Понятие о парадигме программирования. Основные парадигмы программирования. Языки и парадигмы программирования

Важным свойством индустриальных языков является наличие изобразительных средств, поддерживающих различные стили программирования. Совокупность идей и понятий, определяющих стиль программирования, называется **парадигмой программирования**. Основополагающую роль в парадигме программирования играет именно язык программирования, поскольку именно на нем мы выражаем то, как мы «мыслим».

В настоящее время в индустриальном программировании активно используются императивная и объектная парадигмы. Есть основания полагать, что в ближайшее время начнет активно использоваться функциональная парадигма.

Рассмотрим вкратце эти парадигмы и соответствующие примеры программ. Для иллюстрации приведем небольшие программы, решающие одну и ту же задачу, используя разные стили программирования.

Задача состоит в обращении входной последовательности. В стандартном канале ввода (обычно — клавиатура компьютера) задана последовательность символов. Требуется выдать в стандартный канал вывода элементы входной последовательности в обратном порядке (последний элемент входной последовательности — первым, предпоследний - вторым и так далее). Будем считать, что последовательность «не очень длинная», точнее предполагаем, что все элементы могут поместиться в оперативную память компьютера. Такое предположение упрощает решение задачи.

Императивная парадигма

Императивная парадигма (другое название — процедурная) основана на фон-неймановской модели компьютера, названной в честь предложившего её математика Дж. Фон Неймана. Модель Фон Неймана до сих пор является

основой большинства современных архитектур, что обусловило популярность и доминирование императивной парадигмы.

Модель содержит три основные компонента:

- центральное процессорное устройство (ЦПУ);
- оперативная память (ОП);
- устройства ввода/вывода (УВВ).

Оперативная память — это однородная последовательность ячеек. Каждая ячейка имеет свой номер, называемый *адресом*. Адресация начинается с нуля. ЦПУ считывает или записывает информацию в ячейку, используя её адрес. Скорость доступа ко всем ячейкам одинакова и не зависит от адреса. Такие последовательности в программировании называют *массивами*. Ячейки могут содержать как данные (числа или символы), так и команды (то есть команды закодированы с помощью чисел).

Устройства ввода/вывода позволяют обмениваться информацией между ЦПУ и окружающей средой.

«Мозг» компьютера - ЦПУ - состоит из двух частей: арифметико-логического устройства (АЛУ), которое выполняет арифметические и логические команды (сложение, вычитание, умножение, деление, побитовые сдвиги, побитовая инверсия и так далее), и устройства управления, которое отвечает за порядок выборки, декодирование и выполнение команд. ЦПУ содержит ряд специальных ячеек (не из ОП), называемых *регистрами*, каждая из которых выполняет свою роль в работе ЦПУ. Например, регистр команд содержит текущую выполняемую команду, регистр адреса — адрес этой команды, в арифметико-логиеских регистрах находятся операнды арифметико-логических команд.

Команды ЦПУ делятся на следующие группы:

- пересылки между ОП и регистрами ЦПУ;
- арифметико-логические команды;

- команды управления, включающие в себя команды перехода и некоторые специальные команды (например, команда останова);
- команды ввода/вывода — концептуально они похожи на команды пересылки (в некоторых архитектурах команды ввода/вывода отсутствуют и реализуются как команды пересылки для выделенной области ОП).

Команды выбираются из памяти и выполняются последовательно, одна за другой. Исключение составляют команды условного и безусловного перехода — они содержат адрес команды, которая будет выполняться следующей и позволяют менять нормальную последовательность выполнения команд.

Основные понятия императивных языков программирования (ИЯП) представляют собой абстракции основных понятий фон-неймановской модели. В самом деле, любой ИЯП включает в себя понятие переменной (в языке Паскаль - `VAR X:Integer`, в языке С — `int x`), понятие операции (`A * B` – в любом языке), понятие оператора (оператор цикла, оператор присваивания и другие).

Понятие простой переменной абстрагирует понятие ячейки памяти. Кроме простых переменных в императивном языке бывают составные, то есть состоящие из других переменных — массивы и записи (в ряде языков записи называются структурами). Понятие операции обобщает арифметико-логические команды, неслучайно почти для любой операции в ИЯП можно найти прототип - команду в машинном языке. Понятие оператора абстрагирует общее понятие команды. Операторы в императивном языке делятся на три группы:

- оператор присваивания;
- операторы управления;
- операторы ввода/вывода.

Основным оператором в любом императивном языке является оператор присваивания, имеющий вид:

```
V:=E
```

где V — это переменная, а E — выражение. Выражение — это средство комбинирования операций для вычисления некоторого значения (например, `X * (Y+1)/2`). Выполнение оператора присваивания состоит в вычислении значения выражения E и пересылке вычисленного значения в ячейку (или ячейки) ОП, соответствующую переменной V. Таким образом, оператор присваивания в ИЯП может представляться последовательностью команд пересылки, арифметико-логических команд (и даже команд перехода). Это действительно основной оператор в императивных языках.

Операторы управления (циклы, операторы выбора, перехода и тому подобные) абстрагируют машинные команды перехода.

Операторы ввода/вывода обобщают машинные команды ввода/вывода.

Подробнее основные понятия ИЯП разбираются в главе 5 второй части нашего курса.

Как видим, императивные языки концептуально близки машинной архитектуре, поэтому программирование на таких языках позволяет весьма эффективно управлять поведением компьютеров. Это объясняет популярность и распространенность ИЯП. В индустриальном программировании в настоящее время доминируют либо чисто императивные языки (такие, как C), либо языки со смешанной объектно-императивной парадигмой (C++, Java, C#, Delphi, Objective C и многие другие).

Решим нашу пробную задачу реверсирования входной последовательности в императивном стиле на языке C. Для этого стиля характерно представление алгоритма решения задачи в виде последовательности шагов, каждый их которых, в свою очередь, представляется последовательностью более мелких шагов, и так далее, пока мы не дойдем до шага «размером» в один оператор.

Многие задачи по обработке данных (в том числе и наша) сводятся к трем шагам:

- «подготовить данные»
- «обработать данные»
- «завершить»

Условие нашей задачи позволяет хранить всю входную последовательность в ОП, поэтому шаг подготовки данных сводится к вводу данных в некоторую структуру данных, которая позволяет выбирать данные и в прямом порядке, и в обратном (для обращения).

Шаг обработки сводится к реверсированию этой структуры данных, шаг завершения — к выводу обращенной структуры.

Основной вопрос — какую структуру данных выбрать? Нашим требованиям отвечает, например, линейный двунаправленный список, однако такой встроенной структуры в языке С нет. Работу со списками в языке С нужно реализовывать самому программисту. Единственное понятие в языке С, соответствующее последовательности однородных элементов, - это массив. Проблема в том, что массивы в С — это последовательности фиксированной и заранее известной длины. Для того, чтобы упростить решение задачи, предположим, что «не очень длинная» последовательность содержит не более 1024 символов. Тогда мы можем использовать массив символов соответствующей длины.

Прежде, чем привести полное решение, заметим, что можно объединить шаг обработки и завершения: вместо обращения элементов в массиве, можно сразу перейти к выводу, только выводить элементы не с начала, а с конца. Тем самым, выполнение становится несколько быстрее (вместо двух операторов цикла мы имеем один в объединенном шаге).

```
#include <stdio.h>
#define MAX_ELEMENTS 1024
char Input[MAX_ELEMENTS];

int main()
{
     int current, count = 0;
     while ((current = getchar()) != EOF)
```

```
            if (count == MAX_ELEMENTS) {
                 fprintf(stderr, "Слишком много символов");
                 return 1;
            } else
                 Input[count++] = current;
       for (int i = count-1; i >= 0; i--)
            putchar(Input[i]);
       return 0;
  }
```

Первая строка программы сообщает о включении информации о библиотеке стандартного ввода/вывода, в следующих двух объявляется константа MAX_ELEMENTS— предельная длина входной последовательности, и массив Input для хранения последовательности (заметим, что в языке С элементы массива индексируются с нуля). Далее объявляется функция main, с вызова которой начинается выполнение программы на С. В теле функции объявлены переменные current — для ввода очередного элемента последовательности — и count — для хранения количества элементов. Следующий далее оператор цикла while соответствует подготовительному шагу и вводит в массив Input всю входную последовательность. Функция getchar вводит один символ из входной последовательности.

Заметим, что в языке С присваивание «=» является операцией. Эта операция выполняется также, как и оператор присваивания в классических ИЯП, но отличается от последнего тем, что присвоенное значение является одновременно и значением операции присваивания. Таким образом, операция присваивания не только вычисляет значение (как и любая другая операция), но и меняет значение переменной из своей левой части. Такие операции называются операциями с *побочным эффектом*. Как и другие операции, присваивание может комбинироваться с другими операции в выражении.

В нашем примере выражение в заголовке цикла

```
(current = getchar())!=EOF
```

вызывает функцию getchar, далее присваивает её значение переменной current, и после сравнивает это же значение (то есть введенный символ) с

признаком конца ввода EOF (заметим, что EOF — это не особое значение символа, а лишь признак конца, вырабатываемый драйвером ввода операционной системы). Если значение введенного символа не совпадает с признаком конца, то цикл продолжается.

Отметим, что операция присваивания — не единственная операция с побочным эффектом. Так операция ++ в выражении «count++» обладает побочным эффектом, состоящим в увеличении на 1 значения переменной count. А само значение операции равно значению count до выполнения этой операции. Поэтому оператор

```
Input[count++] = current;
```

как присваивает значение current очередному элементу массива, так и увеличивает значение счетчика count на 1.

Аналогично, операция «--» в выражении «i--» уменьшает значение i на единицу.

В случае, если символов слишком много, то в специальный канал вывода сообщений об ошибках (stderr) функция fprintf выводит текст "Слишком много символов". После этого выполнение программы завершается оператором “return 1;” (возврат из функции main).

Последний цикл for выводит элементы последовательности, начиная с конца. Так как индексы элементов массива всегда начинаются с 0, то последний введенный элемент будет иметь индекс count-1, а первый - 0, поэтому параметр цикла i последовательно уменьшается с count-1 до 0.

Последний оператор программы — возврат из функции main.

Объектная парадигма

Объектная парадигма основана на понятии объекта. Объект обладает состоянием и поведением. Поведение состоит в посылке сообщений себе и другим объектам. Для каждого вида сообщения существуют «обработчики», которые могут модифицировать состояние объекта и посылать сообщения

другим объектам. Объекты с одинаковым поведением и набором состояний объединяются в классы. Между классами могут существовать отношения:

- **включение** - «объект-подобъект» - включение объекта класса X в объект другого класса Y, говорят, что объект класса Y владеет объектом класса X;
- **наследование** - «суперкласс-подкласс» - объект подкласса Derived обладает всеми свойствами объекта суперкласса Base, а также, возможно, дополнительными свойствами (специфичными для класса Derived), таким образом, все объекты класса Derived одновременно принадлежат и классу Base, но не наоборот;
- **ссылка** — объект класса W содержит ссылку (но не владеет) на объект класса Ref.

Также существуют и другие отношения.

Объектная парадигма достаточно просто сочетается с императивной парадигмой. Состояние описывается набором переменных, а обработчики сообщений представляют собой процедуры или функции, имеющие доступ к состоянию. Посылка сообщения сводится к вызову соответствующего обработчика.

В результате большинство современных языков индустриального программирования сочетает в себе обе парадигмы. Мы будем говорить об «объектно-императивной» парадигме программирования.

Одно их основных достоинств объектного подхода — возможность создания достаточно гибких и универсальных иерархий классов, которые могут быть использованы во многих прикладных задачах почти без изменения. Неслучайно все индустриальные объектно-ориентированные языки программирования (ООЯП) включают в себя большой набор классов из стандартных библиотек. В случае, если этих классов недостаточно, то ООЯП позволяют сравнительно легко (по сравнению с чисто императивной парадигмой) разрабатывать специализированные классы, либо «с нуля», либо на

основе стандартных классов.

Посмотрим, как задача о реверсировании входной последовательности может быть решена на объектно-ориентированных языках. Вначале рассмотрим решение на языке C#.

Все рассуждения из предыдущего пункта по ходу решения задачи остаются справедливыми и здесь (ведь объектная парадигма в C# расширяет, но не отменяет императивную). Отличие объектного подхода в том, что у нас уже имеются готовые классы, позволяющие быстро решить поставленную задачу. В языке C# есть не только понятие массива, но и набор контейнеров, как универсальных (вектор, список и так далее), так и специализированных (динамическая строка произвольной длины). Мы будем использовать строки и массивы языка C#.

```
using System;
class Program
{
    static void Main(string[] args)
    {
        string s = Console.In.ReadToEnd();
        char[] seq = s.ToCharArray();
        Array.Reverse(seq);
        Console.Write(seq);
    }
}
```

Первая строка программы сообщает об использовании стандартной библиотеки System, в которой нам понадобятся классы Console для ввода/вывода и Array для операций с массивами.

Далее следует объявление класса Program, содержащего описание единственной функции Main. Эта функция играет такую же роль, что одноименная функция в C: с её вызова начинается выполнение консольных программ.

В первой строке мы объявляем строку s и сразу же вводим в неё целиком всю входную последовательность. Класс Console обладает объектом In класса TextReader, представляющим стандартный канал ввода. Объекты этого класса

позволяют ввести целиком всю входную последовательность. Заметим, что мы уже не нуждаемся во введении ограничения на максимальную длину входной строки. Объем ввода лимитируется только размерами свободной виртуальной памяти, доступной процессу.

Далее из введенной строки с помощью функции ToCharArray конструируется массив из символов `(char [] seq)`, составляющих строку. Функция Reverse из класса Array обращает массив (то есть решает нашу задачу). Теперь осталось только вывести реверсированный массив.

Уже можно сделать несколько очевидных выводов. Во-первых, объектное решение и проще для понимания, и короче. Во-вторых, объектное решение позволяет обрабатывать последовательности большей длины. Конечно, мы можем и в первом решении на языке C отказаться от ограничения и использовать либо динамический массив, либо двунаправленный список и тому подобное. Однако такое решение существенно длиннее и сложнее для понимания, чем простой вариант. Причина в том, что в императивных языках типа C достаточно сложно создавать гибкие и одновременно универсальные контейнеры. В каждом конкретном случае приходится создавать такие сущности «с нуля». Это одна из причин популярности ООЯП. При программировании в объектном стиле существенно проще использовать уже готовые объекты. Да и создавать новые объекты проще, чем в императивном языке.

Но императивный стиль программирования тоже обладает некоторыми достоинствами. Например, в приведенной программе на C# не совсем очевиден тот факт, что оперативная память используется в ней расточительно. Входная последовательность хранится в двух экземплярах: в строке s первый экземпляр, а в массиве seq – второй. Конечно, первое решение все равно хуже, но в императивном стиле можно разработать довольно сложное решение, которое будет работать с памятью лучше, чем объектное решение на C#. Однако легкость программирования на ООЯП в большинстве случаев перевешивает.

Кроме этого, и на языке типа C# можно программировать в императивном стиле. В нашем примере мы можем использовать тот факт, что строки, как и массивы, допускают индексирование, и сделать вывод в стиле языка C.

```
using System;
class Program
{
    static void Main(string[] args)
    {
        string s = Console.In.ReadToEnd();
        for (int i = s.Length-1; i>=0;i--)
            Console.Write(s[i]);
    }
}
```

Заметим, что некоторые ООЯП, например, C++, в дополнение к классическому объектному механизму обладают весьма мощными и выразительными средствами обобщенного программирования. В главе 10 (п.10.2) второй части этого курса приводится пример программы на C++, свободной от расточительного использования памяти. Собственно само решение занимает две строки!

Функциональная парадигма

Основные понятия функциональных языков — функция и выражение. Выражение — это комбинация вызовов функций. Наряду с большим количеством стандартных (встроенных в язык) функций программист может определять свои функции. Определение новой функции включает в себя имя функции, список аргументов и выражение-тело функции. Вызов функции состоит из имени функции и списка выражений - фактических параметров. Каждый из фактических параметров соответствует аргументу в определении функции.

Основная операция — вызов функции. При вызове функции вначале вычисляются выражения - фактические параметры, затем их значения подставляются вместо соответствующих им аргументов в выражение-тело функции. Наконец, вычисляется значение тела, которое и будет значением вызова.

Приведем одно из решений нашей задачи на языке Лисп — первом языке программирования, в котором была реализована функциональная парадигма. Мы будем использовать один из самых популярных диалектов Лиспа — Коммон Лисп.

Прежде, чем рассматривать решение, сделаем несколько замечаний о представлении программ на Лиспе. Вызов функции выглядит так:

```
( имя-функции список-фактических-параметров)
```

Например,

```
(+ 2 3)
```

Определение функции выглядит так:

```
(defun  имя-функции (список-имен-аргументов) выражение)
```

Например,

```
(defun plus1(x) (+ x 1))
```

Обращение к стандартной функции ввода:

```
(read)
```

Однако эта функция возвращает не последовательность литер, а более сложную структуру — Лисп-выражение. Подробнее о Лиспе говорится в главе 11 (п.11.2) второй части курса. Здесь сделаем только несколько замечаний необходимых для понимания примера.

Лисп-выражение — это атом, либо список. Атом - это либо символ (идентификатор), либо число. Список — это последовательность членов списка, разделенных пробелами, заключенная в круглые скобки. Член списка — это либо атом, либо список. Есть специальный атом — `nil`, который представляет собой пустой список. Поэтому его другое обозначение - `()`. Это единственный атом, который одновременно является и списком. Нетрудно видеть, что примеры вызова и определения функции сами представляют собой списки.

Иначе говоря, Лисп-программы не только обрабатывают списки, но и сами представляют собой списки. Именно поэтому функция `read` не занимается политерным вводом (как в императивных языках программирования), а читает и строит Лисп-объекты (атомы и списки). Поэтому немного переформулируем задачу для решения её на Лиспе в функциональном стиле. Пусть на входе — список слов (символов). Требуется выдать этот список в обратном порядке. Заметим, что формулировка задачи не упростилась, а наоборот, усложнилась. Однако это не усложнит её решение на Лиспе.

Поскольку функция read сразу вводит весь список, то нам нужно написать функцию обращения списка. Вообще-то, такая функция в Лиспе есть — она называется `reverse`. Тогда решение выглядит тривиально (`print` – это функция вывода в стандартный канал):

```
(print (reverse (read)))
```

Однако, чтобы почувствовать специфику функционального программирования, напишем свой вариант функции `reverse`:

```
(defun rev(x)
          (if (null x)
               nil
               (append (rev (cdr x))(cons (car x) nil))
          )
     )
```

Поясним решение. Заметим, что список — это рекурсивная структура, поэтому и обработка списков тоже рекурсивна по своей природе. Обращение пустого списка — это пустой список, если же список не пустой, то он состоит из первого элемента списка (функция Лиспа `(car x)` возвращает первый элемент списка `x`), и хвоста, который тоже является списком (функция Лиспа `(cdr x)` возвращает хвост списка `x`). Тогда обращение такого списка — это список, который составлен из двух списков: первый - это обращение хвоста (`(rev (cdr x))`), а второй — это список, состоящий их одного элемента — головы `x` (`car x`).

Но первый элемент списка — это не обязательно список, поэтому из него нужно сделать одноэлементный список с помощью функции Лиспа `(cons a b)`. В результате работы этой функции формируется список, головой которого является a, а хвостом - b. Таким образом, (cons (car x) nil) и есть требуемый одноэлементный список.

Функция Лиспа `append` служит для конкатенации списков, а предикат `(null x)` проверяет список `x` на пустоту.

Специальная функция Лиспа `if` вычисляет свой первый аргумент (`(null x)`), и если он истинен, то возвращает вычисленный второй аргумент (`nil`), иначе возвращает вычисленный третий аргумент ((`append (rev (cdr x)) (cons (car x) nil)`)).

Заметим, что в приведенной программе отсутствуют присваивания и циклы. В ней только вызовы функций (один из них — рекурсивный). Это отличительный признак чисто функциональных программ. Также отметим ещё два момента. Во-первых, это совсем не лучший способ реализации функции `reverse`. Во-вторых, рекурсивный способ обращения списка (или массива) можно реализовать на любом императивном языке, в котором допускаются рекурсивные вызовы функций. Поэтому можно программировать в функциональном стиле и на ИЯП. Правда, делать это на императивных языках труднее, чем на языках типа Лисп, явно поддерживающих функциональный стиль.