных предприятийю

Java-файлы могут иметь расширение .java (исходные тексты) или .class (скомпилированные классы). Java-приложениями могут быть как классические консольные приложения, так ijava-апплеты. Java-апплет – это архив с готовыми скомпилированными файлами в байт-коде, имеющий расширение .jar, который обычно выполняется в браузере. Впоследствии программисты на Java стали отдавать предпочтение «полновесным» кроссплатформенным программам, а апплеты стали писать на таких языках, как JavaScript и Adobe Flash (ранее: Macromedia Flash).

Создателем Java была фирма IBM. Язык Java понадобился тогда, когда в компании перешли от поставок к консалтинговым услугам. Как в Solaris, так и в Android существует встроенная JVM. Поэтому Java подходит для создания программ «для себя». Заметим, что для различных браузеров существуют разные реализации JVM. Например, для Internet Explorer существует две реализации, а в Mozilla используется реализация Sun.

Пример (C# и платформа .NET)

Принципы платформы .NET во многом схожи с принципами JRE. Существует спецификация CLI (Common Language Interface), которая стандартизирует промежуточный язык, виртуальную исполнительную систему, а также спкцификацию для типов и Виртуальная исполнительная система называется VES (Virtual Execution System). Промежуточный язык называется CIL (Common Intermediate Language). CLI включает стандартизацию для языка системы (CLS, Common Language Specification) и спецификацию для типов, а также систему представлений для различных значений (CTS, Common Type System). CLR, Common Language Runtime – одна из реализаций VES. Так же существует opensource-реализация Mono.

Отличия подхода Microsoft заключаются в следующем:

нет понятия виртуальной машины

промежуточный язык не является интерпретируемым, есть JIT (Just-In-Time) компилятор. Он транслирует код перед самым его

выполнением (и поэтому большие программы на .NET так долго загружаются).

Существует также версия JIT-компилятора от SUN, но она медленнее.

Формально C# не является .NET языком. Однако C# изначально разрабатывался как язык .NET. Другие .NET языки отстают от C#.

C# – это один из наиболее активно развивающихся языков программирования. Он сложнее, чем Java. В версии 2.0 в язык были добавлены элементы обобщенного программирования, в версии 4.0 – лямбда-функции и ленивые вычисления. Отметим, что существует также язык F# – экспериментальный язык для обкатывания концепций языков программирования.

Итак, Java и C# – это наиболее известные промышленные языки, возникшие в эволюционный период. Ранее Все известные промышленные языки относились к статическим языкам программирования. В 90-х и 2000-х шдах в промышленном программировании перешли к динамическим языкам программирования. Примеров таких языков много: Python, Perl, PHP, JavaScript, Ruby. Причина тому – в развитости WEB-технологий.

WEB-ТЕХНОЛОГИИ И ЯЗЫКИ ПРОГРАММИРОВАНИЯ

HTTP-сервер – это TCP/IP сервер, понимающий HTTP. Протокол его примерно таков: есть команда request – запросить, в которой либо get (получить любой ресурс), либо head (не исполняется), и есть команда respond – ответить. У HTTP-сервера нет состояния: «обработал-отдал-забыл».

URI (URL) – уникальный идентификатор ресурса. Например, пусть есть URL http://mycompany.ru/123/456 - это означает, что ресурс 456 лежит на сервере в папке /123.

Ресурс может быть как простым текстом, так и размеченным текстом. Тексты в HTML, картинки в jpeg и прочее предоставляют статические ресурсы. Поэтому с развитием WEB появилась идея генерировать динамический ресурс. Для этого существует технология CGI (Common Gateway Interface).

Рассмотрим исполняемый файл в URI: сервер делает fork() и запускает программу. URI выглядит так: _____?<параметры программы>. Вывод программы отправляется обратно клиенту. При таком способе программирования можно писать на любом языке (например, C), но это не очень-то удобно. Недостаток fork() в том, что это очень тяжелая операция. Поэтому решили использовать скрипты, которые в данном случае эффективнее. Для таких языков был сделан язык PHP (plain HTML processor).

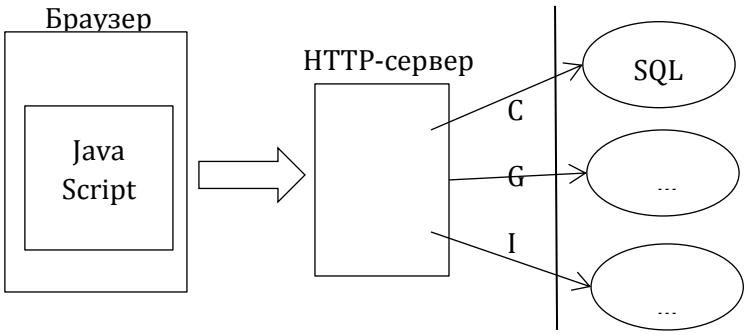
Для WEB-программирования выбираются интерпретируемые языки, потому что они безопаснее, и программировать на них быстрее.

Среди этих языков особняком стоит язык JavaScript

Пример (JavaScript)

JavaScript первоначально назывался LiveScript и предназначался для «оживления» WEB-страниц. Переименовали его маркетологи Sun. Он встраивается прямо в код HTML и исполняется на стороне клиента.

Схему взаимодействия клиента и WEB-сервера можно изобразить следующим образом:



Иногда технологии создания WEB-сервисов сокращают до LAMP, где L означает Linux, A означает Apache, M означает Mysql, а P может означать Perl, Python или PHP.

В JavaScript нет понятия класса, есть понятие прототип. Его иногда называют «ассемблер современного интернета». Существует язык для создания объектов JSON – Java Script Object Notation.

Данные, операции и связывание

Понятия данных и операций – ключевые понятия языков программирования. Формализация этих понятий означает формализацию языка программирования, и более точную формализацию тут придумать нельзя.

В течение нашего курса мы будем обращать внимание на то, что между понятиями данных и операций существует определенный дуализм.

Пример такого дуализма: длина строки символов. Это данные или операция? В языке TurboPascal длина строки хранится в первом элементе строки, поэтому она уже дана, ее не нужно вычислять, значит, это скорее данное. В языке C строка заканчивается символом конца строки, поэтому чтобы узнать ее длину, нужно пройти по всей строке в поиске символа конца строки – значит, в этом языке длина строки – это скорее операция.

Другой пример дуализма: понятие свойства объекта. Рассмотрим подробнее это понятие на примере языка Smalltalk.

Пример (Smalltalk)

В языке Smalltalk есть понятия класса и экземпляра. Переменные могут принадлежать как классу, так и экземпляру. Можно провести аналогию с языком C++ (статические члены-данные и члены-данные экземпляра класса). Переменные в Smalltalk недоступны извне, для доступа используются функции-члены set и get. Свойство в Smalltalk

извне является данным, а изнутри – объектом, потому что изнутри мы обращаемся к нему как к переменной, а извне – как к функции.

Заметим, что в общем случае get и set функции могут быть и нетривиальными. Например, функция установки позиции экрана Win32.SetWindowPos(…) осуществляет системный вызов.

Атрибуты объектов данных

У объектов данных могут быть некие атрибуты:

- имя
- значение
- тип
- время жизни
- область видимости
- область действия
- адрес (осмысленно только для императивной парадигмы)

Связывание атрибутов

Традиционно этому понятию уделяется мало внимания, хотя он столь же фундаментален

Связывание – это процесс установления соответствия между объектами и их атрибутами. Ключевое понятие – время связывания.

Существует три основных вида связывания: связывание о время выполнения, связывание во время трансляции и квазистатическое связывание.

1. Связывание во время выполнения

   Иначе оно называется динамическим связыванием.

   Может выполняться:

   - при входе в блок, однако этот случай больше относится к квазистатической связи.
   - в любой момент работы программы (new, malloc).
2. Связывание во время трансляции

   Может выполняться:

   - по выбору программиста (объявление)
   - по выбору транслятора
   - по выбору компоновщика

   Последние два случая относятся к статическому связыванию.

Пример: объявление `var l;`

Можно ведь и не объявлять переменную явно, а объявлять ее автоматически, при первом упоминании. Однако в этом случае возникает вопрос, а что считать первым упоминанием.

Пример: (C)

`static int i;` – *связывание выполняет транслятор либо компоновщик*

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

`i` *выбирается по правилам ассемблера.*

`static X a;` – *возникает вопрос, в какой момент вызывается конструктор? Такие случаи связывания называют квазистатическим связыванием.*

Во время работы программы можно выделить два этапа, «невидимых» для разработчика программы – пролог и эпилог. В стандартном прологе происходит, например, открытие стандартных потоков stdin, stdout и stderr.

Динамические языки – такие языки, в которых большинство ключевых связываний происходит динамически (в особенности связывание имени и типа). В статическом языке все наоборот, большинство ключевых связываний происходят статически.

Рассмотрим важный вид связывания – связывание определения переменной и ее типа.

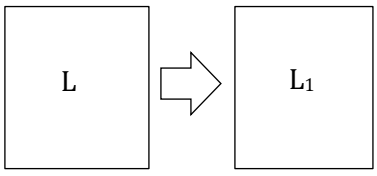
Связывание определения и типа

Оно может осуществляться в различные моменты:

- во время реализации языка (например, предельные системозависимые значения типов данных)
- во время разработки языка

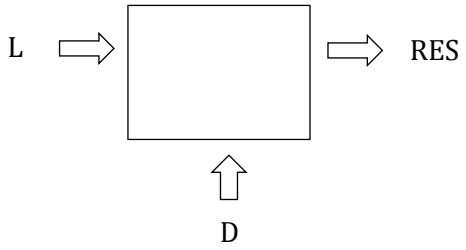
В последнем случае важно, о компилируемых или интерпретируемых языках мы говорим.

1. Компилируемые языки
   Код программы L транслируется в представление L_1.



   Частный случай L_1 – программа на языке ассемблера (очень низкоуровневой, то есть обладающий большой машинной зависимостью).
   В ассемблере может быть три вида предложений:
   - комментарий
   - директива
   - команда (то есть один-в-один машинная команда).

2. Интерпретируемые языки
   Код программы выполняется сразу.



   Здесь L – текст программы, D – входные данные, RES – результат.
   Чистый интерпретатор – это, например, интерпретатор BASIC для IBM 80-х. На современном BASIC можно писать макросы (VBA, Visual Basic for Applications).
   Не-чистый интерпретатор: исходный код сначала компилируется, а потом интерпретируется. Макросы VBA автоматически транслируются – таким образом, нельзя транслировать неверную команду (нельзя набрать endif, можно только EndIf). С некоторой версии можно включать режим компиляции.

С интерпретатором проще работать, чем с компилятором.

Из компилятора тоже можно сделать интерпретатор. Например, Lisp является интерпретатором, потому что в нем есть конструкция eval s. На C тоже можно написать eval с помощью fork(), exec(). Обратно, если язык интерпретируемый, то это не означает, что его нельзя компилировать.

Вернемся к понятию связывания типа и объекта. В интерпретируемых языках связывание типа и объекта происходит динамически. Рассмотрим три языка, на которых объясним отличия связывания типа и объекта у компилируемых и интерпретируемых языков.

Пример (C++, JavaScript, Visual Basic)

C++

```
int a, b;
a + b;
```

Здесь + означает операцию целого сложения, то есть связывание с операцией является статическим.

JavaScript

```
var a, b;
a + b
```

Это интерпретируемый язык, значит, связывание происходит динамически. `a + b` *– это что угодно.*

VisualBasic

```
DIM V[10]
```

В языке VisualBasic объявлять нужно только массивы.

```
S$
```
Второй символ однозначно определяет тип

Связывание объекта и множества значений

Раньше тип определяли как множество значений. Но это странно. Пример:

1	1	1	1	1	1	1	1

Число в ячейке может означать как -1, так и 255 – это зависит от точки зрения. Операции div и mod тоже выполняются по-разному над различными типами. Таким образом, тип данных характеризуется не только множеством значений, но и множеством операций.

Абстрактный тип данных определяет тип как набор операций. Иначе это понятие называют интерфейсом или сигнатурой.

Области видимости и область действия

Область видимости – область, в которой действительно объявление видимого имени.

Области видимости бывают потенциальными и непосредственными.

Пример (JavaScript)

```
{
if(x<0)
  a = 0; – здесь происходит так называемое всплытие имени
alert(a);
}
```

Существует так называемая потенциальная область видимости.

Пример (C++)

```
class M {
int a;
};
```
Вне класса M член a находится в потенциальной области видимости.

```
M::a;
```

Здесь применен оператор расширения области видимости ::

Область действия в классических языках программирования совпадает с областью видимости. Однако в некоторых случаях область действия может превышать область видимости. Примером может служить замыкание в функциональных языках. Если переменная появляется в замыкании, то она «живет» даже при выходе из блока.

Объекты данных характеризуются некоторым набором атрибутов: имя, значение, время жизни, область видимости, область действия

Три последние атрибута сильно связаны.

Время жизни – время, которое объект существует.

Связано с классом памяти(класс памяти можно определять из соображений времени жизни, а можно из соображений реализации).

Классы памяти

Статическая память

Время жизни = время работы программных конструкций, то есть существует, пока работает программа.

Пример – статические объекты в C++:

глобальные – создается перед началом работы программы,

локальные

Пример (C++)

`{ static X a }`– *создается при входе в блок, если в блок не зашли то нет и конструктора*

Заметим, что static может означать и объявление, и определение.

Смыслы ключевого слова static в C и C++:

1. Объект недоступен (локален в данном модуле), например `static void f();`
2. Объект принадлежит статическому классу памяти
    a. Глобальная переменная – локальна в данном файле
    b. Переменная внутри блока – относится к статическому классу памяти
3. (в C++) статический член (тесно связано с первыми двумя смыслами)

Типичный вопрос на устном экзамене: все смыслы ключевого слова virtual? Ответ: виртуальные функции и виртуальное наследование.

В SmallTalk классификация проще для понимания: есть класс, а есть экземпляры класса (instance).

В Smalltalk существует два вида переменных:

1. переменные класса – принадлежат классу (и никакому экзмепляру, в C++ статические),

2. переменные экземпляра – принадлежат экземпляру (нестатические).

Всё что связано со словом «статический» размещается в блоке статической памяти. Время жизни – от начала программы либо момента первого входа в блок, и до конца программы.

Квазистатическая память

Квазистатическая память связана с понятием блок: переменная связана с этим блоком (от момента объявления, точнее, прохода code flow через объявление, до выхода из блока).

Все характеристики почти как у статических объектов, только рассматривается не вся программа, а блок

Размещаются в стеке

Динамическая память

Динамическая память, по крайней мере, в момент жизни зависит от выполнения соответствующих процедур.

Существует специальная операция размещения объекта в динамической памяти (new).

Возникает вопрос: до какого времени живут объекты в динамической памяти? В этом отношении языки делятся на 2 класса

1. языки **с динамической сборкой мусора** – явное удаление объекта из динамической памяти(время жизни между new и delete)

2. языки **без динамической сборки мусора** (отсутствие операции delete – удаляет, когда на него ничего не ссылается (никому не нужен))

Проблемы языков с динамической сборкой мусора:

1. **Висячая ссылка** – ссылка, которая куда то указывает, но куда – непонятно, т.е. указывает на несуществующий объект

Пример (C++, висячая ссылка)

```
X * pa = new X();
X *pb = pa;
delete pa;
// Pb – висячая ссылка
Pb->fld = 0; // неопределенное поведение
```

---

2. Мусор (garbage) объект существует(операция delete не выполнена)

Пример (C++, мусор)

```
X *pa = new X();
Pa = new X();
```

Важным вопросом становится, а что лучше висячая ссылка или мусор?

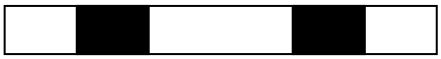
Висячую ссылку легче отловить, легче обнаружить, мусор обнаруживается ошибкой у пользователя, что очень плохо.

В языках без динамической сборки мусора(где нет delete), необходимы дополнительные накладные расходы(правда сейчас сборщики мусора работают параллельно выполнению программы, что несколько уменьшает накладные расходы).

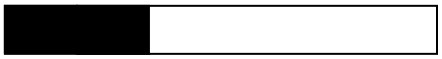
Одним из этапов работы сборщика мусора является дефрагментация памяти.

Она нужна для того ,чтобы можно было выделить нужный объём непрерывной памяти(у нас может быть достаточно свободной памяти, но не подряд идущей).

Промоделируем данную ситуацию в C: будем запрашивать realloc каждый раз чуть больше, чем выделили, получим такую картину:



Динамический сборщик мусора сдвигает всё к одному концу:



В C# можно обращаться к malloc, free и тд. Для этого существует понятие fixed-блоки, в которых можно обращаться к подпрограммам на C/C++.

Fixed (p = 0) – гарантируется,что значение p будет неизменным.

Это необходимо для явного обращения к C, C++ из C#.

Область видимости

Область видимости – это некоторая часть программы, в которой имя можно использовать.

Область видимости характеризует только именованные объекты.

С понятием область видимости связано понятие определяющего вхождения *(*имени объекта данных)

Если определяющих вхождений несколько, то имеет место перегрузка.

В современных языках программирования либо нельзя перегружать пользовательские имена, либо перегрузка допускается только для имён функций (иногда можно перегружать имя типа).

Пример (Name mangling)

```
class X{
...
void f(){};
...
}
```

Здесь f() непонятное имя (сложно отслеживать в отладчике – будет написано «"dfsdf" не определена», ошибку выдаст линкер

Использующих вхождений может быть много(хотя бы 1)

Хороший компилятор должен выдавать предупреждение – имя описано, но не используется

Статическая область видимости – область видимости, которая определяется во время трансляции. Динамическая область видимости – поиск вхождения имени в соответствии с исполнением программы.

Все процедурные языки(например, C, C++,Pascal) – имеют статическую область видимости(или с блочной структурой).

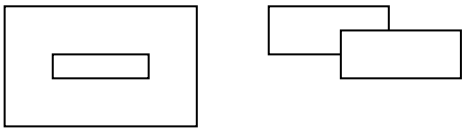
Свойства области видимости

1. Статическая

   Область видимости статическая и динамической быть не может (однако из этого правила есть исключения).

2. Вложенность

   Вложенность областей видимости (может быть как на рисунке слева, но не как на рисунке справа).



В языке С область видимости (блок, файл) – статические. Однако бывают и динамические области видимости

Пример (Паскаль)

```
Module M
var x;//x1
procedure P
var x;//x2
begin
P1; x:=0;
```

```
End P;
Procedure P1
Begin
X:=1;
End P1;
Begin
P;
P1;
End M;
```

Соответствие устанавливается следующим образом: в момент рассмотрения использующего вхождения, компилятор находит определяющее вхождение. Оно проверяется по последней объемлющей области видимости.

Анализ зависит от того, что определяется под областью видимости. Если статическая, то в приведенном примере в P x как x2, в P1 x как x1. Если динамическая, то поиск происходит в зависимости вложенности друг в друга.

Если приведенный выше пример изменить следующим образом:

```
M[x1

P[x2

P1[x2

…
```

то во второй раз в P1 будет ссылка на x1.

Заметим, что поиск обработчика исключений в C++ происходит аналогично поиску в случае динамической области видимости (проводится поиск по стеку вызовов).

Область действия

Область действия – область, в которой объект доступен и может использоваться.

Область видимости(Scope), область действия (extend) – иногда не совпадают.

Если термин «область видимости» применим к имени объекта, то область действия – к самому объекту.

Замыкание(closure) – такое понятие, в котором область действия шире, чем область видимости (характерно для функционального программирования). Бывает лексическое и динамическое.

Пример (Lisp)

`(defun f(x) ((+ x 1)))` – *здесь первый* `x` *– определяющее вхождение, второй* `x` *– использующее вхождение*

```
(sety y 5)
```

`(defun f(x) ((+ x y)))` – *здесь* `x` *– связанная переменная, а* `y` *– свободная переменная и входит в замыкание* `f`
`(sety a 3)`
`(f a)` *– результат 8*
`(sety y 10)`
`(f a)` *– зависит от вида связывания: если динамическое 13, иначе 8*

Другой пример – лямбда-функции в языке C#.

Пример (C#)

Лямбда-функция называется еще анонимная функция. От обычной функции она отличается типом

`(X) => X + 1` *– делегат*
`delegate z` *{ return z + 1 }*
`void P (int x) { return x + 1; }`
`(y) = y + x` *– здесь значение x будет захвачено*

Виртуальная машина языка программирования

Существуют компилируемые и интерпретируемые языки.

Если рассматривать процесс как «черный ящик», то компиляция и интерпретация – это одно и то же. В чем же разница?

Примеры языков: C –классический компилируемый язык, а Javascript интерпретируемый. Они отличаются степенью удобства и накладными расходами.

Интерпретатор – фиксированный, определяется семантикой языка, а компилятор можно оптимизировать.

Есть ли мера того, насколько язык компилируем или интерпретируем (то есть, что для конкретного языка подходит больше – компиляция или интерпретация?) Это Связывание тип ⇔объект данных. При статическом связывании легче генерировать программный код и выгоднее использовать компилятор. При динамическом связывании можно (и проще) использовать интерпретатор.

Большинство современных языков программирования основано на микроядре. В микроядре реализован свой язык программирования. Например, IA 32, x86-64 архитектуры имеют свою систему команд, инструкций (инструкция вызывает функцию на языке микроядра). Машинный код является интерпретирующимся аппаратно. Различные семейства процессоров AMD, Intel имеют одну и ту же систему команд, различающуюся микроядром. Из этого следует, что машинный код можно интерпретировать и программно, отсюда возникает понятие виртуальной машины.

Если можно интерпретировать даже машинный язык, то можно, вообще говоря, представить виртуальный компьютер, у которого машинным языком является наш язык программирования.

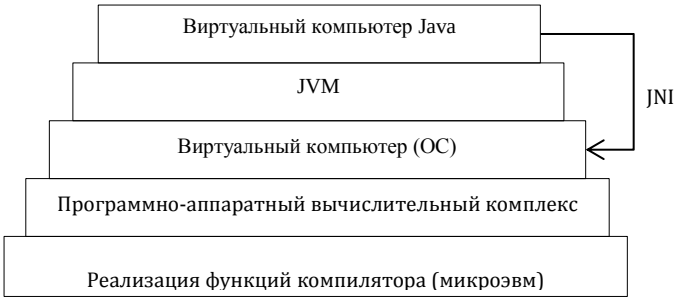
Пример

МИР2,3 – входной язык был язык аналитиков (Algol 60)

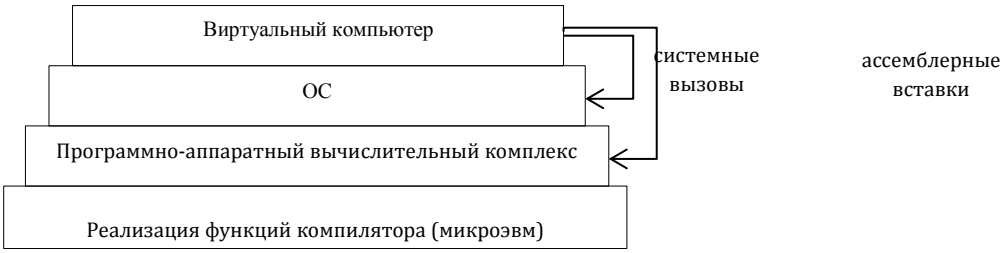
LISP machines – машинным языком был LISP

Процесс программирования можно рассматривать как программирование на иерархии виртуальных машин (на примере Java):

Рассмотрим подробнее виртуальный компьютер языка Java:



Для сравнения рассмотрим виртуальный компьютер языков C/C++:



Существовала аппаратная реализация JVM – picoJava.

Программы, которые не используют интерфейс ОС – более переносимы. JNI (Java Native Interface) – нужна, когда не хватает набора функций виртуального компьютера, для вызова функций операционной системы. Это обертка C/C++ интерфейса в Java-вызовы. В 50-х, 60-х API включал в основном управление внешними устройствами. Дописывая библиотеки на C/C++, мы дописываем компьютер ОС.

Основные понятия императивных языков программирования

Базис императивных языков программирования состоит из двух частей:

1. **Скалярный базис** – примитивные (простые) типы данных и операции

2. **Структурный базис** – составные типы данных, операторы.

Скалярный базис императивных ЯП

Простые типы данных

Простые типы данных делятся на следующие категории:

1. Арифметические
    a. целые – знаковые, беззнаковые;
    b. вещественные – плавающие, фиксированные
2. Символьные
3. Логические
4. Порядковые
    a. диапазон
    b. перечисления
5. Ссылки и указатели
6. Подпрограммные типы данных

Мы будем рассматривать первые 5 типов данных

В основном компьютеры вычисляют, так что начнем с рассмотрения арифметических типов данных

Арифметические типы данных

Целые типы данных

Будем рассматривать целые типы данных различных языков программирования в 5 аспектах:

1. универсальность
2. наличие/отсутствие беззнаковости
3. представление
4. надёжность
5. набор операций

1. Универсальность

В Стандартном Паскале 1 целый тип данных – int

В языке Java:

- byte (0 .. 255)
- short (-32768 .. 32768)

- int (-2^{31} .. 2^{31} - 1)
- long (-2^{63} .. 2^{63} -1)

В языке C++:

- char
- short int
- long long long

Кроме того, ко всем типам добавляются ключевые слова unsigned или signed.

В языке C#:

sbyte

short

int

long

byte

ushort

uint

ulong

Причем sizeof(T1)<sizeof(T2), где T1 выше T2

Зафиксировано представление или не зафиксировано? Если представление зафиксировано, то такие типы переносимые, но менее универсальные.

2. Наличие или отсутствие беззнаковости

Этот пункт тесно связан с надежностью.

Какую программу можно назвать надежной? Ту, в которой мало ошибок. Соответственно, ненадежная программа – та, в которой много ошибок. Ошибка – несоответствие работы программы ожиданию пользователя. Надежные конструкции – такие, которые не провоцирует ошибки.

Беззнаковый тип ненадежен, потому что в некоторых случаях он ошибочно интерпретируется как знаковый (при этом компилятор не выдает предупреждений), что приводит к ошибкам.

Возникает вопрос, а зачем тогда нужен беззнаковый ТД? Потому что есть address – всегда беззнаковый.

Java не содержит беззнаковых тд (кроме byte – но он не базовый).

Безопасное преобразование из T1 в T2:

```
T2(T1(T2))=i
```

3. Преобразования типов

Расширяющие преобразования целочисленных типов это безопасно, язык должен разрешать. Сужающие – опасно и ненадежно,язык должен запрещать. Преобразования из знакового в беззнаковое опасно

Неявное – которое автоматически вставляет компилятор

Явное – (T) expr

Единственная проблема связана с C и с C++ – любой арифметический тип может неявно преобразовываться.

Целочисленные типы в языке Модула 2

- INTEGER
- BYTE
- SHORTINT
- CARDINAL
- LONGINT

В C не запрещены преобразования unsigned-signed, потому что в первый стандарт их не включили, а впоследствии было уже написано множество программ, использующих такие преобразования. Ниша языка C – язык системного программирования.

Пример (C)

```
uint i;
for(i = N-1; i >= 0; --i) S – операция --i не определена
```

Рефакторинг – процесс согласованного изменения программы с целью преобразования вида программы(не функциональности). Обычно это делается с целью улучшения удобочитаемости и понятности кода.

В языке C# существует блок checked{}, в котором выполняется проверка на диапозон значений. По умолчанию стоит unchecked, но можно это изменить, если интересует надежность.

4. Надежность

   См. знаковость/беззнаковость.

5. Набор операций

+ - / * % (операции без побочных эффектов)

<, >, <=, >= ,==, !=

|, & ,^ ,~

<<, >>

++,--,*=

Семантика деления зависит от типа операндов

Хватит ли этого набора операций в языке Java? Нет, 2 сдвига вправо(SAL SAR SHL SHR). Не хватает SHL SHR – необходимо добавить беззнаковый сдвиг вправо.

Кратко напомним основные проблемы типов данных:

- Универсальность
- Надежность
- Представление (фиксировать или нет)

Ввести широкую номенклатуру типов данных и фиксировать представление – так можно легко решить вопросы универсальности и надежности.

Модула 2 – просто запрещает любые преобразования беззнакового типа в знаковый. Во многих ЯП наличие беззнаковости отражено тем, что есть базисный набор типов, для каждого изкоторых есть его без знаковый эквивалент. Например, в языке С:

char	unsigned char
short	unsigned short
int	unsigned int
long	unsigned long

В языке АДА была осуществлена концепция «тип-подтип». Был решен ряд важных проблем. Идея следующая: каждый объект данных принадлежит единственному типу данных.

Типы соответствия:

- множество значений
- множество операций

Возникла проблема надежности и ограниченности представления. Что будет при выходе за диапазон? C,C++ никак не решают эту проблему. В C# есть возможность контроля.

Пример: архитектура IA 32(придумана в 1987 и за 20 лет расширилась, но система команд осталась та же самой). ARM – меньше энергоемкость, проще. CISC – высокая энергоемкость, поедают много энергии (а точнее выделяют) и большая теплоотдача, значит ARM до сих пор поддерживаются и выпускаются. Современные архитектуры ничего не проверяют за выход за границы.

Создатели АДЫ поддерживали 3 требования:

- Надежность (все равно некий компромисс из-за того, что нужна скорость)
- Эффективность (системы в целом)
- Читабельность (не связано с надежностью и эффективностью)

Как же решили все эти проблемы?

Если хотим эффективность и надежность, нельзя использовать беззнаковость. Программист должен выбрать компромисс между типом или подтипом. Типы различны тогда и только

тогда когда различны их имена (именная эквивалентность типам). Различные типы данных несовместимы.

Пример (Ада)

В АДА есть объявление нового типа данных.

```
T
```

```
Type T1 is new T
```
– имена различны, значит и типы данных различны

```
X, y :T1
A, b :T
X + y
A + b
X + a
```
– *так писать нельзя*
```
Y + b
```
– *так писать нельзя*

Разные типы данных несовместимы.

Объявление может сопровождаться уточнением (ограничение диапазона).

```
T
```

```
Type T1 is new T Range (0.. MAXINT)
```

```
Type Natural is new INTEGER range 1..MAXINT
```

```
Type Positive is new INTEGER range 0..MAXINT
```

Любая попытка присвоить типу INTEGER – ошибка

```
X:INTEGER
```

```
Y:NATURAL
```

```
X:=y
```
– нельзя, но допустимо X:= INTEGER (y)

```
Y:= NATURAL (X);
```

Квазистатические преобразования: если компилятор знает текущее значение X, то он осуществит статическую проверку, а если нет, то сразу выдаёт сообщение об ошибке. Ада, Модула 2, стандартный Pascal содержат значительную часть квазистатических проверок.

Перечислимые типы

Языку Assembler противопоказан квазистатический контроль.

```
MOV AL, AX
```

Компилятор генерирует некоторый код – что противоречит сути Ассемблера.

```
a[-1] ~ *(a-1)
```

В C++ появился динамический контроль типа, кроме того, нет квазистатического контроля. Все современные языки с квазистатическим контролем таковы, что где можно вставить проверку, компилятор вставит. А где же тогда гибкость? Концепция типа слишком ограничительна. С одной стороны, хочется поддерживать контроль, а с другой стороны хотелось бы натуральные неотрицательные числа считать разновидностью целых. Если программисту нужна гибкость, то он может использовать концепцию подтипа – некоторое ограничение на множество значений базового типа. Совместимость сохраняется, подтипы одного типа относятся к одному типу и совместимы по операциям. Если они ограниченные, то обязательно происходит квазистатический контроль, конструкции вставляются компилятором.

```
Type Natural is INTEGER range 1..MAXINT
Type Positive is INTEGER range 0..MAXINT
Y:=X
X:=Y
```

Здесь компилятор вставит проверку

```
Z: POSITIVE
Z:=x
Y:=z
```

Должен быть контроль!

В диалектах паскаля (например,Turbo) существуют специальные опции и псевдокомментарии, позволяющие квазистатические проверки. Дональд Кнут иронизировал: продуктивную версию (для эффективного отключения квазистатической проверки) сравнивал с моряком, который носит спасательный пояс на суше, а в море выкидывает.

С одной стороны, эта концепция с точки зрения надежности дает возможность контролируемого поведения программы, исключения надо ловить и обрабатывать (максимальное количество проверок компилятор делает статически), имеется гибкая возможность управлять уровнем надежности и эффективности.

Надежность – запретить преобразования, разрешить преобразования. Но преобразования контролируемы статически или во время выполнения. Проблема беззнаковости решена – если боитесь, не используйте.

Программист имеет право указывать почти произвольный диапазон, а дело компилятора выбрать нужный диапазон представления. Сама по себе концепция оказалось очень мощной, обеспечивая нужную надежность или гибкость.

Другой подход – программист выбирает из готовых диапазонов. Проблема различных архитектур не стоит (Java только на 32 битной, либо 64).

На этом закончим рассмотрение целых типов данных.

Вещественные типы данных

Существуют следующие вещественные ТД:

- Плавающие
- Фиксированные

Плавающие есть во всех ЯП, а фиксированные не во всех. Если взять вообще все архитектуры, то представление вещественных типов данных отличается, что влияет на переносимость программ. Язык Ада, как и в случае с целыми типами данных, задает так называемый «внутрикомпиляторный» подход. Программист работает в терминах абстрактной структуры типов данных, а компилятор уже решает, как это преобразовать в машинное представление. Там и появилось разделение вещественного типа данных на плавающий и фиксированный. Вещественные от целых типов данных с точки зрения машины отличаются тем, что операции на вещественными типами данных не точны, так как в ограниченном количестве битов нельзя представить все возможные вещественные числа. Отсюда самая главная задача при работе с вещественными числами – управление точностью. В Аде плавающий тип данных – это тип данных, в котором зафиксирована точность, например,

```
type myreal is digits 10;
```

такое объявление говорит, что myreal – это вещественный тип данных, для которого компилятор должен подобрать такое представление, чтобы в мантиссе были точно представлены 10 знаков. Т.е. для плавающих типов данных выбрана, так называемая, модельная реализация, вернее модельные числа, которые представлены в виде M*2^p, где 0.1<=M<1. Это нормализованное представление. Здесь мантисса выбирается таким образом, чтобы точно представлять 10 десятичных разрядов. Таким образом получается множество модельных чисел, а компилятор должен выбрать такую реализацию на целевой архитектуре, чтобы нужный тип входил как подмножество в множество чисел реализации. Программист следил за точностью, а компилятор за нужным представлением. Как показала практика, такое представление оказалось несколько тяжеловато, поэтому в языке Ада было просто зафиксировано несколько базовых типов, которые предлагали использовать программистам.

В форме мантисса и порядок записываются следующим образом: S*M*B^p, где

- Знак – S
- Мантисса – M
- Порядок – p
- База – B

Хранятся они отдельно. Представление с плавающей точкой неоднозначно. Нормализация форма мантиссы 1/B <= M< 1 (B не обязательно 2, может быть, например, 16).

Какие основные проблемы возникают с вещественными числами?

Надежность (попадание или непопадание в диапазон операций – ситуация еще хуже чем с целыми – нет диапазона, здесь проблемы переполнения и потери точности). M разрядов под мантиссу и N под порядок. Можно хранить 2^(M+N).

Представление вещественных чисел относительно приближенно.

Пусть M = 24, -126 <= p <= 126, B=2, тогда какое минимальное целое число, которое может храниться? Минимальная мантисса ½, значит минимальное число 2^(-127) – меньше никак не получим.

Overflow/underflow зависят от нескольких параметров: и от B и от точности мантиссы. Существенна погрешность или не существенна, зависит от задачи (и алгоритма решения задачи – должны использоваться алгоритмы, нечувствительные к машинным погрешностям).

Проблема надежности и представления еще более актуальны. Один и тот же алгоритм на разных представлениях вещественных чисел по-разному считает.

На универсальность влияют такие понятия, как модельные и физические числа. Модельные – не зависят от представления, результаты были в пределах относительной обозначенной точности. Модельные числа приняты в языке Ада.

Пример (Ада)

```
Type T is digital N
```

N вычисляется статически, N десятичных цифр – точность представления.

```
Type FLOAT is digital 6;
```

Модельное число в Ада – B – количество бит для хранения мантиссы

B = [log2N]+1

-4B<=p<=4B

$S*M*2^p$

В модельных числах в качестве порядка используется 2.

½ представляется с нулевым значением порядка.

1 представляется точно – мантисса ½, а порядок 1.

p = 0: Δ (расстояние между «засечками», то есть между двумя соседними числами) равно 2^{-B}, 2^B значений мантиссы. От 1 до 2 порядок 1, $\Delta = 2^{-B+1}$.

Для 24 битной мантиссы погрешность 2^{102} между соседними числами (засечками)

P = -126: Δ = - 24-126= -150 (очень велико).

Засечки для модельных чисел должны соответствовать засечкам для физических. В реальности, практика программирования на Ада привела к тому, что концепция удобна, но нереальна. Для каждой реализации определяется несколько типов. Программист должен

сам выбрать тип чисел. Реальные алгоритмы не могут использовать эмуляцию вещественных вычислений.

С течением времени проблема представления утратила актуальность

IEEE 754 – стандарт на применения плавающих чисел (IEEE – некоммерческая организация, стандарты IEEE охотно принимаются индустрией). 802 стандарт – беспроводные сети

IEEE определяет 2 представления чисел: 32 бита и 64.

В Java float (соответствует 32битному представлению), double (64-битному). Float – для оптимизации хранения. Double – сначала все в double а потом уже в float (в C, C++).

По стандарту есть мантисса и порядок и 1 бит под знак:

$M*2^p$

Первый бит всегда 1 (из условия нормализации), поэтому под мантиссу 23 бита (хотя точность 24). Под порядок отводится 8 бит. В 64-битном под порядок 11 битов, а под мантиссу 52(причем точность 2^{-53}).

P+127

Старшее и младшее начального порядка зарезервированы.

Nan = NaN – not a number. 0 = 0 (точно). Существуют специальные представления для +inf и -inf.

Все современные процессоры удовлетворяют этому стандарту.

Итак, получаем следующие стандарты для чисел с плавающей точкой. float, double. long double – зависит от реализации.

Заметим, что представление с плавающей точкой не является универсальным. В технике (инженерии) встречаются разновидности вещественных чисел, которые неудобно представлять в виде с плавающей точкой. Например, рассмотрим АЦП – аналого-цифровой преобразователь. Сигнал преобразуется при определенной частоте дискретизации в регистр разрядностью n бит.

2^n - количество различных значений, которое может измерять, АЦП не выдают значений с плавающей точкой.

точность фиксирована = $(R – L) / (2^N – 1)$

Необходимо представление с фиксированной точкой!

Частая операция преобразования Фурье (выводит спектр)

```
Type FL is digital Range L.. R

Type F is delta H Range L.. R
```

Какие еще проблемные области требуют вычисления с фиксированной точностью? «Давайте я выдам приблизительно 28 копеек?» Торговля, банковские операции. Известное мошенничество: в одной из банковских систем программист сделал «закладочку», которая

всегда «обрубала» долю. (25.554 -> 25.55; .004 переходит на какой-то счет). Нашли мошенника только с помощью налоговых органов. Тенденция современных языков – упрощение базиса.

В Паскале Writeln, Readln – не суть процедуры, это замаскированные операторы ввода-вывода. Язык С порвал с этой традицией, все функции ввода/вывода – части стандартной библиотеки.

То, что было в языке уходило в стандартную библиотеку (ввод/вывод, например). Сейчас происходит обратный процесс. Представьте себе, стандартный ввод в языке Java. Это просто смешно. Библиотеки String, Object – как бы входят в стандартную библиотеку. Они являются частью языка. Есть специальный пакет «java.lang», являющийся частью компилятора.

В языке С вы можете все стандартные функции написать сами. Компилятор не обязан ничего знать о стандартной библиотеке. А в С++ сейчас (начиная с 90-х годов) компилятор уже знает, правда немного: std::exception. Например, bad_cast. Происходит интеграция компилятора с библиотекой. По стандарту std::exception должна существовать, её нельзя заменить.

Заканчивая обсуждение вещественных чисел, упомянем еще один тип – decimal. Он существует в C# и Basic. Попытка реализации фиксированного типа данных.

S - знак, C-коэффициент($0<c<2^{96}$), e-показатель степени 10 ($0<=e<=28$).

$Num = S*C*10^e$

Способ представления очень больших чисел.

IBM/360 -> PL/I -> SQL -> до сих пор остался в нескольких языках для совместимости с базами данных.

Тенденция современных языков программирования – делать крайне простой базис.

Возьмем, например, язык Python. Есть ли там беззнаковые числа? Нет, зачем. Без беззнаковых типов легче. Там есть 2 символьных типа, есть short, int, long, float, double. И все.

Тип Number – число. С точки зрения программиста – неэффективная штука. Проблемы преодолеваются путем огромной нагрузки на время выполнения.

Символьные типы данных

Что такое символ (литера)?

Она характеризуется:

- Названием
- Кодировкой (каждый символ в итоге представляется целым числом)
- Отображением

Charset (CS):

- Номенклатура (у каждого символа есть название)
- Кодировка (набор -> L..R)

SBCS – 0..255

DBCS – 0..65535

MBCS – 1..k

Понятие набора символов тесно связано с алфавитом.

CR(13), LF(10) – появились, когда появилась бумага.

DEL, BS – появились, когда появилась лента.

Сколько у нас символов в латинском алфавите? 26. Плюс еще 10 знаков препинания и т.п. Сколько нужно битов? 60? Это даже для китайцев многовато.

$2^7 = 128$, вот откуда это число. Откуда взялось понятие байт? Старший бит использовался для контроля.

0-127

128-255: для национальных алфавитов. ISO-LATIN1. Вся латиница, что за «железным занавесом». Кому не повезло? Полякам, прибалтам, сербо-хорватам. Компьютеры приравнивались к средству немассового уничтожения. Их можно было продавать союзникам. Турки были союзниками, но им не хватило места в ISO-LATIN1. А грекам хватило. Для старых компьютеров хватало однобайтовой системы кодировки.

Сколько всего было LATIN кодировок? 5.

В эти годы изобрели кодировку КОИ-8Р. Для чего изобрели транслит? Для международных телеграмм. 26->26, а остальные по-разному кодировались. В старшем бите стоит 1. 1A->A; 1Z->3. Есть 1 существенный (хотя, может быть, сейчас и не очень) недостаток – упорядочение.

Существовали языки, которые были в принципе single byte. Эти проблемы впервые появились в Японии. Сколько символов включает он? (JIS) 6000. Мощность charset'а очень велика. Для решения проблемы можно использовать 2-байтовый набор символов. Чем плохо? Нет совместимости. Первые программы писались на C или Ассемблере. 1 байт = 1 символ, строка кончается '\0'. Вот в этом была беда. В DBCS никак не лезет. Если бы использовалось «паскалевское» представление, то этой проблемы бы не было. Проблемы возникала при использовании языка C. Японцы придумали многобайтовую систему кодировки. Та же идея использовалась в кодах Хэмминга, это префиксные коды. Если в первом бите стоит 0 – это 1 байт. А если 1 – это 2 байта.

В 1991 году произошла революция. Ввели UNICODE. Есть еще такая штука UCS. (UCS-2 ISO 16046, UCS-4). Что это такое? Каждый набор символов стандартизует название и кодировку (не отображение).

0-127: ANSI-7

128 – 255: ISO-Latin1

В языке C++ появился новый тип данных - wchar_t (двухбайтный символ) – аналог unsigned short (В C99 _wchar_t).

Проблема UNICODE: совместимость с программами, которые работают с MBCS. Обнаружилась в Visual Studio. Почему? Тут ключевое слово не Visual Studio, а Microsoft. MS

DOS, Windows 95 использовали 8 битную кодировку. Windows NT использовали UNICODE изначально. Поэтому возникла некоторая несовместимость. Эти семейства слились только в XP (2002 год). 16-битные программы не работали под Windows 95.

Вторая проблема. 1040 А – 1298 Л (с крючком). Если взять английский текст, то половина символов там будет 0. Появился UTF – семейство преобразований формата. UNICODE текст на входе, а на выходе непрерывная последовательность из чего-то там. UTF8 – самая популярная кодировка. Тексты в UTF8 прекрасно обрабатывают MBCS программы.

UCS-2 -> UTF8

Рассмотрим правила преобразования чисел из UCS-2в UTF-8:

UCS-2	Диапазон	UTF-8
0|0XXXXXXX	0..127	0|XXXXXXX
00000YYY|YYXXXXXX	128..2047	110YYYYY|10XXXXXX
ZZZZ|YYYY|YYXXXXXX	2048..65535	1110ZZZZ|10YYYYYY|10XXXXXX
WWWZZ|ZZZZYYYY|YYXXXXXX	до 2^{21}	11110WWW|10ZZZZZZ|10YYYYYY|10XXXXXX

Максимальная длина 4 байта.

Каково определение символа в ЯП? В старых ЯП – char. wchar_t.

В C#, Java: char – это Unicode.

Ада (1983 год) : не было проблем с китайскими, японскими и прочими языками, и нужны были лишь различные кодировки английского языка (так как в Пентагоне другие не нужны). Поэтому в Аде было достаточно 8-битной кодировки ASCII ANSI.

Модуль языка Ада Standard поддерживал тип INTEGER (подтип универсального типа – диапазон зависит от реализации), а также тип CHARACTER – разновидность перечислимого типа данных. И все символы определелялись примерно так:

```
TYPE CHARACTER = (‘A’, ‘B’, ‘C’, …)
```

Символы преобразовывались в числа от 0 до 255. Кодировка ISO LATIN1

В языках C#, Java: char можно преобразовать в int.

Возникает проблема переполнения целого числа.

Проблема неоднобайтовых кодировок, основанных на фиксированной выборке:

big endian / little endian

```
#define LITTLE_ENDIAN
```

```
#define BIG_ENDIAN
```

– определяет порядок байт в слове.

Если есть некое целое > 1 байта, то возникает проблема.

Заметим, что UTF-8 – это не совсем кодировка, и в ней такой проблемы нет.

Пример: число 65534

```
X DW 0FFFEh

MOV AL, BYTE PTR X
```

X	X+1

1234 – арабские цифры (индийские): здесь 12 – старшие цифры, а 34 – младшие, то есть прямое представление (BIG ENDIAN). Если наоборот, то это LITTLE ENDIAN.

По соглашению, весь текст Unicode хранился в Big Endian.

TCP/IP пакет: Big Endian числа.

0xFFFE – Byte Mask (записывается в начало) – нужно, чтобы различать Big Endian и Little Endian.

Little 0xFEFF

Big 0xFFFE

Пример: UNIX

В Unix повсеместно используется «кодировка» UTF-8. Пустой файл в Unix сохраняется как 3 байта, потому что в начало нужно записать байтовую марку: 0xFFFE, которая в UTF-8 занимает 3 байта: 11101110/10111111/10111111.

UTF-8 – кодировка по умолчанию файлов XML.

Логические типы данных

Обычно логический тип данных называется bool.

Он принимает значения: true, false.

Определены следующие операции:

- !, not (логическое НЕ)
- &&, and (логические И)
- ||, or (логическое ИЛИ)
- ^, xor (логическое исключающее ИЛИ)
- еще иногда бывает операция тождественного сравнения ≡.

Не следует писать if(i), лучше писать if (i != 0).

Тип bool по умолчанию несовместим с обычными типами данных.

Перевод в true: обрезается до 1 (лишних разрядов не остается).

Проблема логического типа данных ленивость логических операций. Эта проблема связана с понятием порядка.

Рассмотрим выражение a + b + c. С точки зрения математика, порядок будет таким: ((a + b) + c). Но с точки зрения фон Неймановских языков, порядок будет таким: c + (a + b). А в функциональных языках вообще нет порядка вычисления выражений. a + b коммутативная операция, здесь нет большой проблемы. a - (b - c) – здесь уже существенно.

Свойства логического типа в фон-Неймановских языках:

1) не фиксируется порядок. Это делается ради более эффективной работы оптимизатора. Уменьшается поток данных между памятью и регистрами ЦП.
2) ленивость вычислений – может как быть, так и не быть. Ленивость вычислений

Ленивость вычислений – не вычислять логическое выражение, если его значение заранее известно.

Рассмотрим пример: `a++ || --l`. Здесь существенно, есть ли ленивые вычисления, и порядок тоже важен.

Рассмотрим другой простой пример – алгоритм линейного поиска.

```
while A[i] != X and i < N do
      i := i + 1
```

Один компилятор выдаст ошибку. Другой компилятор ошибку не выдаст, и если i < N есть истина, то отработает правильно, а если же i >= N, то программа упадет.

Возникает вопрос, что желать с этой проблемой.

Пример (Ада)

В языке Green, который потом назвали Ада, были специальные ленивые операции and then и or else. Порядок только слева направо. Предыдущий пример перепишется следующим образом:

```
while i < N and then A(i) /= do …
```

В C++ и Modula2 вычисления ленивые. Так же, как и во всех современных языках.

В Pascal не ленивые вычисления.

Все эти дискуссии свидетельствуют о низкоуровневости концепции.

Подводя итог разговора о целых типах данных, мы можем сказать, что набора типов в языках Java и C# хватает для любых архитектур, а также усложнение базиса ведет к неоправданному усложнению базиса языка.

Производные типы данных

Производные типы данных – это такие типы данных, которые опираются на простые типы данных.

Порядковые типы данных

Это диапазоны и перечисления.

Диапазоны

Диапазоны используются в языках Никлауса Вирта (Паскаль, Оберон, Оберон2, Модула, Модула2).

Статья 16 причин, по которым Pascal не является моим любимым языком программирования. Примечание: это верно строго для индустриального программирования. В качестве некоторых причин названы: отсутствие раздельной типизации; а также тот факт, что длина массива – это статический атрибут.

```
L..R of T
```

```
0..R of T
```

«Программирование на языке Модула2»

«Справочник по языку» – 40 страниц, то есть язык очень простой.

Принцип минимальности языковых конструкций – подразделяется на 2 принципа, принцип «чемоданчика» и принцип «сундучка».

Принцип «сундучка» гласит, что нужно включить в язык все полезное. Автор языка при этом должен объяснить, что и зачем включено в язык.

Принцип «чемоданчика» гласит, что нужно включить в язык все то, без чего нельзя обойтись. В языке программирования этот принцип называется еще принципом критичных технологических потребностей.

Итак, принцип минимальности языковых конструкций состоит в том, что языковые конструкции добавляются только тогда, когда функционирование языка без этого невозможно.

Например, вложенные модули были добавлены в Ада, но их не было в Модула2 – на практике они использовались не очень часто.

Пример (Оберон)

Оберон использует принцип «чемоданчика». Это очень простой объектно-ориентированный язык.

Типы данных в языке Оберон:

```
BYTE
SHORT INT
INTEGER
LONG INT
REAL
CHAR
```

Также были функции ORD и CHR.

В Оберон2 появились динамически связываемые процедуры. Компилятор языка Оберон2 занимал менее 1000 строк. В Оберон2 Вирт отказался от диапазонов. Главная концепция Оберона была в расширении типов, а диапазоны плохо поддерживают расширение. Массивы упрощаются:

```
INTEGER 0..N-1
```

```
ARRAY N OF T
```

В современных языках программирования массивы ведут себя именно так. Профессиональные программисты игнорируют эти возможности.

Перечислимые типы

Перечислимые типы были придуманы в языке Паскаль

```
Type T = (val1, …, valN);
```

На самом деле это тот же диапазон, в котором каждому значению присвоен уникальный элемент.

В языке Ада перечислимый тип задается как совокупность литералов, то есть любая переменная, представимая как символьная (не строковая) переменная, может быть литералом перечисления. Поэтому у нас есть некоторая «внутренняя» кодировка (способ присвоения значению литерала некоторого целого числа), и мы не привязаны ни к какой внешней кодировке.

Перечислимый тип данных в С – это короткий способ задать совокупность констант.

enum -> int

int -> enum

Приведения типов являются небезопасными (точнее, безопасно приводить из enum в int, но обратное преобразование небезопасно). Поэтому в языке С это просто наименование целочисленных констант.

Технические проблемы при использовании перечислимого типа:

1) проблема ввода-вывода

1995 Java диапазона нет, перечисления есть

2) наследование

f(T) => derived f(T)

3) неявный экспорт

Ада:

```
type T is (c1, …, cn)
```

```
type T is (w1, …, wN)
```

Константы в разных перечислимых типах данных могли совпадать.

```
type TrafficColor is (Red, Green, Blue)

type Color is (Red, Green, Blue)

Procedure P (X: Color) (Blue)

Procedure P (Y: TrafficColor) (Red)
```

P(Red) – что это за функция?

P(Color.Red) – однозначно можно определить

C# – перечисления 1) надежны 2) поддерживают компоненты

Почему в C# ввели перечисления? Потому что они хорошо сочетаются с языком IDL (Interface Definition Language – язык описания интерфейсов) и компонентным программированием. С точки зрения современных языков программирования, компонента – это класс. Причем класс рассматривается отдельно от способа генерации объектов класса. Для генерации объектов класса используется шаблон Фабрика. Фабрика – это класс, методы которого генерируют объекты связанных классов. В C# есть ключевое слово sealed class, в Java – final class – это классы, от которых нельзя наследовать.

Визуальное отображение компонент: toolbox с иконками/списком компонент. В окне свойств можно задавать имя и значение.

Пусть есть свойство C типа Color, оно может принимать значение из перечислимого типа.

Решение: можно разделить область видимости, сделать отдельную область видимости.

Второе решение: рефлексия – отображение свойств исходного текста в бинарном коде. C, C++, Asm – антирефлексивны.

Пример – имена функций. Атрибут: 1) рефлексия 2) указание компилятору

Атрибут [Flags] целочисленный тип данных. And, or.

Любой перчислимый тип данных совместим с константой 0 (true/false). Используется для инициализации начальных значений.

Java 2005 – много изменений по сравнению с предыдущими версиями языка. Перечень изменений – перечислимые типы данных. Неявно: константа целочисленного типа данных. byte, … -> традиционная интерпретация. Но у Color есть 4 интерпретации (имя перечислимого типа, имя пространства имен, имя класса, переменная-статический член). Пример: enum: enum Color { Red, Green, Blue } . Функции-члены, конструкторы. Можно реализовать toInt(). Расширение компилятора: генерирует класс специального вида.

Порядковые типы данных делятся на классы и примитивные. Выделяется ссылочный тип данных:

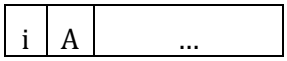
X a; – здесь ссылке присваивается некое нулевое значение.

a = new X();

Smalltalk: что угодно – это объект. Исследовательский язык программирования для генерации прототипов.

Где могут появляться значения: в контексте определения класса.

class X
int i;
A b;



Java: хранится ссылка.

C#: все должно быть объектным; неэффективно в работе с примитивными типами данных.

Упаковка, распаковка. Класс-оболочка – настоящий класс.

s.toString() (или s.ToString())

Чем удобно наличие класса-оболочки?

строка -> целое, можно настраивать

```
Parse {
      int Parse (…, <целое число>)
}
```

Указатели и ссылки

Сильная сторона и большая дыра.

Адрес перешел из языка Ассемблера в языки C и C++.

BCPL.

C отличался типизацией адресов. Тип в языке C – шаблон, которому должен следовать компилятор при распределении памяти.

[] – применимо к любому указателю.

X a[N] равносильно X* const a;

Пример (Fortran)

```
SUBROUTINE P(A, N)
DIMENSION A(N,N)
```

---

f(float* a, int n)

a[i*N + j] – неудобный для математиков способ работы с таблицей в C.

Указатель:

1) ссылка – средство для доступа к объекту без копирования
2) ссылка на анонимные объекты в динамической памяти.

Языки программирования подразделяются на «чистые» и «нечистые» в отношении указателей. В «нечистых» языках, подобных языку Ассемблера, существует абстракция адреса на уровне языка.

Пример (Ада 83)

```
type Pt is access T;

p1

type PT = DT
```

В работе с указателями возникает проблема «курицы или яйца».

```
type T is record

next ptr

…

type PT = pointer to T
```

В C и C++ эта проблема решается при помощи typedef.

```
typedef X* t;
```

Синоним X* теперь t, не определяет нового типа.

X* t – как сделать, чтобы компилятор был счастлив?

Пример (Ада)

```
type Pt is access;

type T is record;

X is PT;
```

Пример операций, которые можно делать с указателями в Ада:

```
X := new T;

P->(c)

P.all ~*p

p^
```

Что еще можно сделать с указателями? – больше ничего.

Проблемы, возникающие при работе с указателями:

1. Висячая ссылка

```
X* pa = new X();

X* pb = pa;

delete pbl;
```

Разыменование – это ошибка. Может упасть, а может и не упасть.

2. Мусор

Вторая проблема – мусор. В языке Ада нет операции delete (из соображений надежности). В нечистых языках сборка мусора невозможна.

Итак, указатель – языковая абстракция понятия адреса. С точки зрения указателей все ЯП делятся на те, в которых есть понятие указателя и в которых нет. Из тех ЯП, которые мы рассматриваем, указатели полностью отсутствуют в языке Java, и они отсутствуют в C# (указатели присутствуют в неуправляемой части кода, но мы ее рассматривать не будем). Понятие указателя заменено на понятие ссылки. Но опять же есть строгие и нестрогие языки.

1.Строгие (АДА, Стандарт Паскаля, Оберон)

Операции: == , !=, ↑ в строгих языках – только для ссылок.

Упрощается жизненный цикл объекта.

Пример (Оберон)

```
TYPE PT = POINTER TO T;
  X: PT;
  X.name
```

Явно указано, что существует динамическая сборка мусора.

В некоторых реализациях АДА рарешена динамическая сборка мусора

Пример (Ада)

```
type PT in access T;
  X : PT;
  X := new T;
```

В некоторых реализациях АДА рарешена динамическая сборка мусора

Как превратить строгий язык в нестрогий? Взятием адреса

```
P±i   //указатель разыменованный, говорить о котором невозможно,
P[i]  // но добавляется гибкость
```

```
~*(p+i)
```

Если в строгий язык добавить динамическую сборку и отказаться от понятия «указатель» (вместо этого использовать ссылку), то можно решить эту проблему

Java, C#, C++/CLI

В C# есть понятие указателя, но взятия адреса нет (можно использовать malloc,realloc…).

Есть конструкция вида `unsafe{…}`. Она необходима для

1) того, чтобы использовать код на C–подобных языках,

2) эффективности C.

Запускать `unsafe`-приложения можно не на всех системах.

Ссылка – аналог имени x.[]

Ссылки есть в следующих языках: C++,Java,Delphi,C#,SmallTalk…

В этих языках ОД соответствует ссылка. Это означает, что реализована объектно-референциальная модель. Ссылка указывает на объект в динамической памяти. Жизненный цикл объекта упрощается. Инициализация возможна только при помощи new().Также существует специальная операция освобождения объекта (free), либо сборка мусора.

Общий вид ссылки: `T& ref`

Инициализация ссылки в C++:

1) При помощи объекта в статической памяти
   Общий синтаксис: `T& ref = a;`

   ```
   int a = 0;
   in& refa = a;
   refa = 1;
   cout << a;
   ```
2) При помощи объекта из динамической памяти
   `T& ref = *pa;`
3) формальный параметр
4) член класса(список инициализации)

Генерация объекта:

- Обращение к менеджеру памяти
- В выделенной памяти размещается объект
- Вызов конструктора

Уничтожение объекта: Free (Delphi)

p.free – операция уничтожения

Возможен случай, когда две ссылки указывают на один и тот же объект:

```
X a;
A = new X();
```

```
X b= a;
//a,b один и тот же объект
```

Проблема эффективности char,int – накладно размещать ссылки в памяти.

В C++ с точки зрения реализации ссылку всегда можно отождествить с адресом

Передача параметров по ссылке:

```
P(var T:q)
```

Область ссылки ограничена формальным параметром

Java(и другие объектно-референциальные модели)

`X a;` ссылка не мнимая

`f(a)` объект можно изменить, ссылку - нет

Встроенной операции копирования объекта в таких языках нет.

C# интерфейс

IDisposable
Метод Dispose – уничтожает объект

Изменить состояние объекта можно только вызовом метода этого объекта. Есть метод Clone.

В Java есть ключевое слово final => нельзя ничего изменять

Smalltalk: всё есть объект

В C# Value Types (примитивные ТД, перечислимые, структурные)

Ада 83 – нет понятия процедурных ТД

Ада 95 – появилось понятие процедурных ТД

Ада 2005:

CALL P «ссылка на функцию»

Extern

Void f(…)

API привязаны к C,C++

Указатель на что угодно – не может быть

```
X : INTEGER
X : aliased INTEGER;
```
– от X можно взять адрес

```
Type PA is access all INTEGER
```
– нестрогий указатель
```
X: PA
X = new INTEGER;
```

```
Y: aliased INTEGER
X:= Y’ access;
```

Так писать нельзя:

```
A: P
A: new INTEGER
A: X’ access
```

Средства дополнительного контроля: у компилятора есть возможность контролировать

Расширение языка с сохранением предыдущих конструкций.

Вывод по примитивным типам данных:

В современных ЯП происходит тенденция упрощения базиса. Сокращение архитектур
Например: исчез диапазон

СТРУКТУРНЫЙ БАЗИС ИМПЕРАТИВНЫХ ЯП

СТРУКТУРНЫЕ ТИПЫ

Массивы

Рассмотрим такие структурный ТД, как одномерные и многомерные массивы и записи. В языке Фортран есть только массивы, что правда не критично – на базе массива можно смоделировать и запись.

Какой множество значений и операций представляет массив? Рассмотрим одномерный массив:

Пример (С)

```
T a[50] // покажем некрасивость C

T b[50];

a = b;  //запрещено
```

Эквивалентно

```
T* const a;  //+распределение памяти

T* const b;  //+распределение памяти

a = b;       //запрещено
```

Если внешний объект или формальный параметр, то распределять память не нужно!

```
T* pa;
```

```
pa = a; //можно
```

Если есть внешний объект или формальный параметр, то распределять память не нужно!

T* pa; pa = a; //можно

Операция сравнения:

В хороших языках программирования существуют операторы:

?= /= (АДА: если массивы одной длины и одинаковый тип – их можно сравнивать)

Операция индексрования []

Множество значений нужно определить, например, чтобы отличить массивы от коллекций (набор элементов с прямым доступом и последовательным доступом). Коллекция обобщает понятие массива. чем отличается коллекция от массива? В коллекцию можно добавить или изъять элемент. Массив – низкоуровневое понятие; последовательность элементов фиксированной длины. Массив непрерывен! Непрерывность – ключевое слово в понятии массива.

Атрибуты массива: базовый тип (T), длина (length), левая и правая границы (L..R), диапазон (Diap).

[]: A,Diap ->T&

```
X:T;
```

```
T:A;
```

```
Y[i] = X;
```

```
X = Y[i];
```
– и то, и другое допустимо

И так в любом языке (кроме C – в нем нет ссылок). Переходя от C к C++, в базисе изменились только ссылки (еще enum – тип, bool, но это косметика). Нельзя полноценно реализовать операцию индексирования без ссылок, поэтому Страуструп и ввел ссылки.

В массивах T и Diap – чисто статические во всех языках. В языках C, C++, Java, Python тип диапазона зафиксирован. В языке Оберон:

L ≡ 0, R ≡ Length -1 => `TYPE ARR = ARRAY N OF T;`

Length, L..R – статические (например, язык стандартный Pascal, C, особенно C89) или квазистатические.

В стандартном Паскале невозможно написать процедуру обработки массива (недемонстрационную). А почему? Атрибут Length статический и принадлежит ТД, а следовательно, этот атрибут наследуется всеми ОД этого типа. Это означает, что длину мы связываем с типом данных. Эта длина не может у нас поменяться. Как следствие в стандартном Паскале нельзя было написать процедуру скалярного произведения произвольных векторов. Полезная процедура, которая в ряде проблемных областей вычисляется очень часто. Процедуру такого рода написать никак нельзя потому, что мы

обязаны именовать соответствующие формальные параметры, приписывать им тип данных, это очевидно тип данных массив. Атрибут длина статический, следовательно, эта процедура будет считать скалярное произведение только для массивов фиксированной длины. Для массивов другого типа нужно переписывать соответствующую процедуру. Это самый большой недостаток языка Паскаль. Ведь массив – это базисная структура для любых видов программирования.

```
Type Arr = array[0..N-1] of char;
```

`1..N` – никак, другой тип =>ни одной полезной обработки процедуры не описать.

А в С нет типа массива, поэтому нет этого недостатка. Передаём указатель и длину (как второй параметр).

```
var X,Y,Z:Arr;
```

`X[i], Y[i], Z[i]` можно вычислить оптимальным образом, т.к. известно распределение памяти.

`X[i]` ≡ адрес X + sizeof(T) + (i-L)

`X[RX]` транслируется в любом языке Ассемблера

В Аде, Модуле-2 и Обероне все распределение памяти под массивы – статическое, а использование – квазистатическое.

Пример (Модула2):

```
TYPE ARR = ARAY [L..R] OF T;
```

Тип и значение тих переменных должны быть известны во время трансляции.

```
VAR X,Y: ARR;
```

Появляется понятие – открытый массив – только физический параметр некой процедуры или функции.

```
ARRAY OF T;
PROCEDURE SCAL (VAR X,Y: ARRAY OF REAL) : REAL
BEGIN
  CNT := 0;
  FOR I:=0 TO HIGH(X) STEP 1 DO
       CNT := CNT + X[I]*Y[I]
  END
  RETURN CNT
END SCAL
```

Фактическим параметром может быть любой массив типа REAL. HIGH(X) выдает максимальное значение диапазона.

```
VAR A: ARRAY[0..N-1] OF REAL

    B: ARRAY[1..N] OF REAL
```

SCAL(A,B); – работает.

Есть еще язык Модула-3 с исключениями, динамическим распределением памяти и многим другим. В Фортране начиналось с 1 диапазона (математический). Начиная с языка С и во всех современных языках диапазон начинается с 0. В Модуле-2 и Обероне отклонение от статики только в формальных параметрах. АДА же более выразительна.

Тип/подтип

Понятие неограниченного ТД:

Неограниченные ТД: `Type BaseArr is Array INTEGER range <> of T;`

Ограниченные ТД: `Type Arr is Array INTEGER range 0..N-1 of T;`

У ограниченного типа все характеристики статические. У неограниченного типа длина вообще не определена. Неограниченный тип нужен для введения подтипов.

```
X,Y: BASE ARR range 0..N-1;
C:Arr;
SUBTYPE BA1 is BaseArr range 0..N-1
SUBTYPE BA2 is BaseArr range 1..N

D:BaseArr – нельзя, будет ошибка
A: BA1; переменная типа BaseArr
B: BA2; переменная типа BaseArr
```

Ba1,BA2 – подтип

X := Y – корректно

X := A – корректно

X := B – корректно, т.к. длина совпадает N-1 = N – 1

X := C – нельзя, разные типы данных

C := X – нельзя, разные типы данных

A’LENGTH – длина

A’LEFT – левая граница

A’RIGHT – правая граница

A’RANGE – диапазон

Пример (Ада)

```
Type A is array integer range <> of float;
Function "*"(A,B:Arr) return float is
  i:integer;
begin
  if A'length != B'LENGTH then
       raise Index_out_of_Range;
  endif;
  for i in A'RANGE loop
       res := res +A(i)*B(i);
  end loop;
  return res;
end "*";
```

Как происходит в современных ЯП (например, Java):

`T`

`int`

`0..N-1` N – квазистатический атрибут

(как только массив порожден, то N – статический)

`T[] a;` `a` – ссылка на массив элементов типа T

Длина не является статическим атрибутом. Конкретно массив появляется при выполнении `new: a = new T[N];` => уже статический

`a.Length` (на C# с большой буквы)

Внутри коллекции всегда есть метод `toArray()`. Коллекции вообще говоря некоторые прикладные классы, имеющие набор интерфейсов.

Старые ЯП имеют «нашлёпку» - динамические массивы (например, языки АДА, С99). Это на самом деле не динамические массивы, т.к. память для них отводится на стеке, они могут быть только локальными. Так что правильней их было бы назвать квазистатическими или локальными.

Пример (Ада):

```
procedure P(N:integer) is
A:array range 1..N of real;
Begin
<Обычная работа с A>
endP
```

Пример (С)

```
void f(float* p,int n){
  float A[n];
  …
}
```

При этом описывать такие массивы, как аргументы нельзя. Эти массивы менее удобны – даже адрес базы надо вычислять через суммирование.

Delphi – гибридный язык. С одной стороны он похож на Модулу2, однако имеет широкую объектную модель. Здесь специально введено понятие динамического массива. Setlength ведет себя как realloc. Диапазон от 0 до N-1.

Многомерные массивы

C/C++:

```
T a[N][M]; // выделяется N*M*sizeof(T) памяти
```

N строк, M столбцов

Пример (Java, C#)

```
int []b;
int [][]a;
b = new int[10];
a = new int[][20];//каждый из 20 элементов ссылка
for (int i = 0; i<20; i++)
  a[i] = new int [i+1];
```

В Java все массивы ступенчатые.

```
T [,] b = new T[20,30]
```

`b[i,j]  b[i][j]` – *форма обращения не важна*

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

– *то есть* `{{1,2,3},{1,2,3},{1,2,3}}`

Без «танцев с бубном» можно работать с многомерными массивами в функциях (в отличие от C/C++).

В C# добавлено [,] для работы со старыми массивами. В Java такого нет, а если хотите использовать, то сами пишите отдельный класс.

Записи

Массив: D* … * D(n раз)

Запись: D1*…*Dn

Операции с записями

1. Присваивание
2. Операция точка(.) a.name

Записи впервые предложил (появились уже в COBOL'е, но были очень кривыми) Чарльз Хаар на конференции НАТО, затем перешло в Паскаль.

Используется `fieldname`, значит необходимо смещение.

Записи обобщены понятием класса. В современных языках программирования их почти не осталось. Зато записи есть, например, в.C#

Рассмотрим точки – у них есть координаты X,Y

```
Point []P;
X = new Point[1000];
foreach (point p in X)
     p = new int(0,0)
```

Добавлены структуры из соображения эффективности:

```
Struct X{
     <описания полей и методов>
}
```

Структуры – обрезанные классы, их нельзя наследовать, они ни от кого не наследуются.

Для всех типов значений существует класс Wrapper – обёртка (оболочка).

Unboxing (распаковка), AutoBoxing (неявная упаковка), Boxing (упаковка).

Объединение типов

Тип – роль данных. Единственна ли эта роль? Почему это важно? Роль определяет поведение объекта. Поведение – набор методов. Единственная роль – единственный тип данных.

Особого смысла придумывать новые языки – нет. 1980г. Ада – вершина процесса придумывания.

Любой объект данных принадлежит единственному типу данных. Один и тот же объект данных может выступать в нескольких ролях.

Объектные языки избавили от аксиомы единственности типа данных. Пример: построение интерфейса пользователя (ИП). До объектных языков не было адекватных ИП. Развитие ОЯП и ИП шло параллельно.

WIMP (Window Icon Mouse Pointer)

Общие принципы: 1970г., Xerox PARC.

Николаус Вирт был там и проникся восхищением, немедленно начал изобретать свою рабочую станцию: Модула 2, Оберон. В 1972г – появился SmallTalk.

Переход системы из состояния в состояние происходит скачкообразно, это можно назвать событием. С окном должен быть связан тип данных. События: клавиатура, мышь, таймер. Для события должен быть обработчик.

Клавиатура: номер клавиши (скан-код), нажата/не нажата, клавиши-индикаторы (Ctrl).

Общее у событий – момент системного времени, больше ничего общего у событий нет. Для решения этих задач служит объединение типов.

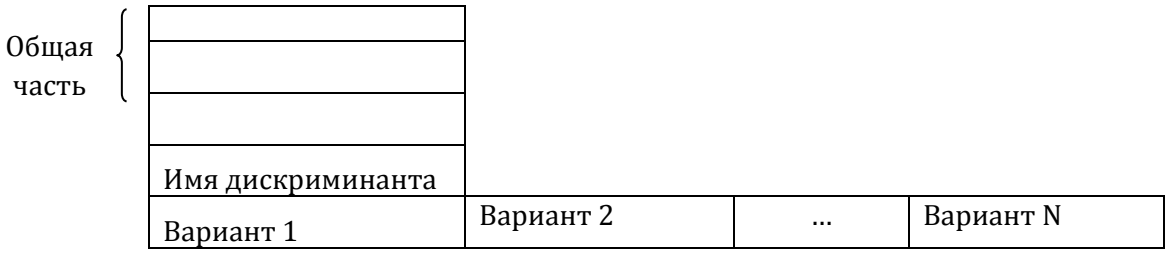
Язык С: union, язык Паскаль/Модула 2: записи с вариантами.

Общий синтаксис:

```
record
      Общая часть
      Вариантная часть
end
case Дискриминант of
      M1: (вариант 1);
      M2: (вариант 2);
      …
      Mn: (вариант n);
      else: (вариант n+1)
end;
```

Дискриминант может быть 2ух видов: имя:тип или тип.

1) Некое поле внутри записи.



   В Модула 2: метки M1,…,Mn. Также есть else – часть (синтаксический сахар).
   Такие события называются дискриминированными. Применительно к событиям дискриминантом может служить тип события.

2) тип

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

   Версия на стандартном Паскале: не мобильна (неразмеченное объединение).

   В языке Си разрешены только неразмеченные объединения.

Программисты на Windows знают VARIANT – запись.

```
VARIANT
Union {
      Struct {
            short vt;
            Bool vt_bool;
      } vt_bool;
      Struct {
            Short vt;
            Pword vt_IM;
      } vt_IM;
}
```

В языке Ада запись с вариантами имеет неограниченный тип.

```
Type Event (et:EventType) is
      record
            Общая часть
            case et:EventType of
                  when KBEvent => … |
                  when MousePressEvent => … |
                  …
                  when others => …
      end record;
```

X:Event – если это не параметр процедуры => ошибка.

X:Event(KBEvent);

new Event(MousePressEvent);

Отвести под переменную память можно, только указав дискриминант.

В любом языке 60-70 годов есть записи с вариантами, а в ООП появилось наследование, заменяющее их. Зачем мы изучаем записи с вариантами? Языки умирают, но приемы программирования остаются. Например, диспетчеризация (механизм динамического выбора). В языке C++ - механизм виртуальных функций. В языке Java все диспетчеризуется.

Недостатки:

1) Можем ошибочно сменить поле типа
2) Большой switch – плохо в C++. Плохой стиль. Причина в важности устойчивости изменений программного продукта. Сопровождать ООП очень удобно, а вот просматривать большой код и исправлять switch'и – нет.

Поэтому в ООП должна отсутствовать диспетчеризация по полю типа. В современных программах нужно этого избегать.

Строки (String)

Область значений – произвольная последовательность символов.

Массив – непрерывная последовательность данных, строка же не обязана быть непрерывной (это зависит от реализации). Например, арабский текст. Его беда в том, что он содержит различные элементы и всегда разделен на сегменты, каждый из которых необходимо читать по-своему. Поэтому реализовать хранение такого текста с помощью непрерывной последовательности данных просто невозможно.

В Delphi, C#, Java => string – массив.

Зачем нужен тип string, если есть char []s? Зачем нужны эти 2 типа: Vector<T>, basic_string<T>? Чем отличается библиотечный конструктор от базисного? Библиотечный не зависит от транслятора => его можно отрезать. String маскируется под библиотечный, но его семантика зашита в транслятор. String – тип из базиса.

C++ настолько богатый язык, что с помощью библиотек можно построить string, неуступающий встроенным string из Java и C#.

В STL есть Vector<T>, basic_string<T>, имеющие схожие операции push_back(), size(). Но эти типы отличаются конкатенацией, а также операцией индексирования. Операция индексирования возвращает ссылку для vector<T>, но string[] возвращает char. String является неизменяемым в большинстве индустриальных ЯП.

```
string s=””; string line;
while ((line = Console.ReadLine()) !=  null)
      s += line;
```

Неэффективный код! Куча времени будет уходить на дефрагментацию.

```
string line; var sb = new StringBuilder();
shile ((line = Console.ReadLine()) !=  null)
      sb.Append(line);
string s = sb.ToString();
```

Существенно более эффективно!

Сложность в распределении памяти под строки состоит в том, что алгоритм распределения должен быть различным для маленьких и больших строк.

Другие структурные базисные типы

Ранее мы рассмотрели: массивы, записи, строки (динамические).

Паскаль (1970г.): существовала куча типов (множества, таблицы, файлы), но строк среди них не было (динамических).

Множества удобно использовать, например, в алгоритме «Решето Эратосфена» (поиска простых чисел). Но минус множеств Паскаля – их маленький размер (до 256).

```
set of T;  |T|<=N
BITSET = SET OF [0..N-1];  в Модула 2
```

В языке Оберон нет множеств вида set of T, т.к. нет типа, который мог послужить в качестве базового. Поэтому там остался просто set.

Таблица – это обобщение множества (в общем случае).

Вывод: в современных ЯП базис упростился.

Операторный базис

Оператор – это нечто, что обладает побочным эффектом. Машинные команды: ввода-вывода, пересылки, арифметико-логические, переходы. Ввод-вывод: прерогатива библиотеки в современных ЯП.

Пересылки => оператор присваивания (для процедурных ЯП)

Арифметико-логические => выражение (в любом ЯП)

Переходы => операторы управления

Оператор присваивания – самый тяжелый, с точки зрения трансляции, оператор.

1967г. – "GoTo statements considered harmful" Дейкстра. Переломная статья в программировании. До 1967г. целью создателей ЯП было «понапихать» туда побольше всего. Например, в языке Фортран существовало 4 оператора перехода, чтобы дать программисту возможность для реализации своих намерений. Существовал термин – DS-programs (dish of spaghetti): программы были похожи на спагетти (нарисованные карандашами переходы с одной строки на другую, переплетающиеся друг с другом).

Оказалось, что хорошие программы содержат ограниченное число структур перехода.

Фактически, статья Дейкстры положила начало новому направлению – структурированное программирование (структурное программирование). Структурное программирование – процесс разделения программы на уровни абстракции, а затем их последующая детализация. С этой точки зрения оператор GoTo – запрещается.

В 1969г. Вирт придумал ЯП, который выразил идеи структурного программирования, в каждом следующем языке встречаются такие же структуры.

Любой последовательный алгоритм можно записать с помощью:

1) `s1;s2` – последовательность
2) `while B do S`

Основная идея Дейкстры состояла в том, чтобы писать программы сразу же в терминах этих структур: блок, развилка и циклы.

В языках появляется составной оператор – частный случай блока. Он применяется там, где по синтаксису подходит один оператор, а требуется несколько. Отличие от блока состоит в том, что в блоке могут появляться локальные переменные.

`begin s1;…sn; end` – составной оператор

`{объявление; операторы}` – блок

Блок – область видимости; тело функции

Выйти из блока тела функции нельзя.

```
void f()
{
      int i;
      …
      if (a>b) {
```

```
            double i;
            …
      }
      …
}
```

В этом примере происходит скрытие. В языке C# компилятор в таких случаях выдает предупреждение – вынуждает поставить заглушку (new).

Блок является не областью видимости, а средством группировки операторов.

`if B1 then if B2 then s1 else s2` – плохо (специально записано в одну строку)

Существует проблема замыкания (непонятно какой else к какому if относится) – это синтаксическая неоднозначность. Проблема была решена так, что else отнесется к ближайшему if, иначе необходимы структурные скобки.

В языке shell, например, отсутствует понятие составной оператор.

Языки Ада, Модула-2, Оберон используют схему явного закрытия операторов.

```
if B1 then
      S11; S12; S13;
end if
if B then
      S1
else
      S2
nd
```

На языке Си грамотно писать:

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

Причины этого: во-первых, пожалейте экран; во-вторых, наглядность.

В языке Python отступы являются правилом. Без их правильного использования нельзя написать синтаксически корректную программу.

Структурные операторы

Структурные операторы – это такие операторы, которые содержат внутри себя другие операторы:

- Развилки
- Циклы
- Блоки или составные операторы

Составной оператор

Составной оператор – это последовательность операторов, объединенных в единую конструкцию.

Пример (Паскаль)

```
begin S1 … Sn end
```

Примеры языков:

- Стандартный Паскаль,
- «паскалоиды» (например, Delphi)
- *Модула 2*
- *Оберон*
- *Ада*

Последние 3 языка особенны в отношении понятия составного оператора: в них также есть понятие блока.

Почему же они особенны? Ведь в стандартном паскале тоже есть понятие блока.

Блок и составной оператор

Что такое блок и чем он отличается от составного оператора?

Блок – это область видимости. В блоке могут быть объявления.

Блок состоит из двух частей:

1. Объявления локальных переменных
2. Операторы

Понятие блока есть во всех языках. Более того, если рассматривать языки, опирающиеся на Алгол 60 (а это почти все современные языки), там не было составного оператора, но был блок.

Языки C/C++ наследуют эту идеологию.

Пример (C, C++)

```
{ объявления; операторы }
```

Область видимости и область действия (как правило, они совпадают, но это не так для функциональных и некоторых современных языков) – это блок.

Подчеркнем, что понимание разницы между блоком и составным оператором важно для принятия тех или иных решений о выборе языка решений.

В Си-подобных языках: объявление блока является слишком жестким.

В некоторых языках (таких, как Java и JavaScript) исчезает разница между объявлением и оператором. Впервые это явление возникло в языке C++. Здесь конструктор (является и определением, и оператором)

Все эти языки поддерживают правило: опережающее объявление объектов.

Правило опережающего объявления объектов

Правило опережающего объявления объектов имеет две цели.

1. Утилитарная. Языки, поддерживающие это правило, проще транслировать. Можем употреблять некоторое имя до его объявления. Как правило, большинство трансляторов справляются с этим легко: на первом этапе выполняется лексический анализ, включающий свертку имен; на втором этапе выполняется синтаксический анализ, причем используемое имя уже известно.
2. Удобство чтения. В стандарте языка Ада об этом было четко сказано. Сейчас от этого правила отступили, но не слишком сильно.

Обратим внимание, что в современных различие между блоком и составным оператором не слишком выделяется.

В стандартном Паскале есть и блок, и составной оператор (заметим, что begin…end – составной оператор в чистом виде), где объявления размещать нельзя. Блок появляется в процедуре. Процедура состоит из заголовок и тело, где тело – это блок. Аналогично – для главной программы, которая тоже состоит из заголовка и тела-блока.

В этих языках обычно принята следующая синтаксическая структура:

```
объявление составной оператор

объявление begin последовательность операторов end
```

В Ада добавляется ключевое слово declare.

Блок служит только как тело соответствующих процедур.

Языки с явным терминатором

Вернемся к вопросу: в синтаксическая разница между стандартным Паскалем и языками Модула, Оберон, Ада?

В этих 3 языках поддерживается правило: любой структурный оператор должен иметь явный терминатор (завершитель, окончание).

Многие языки (Алгол68 и позднее) используют это правило. Там каждый структурный оператор должен иметь завершитель.

В предыдущих примерах в качестве терминатора выступало ключевое слово end.

По правилам Ада терминатор можно дополнить необязательным описанием синтаксической конструкции, которая завершается: end if, end case и так далее. Компилятору легче локализовать ошибку.

Пример сложно локализуемой ошибки:

```
(((a))
```

В этом примере невозможно определить, лишняя ли скобка, или, наоборот, не хватает скобки. Все, что может сказать транслятор: несоответствие скобок.

Точно так же end if, end case упрощает диагностику.

Примечание: C, C++

В языках C, C++: при пропуске }, компилятор выдает огромную диагностику, по которой сложно понять ошибку.
Еще хуже пропустить ; после класса.

```
struct t1 { … }
```
Это потому, что сразу после структуры можно объявить переменную или функцию.

```
Strict time {…} time (struct time*) {…}
```

Проблема в том, что явный терминатор отсутствует (скобка таковым не является).

Хорошим стилем программирования было: блок всегда завершается именем именованной сущности.

Пример

```
procedure P
…
end p
```

Интересно, что у языков с явным терминатором есть 2 свойства:

1. В этих языках составной оператор не нужен, т.к. конструкция begin … end является составной частью конструкции блок: <объявление> begin … end (Ада: тело, Оберон, Модула2: блок).
   В любом структурном операторе есть явный терминатор, поэтому в этих языках везде, где может стоять 1 оператор, может стоять и любое число операторов.

Оператор структурного выбора:

```
If B1 then

S1

Else if
```

Рекомендации по оформлению операторов структурного выбора: следует или все условия на одном уровне, или в одну строчку (чтобы подчеркнуть последовательность, а не вложенность).

Кстати, последовательные операторы легче воспринимаются человеком, чем вложенные. Кроме того, при использовании вложенности нужно писать много end if end if end if.

Для этого ввели конструкции else if, elif, …

```
if B1 then
S1
```

```
elsif B2 then
…
elsif B3 then
…
Else
Sn
End if
```

Частный случай: оператор case/switch.

Оператор CASE

Рассмотрим подробнее case.

Пример

```
Case expr of
Список1:S1
Список2: S2
…
Else
Sn
End
```

В различных языках: одна константа, диапазон, список констант. (Модула 2, Ада).

1,3,4..8,10 (зачем нужен непонятно)

Пример (Ада)

Нужно ключевое слово when

```
When 1,3,4..8,19=>
Others =>
```

В Модула-2 используется другой синтаксис.

Пример (Модула 2)

```
1, 2:S1;S2;… |
```

Case есть практически во всех современных языках. Опционально добавляется else-часть.

Такой оператор можно промоделировать последовательностью if, но это дурной тон.

Во-первых, case отлично моделирует записи с вариантами. Это ответ на проблему объединения типов (Янус-проблему).

Заметим, что синтаксис вариантной записи совпадает с синтаксисом оператора выбора.

Во-вторых, case работает быстрее.

Рассмотрим следующий пример:

Пример

```
T = expr
if(t == 1) S1;
Else if (t==2) S2;
Else Sn;
```

Структурные выборы дискриминирующие. Чем больше номер, тем больше: Таким образом, N-й по счету оператор выполняется только в том случае, если было N-1 проверок. Заметим, что иногда можно путем переупорядочивания увеличить быстродействие.

Выражение внутри case дискретного типа (в С-подобных: интегрального). В языке C#: типа строки.

Возникает вопрос, а как можно эффективно запрограммировать оператор case?

Пример

```
Case I of
0: S1
1: S2
…
255: S255
```

В ассемблерном коде здесь создается таблица переходов.

Пример (Fortran)

```
GOTO(M1,…,Mn) i
```

Это означает GOTO M_i

Если метки идут «вразброс», то используется хэш-таблица или некая поисковая структура.

В C# именно поэтому можно сделать строку в качестве метки case.

Заметим, что если в операторе case всего лишь 2-3 метки, то линейный список эффективнее, чем бинарный поиск. Если больше чем 3, то выгоднее сделать таблицу меток.

Таким образом, case – это вычисляемый оператор перехода.

Циклы.

Набора, который есть в стандартном паскале, хватает на всех.

1. `while B do S`
2. `repeat`
   `S1,…,Sn`
   `until B`
3. `for` (о цикле for мы поговорим отдельно)

Вопрос: зачем использовать циклы, если есть goto?

Здесь уместно вспомнить статью Дональда Кнута «О структурном программировании». Его идея очень простая и очень здравая: не нужно слепо следовать правилам «нельзя использовать оператор goto, Нужно четко понимать, почему нельзя использовать goto.

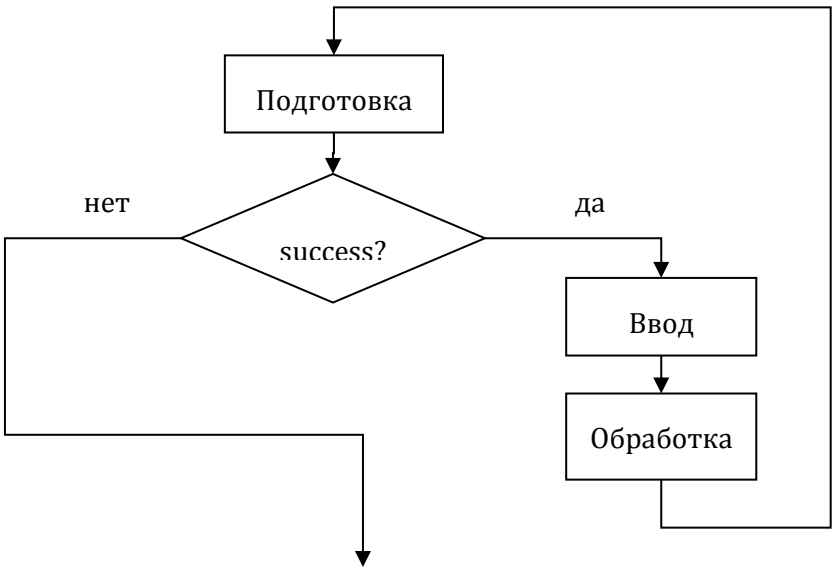
Почему же нельзя использовать goto? Потому что goto портит структуру программы.

Однако, бывают ситуации, когда неиспользование goto портит структуру программы.

Пример (выполняется в цикле)

```
Подготовка ввода
If(success) {
ввести
обработать
}
```

Рассмотрим блок-схему этого алгоритма:



Является ли программа структурной? – Да: один вход, один выход.

Рассмотрим 2 варианта: без goto и с использованием goto.

Пример

```
Подготовка
While (success) do
Begin
    Обработать
    Подготовка
End
```

Мы нарушили структуру алгоритма, так как прибегли к дублированию кода. Дубли – это всегда плохо; потому что в случае изменений нужно менять в двух местах.

Главный принцип хорошего программирования: структура алгоритма должна соответствовать структуре программы.

Хороший способ:

Пример

```
While(true) do
Begin
    Подготовить
    If not success
        Goto out
    end
    Обработать
End
Out:
```

Операторы break, continue, return минимизирует наличие goto.

Однако это не всегда спасает.

Необходимость ограниченного выхода в некоторых случаях необходима.

В Модуле 2 понятие цикла расширено следующим образом:

```
LOOP

      S

END
```

Если можно один оператор, то можно и много.

Только внутри LOOP можно использовать метку EXIT.

```
LOOP
```

```
    IF B EXIT

END

EXIT
```

В языке C: break может встретиться в C в любом цикле.

В языке Ада:

```
loop

операторы

end loop
```

Циклы без выхода – нужны ли? В многопоточном программировании – да. Их можно закончить «снаружи». Но в общем случае наличие таких псевдо бесконечных циклов допустимо.

`when условие => exit` – внутри цикла (или например условного оператора).

`for v in range` – другой вид заголовка цикла.

Оператор break помогает почти всегда, и иногда (очень редко) в C++, в C – относительно часто (обработка аварийных ситуаций). В C нет встроенной обработки аварийных ситуаций. Чистить ресурсы нужно не только в случае аварийных ситуаций, но и вообще.

Пример

```
{
    {
        If(B) goto end;
    }
    End:
    Очистка захваченных ресурсов
}
```

Почему нужно выходить из цикла: Нам нужно выходить из середины 1) того требует структура алгоритма 2) аварийная ситуация: обрабатывать дальше не имеет смысла. В языке C в этой ситуации хорошо употребить goto, потому что если таких ситуаций несколько, то будут повторяющиеся блоки кода.

Один из методов рефакторинга: избавление от повторяющихся участков кода (рефакторинг – это изменение кода без изменения его функционала; необходимый этап в развитии каждого проекта).

Системы рефакторинга выискивают такие повторяющиеся части.

Почему в C++ этого делать не надо?
RAII (resource acquisition is initialization)

Захват ресурсов в конструкторе, освобождение – в деструкторе.

Пример

```
{
    {
        Wrapper w;
        If(B) return;
    }
}
```

Если бы вся работа с ресурсами была локальной, то можно было бы всегда применять эту парадигму. К сожалению, в реальной жизни это далеко не всегда так.

История

Написал однажды И. Г. мобильное приложение. Оно широко использовало связь с сетью. Если есть связь, то всегда можно предусмотреть, что загрузка тянется неограниченно долго и может сломаться.

Приходит пользователь и говорит: зациклилось.

Курсор в форме часов – значит, зациклилась. А на самом деле зациклилась загрузка ресурсов. У курсора есть точка привязки (mouse press). Курсором можно было кликать и сохранить результаты работы.

Почему так получилось

Программа на C++.

```
CWaitCursor wc;
```

В конструкторе сохраняет старый и вставляет новый, в деструкторе очищает.

Длительная операция прерывалась по таймауту и свертки стека не было. Поэтому деструктор не вызывался.

```
CWaitCursor* pwc = new CWaitCursor();
```

Конструкция try – finally создана в Java как раз для таких целей. Finally будет выполнен всегда, и при нормальном выходе, и при исключительной ситуации – как раз для очистки ресурсов.
В итоге код пришлось переписать следующим образом:

```
Try {
}
Catch(…) {
    Delete pwc; throw;
}
Delete pwc;
```

Выбрано решение наверняка

Мораль: не программировать на компиляторе, который не использует свертку стека.

В заключение разговора о составных операторов.

Существует еще один контекст, когда использование goto предпочтительно.

Пример (C)

```
for(…) {
    for(…) {
        for(…) {
            if(m[i][j][k] == m0) goto end;
        }
    }
}
end:
```

Это код поиска в трехмерной матрице. Если нашел – надо делать goto. Если нет – использование дополнительных условий. Это допустимо: структурность сохраняется.

В Java можно использовать конструкцию break.

Пример (Java)

```
M: for(…) {
    for(…) {
        for(…) {
            if(m[i][j][k] == m0) break M;
        }
    }
}
end:
```

Заметим, что в Java goto относится к списку зарезервированных слов (может быть, для того, чтобы была возможность расширить язык?).

Поучительная история

Вся структура программы теряется, если каждый цикл оформлен в виде процедуры.

История от коллеги И.Г.
Коллега дал студентам задание. Оказалось, что они ускорили его в 100 раз. Коллега не поверил, сказал искать ошибку (100 раз – слишком хорошее улучшение). Что же сделали студенты?

```
M: for (…) P;
P: for (…) Q;
Q: for(i, j, k): if(m[i][j][k] == m0) throw new t;
```

Коллега сразу сказал: ищите ошибку.

```
Try {
…
}
Catch(t exp) {
…
}
```

Механизм обработки исключительной ситуации создан для аварийных ситуаций. Время не посчитаешь – время исполнения программы толком не замеряется. Все работает корректно, только очень большие накладные расходы.

Поэтому после того, как убрали исключение, стало в 100 раз быстрее.

Ситуации, когда есть смысл употреблять goto:

4. Обработка исключительных ситуаций.
5. Вложенные циклы.

Цикл FOR

Структура управления абсолютно одинакова.

С-подобные языки:

```
while(B) S
```

```
do S while(B)
```

```
for(;;) S
```

Используются также следующие операторы:

1. Break;
2. Continue;

3. Return;
4. “goto”

Блок – только как тело процедуры (Модула2, Оберон, Ада).

По goto можно уходить только в пределах одного блока.

В стандартном С все блоки на одном уровне, потому что нет вложенных блоков.

Значит, истинная область видимости: блоки, являющиеся телом процедуры или функции. Такие блоки называют еще «запись активации» или «frame».

В языке JS: объявления внутри «поднимаются» наверх во внешний блок.

```
for(e1;e2;e3) S
```

Какой язык ни возьми, можно найти его аналоги. С точки зрения И. Г., это скучно.

Новые вещи возникают либо в параллельном программировании, либо изменяется цикл for.

Пример (Модула 2)

```
FOR v:=e1 TO e2 [STEP e3] DO
    S1;…;Sn
END
```

Отличия других языков: локализация описаний

В 1988 вышло сообщение о языке Оберон, в котором Вирт с удовольствием сообщил, что покончено с традицией, идущей с языка Алгол60. Цикл for убит в языке Оберон.

В 1993 вышел Оберон2. В нем добавили открытые многомерные массивы, виртуальные методы (динамическая привязка процедуры), цикл FOR. Он был сделан не столько Виртом, сколько его сотрудниками.

Оператор FOREACH

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

Коллекция это последовательность, и ее можно пройти. Для этого используется итератор, он неразрывно связан с понятием коллекции. Итератор должен позволять разыменовывать себя и должен быть метод next (похоже на i++). Тогда по соответствующему классу можно устроить проход.

Если с является массивом:

```
for(int I = 0; I < c.length; ++i) {
    T x = C[i];
}
```

Операция прямого доступа (которая может быть не у всех) дороже, чем next. Увеличить указатель на некоторое смещение проще, чем операция прямого доступа.

Что же является коллекцией?

Рефлексивная тенденция в строении современных языков программирования.

Есть понятие интерфейса. Про некоторые интерфейсы язык кое-что знает.

Пример (C#)

Interface IEnumerable

У такого есть итератор, у которого есть current и next.

Если класс реализует IEnumerable, то компилятор знает, как к нему применять foreach.

Пример (Java)

Появился вариант for:

```
for (T x: C)
```

Это эквивалентно foreach.

```
foreach (var x in C)
    x
```

Рассмотрим язык Python. Это язык со смешанной парадигмой.

Пример (Python)

Внешне for в языке Python – это банальный for.

```
for x in C
```

Самое нетривиальное: отступы являются частью синтаксиса. В Python нет begin/end.

```
if B then: S1 else: S2
```

```
if B then:
    S1
    S2
    S3
S4
```

То есть отступ является частью синтаксической структуры.

```
for x in C

S
```

Рассмотрим циклы for. Они очень хорошо демонстрируют общий принцип дуализма операции и данных.

Данные – то, над чем можно выполнять операции.

Операции – то, что можно выполнять над данными.

И ничего более общего сказать нельзя.

Пример (Python)

В Python нет массива. Последовательность с прямым доступом называется списком.

Есть понятие кортеж (tuple).

```
[1,2,3] # список
<1,2,3> # кортеж
```

Кортеж отличается от списка, например, тем, что он immutable.

Цикл for можно рассматривать как генератор некой структуры данных.

Мы можем задать структуру данных, а можем генерировать новый элемент по необходимости.

Коллекция называется ленивой, если ее элементы вычисляются тогда, когда они понадобились.

Рассмотрим множество (в математическом смысле):

{ X | X принадлежит M : P(X) }

```
[ x: for x in M: if P(x) ] # генерация списка.
```

Здесь for – не составной оператор, а элемент генерации списка.

На первый взгляд кажется, что первый x избыточен. Но на его месте может стоять произвольное выражение.

```
M: <key, value>
[ key for <key, value> in M ]
```

Если M состоит из пар, то мы первую называем key, вторую называем value, и берем только key.

Рассмотрим следующий пример.

```
x = set(T)
y = [ w for w in x: if len (w) = 3 ]
```

Практически все необходимо делать во времени выполнения.

Python просто запоминает генератор.

```
z = Y[:50] # нужно взять последние 50 элементов.
```

Будут сгенерированы только последние 50 элементов.

С этой точкой зрения ленивые вычисления, которые тесно связаны с понятием генератора и понятием дуальности вообще, дают большую гибкость.

Можно наблюдать проникновение функционального стиля. Причем эта тенденция характерна для всех современных языков.

На этом заканчиваем разговор о базисе ЯП.
Заметим, что базис современных языков максимально упрощается, а любая более или менее сложная логика переносится в библиотеки. Знания одного языка очень мало, нужно еще знать библиотеки.

Средства развития языков программирования

Языки программирования различаются средствами развития.

Процедурные ЯП:

- данные ⇔ТД: множество значений + множество операций
- операторы: действия с побочным эффектом (аналог многих машинных команд).

Очевидное средство развития – процедура (подпрограмма).

Второе средство развития – создание новых типов данных, для чего необходимо задать структуру данных и операции.

Подпрограммы

Передача управления в подпрограмме

Создание подпрограммы:

```
P(arg1,…,argN)
```

…

У создания подпрограммы есть два этапа:

1. Абстракция – объявление (описание) подпрограммы
2. Конкретизация – вызов и связывание формальных и фактических параметров (коротко – передача параметров).

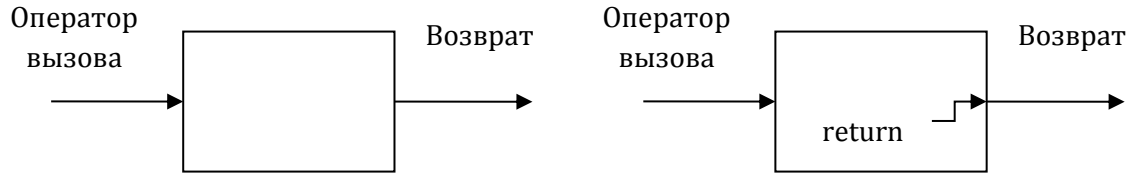
Вид связывания при конкретизации абстракций:

f(e1, e2, e3)

Связывание вызова с телом бывает статическим, а бывает динамическим.

В традиционных языках программирования применяется статическое связывание. Но это тема следующего пункта.

Каждая программа выглядит так:



Каким образом вырабатывается значение:

1) return expr; (return в случае процедуры)
2) имя = expr;

Можно ли использовать функции как процедуры? Большинство современных языков позволяют (исключения: языки Вирта, Ада).

Главное предназначение функции: вырабатывать значение.

Единственное предназначение процедуры: побочный эффект.

Функция – это аналог операции. Ряд языков программирования(Ада, C#, C++) позволяют переопределять стандартные операции в виде функций.

А может ли функция иметь побочный эффект?

Первое желание: запретить

Пример (Ада)

В языке Green (позже: Ада) функции вырабатывают значение, при этом побочные эффекты запрещены. Во-первых, запрещены изменения формальных (фактических) параметров; во-вторых, изменять глобальные переменные тоже запрещено.

Создатели первого варианта языка Ада предлагали:
функции (запрещены любые побочные эффекты)
процедуры (разрешены любые побочные эффекты)
процедуры, возвращающие значение

```
function F(X: T) return T;
```

```
procedure P(X:inout T);
```

Допускаются спецификаторы in, out, inout.

У функций только in, он опускается.

```
Procedure FP(X: inout T) return T;
```

Ведет себя как функция, однако имеет побочный эффект.

```
procedure FP(X: in T) return T;
```

Процедуры, возвращающей значение, в языке Ада нет.

Правило законодателя: создавать только те законы, соблюдение которых можно проверить.

Если проверять побочный эффект в общем случае, то эта проблема алгоритмически неразрешима. Поэтому не нужно делать процедуру с возвращаемым значением.

Максимум: потребуем от функций, чтобы они не меняли входные параметры.

Заметим, что если функция имеет скрытый побочный эффект, то это обычно свидетельствует об ошибке в архитектуре программы

До появления структурного программирования считалось, что программисту нужно давать полную свободу. Сейчас считается, что эту свободу нужно ограничить.

Примеры свободы в старых языках программирования.

ENTRY M

-- альтернативный вход в процедуру

SUBROUTINE P(X, Y, Z, *, *, *)

Некоторые параметры могли быть звездочками.

Формальные параметры * могли соответствовать процедурам.

RETURN *i

Здесь i либо 1, либо 2

P(1,2,-1,56,128)

Глядя на P, нельзя сказать, где окажешься после вызова.

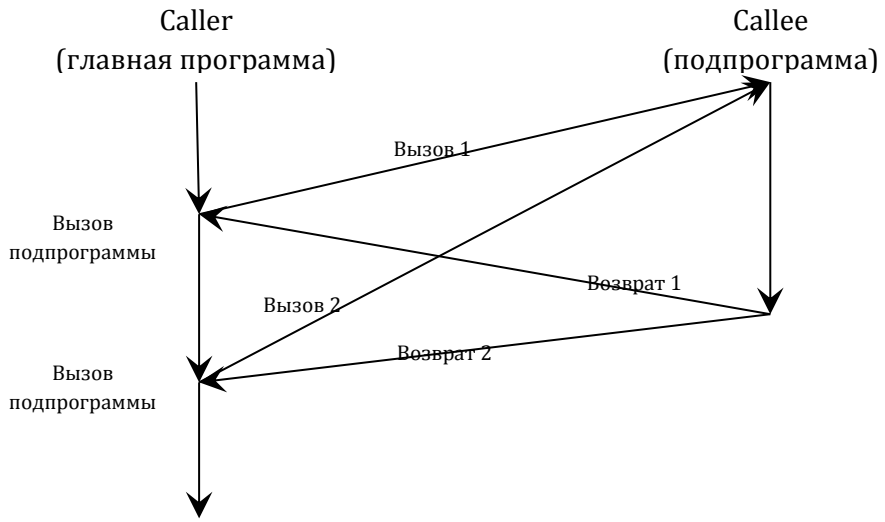
Это полностью противоречит правилам структурного программирования, поэтому от этой возможности сейчас полностью отказались.

Процедура: один вход, один выход.

И.Г.: Единственное категоричное правило: не следовать категоричным правилам.

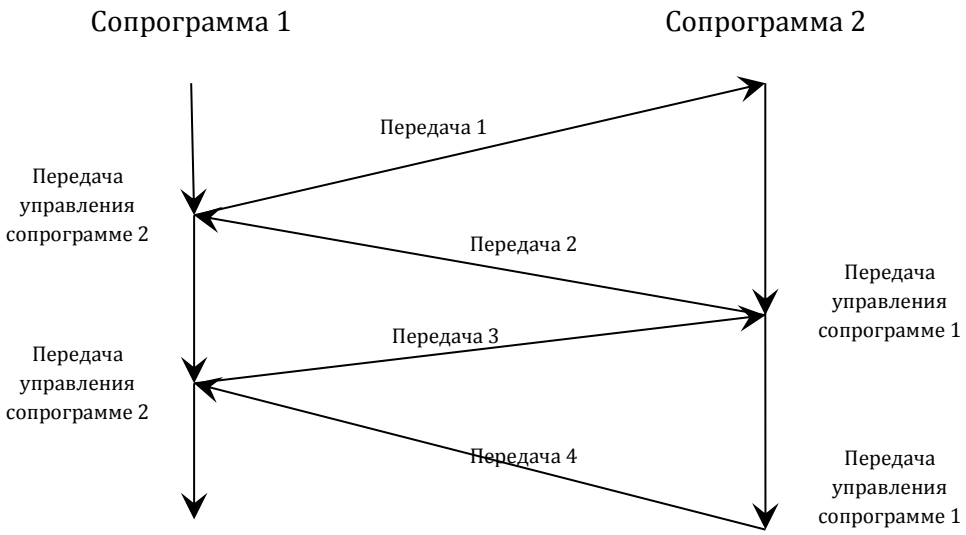
Правило: не делать процедуру длиннее 50, 25, …, N строк является слишком категоричным. Желательно, чтобы процедура отражала законченный алгоритм. Не нужно менять структуру программы в угоду формальным требованиям. Когда не следовать правилам, а когда следовать – подскажет здравый смысл.

Проанализируем подробнее передачу управления. На диаграмме вначале начинает работу главная программа, затем она вызывает подпрограмму (вызов 1), подпрограмма выполняет работу и завершается, передавая управление главной подпрограмме (возврат 1). Затем главная программа работает еще какое-то время, после чего вновь вызывает подпрограмму (вызов 2), подпрограмма выполняет работу и завершается, передавая управление главной подпрограмме (возврат 2).



Видна несимметричность главной программы и подпрограммы. Как создать симметричность?

В начале 60-х годов выдвинули концепцию сопрограммы. Сопрограммы передают управление друг другу поочередно.



Сопрограмма называется subroutine.

CALL – вызов подпрограммы.

TRANSFER P (RESUME P) – возобновление сопрограммы.

Некоторые процессы с использованием сопрограмм описываются проще.

1я статья: Дейкстра.

2я статья: Хоар.

3я: Далл Ньюберг.

Страуструп (датчанин) придумал C++. В этой 3ей статье были описаны сопрограммы, с отсылкой к языку Simula67, и приведен пример задачи, которая очень просто решается при помощи сопрограмм.

Задача: дано 2 упорядоченных файла. Необходимо составить третий файл, объединяющий те два, и тоже упорядоченный.

Решение: пусть есть буфер. Вначале инициализируется большим значением. Каждая подпрограмма пишет в новый файл до тех пор, пока текущее значение не больше значения буфера. Потом она пишет следующее значение в буфер и передает управление.

Другой пример: лексический и синтаксический анализ. Они могут поочередно передавать друг другу управление, при этом схема работы программы будет выглядеть проще.

Из языков высокого уровня, поддерживающих сопрограммы, наиболее известен язык Modula 2.

```
PROCEDURE NEW_PROCESS(P: PROC; VAR C: ADDRESS; N: CARDINAL);

…

TRANSFER(VAR cor1, cor2: ADDRESS);

…
```

Возникает вопрос: в чем смысл третьего параметра процедуры NEW_PROCESS?

Каждая подпрограмма создает свой фрейм (frame), которой есть запись активации (activation record). Он включает в себя:

- формальные параметры,
- адрес возврата,
- локальные переменные.

Запись активации располагается в стеке.

В архитектуре mainframe IBM 360-370 стека не было.

Именно поэтому адрес возврата хранится в записи активации в соответствующей области.

Для сопрограмм записи активации должны храниться в специальном списке (ни в коем случае не на стеке, так как иначе они уничтожатся).

N – это размер записи активации. Про него ничего сказать нельзя.

Понятие сопрограммы – очень хорошее и красивое понятие, но при его использовании появляются низкоуровневые проблемы.

Это одна из причин того, что концепция сопрограмм не использовалась в ЯП.

Сейчас эта парадигма возрождается.

Вирт использовал сопрограммы для квазипараллельного программирования.

Механизм мониторов был реализован в Модула2 на языковом уровне.

Пример из книги Вирта:

SEND(VAR S:SIGNAL)
WAIT(VAR S:SIGNAL)

Если рассмотреть любое API работы с потоками, то можно найти много общего.

В чем разница сопрограмм и потоков? В потоках нет явного RESUME. Планировщик берет на себя роль выделения того процесса, который готов.

Пример использования сопрограмм:

IOTRANSFER – указание передать управление некому драйверу – обработчику прерываний. Параметром задается номер прерывания.

Таким образом, при помощи сопрограмм была реализована концепция драйвера.

В Оберон в таком виде это уже не вошло. Хотя сама концепция программы весьма красива.

Передача параметров в подпрограмму

`P(arg1, …, argN)` – описание подпрограммы

`P(e1, … , eN)` – вызов

Существует два способа передачи параметров в функцию: позиционный и ключевой.

Позиционный способ передачи параметров: нумеруются в том порядке, в каком перечислены. Формі = Факті (при этом имена параметров при описании не важны).

В Python реализован другой способ – ключевой способ: по имени. В языке Ада:

```
имя: expr
```

Объявление процедурного типа

```
TYPE Prc = PROCEDURE(VAR:INTEGER;REAL);
X: Prc
X:= P
X(I,1.0)
```

С точки зрения реализации, такие типы суть указатели.

Пусть дана функция `void f (int = 0);`

Фактически, это две функции.

```
void f(int i) {…}
```

```
void f() {f(0);}
```

Две такие записи эквивалентны. Точно так же `X(int i = 0);` (конструктор).

В C# появились параметры по умолчанию.

Чем удобен ключевой способ? Можно перечислить все параметры, кроме параметров по умолчанию.

Способы передачи параметров.

Под «способом» может подразумеваться 2 вещи:

1. Механизм передачи (как именно осуществляется передача)
   А вообще хорошо или плохо когда начинаем обсуждать механизм?
   Смотря с какой точки зрения! (Например, языки программирования – хорошо или плохо? Asm хорошо или плохо? Смотря с какой точки зрения!)
2. Inout – семантика (семантика изменяемости)

Не по механизму, а по семантике передачи входных параметров.

В языке Ада хотели рассмотреть параметры со 2 точи зрения:

Список имен: спецификатор имя_типа

```
X,Y: out T;
```

```
A: in T1;…
```

Может ли изменяться параметр?

Спецификатор – in,out, inout.

Формальный параметр / фактический параметр: как меняется формальный параметр и меняется ли в соответствии с ним фактический?

В АДА от in требуется только неизменяемое значение (out наоборот). inout – требуется и статус неопределенности и изменяется (может и использоваться и изменяться). Компилятор выберет механизм передачи сам. Если не стоит спецификатор, то по умолчанию в АДА in. А механизм передачи уж определит компилятор.

На первый взгляд идея здравая, если не анализировать последствия (отвечала духу языка АДА – многое перекладывать на компилятор). В итоге: к чему пришли создатели языка АДА?

Обычно под способом имеется ввиду первое – конкретный механизм передачи (в язык входит низкоуровневость).

Вообще говоря, способов 5(если взять все языки)

1. по значению,
2. по результату,
3. по значению/результату,
4. по адресу (ссылке),
5. по имени.

Рассмотрим эти способы

Опишем семантику первых трех способов (семантика копирования – выделяется память под формальный параметр, то есть идёт копирование туда и/или обратно)

Как производится соответствие формальных и фактических параметров?

В современных компьютерных архитектурах аппаратно реализуемый стек – для передачи параметров. В подпрограммах образуют стек, следовательно, место в стеке выделяется под формальный параметр.

По значению: копирование формального параметра в фактический (фактический в стек), вызов процедуры и всё (отсутствует обратное копирование). Даже если фактический параметр меняется, то всё равно, т.к. отсутствует обратное копирование, фактический не меняется. В данном случае реализована in-семантика. Изменение формального параметра не приводит к изменению фактического. Передача по значению чревата последствиями, если параметры велики.

По результату: реализована out-семантика. Производится только обратное копирование перед return.

По значению/результату: и то, и другое. Этот вариант удобен с точки зрения функциональности, но совершенно неудобен с точки зрения удобства.

В Pascal можно передавать параметры по значению и по ссылке.

Советуем массивы передавать как параметр-переменную, чтобы минимизировать копирование массива.

В случае большого фактического параметра – передавать его по ссылке (теряя in-семантику).

По ссылке: вместо значения фактического параметра копируется его адрес.

Ссылка не то же самое, что адрес! Содержит ли ссылка адрес? В общем случае да. Сборщик мусора перезагружает адрес, но ссылка продолжает работать.

Существует ЯП, в котором есть единственный способ передачи параметров – наболее универсальный способ – по адресу(первые версии Бэйсика,Фортрана).

Язык С имеет единственный способ передачи по значению – он не обеспечивает in-out(out можно реализовать путем явной передачей адреса). Есть операция взятия адреса любого объекта. Вообще, если есть понятие указателя, а также возможность получения адреса любого объекта, то можно передать параметр по адресу ссылки.

Передачу параметров по значению нужно рассматривать для совместимости с системными вызовами.

COM API – на C++, макросы на языке С

Поэтому передача по значению, хотя бы для того чтобы была совместимость с языком С.

Чтобы ответить на вопрос об эффективности неизбежно спускаемся на нижний уровень.

К чему в итоге пришли большинство ЯП? Как производится современная передача параметров?

Специфицируется механизм.

Вспомним про язык АДА.

Для массивов передача только по адресу, но при этом следит за in и out семантикой.

На первый взгляд здорово, чего же не хватило? Почему в АДА 95 создатели сказали «простой ТД int будем передавать поз значению, а остальные по адресу (или ссылке)»?

Пришли к тому же способу что и сейчас.

C++ по значению, по ссылке 1 ,4.

Стандартный Паскаль 1,4 .

Java 1: если хотите менять значение переменной, но семантики нет, упаковывай в объект и передавай по ссылке

Простые ТД – значения копируются, а для сложных – копирование ссылок (то есть как бы по ссылке).

4 способ для объектов

Объекты не именуются – именуются ссылки.

Передача по ссылке за бесплатно.

Пример (C#)

В C# два способа: по значению 1, явным образом 4 способ ref(out семантика)

```
void f(X t)
{
  t = //не оказывает влияния на фактический параметр
}

void f(ref X t)
{
  t = // фактический параметр меняется
}

X a = new X();
Ref a
```

Ref требует определённости объекта, а out не требует

F(ref a/out a)

Если неинициализированная ссылка, то возникнет прерывание. То есть ref – inout семантика, а out – только out семантика. Это используется для большей гибкости языка.

Андерс Хейзберг – отец системы TP,Delphi. Почему Хейзберг засунул такие вещи?

В TP есть параметры-значения, параметры-переменные. Передаем параметр-значение, то же самое, что передаем ссылку. А можем ли саму ссылку изменить? В Delphi не можем.

То же самое Хейзберг хотел в C#, осуществив ref и out.

Пример (Ада)

Возьмем невинную процедуру АДА

```
procedure P(X,Y:inout T) is

…

begin

X:=expr1;

//raise error

Y:=expr2;

End P;
```

В чем состоит беда?

Raise(почти что throw). Почему использовал raise – вопрос совместимости(с С конечно).

Появились new,delete и прочее.

А что нельзя перекомпилировать? Библиотеки, системные вызовы.

Чтобы компилировались все стандартные заголовки Linux.

Raise - посылка сигнала самому себе.

```
struct time

  Time(…)
```

Имя типа, а не функции.

Но была обеспечена совместимость с Unix.

Raise error.

Является программа стандартной? Да

Как можно реализовать out-семантику? 3. 4.

P(a,a) если по ссылке и исключение, то а изменит значение, иначе не изменит. Это нехорошо. Единственный способ обеспечить переносимость – явно специфицировать механизмы. Поэтому большинство языков выбирают самый простой – по значению, и самый универсальный – по ссылке.

Образцовый язык – только один (с этой точки зрения): C++

`const T& x` //ссылка на константный объект

Привело ли к усложнению языка? В некотором смысле да.

Плевать хотел на спецификацию стандартных функций в C++, а это нехорошо.

Если указал неявное преобразование для своего класса, то объект получается константный.

```
const T x;
```

Const входит в профиль функции

```
X* const this
```

```
void q() const;
```

```
const X* const this;
```

В STL четко выдерживается in-семантика.

Не просто специфицировать константные ссылки, но и говорить о константных методах.

Если поле отмечено как mutable, то даже в константных объектах оно может меняться.

```
mutable X a;
```

```
const T x;
```

Константный именно объект(а не класс).

По умолчанию можно применять только члены с const-ом.

Const – спецификация this.

```
X *const this;
```

`this` – константа (this = нельзя в C++), но она ссылается на неконстантный объект.

```
void g() const;
```

Это значит что `const X *const this;`

А что же сказать про члены – данные – все константы (публичные и нет).

`X.a = 0` //не статический член

`mutable` – специальный модификатор к членам данным, даже в константных объектах поле может меняться(например, некоторые объекты допускают кэширование; меморизация)

`mutable` – не значит что плохой стиль программирования (наоборот – думаете о константности).

Имманентные свойства современных процедурных языков – должны специфицировать на низком уровне механизм передачи параметров.

Несколько слов о 5 способе передачи – самый забитый, однако самый естественный – по имени.

Был использован в Алгол 60 (в нем еще было реализовано по значению).

`value p;` //p передается по значению

В фортране не было рекурсии => можно было реализовывать запись активации в статическом списке.

Как только появляется рекурсия, необходима динамическая структура.

Если не `value p`, то передавался по имени.

Что записываем в фактический, то и будет формальным. Например,

`P(a)`

Вместо формального параметра подставляется a

Если `P(X[0])` – вместо формального параметра `X[0]`

`P(X[i],i)` – подставляются `X[i],i.`

С точки зрения механизма – самый высокоуровневый.

Пример (Simula 67)

Работал неэффективно, т.к. был построен на базе Алгола60.
Программист говорит «а я не знаю что будет вместо i».

Как компилировался код?

---

procedure PP(x,y);
integer x,y;
Begin
x:= …
y:=…
end
	void PP(xmy) // old-fashion C
	int x,y;
	{
	…
	}

Алгол60 при компиляции программировал thunk – некая процедура – помощница, кот вызывается, когда нужно что-то делать.

Для параметров по значению вычисляют его адрес (не значение), X[i] – вычисляет адрес объекта.

И так много раз

Кроме косвенного обращения нужно еще и вычислить адрес – неэффективно!!!!

Одна из задач в старых учебниках по программированию: доказать, обосновать, что Алгол 60 нельзя написать `swap(x,y)` – написать то можно, но обращение `swap(x[i],i)` будет давать неправильный результат.

А в Lisp как передаются параметры – по значению или по ссылке? В функциональных языках, let x = … - это не присваивание, это отождествление.

В высокоуровневых ЯП нет задачи специфицировать механизм.

А в процедурных ЯП нет возможности программировать без спецификации.

Перегрузка имен

Во многих ЯП существует статический полиморфизм: одной сущности (подпрограмма) может соответствовать несколько форм (несколько тел). Это перегрузка (overloading).

Статический полиморфизм – выбор, какой именно способ статический.

Динамический полиморфизм происходит на уровне выполнения.

Существует два вида статического и один вид динамического полиморфизма.

Динамический полиморфизм – связывание виртуальных методов.

Статический полиморфизм – перегрузка имен и параметрический полиморфизм.

Сущность при перегрузке имен – действие. Способ реализации сущности - подпрограмма (в общем случае).

У сущности есть имя, работаем с именованными сущностями, нет возможности именовать оператор. Действия в общем случае подпрограмма. Действие ассоциируем с именем

Способов реализации действий может быть много. Сущность ассоциируем с именем.

Какая проблемная область чаще всего нуждается в перегрузке? Ввод/вывод. Почему? Да потому что с точки зрения действия есть вывод, а конкретная суть зависит от реализации.

Одна область видимости и несколько определяющих вхождений

Возникает два понятия: определяющее вхождение и использующее вхождение.

Подавляющее большинство ЯП для определяющего вхождения используют объявление. Это и называется перегрузкой имен.

Какие языки не допускают перегрузки имен? Старые(C, Pascal)+ Modula 2,Оберон

Вирт не счел понятие перегрузки необходимым

```
cout << "count="<<cnt<<endl;
```

На Обероне или на МСодуле 2

```
INOUT.writestring("count=");

INOUT.writeint(cnt);

INOUT.writeln;
```

В АДА можно перегрузить

```
type color is (red,green,blue);
```

В одной области видимости red может иметь несколько определяющих значений.

```
X:= pi;
```

Как технически реализовать выбор?

```
Procedure p(x,y: T)

P(a,b);
```

В АДА в одном из первых появилось перекрытие – в разрешение перекрытия входит еще и возвращаемое значение.

```
procedure P(X:T)
function P(X:T) return boolean;
function P(X:T) return integer;
```

По контексту вызова можем распознать все.

Подпрограммный ТД

В большинстве процедурных ЯП (императивных) процедурный ТД – разновидность указательного ТД. Это еще раз подчеркивает низкоуровневость процедурных ЯП. В языке Ассемблера имя процедуры: (CALL P) P – метка. Мы не можем сделать jmp X в языке Ассемблера, а jmp P – можем, P – это метка. И то, и то адреса, но это адрес команды, а не данных.

В языке Си честно указано, что когда мы передаем что-то, то мы так и должны написать:

```
void f(void (*arg)());
```

Или с помощью typedef'a:

```
typedef void vf(); void f(vf *arg);
```

Эти определения эквивалентны. В Си нам явно «тыкают пальцами»: это указатель.

В Си ++ нет разницы между (*arg)() и arg(), не проводят разницы между указателем на функцию и именем функции.

В языке Модула-2 или Оберон делается просто – опускаем имена.

```
TYPE PIF = PROCEDURE (INTEGER):INTEGER;
PRC = PROCEDURE (VAR:INTEGER);
```

Процедурный ТД – это указатели в большинстве языков (большинство – это N-1, где 1 – это C#).

Все объекты первого порядка характеризуются тем, что из одних можно получать другие. Множество значений процедурного типа – это набор имен процедур и функций соответствующих прототипов, описанных в Вашей программе. Каждое имя соответствующей процедуры можно рассматривать как константу. Все, что мы можем делать с процедурным типом: копировать, вызывать, сравнивать (одну с другой). Мы не можем на основе одной процедуры получить другую, поэтому процедуры не являются объектами первого порядка.

В первой версии языка Ада процедурный ТД вообще отсутствовал. Вообще, в каких случаях нам нужен процедурный ТД? 1) передача, как параметров 2) функции обратного вызова (callback'и). Как же можно в Аде обходится без него? Ответ – обобщения (generic). Чаще всего в литературе generic переводится – родовое. Обобщенный – параметризованный, статический.

Возьмем C++, который Вы знаете лучше чем Аду:

```
template<class T,  int size> class Stack {…};
Stack <int, 128> S;
```

На месте 128 Вы можете поставить константу и только константу.

В языке Ада роль функций обратного вызова играли статические переменные. Но это обходилось довольно дорого – объем кода получался большой. Например, библиотека, занимающая на Аде 90000 строк на C++ занимала 11000 строк.

```
type PRC is access procedure (inout INTEGER);
p :  PRC;
```

```
procedure Sub(X:inout integer);
p := Sub access;
```

Есть особый язык C#, он особый, потому что является языком императивного программирования, но в него все чаще «пролезают» вещи из функционального программирования.

C++ - указатель на член. Указатели на член-данное – это не указатели.
Всегда, когда хочешь человека завалить, просишь его – напиши объявление указателя на член класса X.

```
class X {
      static int s;
      int i;
      int j;
      …
};
```

```
int x::*p;
```

```
p=&x::i;
p=&x::j;
```

```
int *pp;
pp = &x::s;
```

.* - обращение к указателю на член.

```
X a;
a.*p
```

Указатели на члены-функции – это не указатели.

`void (x::*pfx)(int));` Тут конечно без «пол-литра» не разберешься.

Тут должен быть более сложный механизм. Нужно учесть виртуальность функций.

Собственно говоря, указатели на члены функции (->*) сделали просто «до кучи», я их никогда не использовал.

```
void (*p)(int);
```

Функции обратного вызова: модель «подписка/уведомление» поддерживается в C#. Мы говорим, что заинтересованы в этом событии – устанавливаем свой обработчик (подписка). Затем происходит обратный вызов (уведомление). Подписчик должен не зависеть от других подписчиков. Некая разновидность процедурного типа.

В C# часто используются статические функции, потому что нет глобальных. Например, функция main. Мы должны ее вызвать до того как порожден какой-либо объект данных. Также, любая математическая функция – sin, cos, exp,… Изобрели специальный класс math для этих функций.

```
import пакет.*;
```

В Java вынуждены писать math.exp. Для того, чтобы этого избежать, необходимо написать
`static import Math.*;`

В языке C# придумали делегатский ТД (делегат). Синтаксис:

```
class X {
      delegate void f(int); //аналог объявления функционального типа
      void g(int x);
      …
      f a;
      class Y {void h(int a) {…} static void l(int b) {…} }
}
```

Что можно сделать с делегатом? Присвоить. Синтаксис в первом C# он был ужасный:

```
a = new f(this.g);
a = new f(Y.l);
var x = new Y();
…
a = new f(x.h);
```

В делегате на самом деле содержится список соответствующих значений. Поэтому возможны операции: +=, -=. (Это фактически – подписка.) Делегат можно сравнивать. Также вызывать a(0); (это уведомление).

Прелесть делегатского типа – event. Event – экземпляр делегатского типа.

```
class Server {
      public event f a {
            add { value }
            remove {…}
      }
};
```

Внутри функций членов сервера можно выполнять все операции делегатского типа.

```
class Client{
      void handler(int a) {…}
      ..
      a += handler; (подписка)
      …
      a -= handler; (отписка)
};
```

А внутри клиентского только подписку и отписку.

Но не это самое интересное в делегатском типе. Появились анонимные делегаты.

```
a = delegate(int x){return x+1;}
```

Это символ – делегат.

До этого приходилось обязательно придумывать эту функцию, а еще и в какой класс ее разместить.

```
delegate int Func1();

private int cnt = 0;

private int c;

Func1 d;

void P(Func1 f){…d=f;…}

void X(){…console.Write(d());}

P(g1); X(); X(); X(); - будет напечатано: 1,2,3

int g1(){return cnt = cnt+1;}


P(g1) ~ P(this.g1)


void Sample() {
      int x=0;
      P(delegate(int){return x=x+1;});
}
```

Теперь вызовем вместо P(g1) функцию Sample();

```
sample(); X(); X(); X();
```

Если локальная переменная используется в анонимном делегате, тогда получается, что область жизни становится больше области действия. В функциональных ЯП это встречается – замыкание(closure).

В Java есть понятие анонимного класса. До этого во всех языках функция – только на уровне глобального действия. Inner class может быть как именованным, так и анонимным. Объект класса существует и методы класса существуют, а имени уже нет.

Понятие замыкания имеет очень много следствий и является принципиальным расширением парадигмы.

Лямбда функции:

```
()=>x=x+1
```

`(x)=>x+1` – нетипизированный делегат

```
Func1 <int,int> f;

F = ((x)=>x+1)
```

Также появились деревья выражений. Функции стали объектами первого порядка.

Определение новых типов данных

Что представляет собой определение новых типов данных?

С этой точки зрения Языки программирования разделяют на, так называемые, старые и новые.

«Старые» ЯП: ТД <-> СД (множество значений).

«Современные» ЯП: тип данных <–> множество операций(набор подпрограмм, процедур или функций).

АТД(абстрактный тип данных) – это всего лишь набор сигнатур. Чтобы задать АТД, необходимо задать множество операций через задание сигнатур.

Как операции становятся операциями?

op_1,op_2,op_3… – необходимо определить их свойства.

Чтобы определить реализацию – добавить тело к сигнатуре – {…}.

К интерфейсу ТД необходимо добавить реализацию (тела подпрограмм+ набор структур данных и переменных).

Есть ли в языке С АТД? Напрашивается ответ нет, но это не совсем так.

FILE* -работа только в терминах набора операций

Если FILE не АТД то что же это тогда? Это же набор сигнатур, а значит АТД. Так что в С есть АТД.

Никто не напишет extern int a; – так писать нельзя, нужен include.

Для поддержки необходимой абстракции необходимо разделить интерфейс от реализации.

В С существовал typedef – но это не определение нового типа данных(это просто синоним – говорили об этом ранее).

В С++ единственная возможность определения нового типа данных – определить новый класс. Иногда хватает плоских классов – без иерархий.

Вторая концепция – логический модуль. Логический модуль - языковая конструкция, которая имеет интерфейс и реализацию. Интерфейс – это, попросту говоря, то, чем можно воспользоваться, хотя у слова «интерфейс» много определений.

Кауфман в своей книге дал такое определение: модуль – то, что проще заимствовать, чем создавать заново. «Заимствовать» – использовать в разных частях программы(или в разных программах).

А что же у нас в интерфейсе? В интерфейсе появляется соответствующее имя ТД, сигнатуры, реализация – раскрывает структуру ТД и, естественно, тела соответствующих подпрограмм, объявленных в интерфейсе.

В яхыках Модула2, Delphi (Turbo Pascal), Object Pascal, Оберон, Ада понятие библиотечного модуля, у которого есть 2 части: модуль определений и модуль реализаций.

```
DEFINITION MODULE M;

     Объявление типов,подпрограмм,констант,переменных
```

```
END M

IMPLEMENTATION MODULE M;
```

Определение подпрограмм из DEFINITION MODULE + дополнительные ресурсы.

В Turbo Pascal и Delphi модуль = unit - одна единица, в М-2 библиотечный модуль это две разных единицы.

В Модуле 2 в INOUT была определена переменная DONE – исключений то там нет.

Она, вообще говоря, должна быть доступна только на чтение, но МОДУЛА 2 не давала средств экспортировать переменную только на чтение (в Оберон уже появились такие средства).

Одностороння связь между модулями

Что это за связь? Связь по использованию.

Есть служебные модули. По большому счёту любой библиотечный – служебный (Service modules). Служебные модули представляют некие вычислительные ресурсы.

Модуль – это вообще возможность объединять и экспортировать как единое целое совокупность вычислительных ресурсов. Существуют также клиентские модули – это те, которые используют служебные модули. Служебные о клиентских ничего не знают – поэтому связь односторонняя, связь по использованию.

Модуль не может использовать прямо или косвенно самого себя.

Любой служебный модуль – интерфейс (.h) + реализация (.c).

В каждый клиентский модуль включать .h только 1 раз.

M1->M2(M1 включает M2).

M->M1->M2(M включает M1, M1 включает M2).

Файл .cpp может включать кого угодно, но обязательно должен начинаться с того, что включать свой hpp.

На языке С чтобы избавить от двойного включения, включается макрос:

```
#ifndef __имя_H__//нечто, что в проекте больше не встретится
#define __имя_H__

… // код файла
#endif
```

При попытке повторного включения этот символ уже определен и не включается более.

Концепция импорта должна быть явной. Импортировать можно только то, что экспортируется.

В Модуле 2 есть слово export(правда только для локальных модулей).

Delphi, Turbo Pascal используют uses M

Бывает видимость 2 видов - непосредственная и потенциальная (для того, чтобы имя стало доступным, мы должны его уточнить).

M.f – f описано внутри M.

Если используем *uses M*, то можно просто *f.* Для чего это нужно? Для удобства.

А для чего не нужно? Чем это плохо? Возникает конфликт имен! Например:

Uses M1,M2,

а в обоих модулях используется *f.*

В C++ каждое пространство имен можно рассматривать как интерфейсный модуль.

:: - оператор расширения области видимости в C++.

Если M1,M2 доступны (с помощью include)

std

std::cin

std::endl

Неудобно? Для этого сделано

using namespace std

Но иногда это плохо – когда непонятно это твоя переменная, или из стандарта, или откуда-то еще – код становится плохо читаем.

Читаем значительно чаще чем пишем, поэтому фашистский стиль программирования можно считать нормальным (И.Г.)

Пример (Оберон)

В Модуле-2 было 3 модуля(состоящих из интерфейса и реализации). В Обероне остались только библиотечные.

```
DEFINITION M
END M.
```

```
MODULE M
END M.
```

Куда делся главный модуль? Модуль экспортирует процедуру без параметров – соответствующая процедура называется командой.

Если доступен M, можем писать M.p – выполняется команда из M.

В окончательном варианте языка Оберон Вирт оставил 1 модуль

```
MODULE M
END M
```

Покусился на святое, на некий принцип который доминировал в ЯП того времени (70-80 годы) – разделение определения и реализации, и использования.

Р<-О->И (реализация, определение, использование)

С++ в некотором роде тоже действует по этой схеме: четко разделены определение и реализация. Вирт уничтожил модуль определения – отказался от священной коровы 70х годов.

Любой современный ЯП (впервые в Eiffel) имеет документатор. Документаторы умеют генерировать текст и гипертекст.

Так вот, в Обероне:

`TYPE T* …` *(*- значит имя импортируется)*

`PROCEDURE P*()`

`DEFINITION` – *псевдомодуль, который генерируется компилятором*

В языке АДА пакет – это модуль, интерфейсная часть которого – спецификация пакета.

```
package M
…
end M
package body M is
реализация
begin
…
end M
```

Видимость:

```
package M is
Procedure PRC
...Все имена импортируются далее
END M
...
```

```
PRC доступна здесь (но потенциально)
```

Экспортируемые имена доступны потенциально

Чем АДА дает дополнительную гибкость?

Перегрузка:

```
Function "+" (x,y : T) return T;

a,b,c : T;

C:= a+b;
```

Компилятор c := "+"(a,b)

(Только там где T виден непосредственно).

```
Package Def T i

Type T is …

Function "+" (x,y:T) return T;

End M

A,b,c:T;

C:= a+b; //не знает никакого плюса

C := Def T."+"(b,c)//нужно так – никакого смысла перегрузки операции
```

Важно, что мы смотрим на приоритеты A+B*C

`def T."+"(a, def T."+"(b,c))` // простое выражение в такую штуку? нет уж!

Как решили проблему в АДА? Для использования записей инфиксных операций необходима явная видимость

1. использовать `use def T`

   ```
   a,b,c:T

   c:=a+b;
   ```

   по ТД компилятор поймет какой модуль имеется ввиду(если еще def P)

2. `имя1 renames имя2`

   `имя1` как правило – сложное локальное.

   `std::cout << i << std::endl;` // что не так?

   если бы писали на АДА, то есть противоречие. Нет области видимости оператора <<. В C++ std::<< писать не требуется, хотя << тоже в std.

   std::operator<<(std::cout,i)

Правило C++: при обращении к функции f(a,b) имя f ищется в локальном пространстве имен и в пространствах имен ее фактических параметров. Это соглашение с пространствами имен и перегрузкой операций.

Механизм раздельной трансляции

Этот механизм поддерживают все рассматриваемые ЯП, кроме Ады. Механизм заключается в том, что одному модулю соответствует одна единица трансляции без деления и объединений.

В Аде выделяются первичные единицы компиляции и вторичные единицы.

К первичным относится например спецификация пакета. Ко вторичным – тело пакета, подпрограммы.

На каждую функцию в стандартной библиотеке языка Си приходится отдельный модуль. В Си++ этого уже не нужно. Можно написать «умный» линковщик, делающий объектный модуль с нужными функциями.

Чем отличаются первичные и вторичные единицы компиляции? Первичные – содержат в себе только таблицы имен, а вторичные – это объектный код. Это верно только для односторонней связи. Заметим также, что первичные единицы компиляции можно транслировать независимо от вторичных.

В Аде, в случае раздельной трансляции, есть специальное указание контекста: with.

```
with M; use M;
```

Типичный случай односторонней связи.

В Оберон аналогом with является import.

Двусторонняя связь между модулями

В Ада также есть и двусторонняя связь: вложенные пакеты.

```
package Outer is
      …
      package Inner is
      …
      end Inner;
      …
end Outer;
```

Единица компиляции:

```
package M is
…
end M;

package P is
…
end P;
```

Единица компиляции:

```
with P;

package body M
      P.X
End M;

package body Outer is
      …
      Type OT is …
      Package body Inner is
      OT – тут доступно (демонстрация двусторонней связи)
      …
      end Inner;
      …
end Outer;
```

Вложенный пакет является клиентом своего «папаши».

В принципе двустороннюю связь можно имитировать и на языках Си и Си++. Но только имитировать.

Интерфейс мы не можем разбивать. Реализацию пакета можно оттранслировать отдельно. Как раз это нам и нужно. Естественно потом нужно будет пересобрать.

Отдельно оформим спецификацию пакета:

```
package body Outer is
      <все, что описано тут>
      …
      Package body Inner is separate;
      <видимо тут>
      Procedure S is separate;
      …
end Outer;
```

separate – это в данном случае stub(заглушка)

```
separate (Outer)

package body Inner is
      …
end Inner;

separate (Outer)

procedure S is
      …
end S;
```

В общем случае двустороння связь возникает из-за вложенности. На этом покончим с модульными языками.

Абстрактный тип данных

Интерфейс модуля дает: имя типа, набор операций. Поговорим об абстрактных типах данных. Пример, на языке Ада:

```
Package M is
      Type T is …;
      Procedure Op(X:inout T);
```

И где же тут абстракция? Тип T не инкапсулируется. На самом деле, язык Ада заставляет людей инкапсулировать типы данных.

Рассмотрим пример на Модула-2:

```
DEFINITION MODULE Stacks;
      TYPE Stack;
<Скрытый тип данных>
      PROCEDURE POP(VAR S:Stack; VAR X:INTEGER);
      PROCEDURE PUSH(VAR S:Stack;X:INTEGER);
      PROCEDURE IsEmpty(VAR S:Stack):Boolean;
      PROCEDURE Init(VAR S:Stack);
      PROCEDURE Clear(VAR S:Stack);
      VAR Done:Boolean;
end Stacks;
```

Первичные единицы компиляции можно транслировать отдельно от вторичных. В этом и весь смак. Только когда нам нужен исполняемый код – мы создаем объектный модуль.

```
MODULE Main

      IMPORT Stacks;
      …
      S: Stacks.Stack;
      …

BEGIN

      Stacks.Init(S);
      Stacks.Push(S,1);
      …

END Main.
```

Указатель и целое совпадают в плоских 32 – битных архитектурах. Но, например, в современных 64-битных системах: x86/64 – 32 бита на целое, а 64 бита на адрес. Она специально так сделана, чтобы старые программы выполнялись без существенного замедления.

Пример абстрактного типа данных – файловый дескриптор в системах Unix. Что такое файловый дескриптор? Целое число.

Можно реализовать стек динамически в виде списка, а можно и в виде статического массива.

К скрытому ТД применим следующий набор операций: во-первых, все, что описаны, также операция присваивания (именно указателей, не операция копирования), сравнения на равно/не равно.

```
package Stacks is
      type Stack is private;
      procedure Push(S:inout Stack; X:T);
      procedure Pup(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;
```

```
use Stacks;
…
S:Stack;
X:T;
Push(S,1);
Pop(S,X);
S.top := 1; – ошибка!
```

В программировании различают 2 вида копирования: поверхностное или глубокое. Различаются они только если структура данных является неплоской. В данном случае реализация стека – плоская. Альтернативная реализация – динамический массив. В этом случае структура данных является неплоской, и операция присваивания не пройдет.

Для этого в языке Ада есть специальное слово limited

type Stack is limited private; - целиком закрыть всю структуру данных

В этом случае получается настоящий абстрактный тип данных.

На этом закончим рассмотрение модулей в языке Ада. Начнем рассмотрение классовых ЯП.

Класс

Структура класса в общем случае такова:

Класс:

- имя = имя типа
- набор операций = функции члены (доступ)
- Реализация:
    - Структура данных = набор членов-данных
    - …
    - <возможно что-то еще>

Класс – это ТД и контейнер одновременно. Иногда класс нужен нам только как контейнер. Например, в языке Java есть модуль Math. Существуют «методы доступа»: exp,…. Которые не являются ими на самом деле. В этом случае класс нам нужен именно как контейнер.

C++, C#, Java, Delphi, php, python – всё это классовые языки.

```
class имя {
      <в C++ можно:>
      void f();
      <в Java обязательно:>
      void f(){…}

}
```

Delphi требует от нас разделения определения и реализации – это пошло еще с 70-х годов из объектного Паскаля.

SmallTalk

- класс: экземпляр класса
    - переменные (класса, экземпляра класса)
    - методы доступа (класса, экземпляра класса)

Object – Объект высшего класса (Small Talk, C#, Java, Delphi)

C++: дерево объектов, в котором нет единого корня. Может быть и лес.

В Small Talk и Java есть суперкласс (Object) и подкласс – любой производный класс.

В C++ есть базовые и производные классы(своя терминология).

Я смеялся над терминологией Small Talk'a☺ Почему? С точки зрения представления памяти, под-экземпляр – производный класс больше. Подкласс получается «больше» суперкласса, хотя из названия кажется, что должно быть наоборот.

С точки зрения математики – терминология Small Talk понятна. Любой объект производного является объектом базового. Количество элементов базового класса будет больше количества элементов производного. T1 и T2 - ковариантные ТД – если один из них является подмножеством другого. Очень редко используется.

Члены

- данные(статические)
- функции(статические)
- классы

```
extern int a;
…
int a;
…
a=1;
```

Если отсутствует int a; будет ошибка компоновки

```
class X {
      static int a;
      …
      int X::a = 0;
      ..
}

X x;

x.a; // -можно писать в C++, компилятор не запрещает

X::a; //- нужно обращаться так
```

В Си++ есть просто глобальные функции, а в C# всегда приходится придумывать в какой класс поместить функцию, т.к. глобальных функций нет. Поэтому в C# придумали static class.

```
static import Math.*; //все статические члены класса становятся
видимыми без всякого указания модуля
```

можно писать: exp(x) вместо Math.exp(x)

В Delphi статические функции тоже есть. Вообще говоря, чтобы иметь доступ к приватным членам соответствующего класса.

Статическое данное

Запись – набор переменных

```
class X {
      int a;
      static int foo;
      void f();
      static void g();
};

const X x;

x.foo = -1; //ошибки не будет (статический член к экземпляру отношения
не имеет)
```

Если убрать функцию g из описания класса X, то его размер не изменится.

Если убрать функцию f из описания класса X, то его размер не изменится.

```
class Y:X {
      void f1(){}
      void f2(){}
      void f3(){}
      …
      void f100000(){}
};

sizeof(Y) == sizeof(X)

X * const this;
```

```
X x;
```

Обращение к нестатической функции: x.f();

Члены класса

Что может быть членами класса?

- члены – данные
- методы, члены – функции (статические и нестатические)
- вложенные ТД, из которых больше всего нас будут интересовать классы

Существует ли аналог статических и нестатических членов для вложенных классов?

Да, например (C++)

```
class Outer{

      public:

            class inner{}    //к чему он ближе? К статическому или нет,
                             //как член класса? Конечно к статическому

};
```

Снаружи к Inner обращаться с помощью Outer::Inner,как к любым другим статическим членам. По синтаксису это статический тип данных или вообще не тип? Формальный ответ – посмотрите по контексту. Вообще говоря, не всегда контекст Outer::Inner однозначен. Компилятор может не видеть такую вещь в шаблонах – внутри шаблона мы обращаемся к статическому члену.

```
template <class T…>
class sample{
      …
      T::smth;//контекст может быть не определён
};
```

Как решить, является ли что–то типом или не типом? По умолчанию статический член данных T:: smth (только внутри шаблонов!!). Как разрешается неоднозначность? По умолчанию это член данных, иначе нужно написать typename.

Вложенные ТД (класс или перечисление, или еще что) встречаются достаточно часто, typename можно использовать и вместо ключевого слова class.

Template < typename T…>//Может быть любым типом (например int)

Вложенные ТД принадлежат классу целиком – полный аналог статических членов.

В C# также есть вложенные классы, и они ведут себя как в C++, но есть нечто под названием static (перевод плохой, но и без него бардак). Если присутствует static, то все члены класса должны быть статическими. Естественно, класс не может наследоваться.

Static в Java по отношению к членам класса играет такую же роль, какую играет по отношению к классам. Пишется static перед именем вложенного класса.

Java:

```
public class outer{
      public static class inner{}
}
```

Эквивалентно тому, что мы бы записали на C# с использованием слова static. Static в Java - свойство члена. Только вложенные классы могут быть статическими. В Java понятие static наиболее логично.

А что такое нестатический вложенный класс? Перед ним всего – навсего не стоит ключевое слово.

```
class Outer{

      class Inner{}

}
```

То есть Inner является продолжением своего внешнего класса.

Экземпляр inner находится в контексте какого–то экземпляра Out. Вопрос - а какого?

```
class Outer{

      T a;

      class Inner{a. …}

      this.new Inner();

}//порождает новый экземпляр Inner который привязан к this
```

Это не просто отношение включения (когда объект 1го класса содержит ссылку на объект другого класса). Класс inner всегда имеет смысл, но его this является ссылкой и на объект outer класса.

```
inner.i = new Inner();
```

Можно обратиться и извне

```
outerObj

Outer.Inner obj.new Inner();
```

Создатели java пошли дальше, у них еще появились локальные классы (что, правда, до предела запутывает ситуацию). Однако посмотрим, что такое локальный класс? Локальный класс – это класс, который описан внутри блока, а блок внутри метода, а метод внутри класса (других блоков то нет). Не является в полной мере вложенным классом (не является ни статическим, ни внутренним).

```
{
      class local{
            void f(){…}
      }// является частью блока
}
```

Этот класс имеет доступ к объектам, которые объявлены внутри блока.

Область видимости и область действия

Область действия – часть программы, в которой действуют определяющие вхождения. Как правило, областью действия является блок. Области действия обычно являются вложенными друг в друга. Область действия связывается только с объектами данных, у которых есть имя. Определяющее вхождение имени – объявление (в таких языках, как C, Pascal). Использующее вхождение имени – использование переменной. Обычно определяющее вхождение в статических языках должно предшествовать использующему (за исключением случае forward-декларации).

В некоторых ЯП понятие области действия более широкое.

Область действия может быть больше блока? И как? Да, см. пример ниже.

В Java это актуально из–за наследования, в других языках – бессмысленно.

```
class X(){
      public:
            void f(){…}
}

class Y(){
      public:
            X g (){
                  class local extended
                  …
                  return new local;
            }
}
```

Похоже на концепцию замыкания на языке C#. Переменные объемлющего блока входят в замыкание. Что это облегчает? Если в замыкание входят только читаемые переменные, то при доступе к ним можно не тратить ресурсы на синхронизацию соответствующих потоков.

Как только в java появились lock–внутренние классы, появилась концепции анонимных локальных классов – это анонимные классы, только без имени.

Пример применения внутреннего класса:

Транзакция, т.к. никакая транзакция не имеет смысла вне банковского счета.

В Java отношение внутреннего класса инкапсуляцию не нарушает. Члены класса Outer не имеют доступа к внутренним членам private Inner.

Специальные функции – члены класса. Компилятор знает о них кое–что дополнительное – поэтому они специальные. Компилятор умеет их автоматически вызывать (если б не умел, они не были бы специальными).

Аналогичные члены есть в любых языках, только разное количество разновидностей. Они придуманы для того, чтобы решать несколько общих проблем.

1) Инициализация и уничтожение (конструктор, деструктор, финализатор)

2) Копирование (Конструктор, интерфейсы)
3) Проблема преобразования (операции преобразования)

В чем проблема общей инициализации? Простого обнуления недостаточно. Нужно вызывать конструктор сразу, как только объект разместился. Проблема в обеспечении вызова этой функции. Ни один класс адекватно не решает эту проблему.

Классификация конструкторов (C++):

1. Умолчания X() – когда программе не указано явно, какой конструктор вызывать

2. Копирования X(const X1) – инициализация путем копирования

3. Преобразования X(T) – определяет преобразование из X в T

4. Остальные

`operator имяТипа ()` – находится внутри класса X.

Определяет преобразование из T в X, т.е. обратное. Когда он нам нужен? Почему нельзя для T создать соответствующий конструктор? Как написать конструктор для класса, если он классом не является? Или вдруг нам по каким то причинам вдруг нужно преобразовывать из базового в производный тип. Вообще говоря, поддержка неявных преобразований – вещь неприятная.

Что такое сильная и слабая типизация?

Сильная – избегнуть нельзя (например, АДА – нет преобразований между типами).

Слабая – C и C++ (т.к. сознательно допускаются неявные преобразования).

В начале 80-х была мода на сильную типизацию

`INTEGER`

`CARDINAL`

В C с классами не было никаких явных преобразований. Новые ТД становились неотличимыми от старых. Страуструп добавил только ссылочный тип и надежный ввод/вывод.

Рассмотрим выражение с арифметическими операциями */+-

`a = f(x) * D* k* exp(-i*j)/B`, где f – вещественное, k – комплексное

Как записать использую только префиксную форму?

`*(F(x),*(D,*(k,*(/(exp(-i,j),B))))`

Придется написать еще кучу функций для разных наборов аргументов. Получается слишком много операций, либо явные преобразования. А хотелось бы работать как со встроенными числовыми типами. Если у нас есть 10 типов и 6 операций, то понадобится около 120 функций! Причем они будут тривиальными.

Есть языки, в которых нет тривиальных преобразований: Java, Delphi, классическая Ада, Модула-2 и так далее. Там нет переопределения стандартных операций.

В C# неявные преобразования допустимы только между классами (из одного класса в другой). Преобразователь обязан быть статической функцией.

`static operator X(X a, Y b) {…}` – определяет только явное преобразование.

```
Y y;
```

`(X) y;` – явное преобразование

C++: проблема неявного вызова оператора преобразования.

```
class Vector {
      T* body;
      int size;
      public:
            explicit Vector(int size); – оператор явного преобразования
            ~Vector();
            T& operator [](int i);

};

Vector V(128);

v = 0; // если не писать explicit, то компилятор не выдаст ошибку
```

В C# есть два ключевых слова – explicit и implicit. В C++ следовало по умолчанию выбрать implicit, но переделывать не стали из-за слишком большого объема уже написанного кода. В C# нельзя переопределить = , так как копируются ссылки, а не объекты. [], () тоже нельзя переопределять, что неудобно. Для обхода этого «прибили» индексатор.

```
class WithIndexer {
      T this [int index] {
            get {…}
            set { value: …}
      }
}

WithIndexer c;

c[0] = 0; // вызывается set
a = c[1]; // вызывается get
```

Типы индекса могут быть самыми разными:

В языках со ссылочной семантикой ситуация проще. Таких языков значительно меньше. Проблемы копирования решаются с помощью других средств.

`a = b` – вообще говоря, присваивание ссылок

`X a(b) ~ X a = b ~ a = b` (в ссылочных языках)
Остается конструктор умолчания – единственный конструктор, который вызывается неявно, все остальные вызываются явным образом.

`X (…): … {…}`

Системная часть (генерируемая) – код, который автоматически генерируется компилятором.

Пользовательская часть – {…}.

Все подобъекты в ссылочных языках – ссылки. Допустимы инициализаторы такого вида:

```
class X {
      Y a = new Y();// такой инициализации чаще всего хватает
      X(){…}
};
```

Вначале вызываются конструкторы подобъектов (автоматически). Существует только 1 случай, когда нужно вызывать явно конструктор – вызов конструкторов базовых классов. Любой конструктор начинается с того, что вызывает конструкторы базовых классов, тот своего базового, и т.д. – Object.

В Java:

```
X(…){super(0);…}
```
//терминология SmallTalk

```
X(…){super();…}
```
// такой код генерируется автоматически

```
X(…): base(0)

      this(0,0)
```

//синтаксический сахар в стиле C++

Все остальные конструкторы – явные: new X(…)

За счет ссылочной семантики сами языки значительно упрощаются.

Копирование: универсального способа решения нет.

Уничтожение. Особняком стоит Delphi:

```
type X = class constructor Create;

           destructor Destroy;

           end;
```

Названия такие, потому что такие же названия в классе TObject, мы просто переопределяем.

```
X a;
a = X.Create;
```

Возникает очень интересная вещь в Delphi. Все конструкторы вызываются только явным образом. Компилятор языка Delphi никогда не генерирует автоматически вызов конструктора.

```
constructor X.Create;

begin
```

```
      inherited Create;//вызывается соответствующая функция из базового
класса

      …

end;
```

В других языках используется синтаксис C++: X(), ~X(). Автоматически генерируется конструктор умолчания, если нет другого конструктора.

Дополнительные конструктор/деструктор:

```
Load
```

```
Save
```

Для сериализации объектов (сохранения объектов во внешней памяти).

```
static int i = 0;//C++ запрещает
```

Что такое виртуальные функции? Это функции, вызов которых динамически связывается с телом.

```
X.f();
```

```
X->f();
```

```
//«Дайте мне X и я переверну Землю.»
```

Только через указатель или ссылку можно вызвать виртуальную функцию.

Во всех языках со ссылочной семантикой существует понятие метакласс – класс, который работает с произвольными классами. Метаклассы ипользуются для цели рефлексии. С ее помощью можно получить имя типа. Рефлексия – развитие идеи динамической идентификации типа.

Программными средствами языка можно реализовать динамическую подгрузку модулей. В отличие от динамически загружаемых библиотек, проверяется корректность сигнатур.

```
constructor Create; virtual;//описание в классе TObject
```

Заметим, что в других языках, в которых конструкторы не бывают виртуальными, конструктор – это некая виртуальная функция, которая генерирует объекты определенного типа.

```
virtual X*MakeX(); //виртуальный конструктор в языке Delphi
```

```
~X(); - в C# зачем-то ввели такую штуку, на самом деле она сбивает с толку
```

В языке C++ все просто, но за это у нас нет автоматической сборки мусора.

Если язык имеет ссылочную семантику, то как отловить момент, когда удалять объект?

В Delphi есть метод Free, он вызывает деструктор объекта, а затем освобождает память.

```
a.Free;
```

```
a:=nil;
```

RAII: Resource Acquisition Is Initialization: захват ресурса есть инициализация.

RAII => X(); - захват, ~X(); - освобождение

RAII методика удобна только для коротко живущих объектов.

Пример (C# или Java)

```
try {
}

finally {
}

try
<операторы>

finally
<операторы>

end
```

Фактически, с помощью finally части мы можем сами вызывать свертку стека.

В Java:

`protected void finalize();` – автоматически вызывается тогда, когда объект перестает существовать

В библиотеке .Net есть класс Image. У него есть шикарный статический метод `Image.FnewFile("f.jpg");`

Чем плохо? Есть блокировка файлов, например, пока мы его считываем, то блокируем от записи. Пока мы работаем с этим образом, он не будет изменен. Блокируется в деструкторе, а тот будет когда вызван? Никогда. Ну естественно когда-то то будет вызван, но пользователю то об этом не скажешь.

Интересно, что как только в языке появляется сборка мусора, появляется метод для удаления объектов.

```
protected void finalize() {
      Close();
      Dispose();
…
}
```

Динамическая сборка мусора помогает только в очень простых программах. Часто вызывает непредвиденные накладные расходы.

Надежные отказоустойчивые сервера пишут только на C/C++.

ILDASM – программа, предоставленная Microsoft.

Любой компилятор .Net генерирует промежуточный код, а дизассемблировать его можно с помощью этой утилиты. Если мы дизассемблируем деструктор, то вызывается приватный метод класса Object – finalize().

```
FileStream fs = new FileStream();
try {
}
finally {
      fs.Close();
}
```

```
IDisposible:
void Dispose();
```

```
FileStream fs = new FileStream();
try {
}
finally {
      ((IDisposible)fs).Dispose();
}
```

Большинство программ имеют соответствующую структуру.

```
Using (v = expr){

      блок

}

v = expr;

try {
}

finally {
      ((IDisposible)v).Dispose();
}
```

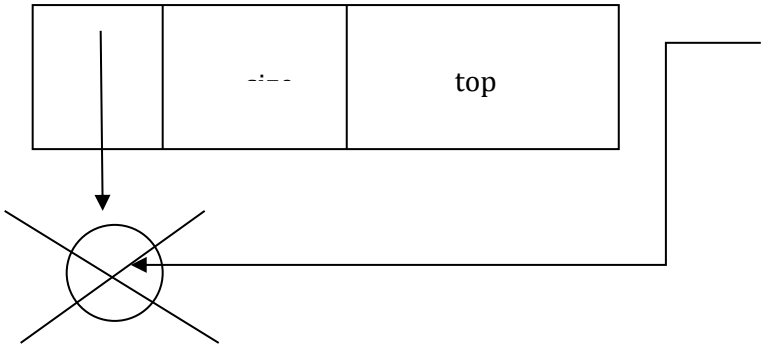
В языке C++ этого нет, нужно пользоваться методикой RAII.

Во всех остальных языках она есть, фактически является заменителем свертки стека.

```
X a = b;

a = b;
```

`Copy(); shallow copy` – поверхностное копирование (для плоских объектов)

`Clone(); deep copy`



Java:

```
protected Object clone();
C#: protected Object
      Memberwise Clone();
```

NB: ICloneable – безграмотно с точки зрения английского языка, правильно было бы IClonable

```
ICloneable
      Object Clone();
```

Нельзя сказать глубокая или неглубокая была копия. В Java была попытка более ловко решить проблему клонирования.

Microsoft советует: Если Вы хотите реализовать копирование – то называйте интерфейс как хотите, но опишите подробно в комментарии. Не надо реализовывать ICloneable.

Cloneable в Java - пустой интерфейс.

```
void f() throws (x,y,z){…}

throw x,y,z
```

Если в программе возникает исключение какого-либо другого типа, то выполнение программы прерывается.

В Java такая ситуация, что выскочит что-то кроме x,y,z, невозможно.

В Java введено 4 уровня поддержки операции копирования:

1) Полная поддержка
   ```
   class X implements Cloneable {
         public X clone() throws(){…}
   }
   ```
   Пользоваться таким классом безопасно, он делает честную копию.
2) Условная поддержка
   ```
   class X implements Cloneable {
         public X clone(CloneNotSupported) throws(){…}
   }
   ```
3) Условная неподдержка
   Не реализуется Cloneable. Извне не доступен.

4) Запрет
   Не реализует интерфейс. От клонирования остался Protected Object Clone(). Перекрывается: throw new CloneNotSupported(); - меня копировать нельзя. Доступен производным, а также всем классам из данного пакета. Сволочь, которая может вас скопировать – это классы из Вашего пакета, либо наследник.

В общем случае это говорит о сложности проблемы копирования. Проблемы у программистов на C# такие же.

Абстрактные ТД в ОО ЯП

В C#, Java, Delphi есть спецификатор abstract, который определяет задание абстрактного класса.

Что означает инкапсуляция с точки зрения понятия класса? И (чисто формально) как будет выглядеть модуль АТД с точки зрения инкапсуляции? Определяется ТД и множество операций, причем понятие операций сводится к понятию функций, т.е. определяем ТД и соответствующие функции. От чего мы абстрагируемся? От структуры ТД.

Обсудим, как инкапсуляция организована в классах.

Закрытость членов классов для доступа извне реализуется при помощи управления доступом и управления видимостью. Что это такое?

Управление видимостью – говорим, что некоторые члены класса видимы.

Есть некий класс. С точки зрения инкапсуляции все члены равноправны:

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

доступ к i и j полностью открыт и равноправен. А как же осуществляется доступ с точки зрения внешних функций (глобальных функций или других методов)?

В Java, например, нет глобальных функций, есть только методы. В C#, Java существуют также механизмы, описывающие совокупности классов.

Необходимо рассмотреть, что в разных ЯП значат такие понятия, как единица дистрибуции и единица области действия(в данном случае имени).

В C# единица видимости – пространство имен namespace(из C++).

```
using namespace …;
```

Все имена, доступные в пространстве имен, становятся непосредственно видимыми.

В C# есть понятие частичных классов. Частичные классы – конструкция, позволяющая определить один класс в нескольких файлах. Зачем нужен частичный класс? Чтобы отделить то, что сгенерировано, и трогать не нужно, и то, что трогать можно. Второе выделяется в отдельный класс.

```
partial class …{…}
```

1 файл генерируется программными средствами, а второй для изменяемости – чисто технологичная штучка. Ничего с точки зрения принципиального, разделение не дает.

В Java появилось понятие пакета (package).

```
package имя_пакета;
```

Обязательно ли писать package и using? Package обязательно (в отличие от using)!

Как использовать пакет? Можно с помощью

```
import имя_пакета *; //все имена из пакета
```

Или

```
import  пакет.имя; // если нужно только 1 имя
```

А что же такое все-таки получить доступ к соответствующей дистрибуции?

Java снова упрощает вещи. Это не всегда хорошо, но в данном случае это удобно.

В Java сказано, что пакет = единица дистрибуции

Класс не является единицей дистрибуцией, только пакет! Чтобы использовать классы из пакета надо обратиться к соответствующему jar.

В C# все запутанней.

Можно одно и то же пространство имен «размазывать» по сборкам. Пространство имён и пакеты иерархичны, и это существенно с точки зрения видимости и поиска. Но к вложенности файлов не имеет отношения. Можно несколько пространств имен поместить в 1 сборку или разнести («размазать») по нескольким.

Что такое сборка? Сборка – двоичный файл, содержащий управляемый код. Вообще это достаточно рыхлое понятие, но понятие инкапсуляции вводится именно в терминах сборки.

При управлении видимостью: инкапсуляция говорит о том, что члены классы невидимы для других классов.

При управлении доступом: любые имена видимы (правда есть скрытие), а если имя видимо, то оно может быть доступно или недоступно.

В C++ реализовано управление доступом, в Java и то, и то (когда речь о пакете, то управление доступом, но когда речь о разных пакетах, то управление видимостью).

В чем тонкости?

Рассмотрим на примере поиска имен (C++): берется использующее вхождение, нужно найти определяющее вхождение.

Как будем искать? Сначала в блоке, если в нем не нашли, смотрим в объемлющем блоке – например, базовом классе, если в нем не нашли, ищем еще в объемлющем блоке, например, пространстве имен. Так, рано или поздно, дойдем до корня, тогда уже точно это будет namespace. Когда будет найдено определяющее вхождение, поиск прекращается.

Специфика управления доступа: нашли имя – остановились, даже если доступа нет, имя все равно видимо.

При управлении видимостью – проскакиваем мимо имени.

```
int f;

class X{
     private void f()
}

class D:X{
     f = 0;// то, что есть функция f из X никого не волнует
}
```

Правило доступа определяет доступ.

В Java:

```
package p;

class X{
     private void f()
}

// если тут разделение пакетов, то все private имена недоступны

import p.X

class D extends X{
     public void f(){…}
}
```

Если всё в 1 пакете, то ситуация вообще говоря нехорошая:

```
package p;

class X{
     private void f()
     public void g() {f();}//в C++ такое невозможно
}

import p.x

class D extends X{
     private void F(){…}
     public void smth{g();}// вызовется f из этого класса(из-за
                           // виртуальности)
}
```

В C# такие вещи вообще запрещены, т.к. нельзя замещать приватную функцию.

Мораль: нельзя делать приватные виртуальные функции в C#.

А в Java можно?

Если всё в 1 пакете, то можно, а если разделение пакетов, то нет.

F замещает f? Нет, потому что первой f не существует для Java - другой пакет!

G вызовет f из ее класса («даже если мы заместим в другом классе из другого пакета, то мы ее не заместили»).

Важно понятие, что такое другой класс. Рассмотрим на примере C++. Основные типы членов класса:

- `private` - доступ открыт самому классу (т.е. функциям-членам данного класса) и друзьям (friend) данного класса, как функциям, так и классам
- `protected` – доступ открыт «своим» + наследникам
- `public` – доступ открыт кому угодно

Друзья(friend) – всего лишь «нашлепка». Основные первые три.

Чем плоха эта модель? Если есть совокупность взаимодействующих классов, то она слишком жесткая.

В C#, Java вводится дополнительный уровень доступа `internal`, в Java пакетный (слова зарезервированного для него нет). Если в Java опущено явное слово, то это пакетный уровень доступа. `internal` – разрешается доступ всех классов из одной единицы дистрибуции (!) Пакетный – разрешается доступ к любым классам из пакета (в C# из сборки).

«Неудобно когда пространство имен включает другие пространства имен, которые оказываются в другой сборке – но это лично моё мнение» – И.Г.

`public` перед именем члена означает доступ к этому члену из любого другого класса (класс-наследник, из данной единицы дистрибуции и из других единиц дистрибуции).

`private` существует во всех языках, и обозначает(ура!) одно и то же –доступ только из данного класса.

Осталось понять, что такое protected.

В C++ доступ из любых классов наследников.

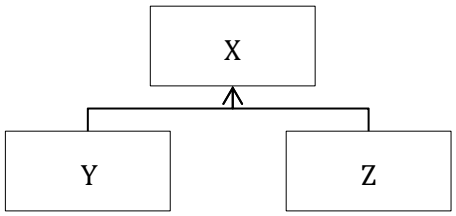
В Java protected доступны с точки зрения любых производных классов независимо от пакета, НО! Ограничение: доступ только через ссылку на этот же класс.

```
class X{

      protected prot;

}

class Y extensed X{
```

```
      prot//через ссылку на Y или его наследника

}
```

C++, C#:

Рассмотрим иерархию классов




```
void Y::f{ Z z;
           z.prot;
           }
X x;
x.prot;
```

В Java через ссылку this

```
X x = new X();
x.prot;
```

В C# по сравнению с Java семантика protected как в C++.

Но есть еще уровень `protected internal`.

Это уровень объединения доступов, т.е. доступ к членам класса имеют наследники класса или любой класс из данной единицы дистрибуции, т.е. доступ из наследников и из пространства имен.

Если речь об объединении можно было бы сделать и пересечение. Оно и сделано в C++/CLI в .Net, а также в MSIL, но в C# такого уровня нет.

Как же реализуются АТД? (не следует путать с абстрактными классами).

Чисто формально, АТД эквивалентен публичному интерфейсу (только методы).

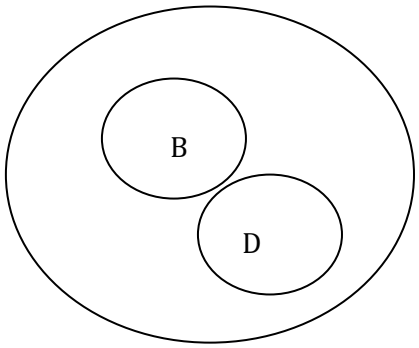
В ООП есть понятие абстрактного класса, правда то понятие не имеет отношения к АТД.

НО! Есть понятие «интерфейс» (понятие чисто языковое), вот оно то имеет большое отношение к АТД!

Что такое абстрактный класс?

Есть понятие мощности для бесконечных множеств (например, можно говорить,что Q принадлежит множеству всех целых чисел).

Абстрактный класс: чисто виртуальные функции (ЧВФ).



B – базовый, D – производный.

В классе B: `void f()`

В классе D:`void f(int)` – не есть перегрузка(разные области видмости), это скрытие (hiding)

К f из B можно обратиться `B::f()` или `B.f()`

В другом случае:

В классе B: `virtual void f()`

В классе D: `void f()` – функция-заместитель

Ковариантные типы - это T1 и T2 такие, что T1 – подмножество T2

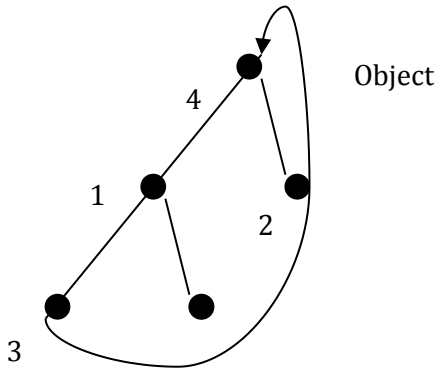
Производный тип всегда ковариантен базовому.

Контрвариантные типы – это (наоборот) T1 и T2 такие, что T2 – подмножество T1

Инвариантные типы - этоT1 и T2 такие, что T1 и T2 не ковариантные и не контрвариантные.

В C++ легко создать инварантные ТД.

В C#, Java:



1 и 2 – инвариантные, на разных уровнях иерархии

3 и 4 – ковариантные

Заместитель: имеет тип, ковариантный замещаемому. Совпадающий тип ковариантен самому себе.

Статический ТД	Динамический ТД
(дан при объявлении)	(ссылки, указатели)
	динамический тип ковариантен статическому

Виртуальный вызов

Если функция вызывается через указатель(ссылку)

```
X *px;

px -> f();
```

Если f() объявлена, как virtual, то вызывается заместитель, который ближе к динамическому типу.

X – f()

…

… ← Цепочка классов-наследников

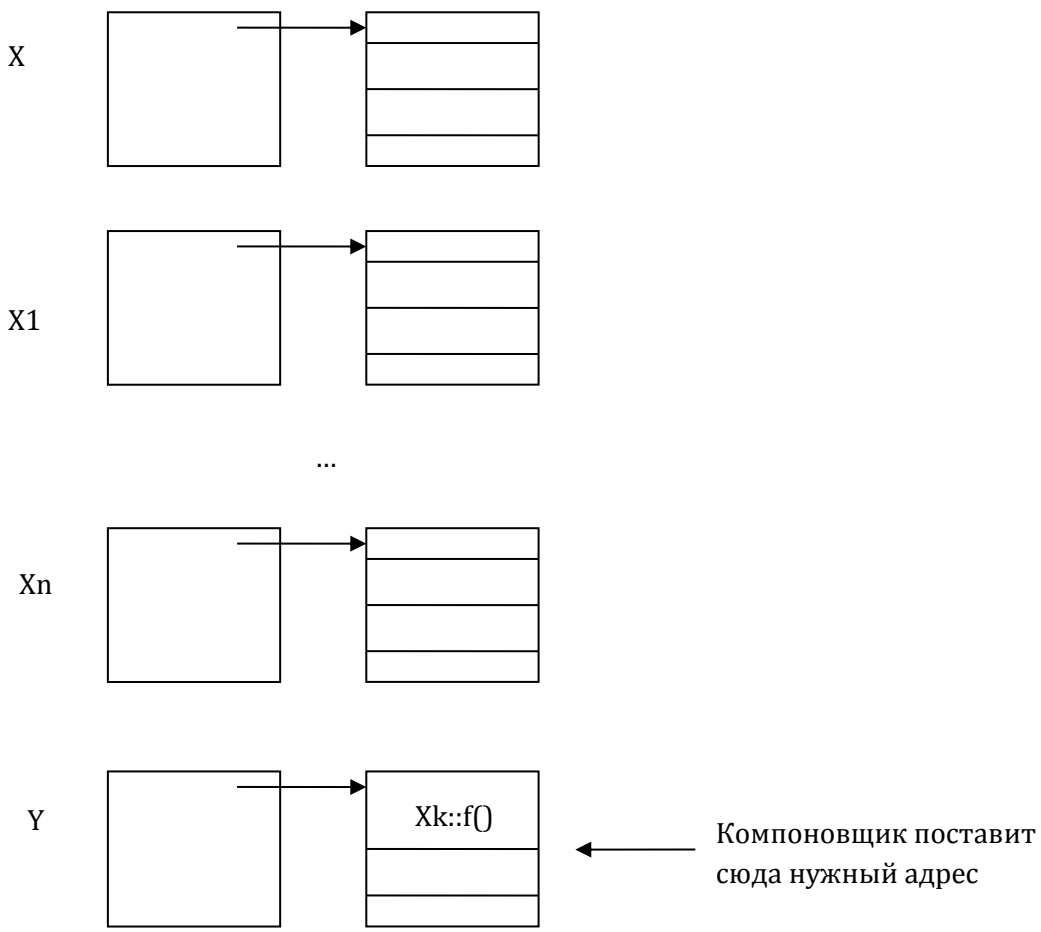
…

Y

Снизу вверх будет произведен выбор ближайшей функции

В C++

На самом деле поиска заместителя производиться не будет, выберется из таблицы виртуальных методов:



А вот в SmallTalk произойдет настоящий поиск. В нем все методы виртуальные, поэтому язык крайне неэффективен. Зато можно дать возможность программисту выбрать какую вызвать функцию. Язык гибкий (примерно также в JavaScript).

В C++ как снять виртуальность?

```
Px -> X::f();

{ x.f();}

void g(X&a);
      a.f(); - вызов виртуальный
X x;
Y y;
X =>Y
g(x);
g(y);


void g(X a)
      a.f()
X x;
Y y;
X =>Y
```

```
g(x);
g(y)
// виртуального вызова нет
```

На C++ «харакири» нам сделать никто не помешает:

```
a = new X();
```

`a.f();` – компилятор снимает виртуальность

`final` – запрет замещения

```
final class X{}
```

`final void f{}` – все вызовы невиртуальные

`X f()` – виртуальная

`Y f()` – не может быть невиртуальной, т.к. это заместитель

В Java невиртуальных функций просто не бывает. В C# и Delphi мы можем как заместить, так и не замещать.

```
class X{
      public virtual void f(){…}
}
```

```
class Y:X { … public void f(){…}…}
```

В C# нужно явно поставить ключевое слово `override`.

Если его не написать, то компилятор будет выдавать предупреждения, которые можно «заткнуть», только поставив слово new.

Довольно хорошая практика: все предупреждения должны трактоваться как ошибки.

Если в определении языка описано, что какую-то инструкцию можно употреблять, не значит, что ее нужно употреблять.

Мы можем поставить virtual после new, но это означает, что виртуальность начинается заново с этого уровня.

Собственно все про механизм виртуальности. Вернемся к понятию абстрактный класс (АК).

Вспомним пример – класс Figure.

```
class Figure {
      protected:
            int x,y
      public:
            virtual void Draw();
            void Erase();
            void Move(int dx, int dy) {
                  Erase(); x += dx; y += dy;
                  Draw();
```

```
            } //непонятно как будет работать универсальный метод Erase
};


class Point: public Figure {
//    переопределяем Draw()
      …
}

list <Figure*> objects; // статический список всех объектов
foreach(objects.begin(),objects.end(),Draw);//метод reDraw
```

Мы не можем написать реализацию метода Draw для произвольной фигуры. Это хороший кандидат на виртуальный метод.
При таком описании произойдет ошибка. Нет реализации метода Draw в классе Figure.

Ошибку выдаст компоновщик.

Для этого нужны абстрактные методы.

virtual void Draw() = 0;

Появилось понятие чисто виртуальной функции. У нее необязательно должно быть тело. Компилятор запретит создание объектов такого типа.

Любой ОО язык поддерживает концепцию абстрактных классов.

В языке C# и Java определение абстрактного класса:

```
abstract class
public abstract virtual void Draw();
```

Абстрактный класс – это класс, перед описанием которого написано слово abstract.

Объекты абстрактного класса заводить нельзя.

В классе Figure лучше всего сделать все 3 функции виртуальными.

Когда мы проектируем некоторую иерархию, то лучше всего на верхнем уровне сделать класс, все члены которого – публичные чисто виртуальные (абстрактные) функции. Такие классы называют интерфейсами.

Если в классе есть виртуальные методы, то деструктор должен быть виртуальным. Если он будет невиртуальным, то при работе через указатели:
```
Figure* pfig;

delete pfig;
```

будет вызван не тот деструктор.

```
virtual ~Figure();
```

В 99,9% случаях, если в классе есть виртуальные функции, а деструктор является невиртуальным, то это ошибка.

Рассмотрим теперь множество (set).

```
class ISet {
      public:
            void Incl(T x) = 0;
            void Excl(T x) = 0;
                  …
            Virtual ~ISet() {…};

};

Class Keys: public ISet,
            private set <string> {
      …
};//это и есть в чистом виде инкапсуляция
```

Интерфейс – чистый контракт. Содержит только заглушки для методов.

В языках C# и Java есть понятие interface.

```
interface имя {

      объявление метода; (или константы)

}

class X extends:Y  implements список интерфейсов

class X: Y, I1, I2,…,IN {…}
```

iostream – одна из первых иерархий, в которой нет ни одного виртуального метода.

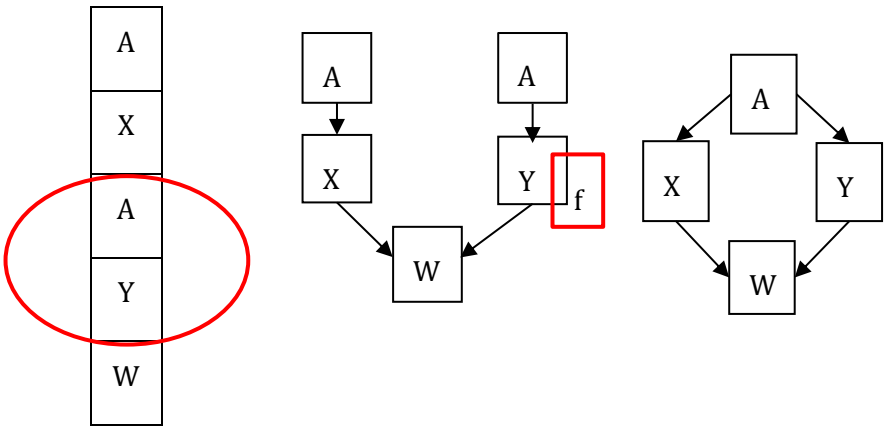
```
ios, istream, ostream, iostream, fstream, strstream
```

Но большинство других иерархий требует наличия виртуальных, абстрактных методов, классов, функций.

Фабричный класс – методы: создать меню, создать окно, …

Программирование с помощью интерфейсов – легкий способ создания абстрактных классов. Эту реализацию поддерживают почти все языки (может быть, кроме Delphi).

Что плохого в множественном наследовании? С конфликтом имен как-то можно разобраться. В языке Objective C реализовать множественное наследование не удастся:




```
W * pw = pw -> f(); //Какой адрес должен быть передан?
X* px; Y* py;
px = &w; py = &w;
px->f();//будет работать
py->f();
```

В таблицу виртуальных функций добавляется еще одно поле – дельта (насколько смещать указатель this). Появляются дополнительные накладные расходы.

В ромбовидном случае:

```
class w: public X, public Y {…}

class X: virtual public A,

      Y: virtual public A
```

Вот поэтому современные языки множественное наследование не поддерживают, они поддерживают интерфейсы.

`Draw` – нарисовать

`Draw` – раздать карты

Java: выбрать тот метод, что нравится, другой переименует.

В C# делается так:

```
I1:

      Process();

I2:

      Process();


class X: I1,I2 {
      Public void Process(){…}// неявная реализация интерфейсов
}
```

```
class X: I1,I2 {
      void I1.Process(){…}// явная реализация
      void I2.Process(){…}
}

X x = new X();
I1 i1 = x;
i1.Process();


((I1)x)Process();



class X:IEnumerable<T> {
      public T GetEnumerator<T>(){…}
      Object IEnumerable.GetEnumerator(){…}
}
```

Список исправлений (версия 12 января)

с. 5 – исправлена схема архитектуры ЭВМ.

с. 6 – исправлен пример на C (название константы, переменная i определяется вне цикла) и описание этого примера.

с. 7 – исправлен абзац про прототипные языки

с. 7 – исправлен абзац про RAD

с. 8 –поправлен заголовок

с. 9 – убран непонятный абзац про C++

с.9-11 – исправлен синтаксис в примерах Lisp

с. 41 – исправлено предложение «Область действия в классических языках программирования совпадает с областью действия».

Спасибо за исправления Владу Шахуро, Михаилу Старцеву и Александру Фомину.