Основные понятия ЯП - переменные

Переменные - абстракция объекта данных

Основные атрибуты данных

- имя
- тип (данных)
- значение
- время жизни
- область видимости (для именованных переменных)
- область действия
- (для императивных) адрес.

Основные понятия ЯП - переменные

Переменные - именованные и анонимные

Пример анонимных объектов - объекты в динамической памяти

Референциальные языки (C#, Java, все динамические ЯП - JS, Python,...) - вводится понятие референциальных типов, именованными могут быть только ссылки на объекты этих типов, а сами значения (объекты) этих типов - только в динамической памяти.

Референциальные типы: массивы, классы, интерфейсы.

В динамических ЯП - любая именованная переменная - это ссылка на значение (неважно, где - в ДП или таблице значений....)

Основные понятия ЯП - переменные

Переменные: типы данных и значения

Статические языки: имя и тип связываются в спец. конструкциях - объявлениях переменных, изменить это связывание далее уже нельзя (как правило).

Автовыведение типа (C++, C#, Go....) - обязателен инициализатор

C++:

```
int i; string s;
auto j = i;
auto ss = s;
auto& css = s;
const auto& crs = s;
```

Основные понятия ЯП - переменные

Переменные: типы данных и значения

Статические языки: имя и тип связываются в спец. конструкциях - объявлениях переменных, изменить это связывание далее уже нельзя (как правило).

Автовыведение типа (C++, C#, Go....) - обязателен инициализатор

C#:

```
int i; string s;
var j = i;  // тип j - int
var ss = s;  // тип s - string
// аналогов последних двух объявлений C++ нет - почему?
```

Основные понятия ЯП - переменные

Переменные: типы данных и значения

Статические языки: имя и тип связываются в спец. конструкциях - объявлениях переменных, изменить это связывание далее уже нельзя (как правило).

Автовыведение типа (C++, C#, Go....) - обязателен инициализатор

Go:

```
f, err := os.Open("myfile.txt")
if err != nil {
    log.Fatal(err)
}
i := 1
// про ссылки также нет, так как их нет (значения и указатели)
// правда, есть интерфейсы...
```

Основные понятия ЯП - переменные

Динамические языки - имя связывается со значением любого типа динамически - при присваивании (инициализации). Конструкция "объявление переменной" либо вовсе отсутствует, либо присутствует для надежности и читабельности, но не содержит никаких ссылок на тип.

В чисто динамических языках нет понятия статического типа в программе (хотя сами ТД конечно есть).

В императивных языках значение связывается с переменной в специальной конструкции - операторе присваивания, для констант - при инициализации

В функциональных ЯП значение связывается с переменной при инициализации (сравним с ИЯП).

Основные понятия ЯП - переменные

Переменные - время жизни

ИЯП:

- статические переменные - пока работает программа
- локальные переменные - пока активен блок (подробнее - ниже)
- динамические переменные - любое...

Основные понятия ЯП - переменные

Переменные - время жизни

ИЯП:

- статические переменные - пока работает программа
- локальные переменные - пока активен блок (подробнее - ниже)
- динамические переменные - любое...

Классы памяти (C, C++) <=> время жизни

Основные понятия ЯП - переменные

Переменные - время жизни

ИЯП:

- статические переменные - пока работает программа
- локальные переменные - пока активен блок (подробнее - ниже)
- динамические переменные - любое...

Классы памяти (C, C++) <=> время жизни

Классы памяти <=> время связывания атрибута "адрес"

А какое время жизни отсутствует?

Основные понятия ЯП - переменные

Переменные - время жизни
ИЯП:

- статические переменные - пока работает программа
- локальные переменные - пока активен блок (подробнее - ниже)
- динамические переменные - любое...

Классы памяти (C, C++) <=> время жизни
Классы памяти <=> время связывания атрибута "адрес"
А какое время жизни отсутствует?
Вечно? Ну хотя бы между запусками программы...
Persistent objects - только в библиотеках/API.
В базисе ЯП - пока нет... (зависимость не только от ОС, а от конфигурации ВС)

Основные понятия ЯП - переменные

Переменные - время жизни

Функциональные ЯП:

пока активен блок (тело вызова функции), либо

- локальные переменные - пока активен блок (подробнее - ниже)
- динамические переменные - любое...

Классы памяти (C, C++) <=> время жизни

Классы памяти <=> время связывания атрибута "адрес"

А какое время жизни отсутствует?

Вечно? Ну хотя бы между запусками программы...

Persistent objects - только в библиотеках/API.

В базисе ЯП - пока нет... (зависимость не только от ОС, а от конфигурации ВС)

Основные понятия ЯП - переменные

Три понятия: время жизни, область видимости и область действия - очень тесно связаны друг с другом.

Время жизни - обсудили (но вернемся к нему, когда будем обсуждать область действия)

Область видимости (scope) - относится, скорее, к именам, чем к объектам, например, переменным.

ОВ имени - часть программы, в которой имя связано с одной и той же сущностью (переменной, константой, типом, подпрограммой и т.д.).

Основные понятия ЯП - переменные

Три понятия: время жизни, область видимости и область действия - очень тесно связаны друг с другом.

Время жизни - обсудили (но вернемся к нему, когда будем обсуждать область действия)

Область видимости (scope) - относится, скорее, к именам, чем к объектам, например, переменным.

ОВ имени - часть программы, в которой имя связано с одной и той же сущностью (переменной, константой, типом, подпрограммой и т.д.).

В каждом языке - свои ОВ - их нужно знать обязательно.

Что общее - проективность ОВ с точки зрения вложенности. 2 различные области видимости - либо вложены одна в другую, либо не пересекаются. "Нахлестов" нет.

Основные понятия ЯП - видимость имен

Пример 1 - С

Области видимости:

- файл
- блок
- структура
- глобальная область видимости (все имена нестатических переменных , ОВ которых - файл + все имена нестатических функций)

ОВ функций? Только файл (вложенных функций до сих пор нет - только в диалектах)

Проблема в С (особенно с точки зрения С++) - видимость тэгов структур.

Тэги структур - отдельная (файловая) область видимости, независимо от вложенности объявлений.

Нет единообразия с точки зрения видимости имен.

Основные понятия ЯП - видимость имен

Пример 2 - C++

Области видимости:

- файл
- блок
- класс (структура)
- пространство имен
- глобальное пространство имен (все имена нестатических переменных , ОВ которых - файл + все имена нестатических функций) - для совместимости с C

ОВ функций? Только файл (вложенных функций до сих пор нет - только в диалектах)

ОВ функций-членов - класс.

Для совместимости с C - struct имя {}; - имя может совпадать с именем неклассовых сущностей (функций и переменных). В остальном имена классов (структур) подчиняются правилам вложенности.

Основные понятия ЯП - видимость имен

Вложенные области видимости:

скрытие имен - имя сущности из вложенной ОВ скрывает то же самое имя сущности из объемлющего блока.

В каком случае происходит (или не происходит) скрытие, можно ли получить доступ

ВНИМАНИЕ!!!

Проблемы с видимостью имен - одни из самых запутанных в ЯП.

Пример: Ада

Основные понятия ЯП - видимость имен

Вложенные области видимости:

скрытие имен (hiding) - имя сущности из вложенной ОВ скрывает то же самое имя сущности из объемлющего блока.

В каком случае происходит (или не происходит) скрытие, можно ли получить доступ - все индивидуально

ВНИМАНИЕ!!!

Проблемы с видимостью имен - одни из самых запутанных в ЯП.

Пример: Ада

Некоторые (но отнюдь не все!) источники проблем:

- перегрузка имен
- перегрузка станд. операций (+, -, << и т.д.)
- неявный импорт имен
- в ООЯП - взаимодействие видимости и управления доступом

Основные понятия ЯП - видимость имен

Пример проблем и неочевидных решений (C++):

```
namespace NS {
   struct T {};
   void Foo(T x, const char * s);
   T a;
}
Foo(a, "Hi!");; // очевидно ошибка
NS::T b ; // ОК
Foo(NS::a, "Hi!");  // очевидно что? - ошибка? - NS::Foo(NS::a, "Hi!");
Foo(b, "Hello!");  // ошибки нет!!! Можно не писать NS::Foo(b, "Hello!");
```

Основные понятия ЯП - видимость имен

Тот же самый пример, но другие имена(C++):

```
#include <iostream>
....
cout << "Hello, world\n"; // очевидно ошибка
```

Основные понятия ЯП - видимость имен

Тот же самый пример, но другие имена(C++):

```
#include <iostream>
....
cout << "Hello, world" << endl; // очевидно ошибка
```

Исправление:

```
std::cout << "Hello, world\n"; // очевидно нет ошибки
```

НО:

```
operator << (std::cout, "Hello, world\n");
```

Основные понятия ЯП - видимость имен

Общее правило C++: поиск имени функции ведется не только в текущей области видимости (пространстве имен), но и пространствах имен, где определены типы аргументов функции.

Зачем? Для возможности использовать без уточнения перегруженные стандартные знаки операций.

Другое решение проблемы - переименование (Ада).

прототип1 RENAMES прототип2;

Основные понятия ЯП - видимость имен

Имя: определяющее вхождение-использующее вхождение

Конструкция, в которой происходит связывание имени с сущностью, - *определяющее вхождение имени*

Остальные вхождения - использующие.

(Пролог - Рефал - там есть разница?)

Определяющее - только одно, использующих - любое количество

Перегрузка - в одной области видимости (обязательно одна!!!) несколько определяющих вхождений одного имени.

Везде, где есть перегрузка имен, работает правило - перегружаемые имена могут быть только у ОДНОГО вида сущностей - ПОДПРОГРАММЫ (функции).

Также - разрешение перегрузки - только статически (по типам фактических параметров) => она есть только в статических языках.

В динамических языках - подпрограммы с переменным числом параметров (сама подпрограмма динамически "разбирается" с ними)

Основные понятия ЯП - видимость имен

2 проблемы

- что есть определяющее вхождение?
- как по использующему вхождению найти определяющее?

Основные понятия ЯП - видимость имен

Первая - что есть определяющее вхождение (ОпрВ)?

Статические языки - нет проблем

Объявление (определение) - и есть ОпрВ.

Функциональные ЯП (неважно - статические или динамические) - тоже нет проблем - либо явное объявление с возможным автовыведением типа (статические ФЯП) , либо инициализация (динамические ФЯП)

А вот динамические императивные ЯП - вот тут и проблема...

Либо вводим специальное объявление (JS - var), либо - первое присваивание. В чем проблема?

Что считать первым присваиванием?

Основные понятия ЯП - видимость имен

Пример 1 - Python

ОВ в Python - модуль (глобальная ОВ), тела функций (локальные ОВ)

Пример глобальной переменной

```
x = "this is global X"  # это - определяющее вхождение
def foo():
        print(x) # это - использующее вхождение
foo()
```

можно и так:

```
def foo():
        print(x) # это - использующее вхождение
x = "this is global X" # это - определяющее вхождение
foo()
```

Основные понятия ЯП - видимость имен

Пример 1 - Python

ОВ в Python - модуль (глобальная ОВ), тела функций (локальные ОВ)

Пример глобальной и локальной переменных

```
x = "this is global X"  # это - определяющее вхождение
def foo():
        x = "this is local x"  # это - определяющее вхождение локальной переменной
        print(x) # это - использующее вхождение
foo()
print(x)
```

Основные понятия ЯП - видимость имен

Пример 1 - Python

ОВ в Python - модуль (глобальная ОВ), тела функций (локальные ОВ)

Чуть более нетривиальный пример

```
x = "this is global X"  # это - определяющее вхождение
def foo():
        x = "this is local x"  # это - определяющее вхождение локальной переменной
        print(x) # это - использующее вхождение
foo()
print(x)
```

Основные понятия ЯП - видимость имен

Пример 1 - Python

ОВ в Python - модуль (глобальная ОВ), тела функций (локальные ОВ)

Пример глобальной и локальной переменных

```
x = 1
def foo(a):
    if a>0:
        x = 5
    print(x)

foo(1)
#foo(-1)
print(x)
```

Что будет при раскомментировании foo(-1)?

Основные понятия ЯП - видимость имен

Пример 1 - Python

ОВ в Python - модуль (глобальная ОВ), тела функций и классы (локальные ОВ)

Можно ли менять глобальную переменную в локальной ОВ?

Нельзя (так как плохой стиль - "Global variables considered harmful" - 1973)

Это побочный эффект. Но если очень хочется, то можно...

```
x = 1
def foo(a):
    global x # это не объявление переменной, но указание, где искать ее определение
    if a>0:
        x = 5
    print(x)
foo(-1) # переставил, чтобы было интереснее
foo(1)
print(x)
```

Основные понятия ЯП - видимость имен

Пример 1 - Python

ОВ в Python - модуль (глобальная ОВ), тела функций и классы (локальные ОВ)

Можно ли менять глобальную переменную в локальной ОВ?

Для python 1...2 - хватало, но развитие языка привело к широкому использованию вложенных ОВ (функции внутри функций)=> как менять локальные переменные из объемлющей ОВ изнутри вложенной ОВ

так называемые нелокальные ОВ (которые для кого-то локальные, а для кого-то - глобальные...)

Основные понятия ЯП - видимость имен

Пример 1 - Python

ОВ в Python - модуль (глобальная ОВ), тела функций и классы (локальные ОВ), нелокальные ОВ (объемлющие тела функций)

Как менять локальные переменные из объемлющей ОВ изнутри вложенной ОВ

```
x = 1
def foo(a):
    x = 5
    def inner_fun(y):
        nonlocal x # это не объявление переменной, но указание, где искать ее определение
        x += y
    inner_fun(a)
    print(x)

foo(-1)
foo(1)
print(x)
```

Основные понятия ЯП - видимость имен

Пример 2 - JavaScript - поднятие имен - "hoisting"
ОВ в JavaScript - глобальная ОВ, локальные ОВ - тела функций
Блок - синтаксически есть, но он НЕ ЯВЛЯЕТСЯ ОВ!
Объявление ***var x*** добавляет x в текущую ОВ

```
function foo() {
    if (false) {
        var x = 1;  // поднимается вверх, но без инициализации
    }
    return;
    var y = 1; // поднимается вверх, но без инициализации
}
```

Основные понятия ЯП - видимость имен

Пример 2 - JavaScript - поднятие имен - "hoisting"

```
function foo() {
   if (false) {
      var x = 1;  // поднимается вверх, но без инициализации
   }
   return;
   var y = 1; // поднимается вверх, но без инициализации
}
```

Эквивалентно:

```
function foo() {
   var x, y;
   if (false) {
      x = 1;
   }
   return;
   y = 1;
}
```

Основные понятия ЯП - видимость имен

Пример 2 - JavaScript - поднятие имен - "hoisting"

То же самое относится к объявлению глобальных переменных и объявлению функций

```
foo();
function foo() {
   x = 1;
   alert (x);
}
alert(x);
x = 2;
alert(x);
```

Основные понятия ЯП - видимость имен

Пример 2 - JavaScript - поднятие имен - "hoisting"

В расширениях JS - обязательное опережающее объявление переменных (никаких "голых" присваиваний)

Основные понятия ЯП - видимость имен

Итак:

1). Что такое ОВ для ЯП.

2). Что является определяющим вхождением для имени

3). Как связать использующее вхождение с определяющим (тоже разновидность связывания)

По поводу п.3 - в статических языках - меньше проблем (особенно, если нет перегрузки имен и неявного импорта) и ВСЕГДА - статическое связывание (иногда и со связыванием адреса)

Основные понятия ЯП - видимость имен

Итак:

1). Что такое ОВ для ЯП.

2). Что является определяющим вхождением для имени

3). Как связать использующее вхождение с определяющим (тоже разновидность связывания)

По поводу п.3 - в статических языках - меньше проблем (особенно, если нет перегрузки имен и неявного импорта) и ВСЕГДА - статическое связывание (иногда и со связыванием адреса)

В динамических - "лексическое" и "динамическое" связывание (lexical/dynamic scoping). Все примеры выше - используют лексическое связывание - в момент трансляции определяется, какое именно определяющее вхождение имеется ввиду.

Динамическое связывание - ранние версии Лиспа

Основные понятия ЯП - видимость имен

Динамическое vs лексическое связывание - можно понять, применяя логику реализации - в какой момент происходит поиск и добавление имени в таблицу имен для ОВ?

Если при трансляции - лексическое связывание, если при выполнении - динамическое связывание.

Пример на гипотетическом ЯП:

```
fun outer() {
  var x = 1 // ОпрВ1
  fun inner1() {
     var x = 2 // ОпрВ2
     inner2()
  }
  fun inner2() {
     write(x); // ЛС => всегда связано с ОпрВ1
  }
  inner1();
  inner(2);
}
```

Основные понятия ЯП - видимость имен

```
fun outer() {
  var x = 1 // ОпрВ1
  fun inner1() {
     var x = 2 // ОпрВ2
     inner2()
  }
  fun inner2() {
     write(x); // ЛС => всегда связано с ОпрВ1
  }
  inner1(); // при ДС: ОВ outer (ОпрВ1) => ОВ inner1 (ОпрВ2) => ОВ inner2(пуста)
  inner(2); // при ДС: ОВ outer (ОпрВ1) => ОВ inner2(пуста)
}
```

При ЛС ОпрВ2 нигде не используется (нет обращений из статически вложенных ОВ)

=> 1 1

При ДС => 2 1

Основные понятия ЯП - видимость имен

Связь между определяющим и использующими вхождениями

Как правило, от точки определения до конца некоторой области программы (блок, файл, модуль, пространство имен....)

Но и тут есть исключения :

- "поднятие имен(hoisting)" в JS
- в языках с классами - видимость члена - не от точки определения, а с начала класса.

Блок (с Алгола 60) - именно как область видимости.

Модули (классы) - экспорт имен в объемлющую область видимости

Видимость - потенциальная и непосредственная

Потенциальная - через уточнение (qualifying).

Везде - операция уточнения - "точка" (кроме C++ - там Outer::inner)

OuterScopeName.aName - так можно обратиться к aName

Если можно без OuterScopeName - то это уже непосредственная видимость

Основные понятия ЯП - видимость имен

Видимость - потенциальная и непосредственная (речь идет уже об использующих вхождениях)

Потенциальная - через уточнение (qualifying).

OuterScopeName.aName - так можно обратиться к aName

Если можно без OuterScopeName - то это уже непосредственная видимость

Всегда непосредственно видимы - имена из текущей области видимости (то есть опред. вхождение - в этой же области).

Импортируемые имена видимы (как правило) - потенциально.

```
std::cout << "Hello, world!" << std::endl;
```

Основные понятия ЯП - видимость имен

Видимость - потенциальная и непосредственная

Проблемы (немедленно и обязательно) возникают при снятии непосредственной видимости.

C++:

using namespace std; // неявный импорт всех доступных в этой точке имен из std

cout << "Hello, world!" << endl;

Для std проблем меньше, так как имена достаточно известны, но все равно есть.

Основные понятия ЯП - видимость имен

C++:

```
namespace HisNS {
  double sss(double);
}
namespace MyNS {
  double sss(int);
}
using namespace MyNS;
sss(0.0); // MyNS::sss
sss(0); // MyNS::sss
```

Основные понятия ЯП - видимость имен

C++:

```
namespace HisNS {
  double sss(double);
}
namespace MyNS {
  double sss(int);
}
using namespace MyNS;
using namespace HisNC; // программа компилируется, но семантика изменилась!!!
sss(0.0); //  HisNS::sss
sss(0); // MyNS::sss
```

Основные понятия ЯП - видимость имен

Если нет неявного импорта, то эта проблема исчезает.

Пример: язык Оберон - нет ни неявного импорта (потенциальную видимость снимать нельзя!), нет ни перегрузки...

Проблем нет.

Но есть вот что:

Основные понятия ЯП - видимость имен

Если нет неявного импорта, то эта проблема исчезает.

Пример: язык Оберон - нет ни неявного импорта (потенциальную видимость снимать нельзя!), нет ни перегрузки...

Проблем нет.

Но есть вот что:

C++:

```
cout << "counter=" << cnt << '\n';
```

C:

```
printf("counter=%d\n", cnt);
```

Оберон:

```
IMPORT InOut; ....
InOut.WriteString("counter=");  InOut.WriteInt(cnt); InOut.WriteLn;
```