Высшее профессиональное образование

БАКАЛАВРИАТ

И. Г. ГОЛОВИН, И. А. ВОЛКОВА

ЯЗЫКИ И МЕТОДЫ ПРОГРАММИРОВАНИЯ

УЧЕБНИК

Допущено
Учебно-методическим объединением
по классическому университетскому образованию
в качестве учебника для студентов высших учебных заведений,
обучающихся по направлениям 010400 «Прикладная математика
и информатика» и 010300 «Фундаментальная информатика
и информационные технологии»

ACADEMIA

Москва
Издательский центр «Академия»
2012

УДК 004.43(075.8)
ББК 32.973-018.1я73
Г611

Рецензенты:
доктор физико-математических наук, профессор *А. В. Гладкий*;
доктор физико-математических наук, профессор *Н. В. Семенов*

Головин И. Г.

Г611 Языки и методы программирования : учебник для студ. учреждений высш. проф. образования / И. Г. Головин, И. А. Волкова. — М. : Издательский центр «Академия», 2012. — 304 с. — (Сер. Бакалавриат).

ISBN 978-5-7695-7973-8

Учебник создан в соответствии с Федеральным государственным образовательным стандартом по направлению подготовки 010400 «Прикладная математика и информатика» (квалификация «бакалавр»).

Рассмотрены основные парадигмы программирования: процедурная, объектно-ориентированная, функциональная. Особое внимание уделено семантике и прагматике языковых понятий, их связи с методами и технологией программирования. Материал представлен на примере современных языков индустриального программирования: C++, C#, Java. Отражены теоретические и практические вопросы реализации языков программирования.

Для студентов учреждений высшего профессионального образования.

УДК 004.43(075.8)
ББК 32.973-018.1я73

Учебное издание

Головин Игорь Геннадьевич, Волкова Ирина Анатольевна

Языки и методы программирования

Учебник

Редактор *В. И. Махова*. Технический редактор *О. Н. Крайнова*
Компьютерная верстка: *Г. Ю. Никитина*. Корректор *А. П. Сизова*

Изд. № 101115871. Подписано в печать 28.10.2011. Формат 60 × 90/16. Гарнитура «Ньютон». Бумага офсетная № 1. Печать офсетная. Усл. печ. л. 19,0. Тираж 1000 экз. Заказ № 30

ООО «Издательский центр «Академия». www.academia-moscow.ru
125252, Москва, ул. Зорге, д. 15, корп. 1, пом. 266.
Адрес для корреспонденции: 129085, Москва, пр-т Мира, 101В, стр. 1, а/я 48.
Тел./факс: (495) 648-0507, 616-00-29.
Санитарно-эпидемиологическое заключение № РОСС RU. АЕ51. Н 14964 от 21.12.2010.
Отпечатано с электронных носителей издательства.
ОАО «Тверской полиграфический комбинат», 170024, г. Тверь, пр-т Ленина, 5.
Телефон: (4822) 44-52-03, 44-50-34. Телефон/факс: (4822) 44-42-15.
Home page — www.tverpk.ru Электронная почта (E-mail) — sales@tverpk.ru

ISBN 978-5-7695-7973-8

ПРЕДИСЛОВИЕ

Неоднократные попытки создания универсального языка программирования не привели к успеху: люди не только продолжают использовать старые языки, но и постоянно придумывают новые. Поэтому изучать приходится несколько языков программирования. При этом набор постоянно используемых языков со временем изменяется: сначала нельзя было обойтись без знания Фортрана, затем необходим был ТурбоПаскаль, сейчас трудно обойтись без знания C++. Что будет завтра? Авторы затрудняются ответить на этот вопрос, но уверены в том, что «фаворит» снова поменяется.

Перед вами учебник, который, как надеятся авторы, поможет войти в сложный, но необычайно интересный мир языков программирования.

Данный учебник основан на курсах «Языки программирования» и «Системы программирования», читаемых студентам факультета ВМК МГУ им. М.В.Ломоносова. Цель учебника — объяснить основные понятия, конструкции, принципы разработки и реализации современных языков индустриального программирования.

В учебнике представлен сравнительный анализ основных понятий и возможностей современных языков. Понимание фундаментальных концепций поможет вам не только лучше использовать знакомый язык программирования, но и быстрее и легче освоить новый (что, по мнению авторов, рано или поздно будет необходимо).

Первые две части учебника посвящены введению в основные концепции современных языков программирования. В качестве примеров рассмотрены языки C++, Java и C#. Такой выбор обусловлен, во-первых, популярностью этих языков, во-вторых, доступностью реализаций на популярных платформах, а в-третьих, тем, что они демонстрируют основные направления развития современных языков программирования.

Третья часть учебника посвящена основному механизму описания формальных языков — формальным грамматикам, а также основным приемам, методам и алгоритмам создания программ — анализаторов формальных языков. Формальные языки — это не только языки программирования, но и любые искусственные языки, имеющие какую-либо внутреннюю структуру (например, язык алгебры). Зна-

ние теории и практики реализации формальных языков должно быть неотъемлемой частью фундаментального образования любого профессионального программиста. Сведения, представленные в учебнике, помогут грамотно и эффективно решать разнообразные задачи, связанные с обработкой любых формальных языков.

Авторы глубоко признательны Вылитку Алексею Александровичу за ряд ценных замечаний по третьей части учебника.

ЧАСТЬ I
ОСНОВЫ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ

Глава 1
ПОНЯТИЕ ЯЗЫКА ПРОГРАММИРОВАНИЯ

1.1. Определение языка программирования

Одно из наиболее общих определений языка программирования дано в [18]: **язык программирования** — это инструмент для планирования поведения некоторого устройства-исполнителя. Под такое очень общее (хотя и полезное) определение подходят, например, как средства составления программ для жаккардовых ткацких станков (XVIII в.) и калькулятора Ч. Бэббиджа (XIX в.), так и системы программирования современных компьютеров. Исходя из целей курса ограничим это определение: в качестве управляемого устройства будем рассматривать только компьютеры (вычислительные системы). Планы, управляющие поведением компьютеров, называются компьютерными программами. Тогда уточненное определение будет выглядеть так: **язык программирования** — это нотация для записи компьютерных программ [6].

Однако области применения современных компьютеров настолько обширны и разнообразны, что существует большое число специализированных языков, подпадающих под наше определение. Например, это язык программирования баз данных SQL [12]. Исполнителем для этого языка является система управления базами данных (СУБД). Данный язык позволяет описывать и создавать структуры баз данных, манипулировать данными (добавлять, модифицировать и удалять данные из базы), составлять запросы на выборку данных.

Другой пример специализированного языка — язык разметки гипертекста HTML [19]. Тексты на HTML управляют отображением гипертекста на экране компьютера. В данном случае исполнителем является программа-обозреватель (веб-браузер). Вообще можно сказать, что любая прикладная программа с нетривиальным форматом

обрабатываемых ею данных определяет некоторый узкоспециализированный язык.

Конечно, мы не будем рассматривать весь спектр специализированных языков, а ограничимся универсальными языками программирования. Отличие универсальных языков — алгоритмическая полнота, т.е. возможность описания на таком языке любого вычисления (алгоритма). Кроме того, на универсальных языках можно программировать широкий класс прикладных задач (теоретически — все возможные), решаемых на компьютере. Ни SQL, ни HTML не являются универсальными, тогда как на универсальном языке программирования (например, Java или C++) можно реализовать как программу просмотра гипертекста, так и СУБД.

Отметим, что многие понятия и концепции, рассматриваемые в курсе, применимы при анализе и изучении любых языков программирования: и универсальных, и специализированных. Особенно это относится к третьей части учебника.

1.2. Языки и основные парадигмы программирования

Как уже отмечалось, спектр применения современных компьютеров очень широк, поэтому общая совокупность разработанных компьютерных программ весьма велика и с трудом поддается классификации. Тем не менее, все программы (как и деятельность по их созданию) можно разделить на два больших типа: программы «для себя» и программы «для других».

Создание программ «для себя» назовем (несколько условно) научно-развлекательным программированием. Люди создают программы для развлечения (когда развлечением является сам процесс программирования, а не использование уже готовых программ), для изучения программирования (студенты, выполняющие задания практикума работы на ЭВМ), для экспериментальных расчетов, демонстрирующих работоспособность математических моделей (физики, математики, химики, биологи и т.д.), для решения каких-либо сиюминутных задач (ученые, инженеры).

При научно-развлекательном программировании число создателей программы крайне невелико (как правило, один-два человека), также невелико число пользователей программы (как правило, только авторы и небольшое число коллег). Такие программы невелики по размеру (тысячи и реже несколько десятков тысяч строк), «живут» (т.е. активно используются) они недолго (дни, месяцы, реже годы) и в процессе жизни редко модифицируются.

Языки научно-развлекательного программирования должны быть легко изучаемыми, несложными для понимания и реализации.

Основной критерий их качества — удобство применения для соответствующих целей (учебы, науки, развлечения). Самые известные и широко используемые до сих пор языки — Фортран, Бейсик, Паскаль.

Создание программ «для других», т. е. программ, специально создаваемых в расчете на эксплуатацию пользователями, не имеющими отношения к авторам, значительно отличается от научно-развлекательного. Такие программы называют **программными продуктами**, а процесс их создания — **индустриальным программированием**.

Число разработчиков программного продукта варьируется в широких пределах (как правило, их десятки, а иногда и сотни), то же относится и к его размерам (сотни тысяч и миллионы строк кода), и к числу пользователей. Успешные программные продукты «живут» годы (иногда десятилетия) и в процессе использования активно модифицируются (исправляются обнаруженные в ходе их эксплуатации ошибки, добавляются новые функции, выполняется адаптация к новому оборудованию или программному окружению).

Языки индустриального программирования отличаются от языков научно-развлекательного программирования. Первые сложнее в изучении и реализации, включают в себя большое число концепций и понятий, обладают объемными библиотеками.

В рамках нашего курса сосредоточимся на основных понятиях современных языков индустриального 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 это клавиатура компьютера) задана последовательность символов. Требуется выдать в стандартный канал вывода элементы входной последовательности в обратном порядке (т. е. последний элемент входной последовательности первым, предпоследний — вторым и т. д.). Будем считать, что эта 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 ее название процедурная) основана на фон-неймановской модели компьютера, названной в честь предложившего ее 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 основных 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 ее адрес. Скорость доступа ко всем ячейкам одинаковая и она не зависит от адреса. Такие последовательности в программировании называют *массивами*. Ячейки могут содержать как данные (числа или символы), так и команды (т.е. 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 т.д.), 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, а в арифметико-логических регистрах находятся операнды арифметико-логических команд.

Команды ЦПУ подразделяются следующим образом:

- пересылки между ОП и регистрами ЦПУ;
- арифметико-логические команды;
- команды управления, включающие в себя команды перехода и некоторые специальные команды (например, команду 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 переходов, содержащие адрес команды, которая будет выполняться следующей, и позволяющие изменять нормальную последовательность выполнения команд.

Основные понятия императивных языков программирования (ИЯП) представляют собой абстракции основных понятий фоннеймановской 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 др.).

Понятие простой переменной абстрагирует понятие ячейки памяти. Кроме простых переменных в императивном языке содержатся составные (т. е. 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; `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 и арифметико-логических команд (и даже команд перехода). Это действительно основной оператор в императивных языках.

Операторы управления (циклы, операторы выбора, перехода и т. п.) абстрагируют машинные команды перехода.

Операторы ввода-вывода обобщают машинные команды ввода-вывода.

Подробнее основные понятия ИЯП разбираются в гл. 5 нашего учебника.

Мы видим, что 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 С), 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 С. Для этого стиля характерно представление алгоритма решения задачи в виде последовательности шагов, каждый из которых в свою очередь представляется последовательностью более мелких шагов, и т.д., пока не получим шаг «размером» в один оператор.

Многие задачи по обработке данных (в том числе и наша) сводятся к следующим 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 должен реализовывать сам программист. Единственное понятие в языке С, соответствующее последовательности однородных элементов, это массив. Проблема в том, что массивы в языке С — это последовательности фиксированной и заранее известной длины. Для того чтобы упростить решение задачи, предположим, что «не очень длинная» последовательность содержит не более 1 024 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 следует не с начала, а с конца. В этом случае выполнение становится несколько более быстрым (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 данной программы сообщает о включении информации о библиотеке стандартного ввода-вывода, в следующих двух строках объявляются 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 этой функции объявлены переменные current (для ввода очередного элемента последовательности) и count (для хранения числа 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 так же, 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, операция присваивания может комбинироваться с другими операциями в выражении.

В нашем примере выражение в заголовке цикла

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

вызывает функцию getchar и затем присваивает ее значение переменной current, после чего сравнивает это же значение (т.е. 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 — это 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 следующие отношения:

- включение — «объект—подобъект» — включение объекта класса `X` в объект другого класса `Y`, т.е. 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 (но не владеет) ссылку 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 — это 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 на объектно-ориентированных языках решается задача о реверсировании входной последовательности. Сначала рассмотрим решение на языке C#.

Все рассуждения из предыдущего пункта по ходу решения задачи остаются справедливыми и здесь (ведь объектная парадигма в языке C# расширяет, но не отменяет императивную). Отличие объектного подхода в том, что здесь 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 т.д.), так и специализированных (динамическая строка произвольной длины). Используем 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 языке C: с ее вызова начинается выполнение консольных программ.

В первой строке мы объявляем строку `s` и сразу же вводим в нее целиком всю входную последовательность. Класс `Console` обладает объектом `In` класса `TextReader`, представляющим собой 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 (т.е. решает нашу задачу). После чего осталось только вывести реверсированный массив.

Теперь 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.

Однако императивный стиль программирования тоже обладает некоторыми достоинствами. Например, в приведенной программе на языке 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 языке C#. Однако легкость программирования на ООЯП в большинстве случаев перевешивает. Кроме того, и на языке типа C# можно программировать в императивном стиле. В нашем примере можно 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.

Функциональная парадигма

Основные понятия функциональных языков — функция и выражение. Выражение — это комбинация вызовов функций. Наряду с большим числом 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 сначала вычисляются выражения — фактические параметры, а 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 [34] — первом языке программирования, в котором была реализована функциональная парадигма.

Будем использовать один из самых популярных диалектов Лиспа — Коммон Лисп.

Прежде чем рассматривать решение, сделаем несколько замечаний о представлении программ на языке Лисп. Вызов функции имеет вид

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

Например,

```
(+ 2 3)
```

Определение функции имеет вид

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

Например,

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

Обращение к стандартной функции ввода следующее:

```
(read)
```

Однако эта функция возвращает не последовательность литер, а более сложную структуру — Лисп-выражение. Подробнее язык Лисп

рассматривается в подразд. 11.2. 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 увидеть, что примеры вызова и определения функции сами представляют собой списки. Иначе говоря, Лисп-программы не только обрабатывают списки, но и сами представляют собой списки. Следовательно, 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 ее на языке Лисп в функциональном стиле. Пусть на входе имеется список слов (символов). Требуется выдать этот список в обратном порядке. Заметим, что формулировка задачи не упростилась, а наоборот, усложнилась. Однако это не усложнит ее решение на Лиспе.

Поскольку функция read сразу вводит весь список, то необходимо написать функцию обращения списка. И такая функция в Лиспе имеется, 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 рекурсивная 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`). Обращение такого списка — это список, который состоит из двух списков: первый список представляет собой обращение хвоста (`(rev (cdr x))`), а второй список состоит из одного элемента — головы `x(car x)`.

Однако первый элемент списка — это необязательно список, поэтому из него следует сделать одноэлементный список с помощью

функции Лиспа `(cons a b)`. В результате работы этой функции формируется список с головой `a` и хвостом `b`. Таким образом, `(cons (car x) nil)` и есть требуемый одноэлементный список.

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

Специальная функция `if` Лиспа вычисляет свой первый аргумент (`(null x)`), и если он истинный, то возвращает вычисленный второй аргумент (`nil`), а в противном случае 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 (и один из них рекурсивный). Это отличительный признак чисто функциональных программ. Также отметим еще 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.

Следовательно, можно программировать в функциональном стиле и на ИЯП. Правда, делать это на императивных языках труднее, чем на языках типа Лисп, явно поддерживающих функциональный стиль.

1.3. Схема рассмотрения языков программирования

Конструкции языков программирования будем рассматривать по следующей схеме (подробнее см. в [18]): базис, средства развития и средства защиты.

Базис — это понятия и конструкции, встроенные в язык программирования, иначе говоря, это то, что «понимает» транслятор. Базис подразделяется на скалярный и структурный.

В скалярный базис входят элементарные (неделимые) типы данных и элементарные операции. Тип данных `integer` с машинной точки зрения имеет некоторую структуру, представляющую собой последовательность либо битов, либо байтов. Однако в большинстве ЯП целый тип представляет собой именно скалярную величину и относится к скалярному базису.

К структурному базису относятся встроенные в язык конструкции, которые имеют внутреннюю структуру, т.е. включают в себя другие конструкции языка. В структурный базис императивных языков входят составные типы, например массивы и записи (структуры), большинство операторов языка (за исключением совершенно тривиальных типа `break` или `continue` в C).

Базисы императивных языков программирования похожи друг на друга. И то, что появляются новые языки, а старые продолжают жить, обусловлено различиями не в базисе, а в средствах развития и защиты.

Средства развития — это аппарат, позволяющий добавлять в программы новые понятия (абстракции), которых не было в базисе. Уже в самом первом языке программирования Фортране появилось такое средство развития, как подпрограмма. Основное требование к средствам развития — возможность определять новые типы данных. В идеале новые типы почти не должны отличаться от базисных типов. Самыми мощными средствами развития из тех, которые мы будем рассматривать, являются классы.

Однако только одного аппарата развития недостаточно. Например, мало иметь средства создания новых типов данных. Необходимо определять их таким образом, чтобы соответствующие абстракции можно было легко употреблять, а также контролировать их цельность и корректность поведения. Именно этим занимаются **средства защиты** в языках программирования. К средствам защиты относятся, например, средства обработки исключительных ситуаций, механизм абстракции данных и др. Современные языки программирования отличаются от более ранних языков прежде всего тем, что у первых значительно усилены именно средства защиты.

Глава 2
ИСТОРИЧЕСКИЙ ОЧЕРК РАЗВИТИЯ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ

Можно условно выделить три периода развития языков программирования:

- период зарождения (1950 — начало 1960 гг.);
- период бурного роста (1960 — 1980 гг.);
- эволюционный период (1980 гг. и по настоящее время).

Период зарождения

Первым языком программирования в современном смысле этого слова стал язык Фортран, разработанный в 1954 — 1957 гг. в IBM под руководством Джона Бэкуса. Фортран (Fortran) — это транслятор формул (Formulae Translator). Авторы языка поставили целью создание инструмента, который позволил бы ученым и инженерам использовать близкую им нотацию, а не машинный код для создания программ. Фортран — пример языка, создатели которого достигли тех целей, которых добивались. Несмотря на то что с современной точки зрения Фортран подвергается справедливой критике, он является одним из самых успешных языков программирования. На этом языке активно программируют до сих пор.

Определим *языковую нишу* как совокупность проблемных областей, для работы в которых предназначен данный язык программирования. Одна из главных причин успеха Фортрана состоит в том, что он первым занял нишу языка научно-технических расчетов (НТР) для разнообразных математических и физических моделей, использующих интегральное и дифференциальное исчисления, разностные методы. Фортран позволил ученым (математикам, физикам и др.) самостоятельно разрабатывать программы для своих целей. Огромным достоинством Фортрана явилась его доступность на основных машинных архитектурах. В результате программистская база существенно увеличилась. Фортран показал, что язык высокого уровня помогает привлечь к программированию большее количество прикладных специалистов. Компьютер становится более полезным. Фортран продемонстрировал, что язык программирования высоко-

го уровня позволяет обеспечить *мобильность* (или *переносимость*) программ и знаний.

Существует два аспекта мобильности, обеспечиваемой использованием языков программирования: мобильность программного обеспечения и мобильность знаний. До появления Фортрана программа, написанная на одном машинном языке, не могла выполняться на машине другой архитектуры из-за несовместимости на уровне двоичных кодов. После появления Фортрана были разработаны миллионы строк программного кода, которые могли выполняться на машинах разной архитектуры, хотя до сих пор обеспечение полной мобильности программного обеспечения не достигнуто. Чем более сложной является задача, тем труднее перенести ее с одной машины на другую.

Однако более важным аспектом является мобильность знаний. До появления Фортрана при появлении новой или просто другой модели компьютера приходилось изучать новую систему команд, привыкать к новой управляющей программе (операционных систем тогда не было), переписывать все программы на новом машинном языке. После появления Фортрана при переходе с одной машины на другую необходимость переучиваться и переписывать программы отпала, что позволило специалистам сосредотачиваться на проблемной области, а не на изучении новых инструментов.

Фортран по многим причинам непригоден для индустриального программирования, однако в области высокопроизводительных вычислений он является ведущим языком. В настоящее время существует несколько версий Фортрана, самые популярные из которых Фортран 77 и Фортран 90.

Успех Фортрана дал толчок к появлению новых языков. В 1960 г. появился Алгол 60. Этот язык оказал огромное концептуальное влияние на последующие языки. Так, например, цикл `for` современных языков унаследован из Алгола. В Алголе впервые появилась *рекурсия*, а также понятия *стека* и *вложенных областей видимости*. Алгол был разработан группой ученых как язык описания алгоритмов — ALGOrithmic Language. В отличие от Фортрана (разработчики которого, как и все первооткрыватели, действовали «на ощупь») Алгол 60 проектировался под влиянием определенных языковых концепций. Впервые синтаксис языка был описан формально (с помощью так называемых бэкусо-науровых форм — БНФ). Это описание дало мощный толчок к исследованиям в области теории и практики реализации языков программирования. После Алгола описания языков программирования стали выглядеть следующим образом:

- словесное описание лексики;
- формальное описание синтаксиса с помощью БНФ (или их разновидностей);
- словесное описание семантики введенных понятий.

Была достигнута определенная унификация в описаниях языков программирования. Хотя Алгол и был языком «номер два» в своей нише (НТР), но он получил достаточно большое распространение в Европе (в том числе и в СССР) именно как первый язык программирования, на котором учили программированию студентов. Языковая ниша Алгола 60 — образовательная, ее он удерживал три десятка лет, доминируя в университетах.

Не будет преувеличением сказать, что Алгол 60 — предок большинства современных языков программирования, включая Паскаль, Ада, С и др.

Примерно в одно время с Алголом (1959 г.) по инициативе Министерства обороны США появился Кобол (COBOL — Common Business-Oriented Language), который использовался для коммерчески ориентированных приложений (бухгалтерских и финансовых отчетов). Вследствие ранней стандартизации язык получил широкое распространение. Кобол до 1990 гг. XX в. занимал господствующее положение в нише программ для работы с большими БД и программ для автоматизации финансовых расчетов. «Проблема 2000 года» появилась из-за того, что в Коболе был специальный тип данных — дата, под год в котором отводилось две десятичные цифры. Поэтому на рубеже тысячелетий пришлось переписывать многие приложения для работы с финансовыми базами данных. Если бы не проблема 2000 г., возможно, многие программы на Коболе до сих пор бы использовались.

Важное влияние на последующее развитие программирования оказал язык Лисп (Lisp — List Processing), разработанный Джоном Маккарти в 1958 г. для решения задач, позже названных задачами искусственного интеллекта. Лисп активно используется до сих пор, он стал родителем целого направления — функционального программирования, хотя следует отметить, что в современных вариантах Лиспа используются все основные парадигмы, в том числе императивная и объектно-ориентированная. В настоящее время самый популярный диалект Лиспа — Common LISP. Широко известен близкий потомок Лиспа — Scheme.

В настоящее время функциональные языки не относятся к числу индустриальных. Основная причина — относительная неэффективность (по сравнению с языками, основанными на императивной парадигме). Еще в 1978 г. «отец Фортрана» Джон Бэкус предложил отказаться от парадигмы Фон Неймана (императивного программирования) и использовать методику функционального программирования. В настоящее время идеи функционального программирования переживают «ренессанс» и начинают проникать и в индустриальные языки.

Заметим, что уже в период зарождения языков программирования были высказаны и частично реализованы идеи, которые до сих пор оказывают огромное влияние на их развитие.

Период бурного роста

В этот период ежегодно появлялись сотни новых языков программирования. К 1967 г. только в США активно использовалось порядка 400 языков программирования. В этот период окончательно сформировались основные концепции языков программирования (абстрактные типы данных, инкапсуляция, полиморфизм и многие другие), которые развиваются до сих пор.

Также в этот период были созданы языки C и Паскаль.

Интересно, что в течение всего данного периода неоднократно предпринимались попытки создания единого универсального языка программирования, который покрывал бы практически все проблемные области и был бы достаточно адекватен, чтобы на нем программировать. Заметим, что все эти попытки провалились. Единого языка не существует до сих пор, хотя в разных проблемных областях (нишах) можно выделить лидеров.

Язык PL/1 был одним из первых претендентов на роль единого языка (1964 г.). PL/1 был создан как объединение конструкций из самых популярных к тому времени языков (Фортрана, Алгола 60, Кобола и некоторых других). Этот язык широко использовался для программирования на больших компьютерах (мейнфреймах) компании IBM (для которых он и создавался), но для других архитектур он оказался непригоден, и в настоящее время практически не используется. Основное концептуальное влияние PL/1 состояло в том, что разработчики языков программирования осознали, что простое объединение разнородных механизмов без тщательной проработки и адаптации невозможно.

Другой претендент на роль универсального языка — Алгол 68 интересен прежде всего с теоретической точки зрения. Этот язык разрабатывался как преемник Алгола 60, но не получил такого широкого распространения. Главная проблема Алгола 68 — сложность: он имел очень сложное описание и оказался сложным для реализации. Данный язык воплотил принцип *ортогональности* языковых конструкций, т.е. их независимость друг от друга. Принцип ортогональности требует, чтобы везде, где встречается одна конструкция, могла встречаться и другая. Например, любая конструкция Алгола 68 возвращает некоторое значение (т.е. может быть элементом выражения), в том числе и оператор. С другой стороны, любое выражение может рассматриваться как оператор. Понятие «выражение-оператор», которое есть сейчас во многих языках программирования (C, C++, C#, Java), досталось в наследство от Алгола 68.

Язык Паскаль, разработанный Никлаусом Виртом в 1969—1970 гг., явился полной противоположностью Алгола 68. Простой, понятный и мощный язык стал одним из основных языков для обучения студентов программированию. На основе Паскаля Н.Вирт позже создал языки Модула 2, Оберон и Оберон 2.

Самым популярным диалектом Паскаля стал ТурбоПаскаль (продукт компании Borland), отличавшийся мощным механизмом модульности и объектно-ориентированными возможностями. Автор этого языка Андерс Хейльсберг позже создал язык Delphi на основе ТурбоПаскаля, а после перехода в компанию Microsoft — язык C#.

Практически одновременно с Паскалем был разработан язык С. Автор этого языка Деннис Ритчи создал его, с одной стороны, весьма простым, с другой — очень гибким. Конструкции языка С естественным образом отображались на машинную архитектуру (как и конструкции языка Паскаль) и позволяли использовать фундаментальные особенности архитектуры (что не всегда можно сделать на Паскале). В результате язык С прочно занял нишу, которая раньше была занята машинными языками, — системное низкоуровневое программирование (разработка операционных систем, драйверов устройств ввода-вывода и т.д.).

На основе языка С в 1980-х гг. были разработаны языки C++ и Objective C, дополнившие С объектно-ориентированными свойствами.

Автор языка C++ Бьярне Страуструп сделал язык максимально совместимым с языком С (что позволило, например, включить в язык C++ стандартную библиотеку языка С). Объектно-ориентированные свойства языка C++ опирались на понятия языка Симула 67, разработанного в 1960-е гг. в Норвегии К. Нюгардом и У. Далем. Язык C++ в настоящее время является одним из самых популярных языков программирования.

Язык Objective C так же, как и язык C++, использовал С как базу, но объектно-ориентированные свойства Objective C использовали концепции языка SmallTalk — первого «чистого» объектно-ориентированного языка. Язык SmallTalk оказал большое влияние на развитие других объектно-ориентированных языков, но по эффективности сильно им уступал. Язык Objective C одно время оказался в тени своего конкурента языка C++, но в настоящее время обрел «второе дыхание» в связи с распространением платформы iOS компании Apple, в которой он является одним из основных языков разработки как системных, так и прикладных программ.

Одним из самых значительных языков программирования рассматриваемого периода стал язык Ада (1980 г.) — самый яркий претендент на роль универсального единого языка. Он был разработан на основе языка Паскаль и воплощал в себе базовые концепции технологии программирования в том виде, как ее понимали в конце 1970-х гг. Язык был разработан по заказу Министерства обороны США в соответствии с тщательно сформулированными требованиями и в результате открытого конкурса. Язык Ада должен был заменить более трехсот пятидесяти языков программирования, использовавшихся подрядчиками Пентагона при разработке заказного программного обеспечения (в основном в области встроенных

систем реального времени). Несмотря на мощную финансовую, законодательную и административную поддержку, язык Ада так и не стал основным языком для разработки программного обеспечения (даже в системе Минобороны США). Основными причинами этого явились большая его сложность (проблему усугубило официальное запрещение создания подмножеств Ады) и отсутствие поддержки объектно-ориентированной парадигмы (объектно-ориентированное расширение Ады появилось только в 1995 г.). Тем не менее язык продолжает использоваться и в настоящее время (хотя и не так активно, как предполагалось при его создании).

Эволюционный период

Последние двадцать лет развития языков программирования характеризуются уменьшением числа вновь созданных языков. Они продолжают регулярно появляться, но говорить можно только о единицах новых появившихся языков в год (в отличие от десятков и даже сотен ежегодно появлявшихся языков в предыдущие десятилетия).

Концептуально почти все новые языки поддерживают объектно-ориентированную парадигму.

Особенность эволюционного периода — появление большого числа языков для web-программирования, т. е. языков, предназначенных для встраивания в web-серверы и клиентские web-приложения. Самые популярные языки этого вида — Python, Perl, PHP, Ruby, JavaScript. Из-за специфики проблемной области большинство связываний в этих языках — динамические (понятие связывания рассматривается в гл. 3), поэтому большинство этих языков интерпретируемые.

В 1995 г. компания Sun объявила о разработке языка Java — одного из самых популярных в настоящее время. Синтаксис Java и ряд понятий в нем похожи на C++. Основное достоинство Java — независимость от платформы (как это достигается см. в гл. 4). Кроме того, по сравнению с конкурентами (такими, как C++ и C#) Java более простой язык. Он обладает большим набором библиотек и имеет множество реализаций. На мобильных платформах язык Java является одним из основных средств разработки.

Язык C#, появившийся в 1999 г., достаточно похож на Java (хотя имеются и серьезные отличия). В настоящее время он является основным языком программирования для платформы .NET компании Microsoft. Особенностью этой платформы является многоязычность. Все языки платформы поддерживают общеязыковую инфраструктуру (CLI — Common Language Infrastructure), которая включает в себя, в частности, общую систему типов (CTS — Common Type System). Все базисные типы языка C# имеют соответствующий аналог в CTS. В CLI входит также единый набор стандартных библиотек, следовательно, при переходе от языка к языку не надо изучать новые

библиотеки. Все языки транслируются в промежуточный код (IL — Intermediate Language). Отличие .NET от других систем с промежуточными языками состоит в том, что код на IL не интерпретируется, а транслируется в двоичный машинный код непосредственно при загрузке программы на выполнение. Такая технология называется *динамической трансляцией* (JIT — just-in-time).

В отличие от языков C++, Objective C, Delphi языки Java и C# изначально проектировались как чистые объектно-ориентированные языки.

Подробнее о генеалогии языков программирования см. [4, 21].

Заключая исторический очерк, отметим, что не существует «лучшего» во всех отношениях языка программирования. У каждого языка есть свои достоинства и недостатки. В этой связи особенно актуально изучение основных концепций языков программирования в сравнении.

Глава 3

ОСНОВНЫЕ ПОНЯТИЯ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ: ДАННЫЕ, ОПЕРАЦИИ И СВЯЗЫВАНИЕ

Прежде чем рассматривать основные понятия современных языков индустриального программирования, обратимся к базисным понятиям, которые не зависят от стиля программирования и, следовательно, применимы к любым его языкам. Например, понятие «оператор» актуально только для императивных языков, поскольку основным назначением оператора является изменение состояния (памяти, порядка выполнения и т.д.), т.е. побочный эффект. В функциональном программировании побочные эффекты запрещены, поэтому понятия «оператор» в чистом функциональном языке быть не может. Является ли универсальным понятие «класс»? Тоже нет, поскольку существуют объектно-ориентированные языки, в которых нет такого понятия. К их числу относятся так называемые «прототипные» языки, например JavaScript [14, 16], Self, IO.

К фундаментальным понятиям любого языка программирования относятся **данные, операции и связывание**.

Любая компьютерная программа обрабатывает некоторые данные, причем процесс обработки включает в себя выполнение некоторых операций (вычислений) над этими данными. Конкретная номенклатура данных и операций определяется каждым языком (входит в базис языка — см. подразд. 1.3) и может сильно варьироваться. Например, язык JavaScript содержит единственный числовой тип `Number`, который включает в себя всевозможные представимые в компьютере числа. Язык Java имеет два вещественных типа (`double`, `float`) и четыре целых (`byte`, `short`, `int`, `long`). В языке Ада можно определить потенциально бесконечное множество числовых типов, а в некоторых языках числового типа нет вообще (как правило, эти языки имеют строковый тип данных, который может преобразовываться в числа для выполнения арифметических операций).

Заметим, что данные и операции тесно связаны, иногда настолько, что можно говорить о *дуализме данных и операций*.

В качестве примера такого дуализма рассмотрим строковый тип данных. Множество его значений — всевозможные последовательности литер. Что такое «длина строки»? На первый взгляд — безусловно операция. Например, в языке С есть специальная библиотечная функция `strlen(s)`, которая вычисляет длину строки `s`. Так как

строка в C представляется последовательностью байтов, заканчивающейся нулем, то длина строки всегда подразумевает вычисление, т.е. это операция. С другой стороны, во многих языках длина строки — это данные, хранимые вместе со строкой. Так что вызов функции `Len(s)` в языке Бейсик или функции `Length(s)` в языке Delphi или TurboPascal — это считывание соответствующей переменной из внутреннего представления строки и, конечно, транслятор не вызывает в этом случае никакой функции и ничего не вычисляет.

Другой пример дуализма данных и операций — использование подпрограммы-генератора данных (т.е. вычисления) вместо хранения данных в какой-либо структуре.

Понятие дуализма данных и операций отражено в ряде объектно-ориентированных языков программирования (C#, Delphi, Visual Basic) с помощью конструкции «свойство» (property). Свойство выглядит как единица данных с точки зрения пользователя и как пара функций с точки зрения реализации. Одна функция возвращает значение свойства, а другая устанавливает его, причем обе функции могут выполнять довольно значительные вычисления. Подробнее механизм свойств рассматривается в гл. 7.

Однако при проектировании новых типов данных вполне правомерен следующий вопрос: что «первичнее», данные или операции? Современный взгляд на типы данных обеспечивает достаточно однозначный ответ на этот вопрос: операции первичнее. Даже если часть состояния объекта представляется как некоторая переменная (например, длина), то доступ к этой переменной должен предоставляться с помощью функции. Механизм свойств реализует эту точку зрения. Другими словами, пользователь нового типа данных не должен зависеть от того, является ли свойство функцией (операцией) или переменной. Понятно, что такая независимость достигается, если новый тип данных используется только посредством вызовов функций (подпрограмм), представляющих собой единственно доступные операции над типом. Такие типы данных, называемые **абстрактными**, очень важны в современном программировании. Абстрактные типы данных в языках программирования рассматриваются в гл. 7.

Каждый объект данных (иногда объекты данных называют *переменными*) характеризуется набором атрибутов. Полный набор атрибутов, конечно, специфичен для конкретного языка программирования, однако можно выбрать достаточно универсальный набор атрибутов [4], состоящий из шести элементов:

- имя;
- адрес;
- тип;
- значение;
- время жизни;
- область действия.

Перед обсуждением этого набора рассмотрим понятие связывания, которое не менее важно, чем данные и операции, хотя и менее очевидно.

Связывание — это процесс установления связи между элементом программы и конкретным атрибутом или характеристикой. Как правило, связывание сводится к выбору атрибута из (конечного) набора атрибутов. Примеры связываний будут приведены далее. **Время связывания** — это момент установления этой связи. При этом нас, конечно, интересует не конкретное (астрономическое) время, а несколько основных видов времени связывания [25].

Во время выполнения программы. В этом случае вид связывания называется ***динамическим*** и включает в себя следующие категории:

- *при входе в блок (тело подпрограммы).* Например, в этот момент локальные переменные блока и формальные параметры подпрограммы связываются с атрибутом-адресом. Связывание фактических и формальных параметров тоже происходит в этот момент. Такой способ связывания называется ***квазистатическим***;
- *в произвольной точке программного кода.* К этому виду связывания относится, например, связывание переменной и значения, происходящее во время выполнения оператора присваивания, а также связывание адреса и объекта данных при динамическом распределении памяти и выбор обработчика исключительной ситуации при распространении исключения (см. гл. 9).

При трансляции. В этом случае вид связывания называется статическим и подразделяется на следующие категории:

- *по выбору программиста.* Это, например, связывание объекта данных с именем, а имени с типом (в языке, подобном Паскалю) и т.д.;
- *по выбору транслятора.* Это, например, связывание относительного адреса (не путайте с абсолютным) локальных переменных (в языке, подобном Паскалю или С). Конкретный вариант такого связывания, как правило, определяется реализацией языка;
- *по выбору компоновщика (загрузчика, редактора связей).* Все индустриальные языки допускают (или даже требуют) написание многомодульных программ. При ссылке на переменную или вызове подпрограммы из другого модуля связать конкретные адреса с местом ссылки или вызова можно только на этапе компоновки, когда доступны все используемые модули и библиотеки и известен их относительный порядок.

Можно выделить также моменты связывания *во время реализации языка* (например, выбор максимального и минимального значений типа `int` при реализации транслятора с языка С) и *во время определения языка* (например, выбор номенклатуры типов данных в языке).

Понимание семантики и времени конкретного связывания часто является решающим при освоении языка. Например, отличие семантики виртуальных функций в языках C++ и C# от невиртуальных (см. подразд. 8.2) состоит *только* во времени связывания вызова функции с конкретной функцией класса. Для виртуальных функций (и при вызове через ссылку или указатель) это время динамическое, а для невиртуальных — статическое.

Еще одно важное понятие, присутствующее в любых языках программирования, — это имя.

Имя — это строка символов, служащая для обозначения некоторой сущности в программе. В большинстве языков имя — это идентификатор, т.е. последовательность букв и цифр, начинающаяся с буквы. Связывание сущности с именем, как правило, происходит статически в специальных конструкциях языка, называемых объявлениями. Такое вхождение имени в текст программы называется *определяющим*. Подавляющее большинство индустриальных языков программирования требует, чтобы определяющее вхождение имени всегда предшествовало его использованию (т.е. *использующим* вхождениям). Исключение делается только для указательного типа в случае рекурсивных структур данных. Например, в Паскале, C, C++ допускается использующее вхождение имени типа, когда объявляется указатель на тип (указатели рассматриваются в гл. 5):

```
typedef struct link * plink;
     //имя plink еще не определено, но уже использу-
ется
struct link {
     int info; plink next;
};
```

Заметим, что не все объекты данных могут иметь имя. Например, литеральные константы в C++ (целые, строковые и т.д.):

```
int i = 128;
ifstream input("input_file.txt");
```

Однако понимать и (главное) модифицировать такие программы легче, если таким константам будут присвоены явные и ясные имена:

```
const BUFFER_SIZE = 128;
const char * INPUT_FILE_NAME = "input_file.txt";
```

В то же время существуют *анонимные* объекты данных, которые не могут иметь имени. Например, это объекты, размещаемые в динамической памяти.

Пример на языке C++:

```
class X;
X * pX = new X();
```

Указатель pX ссылается на объект класса X, размещенный в динамической памяти. Этот объект никак не именуется и доступен только через указатель на него.

В языках Java и C# принята так называемая референциальная модель объекта, в которой все объекты классов размещаются исключительно в динамической памяти. Имена есть только у объектов простых типов данных и ссылок на объекты. Такой подход характерен для большинства объектно-ориентированных языков (примечательным исключением является язык C++).

Глава 4

ВИРТУАЛЬНАЯ МАШИНА ЯЗЫКА. ИЕРАРХИЯ ВИРТУАЛЬНЫХ МАШИН

Рассмотрим подробнее процесс выполнения на компьютере программ, написанных на каком-либо языке программирования. Каждый компьютер обладает своей системой команд (машинных операций), которые обрабатывают данные, записанные в оперативной памяти. Данные представляются в двоичной форме (наиболее просто реализуемой с точки зрения аппаратуры). Машинные команды, как и обрабатываемые ими данные, также записываются в оперативную память, откуда последовательно выбираются для обработки центральным процессором. Таким образом, машинные программы тоже представлены в двоичной форме, и компьютер может выполнять только такие программы. Непосредственно выполнить на таком компьютере программу, написанную на языке программирования типа С или Лисп, нельзя.

Для выполнения программы на каком-либо языке программирования существует специальная программа — *транслятор*. Транслятор сопоставляет программе на языке программирования эквивалентную (т. е. осуществляющую эквивалентные вычисления) машинную программу. По способу реализации такого сопоставления трансляторы подразделяются на две категории: компиляторы и интерпретаторы. Обе категории трансляторов обрабатывают программу на языке программирования (назовем его L), но делают это по-разному.

Компилятор читает программу на языке L и переводит ее в двоичную программу на машинном языке. Далее эта программа непосредственно выполняется на компьютере.

Интерпретатор читает программу на языке L и сразу же ее выполняет. Таким образом, интерпретатор не порождает эквивалентную программу на машинном языке, а сразу выполняет эквивалентные вычисления. Подробнее процессы компиляции и интерпретации разбираются в ч. III данного учебника. Заметим, что для любого языка L можно написать как компилятор, так и интерпретатор.

По разного рода причинам (часть из них обсуждается в ч. III данного учебника) время выполнения откомпилированной программы меньше, чем время на интерпретацию той же программы (в этом смысле говорят, что компиляторы «эффективнее», чем интерпретаторы). При этом разница во времени выполнения откom-

пилированной и интерпретируемой программ на некоторых языках весьма существенная. Такие языки называют **компилируемыми**. Большинство индустриальных языков программирования относятся к этому типу.

Однако есть языки, для которых разница между компиляцией и интерпретацией не столь велика, а интерпретатор проще реализовать, чем компилятор. Такие языки называют **интерпретируемыми**. Типичным примером интерпретируемого языка является Лисп. Вспомним, что программы на Лиспе обрабатывают списки. С другой стороны, программы на Лиспе представляются в виде списков. Неслучайно в Лиспе есть встроенная функция `eval` (сокращение от evaluate — вычислить значение), которая имеет один аргумент — список. Этот список рассматривается как Лисп-программа и вычисляется. По определению, `eval` — это интерпретатор языка Лисп. Можно, конечно, написать и компилятор языка Лисп (и такие реализации существуют), но интерпретатор Лиспа будет компактнее и удобнее компилятора.

Какие же свойства языка влияют на компилируемость или интерпретируемость? Вспомним понятие связывания. Чем больше статических связываний в языке программирования, тем более эффективна компиляция. Рассмотрим несколько связываний, которые сильно влияют на эффективность откомпилированных программ.

Первый пример — связывание переменной с типом данных. Языки, в которых это связывание статическое, называются языками со *статической типизацией* (или *статическими* языками). Как правило, эти языки требуют опережающего объявления имен. В таких языках есть понятие объявления, связывающего переменную с ее атрибутами (в частности, с типом данных). Изменить тип переменной нельзя. Значение переменной может относиться только к объявленному типу данных. Если тип значения, помещаемого в переменную (например, при выполнении оператора присваивания), отличен от типа переменной, то либо транслятор выдает сообщение об ошибке, либо значение преобразуется к типу переменной (такая операция преобразования иногда называется *приведением типа*).

Языки, в которых связывание переменной с типом динамическое, называют языками с динамической типизацией (или *динамическими* языками). Иногда в литературе встречается термин «бестиповые языки», но мы не будем его использовать, так как даже в бестиповых языках типы данных есть (языков без типов данных просто не существует). Правильнее говорить о бестиповых переменных в динамических языках. Такие переменные могут хранить значения любого допустимого типа. Типичный пример динамически типизированного языка (как Лисп) — язык JavaScript. Переменные в нем объявлять необязательно, но при этом существует понятие объявления переменной, например:

```
var x = 0,y = "string", z;
```

В этом примере x, y, z связываются со своими именами. При этом две переменные также инициализируются значениями целого и строкового типа (третья получает неопределенное значение). Однако далее в эти переменные можно поместить любые другие значения:

```
x = y; y = -1; z = x;
```

Интересно, что в статически типизированном языке C# (начиная с версии 3.0) существует синтаксически похожее понятие:

```
var x = 0,y = "string";
```

Однако смысл этого объявления сильно отличается от предыдущего примера. Здесь переменные связываются не только с именами и инициализирующими значениями, но и с типами, которые определяются типами инициализирующих значений (очевидно, что инициализаторы в таких объявлениях обязательны, именно поэтому переменная z из примера на JavaScript опущена в объявлении на C#). Говорят, что транслятор *выводит* тип переменной из инициализатора, поэтому это объявление эквивалентно следующему:

```
int x = 0; string y = "string";
```

Статическая типизация переменных обусловливает статичность многих других видов связывания, например связывание знака встроенной операции с ее смыслом. Так, во фрагменте программы на языке C# первое вхождение операции «+» — это операция над целыми числами, а второе — операция сцепления двух строк:

```
int a = 0, b = 1, c;
c = a + b; //сложение целых
string s1 = "STR", s2 = "ING", s3;
s1 = s2 + s3; // сцепление строк,результат – STRING
```

Теперь рассмотрим фрагмент на языке JavaScript:

```
var a, b, c;
... //какие-то операторы, рассмотрим их позже
c = a+b;
```

Какая конкретная операция соответствует здесь знаку «+» сказать нельзя, так как связывание динамическое и происходит в момент вычисления выражения a+b. Определяется типами значений переменных a и b. Если вместо троеточия в примере стоят операторы `a = 1; b = 2;` то выполнится операция сложения целых, а если операторы `a = "STR"; b = "ING";` то выполнится операция сцепления строк.

Очевидно, что выполнение программ в статических языках требует меньше времени. Кроме того, ошибки в программах на статических языках легче обнаружить, так как типы операндов известны транслятору, поэтому некорректные операции, например сложение целого числа и строки, обнаруживаются во время трансляции.

Относительность процессов компиляции и интерпретации подчеркивает тот факт, что в реализации некоторых компьютерных архитектур (в том числе и распространенной архитектуры x86) машинный язык сам является интерпретируемым. Это связано с тем, что машинный язык в архитектуре x86 является довольно сложным и включает в себя сотни машинных операций, множество режимов адресации и т. п.

Непосредственная реализация процессоров этой архитектуры весьма сложна, поэтому инженеры-системотехники используют понятия *микрокода* и *микроядра*. Микроядро можно рассматривать как специализированный компьютер, являющийся частью более крупного центрального процессора. Микроядро обладает своей архитектурой, памятью и системой команд (микроопераций). Набор микроопераций достаточно невелик (существенно меньше, чем набор команд архитектуры x86) и проще реализуется. Специализация микроядра заключается в интерпретации команд основного машинного языка. Упрощенно говоря, каждой команде процессора x86 соответствует набор микрокоманд (называемый микропрограммой, или микрокодом), который и интерпретирует (выполняет) соответствующую команду. Таким образом, микроядро является интерпретатором микрокода, реализованным аппаратно, в отличие от интерпретатора языка Лисп, реализованного программно. Подробнее машинные архитектуры (включая микропрограммные) рассмотрены в [10].

Заметим, что пользователям компьютеров и даже прикладным программистам совершенно безразлично, как именно реализована машинная архитектура: непосредственно или с помощью микроядра.

Из всего сказанного вытекают два следствия.

1. Если машинный язык можно интерпретировать с помощью аппаратно реализованного интерпретатора, то это можно сделать и с помощью программного интерпретатора. Если мы имеем такую программу — интерпретатор машинного языка какой-либо архитектуры, то мы можем выполнять любые программы для этой архитектуры. Иначе говоря, мы имеем программно-реализованный компьютер. Такой компьютер называют *виртуальным* (вообще в программировании виртуальным называют любую сущность, которая полностью или частично реализована программно). Современные процессоры включают в себя команды, специально поддерживающие виртуализацию (т. е. моделирование исполнения команд). Именно благодаря такой программно-аппаратной реализации на современных компьютерах возможен одновременный запуск различных

вариантов операционных систем и прикладных программ. Правда, моделируемая машинная команда работает медленнее, чем команда, непосредственно выполняемая аппаратурой, поэтому программы под управлением виртуальных машин работают «медленнее», чем на «родном» компьютере, однако в ряде случаев такими накладными расходами можно пренебречь.

2. Язык программирования может интерпретироваться не только программно, но и аппаратно (по аналогии с машинным языком). Действительно, некоторые реализации трансляторов для языков программирования были выполнены аппаратно (точнее, программно-аппаратно). Также были разработаны компьютеры с машинным языком, близким к языкам Алгол-60, Лисп, Ада и др. Правда, широкого распространения эти машинные архитектуры не получили, поскольку наилучшего для всех проблемных областей языка не придумано (вспомним гл. 2), и постоянно появляются новые языки, в чем-то превосходящие уже существующие. Следовательно, выгоднее иметь универсальную машинную архитектуру и реализовывать трансляторы с новых языков в этой архитектуре.

Еще раз отметим, что прикладным программистам-пользователям машинной архитектуры в принципе безразлично, как именно (чисто программно, аппаратно, программно-аппаратно) реализована эта архитектура. Таким образом, мы приходим к концепции *виртуальной машины языка программирования*. Машинным языком такой виртуальной архитектуры является собственно язык программирования, а способ реализации (программно-аппаратная интерпретация, чисто программный интерпретатор, компилятор) не имеет значения.

Понятие виртуальной машины полезно, потому что позволяет почти полностью абстрагироваться от особенностей машинного языка и конкретной архитектуры компьютера, разновидности процессора и других технических характеристик вычислительной системы.

Однако интересен следующий вопрос: от чего нельзя абстрагироваться при программировании на некотором языке (или, что то же самое, при программировании на виртуальной машине языка)? Иными словами, что нужно знать при программировании на языке, кроме самого языка программирования? Ответ на этот вопрос такой: необходимо знать способы взаимодействия виртуальной машины языка с *операционной системой*. Операционной системой (ОС) компьютера называется комплекс программ, управляющих ресурсами компьютера. Заметим, что компьютер под управлением операционной системы тоже можно рассматривать как виртуальный. Подобно тому, как виртуальная машина языка позволяет абстрагироваться от машинной архитектуры, виртуальный компьютер ОС позволяет абстрагироваться, например от деталей взаимодействия с внешними устройствами ввода-вывода. Виртуальный компьютер ОС предоставляет возможность управления внешними устройствами, файлами дан-

ных, процессами и много других. Подробнее устройство и функции операционных систем рассматриваются, например в [30].

Форма представления интерфейса с виртуальной машиной ОС зависит, прежде всего, от языка программирования. Например, в языках C и C++ — это библиотеки системных вызовов и структур, в языке C# — библиотеки классов, объединенных в сборки, а в языке Java — библиотеки классов, объединенных в пакеты, и т.д. В данном учебнике мы не будем заниматься интерфейсами с ОС.

В некоторых случаях понятие виртуальной машины языка (и ее реализации) тесно связано с понятием промежуточного языка.

Дело в том, что конструкции языков программирования плохо подходят для непосредственной интерпретации. Они имеют сложную структуру, представляются в виде последовательности символов и ориентированы не на машинную реализацию, а на восприятие человеком-программистом (что неудивительно). Поэтому «чистых» интерпретаторов языков программирования крайне мало. Почти все реальные интерпретаторы используют промежуточный язык (ПЯ), который обладает следующими двумя свойствами:

- легкость интерпретации (по сравнению с исходным языком L);
- легкость перевода программ с языка L в ПЯ (по сравнению с переводом в машинный код).

В этом случае процесс трансляции и выполнения программы включает в себя два этапа: перевод программы с языка L в ПЯ и непосредственная интерпретация текста на ПЯ. Интерпретатор ПЯ называют в этом случае *исполнительной системой* (или *системой времени выполнения*). Выгода этой схемы состоит в том, что в силу свойств ПЯ обе компоненты (транслятор в ПЯ и интерпретатор ПЯ) достаточно просты и компактны. Кроме того, схема обеспечивает дополнительные преимущества.

Рассмотрим задачу реализации языка L для N различных машинных архитектур. При этом требуется, чтобы все реализации работали одинаково (т.е. оттранслированные программы должны обеспечивать получение одних и тех же результатов на всех архитектурах). В обычном случае требуется разработать N различных и достаточно сложных программ-трансляторов (неважно компиляторов или интерпретаторов). Если же использовать единый промежуточный язык, то требуется одна программа-транслятор с языка L в ПЯ и N интерпретаторов ПЯ для каждой архитектуры (как вы думаете, на каком языке лучше всего написать программу-транслятор?). Каждая из этих программ существенно проще, чем интерпретатор или компилятор с языка L, поэтому суммарные усилия по реализации всех трансляторов меньше. Кроме того, добавление реализации для новой архитектуры также существенно проще, потому что интерпретатор ПЯ существенно проще компилятора или интерпретатора L. Подробнее ПЯ и его интерпретации рассматриваются в ч. III данного учебника.

Одна из первых реализаций этой схемы — система UCSD-Pascal, использовавшаяся для организации выполнения студенческого практикума на множестве компьютеров разных архитектур. Входной язык этой системы Паскаль, а промежуточный язык — P-code.

Такой же подход использовался при реализации первого чистого объектно-ориентированного языка SmallTalk. Авторы SmallTalk предложили свой вариант ПЯ — байт-код. Такое название выбрано, потому что каждая команда байт-кода имеет простую структуру — код операции размером один байт и далее операнд (операнды) переменной длины. Программы на SmallTalk транслируются в байт-код, который далее интерпретируется.

Еще одним важным примером использования ПЯ является язык Java. Вместе с Java была разработана спецификация виртуальной Java-машины (Java Virtual Machine — JVM). Машинным языком JVM является байт-код JVM. Байт-код не зависит от конкретной машинной архитектуры и достаточно легко интерпретируется. Спецификация JVM содержит описание байт-кода и описание среды времени выполнения (Java run-time environment — Java RTE). Функции Java RTE похожи на функции ОС, но не зависят ни от одной ОС. Разработчик языка (компания Sun Microsystems) представила компилятор javac, транслирующий программы с языка Java в байт-код JVM. Таким образом, JVM в терминах виртуальных архитектур — это прослойка между виртуальной машиной языка Java и виртуальным компьютером ОС. Наличие JVM позволяет абстрагироваться не только от конкретной архитектуры, но и от операционной системы, поэтому программы на Java могут выполняться везде, где есть реализация JVM. Реализация JVM подразумевает реализацию интерпретатора байт-кода и реализацию Java RTE. В настоящее время реализации JVM существуют практически для всех операционных систем.

Некоторым недостатком систем с ПЯ является невысокая скорость интерпретации ПЯ. Одно из очевидных средств увеличения

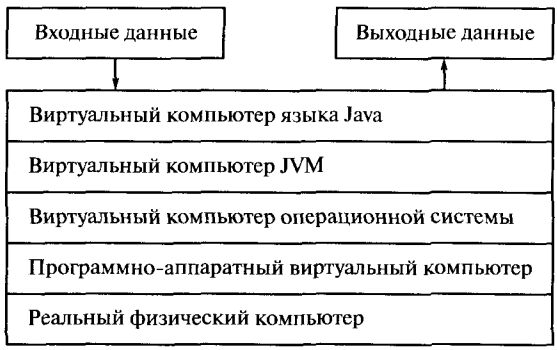


Рис. 4.1. Иерархия виртуальных компьютеров для Java-приложения

их быстродействия — аппаратная реализация ПЯ. Например, были разработаны микропроцессоры, непосредственно интерпретирующие байт-код SmallTalk (микропроцессор Катана [32]), а также архитектура процессоров picoJava для интерпретации байт-кода JVM [29].

Итак, при создании приложений программист использует виртуальную машину языка программирования, которая представляется на самом деле иерархией виртуальных компьютеров. Рассмотрим эту иерархию на примере приложения, написанного на Java (рис. 4.1). В основе иерархии лежит реальное физическое устройство — компьютер. Машинный язык этого компьютера — микрокод. Следующий уровень иерархии — программно-аппаратный виртуальный компьютер (ПАВК), машинный язык которого интерпретируется микропрограммно. Далее находится слой виртуального компьютера ОС, реализуемый программами на языке ПАВК. Следующий слой — виртуальная машина байт-кода (реализация JVM), машинным языком которой служит байт-код JVM. Вершиной иерархии является виртуальная машина языка Java (машинный код — язык Java).

ЧАСТЬ II

ВВЕДЕНИЕ В СОВРЕМЕННЫЕ ЯЗЫКИ ПРОГРАММИРОВАНИЯ

Глава 5

БАЗИС СОВРЕМЕННЫХ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ

5.1. Простые типы данных, операции над ними

В соответствии со схемой, приведенной в подразд. 1.3, рассмотрим основные конструкции языков программирования, начиная со скалярного базиса.

Все значения простых типов данных (называемых иногда *элементарными*, или *примитивными*) являются атомарными, т.е. они не имеют внутренней структуры.

Классификация простых типов данных приведена на рис. 5.1. В данном подразделе рассмотрим все простые типы данных, кроме подпрограммного, который имеет смысл обсуждать вместе с понятием подпрограммы (см. гл. 6).

Арифметические типы данных

В компьютерах эти типы данных представляют числа, и поэтому действительно являются основными.

Как уже отмечалось, в некоторых языках (JavaScript) существует единственный числовой тип (Number), представляющий все допустимые числа. С точки зрения упрощения программ это, конечно, удобно, однако в большинстве индустриальных языков программирования арифметические типы данных разделяются на два вида: целые (для представления целочисленных значений) и вещественные (для представления чисел с дробной частью). Основная причина такого разделения состоит в том, что представление целых и вещественных чисел в современных компьютерах различное, а также различается и набор операций. Например, операции сложения целых и вещественных чисел представляются разными машинными командами.

Рис. 5.1. Классификация простых типов данных

Сама номенклатура машинных операций над целыми и вещественными числами различается (например, для вещественных чисел отсутствуют побитовые операции). Реализация операций над вещественными числами сложнее с аппаратной точки зрения, поэтому в ряде архитектур вещественные числа вообще могут отсутствовать. Для того чтобы программист мог учитывать эти нюансы, многие языки (в том числе Паскаль, C++, Java, C#) разделяют числовые типы на два вида.

Целые типы данных

Для целых типов данных необходимо рассмотреть следующие основные вопросы:

- универсальность (насколько полно учтены машинные типы);
- наличие (или отсутствие) беззнаковых типов;
- представление (размер значения, диапазоны значений);
- надежность (какие ошибки могут возникать при выполнении операций с целыми значениями);
- набор операций.

Универсальность. Язык Паскаль предоставляет единственный целочисленный тип данных — integer. Этого вполне достаточно для учебного языка программирования, но неприемлемо в индустриальном программировании. Языки C++, Java и C# представляют универсальную номенклатуру целых типов, которая соответствует большинству современных архитектур. В этих языках определены однобайтовые целые числа (char — в C++, byte — в Java и C#), короткие целые (short), основные целые (int), длинные целые (long).

Беззнаковые типы. Практически все компьютерные архитектуры в дополнение к знаковым целым числам поддерживают и беззнаковые типы, т.е. целочисленные типы, содержащие только неотрицательные значения. Это обусловлено необходимостью выполнения операций над адресами в машинных программах.

Адреса представляются беззнаковым целым типом (вспомним, что адрес — это номер ячейки памяти, начинающийся с нуля). Операции над адресами называются *адресной арифметикой.*

Также причиной использования беззнаковых чисел является то, что при одинаковом размере максимальное значение беззнакового типа больше, чем максимум знакового (ведь не требуется хранить информацию о знаке). В случае если диапазон целых значений невелик, использование беззнакового типа иногда необходимо.

Язык Java не содержит беззнаковых типов, что упрощает реализацию JVM и позволяет избежать ряда проблем, связанных с надежностью программ (см. далее). Языки C++ и C# для каждого размера целого типа содержат знаковый и беззнаковый варианты. Прежде всего, это диктуется требованием универсальности.

Представление. Языки Паскаль и C++ не фиксируют представление целого типа. Размер и диапазон значений определяются реализацией. Это связано с тем, что эти языки были реализованы для большого числа машинных архитектур, существенно различавшихся по представлению чисел. Фиксация представления дала бы необоснованное преимущество конкретной архитектуре, поскольку реализации в других архитектурах были бы более сложными и менее эффективными. Язык C++ даже не фиксирует представление однобайтового типа `char`. В зависимости от реализации он может быть как знаковым (`signed char`), так и беззнаковым (`unsigned char`). Про размеры типов в языке C++ известно следующее:

```
sizeof(char)=1
sizeof(char)<=sizeof(short)<=sizeof(int)<=sizeof(long)
```

Здесь `sizeof` — это статическая операция языка C++, которая применима как к именам типов, так и к объектам данных, и возвращает размер типа (объекта данных) в байтах.

Языки Java и C# полностью регламентируют размер и диапазон значений всех типов данных. Это связано с тем, что архитектуры, для которых разрабатывались языки, вполне определены (для Java это JVM, а для C# — платформа .NET на архитектуре IA-32). Представление типов описывается в табл. 5.1 и 5.2.

Таблица 5.1. **Целые типы в языке C#**

Размер	Знаковый	Диапазон	Беззнаковый	Диапазон
1	`sbyte`	–128...127	`byte`	0...255
2	`short`	–32 768...32 767	`ushort`	0...65 535
4	`int`	$-2^{31}...2^{31}-1$	`uint`	$0...2^{32}-1$
8	`long`	$-2^{63}...2^{63}-1$	`ulong`	$0...2^{64}-1$

Таблица 5.2. **Целые типы в языке Java**

Размер	Имя	Диапазон
1	`byte`	–128...127
2	`short`	–32 768...32 767
4	`int`	$-2^{31}...2^{31}-1$
8	`long`	$-2^{63}...2^{63}-1$

Надежность. Надежность языковых конструкций определяется тем, насколько эти конструкции предохраняют программиста от случайных ошибок. Наличие беззнаковых типов снижает надежность работы с целыми числами.

Рассмотрим следующий фрагмент программы на C#:

```
int cnt = 0;
for (uint i = 256; i >= 0; i--) cnt++;
```

Этот цикл никогда не закончится, так как переменная цикла `i` всегда неотрицательная. Проблема в том, что в большинстве машинных архитектур целочисленные операции сложения и вычитания не генерируют ошибку при выходе результата за пределы допустимого диапазона. Решение о корректности значения операции возлагается на программиста. Поэтому в приведенном фрагменте программы значение переменной `i` сначала достигает нуля, а затем вычитание 1 дает в результате (на архитектуре x86) сразу значение $2^{32}-1$ (максимальное значение типа `uint`). Результат неверный, но ошибки не возникает, и программа «зацикливается». Аналогичный результат получается и при переписывании фрагмента на язык C++ (в этом случае `uint` следует заменить на `unsigned`). Правда, язык C# позволяет проконтролировать корректность результата целочисленных операций, для чего служит конструкция `checked`:

```
checked
{
    for (uint i = 256; i >= 0; i--) cnt++;
}
```

В блоке, следующем за ключевым словом `checked`, все целочисленные операции проверяются на корректность, поэтому при возникновении переполнения (при попытке вычесть 1 из нулевого беззнакового значения) генерируется ошибка. Разумеется, время выполнения операций растет, поэтому по умолчанию операции не контролируются.

Заметим, что в языке Java сделать подобную ошибку просто невозможно.

Еще один источник возможных ошибок при работе с целочисленными типами — преобразования значений одного типа в значения другого типа. Общий синтаксис преобразований типов в языках C++, Java, C#:

```
(X) выражение
```

Здесь `X` — это имя типа. Пусть `выражение` вычисляет значение типа `Y`. Тогда можно сказать, что приведенная конструкция осуществляет *преобразование из типа `Y` в `X`* (или *приведение типа `Y` к `X`*). Преобразования арифметических типов бывают двух видов: расширяющие и сужающие. Если множество значений типа `Y` есть подмножество значений типа `X`, то преобразование является *расширяющим*, а в противном случае — *сужающим*.

Очевидно, что расширяющие преобразования безопасны, так как преобразуемое значение сохраняется. Сужающие преобразования потенциально опасны. Рассмотрим фрагмент программы на C#:

```
short i = -1;
ushort ui = (ushort)i;
int k = 256;
sbyte sb = (sbyte)k;
```

Здесь два сужающих преобразования: из `short` в `ushort` (знаковый и беззнаковый типы одного размера) и из `int` в `sbyte` (знаковые типы разных размеров). В обоих случаях результат может обескуражить: из –1 получится 65 535, а из 256 получится 0. Если переписать данный фрагмент на языке C++, то получим такие же результаты на компьютерах архитектуры x86. В других архитектурах результат может быть другим, но в любом случае преобразуемое значение будет потеряно.

Конечно, не все сужающие преобразования ведут к ошибке: если преобразуемое значение типа `Y` одновременно входит и в тип `X`, то оно не изменится при приведении (например, нуль принадлежит всем числовым типам в наших языках, поэтому его безопасно приводить к любому числовому типу).

В языке C++ корректность сужающих преобразований не контролируется (программист берет на себя всю ответственность), а в языке C# контролируется только внутри `checked`-блока.

Еще опасней ситуация при неявных преобразованиях. Преобразование типа *неявное*, если оно вставляется или выполняется транслятором. В языках C# и Java неявными могут быть только расширяющие преобразования, а в языке C++ абсолютно все преобразования между любыми двумя арифметическими типами (не только целыми) могут быть неявными. Говоря другими словами, в языке C++ значение одного арифметического типа можно присваивать переменной любого другого арифметического типа. Компилятор просто вставит неявное преобразование. При этом никакого контроля корректности

преобразования не проводится. Такая ситуация может приводить к ошибкам, которые трудно обнаружить.

Набор операций. Языки C++, Java, C# поддерживают практически единый набор операций над целыми значениями:

- арифметические (сложение «+», вычитание «–», умножение «*», деление нацело «/», остаток от деления «%»);
- побитовые (побитовое *или* «|», побитовое *и* «&», побитовая *инверсия* «~», побитовое *исключающее или* «^»);
- операции сдвига (влево «<<», вправо «>>»).

Все операции возвращают тот же тип, что и тип операндов (операнда).

Интересно, что отсутствие беззнаковых типов в языке Java обеспечивает появление дополнительной операции: логического сдвига вправо >>>. Дело в том, что машинные операции сдвига вправо имеют разный эффект для знаковых и беззнаковых типов (подробнее сдвиги рассматриваются, например в [29]). Для беззнаковых типов используется логический сдвиг (сдвигаемые биты замещаются нулями), а для знаковых типов — арифметический сдвиг (знак сохраняется, и сдвиг эквивалентен делению на степень числа 2). При трансляции операции >> компилятор применяет операцию логического сдвига, если операция применяется к беззнаковому типу, и операцию арифметического сдвига для знаковых типов. Однако в языке Java есть только знаковые типы (поэтому операция >> всегда представляет собой арифметический сдвиг), а операция логического сдвига (игнорирующая знак) иногда необходима. Вот в этих случаях и применяется операция >>>.

К целым типам применимы операции сравнения (равенство «==», неравенство «!=», меньше «<», больше «>», меньше или равно «<=», больше или равно «>=»), которые выдают значения логического типа (см. далее). Обратим внимание, что равенство задается символами «==», так как символ «=» занят операцией присваивания (эти операции могут смешиваться в выражениях, поэтому необходимо различать их лексически).

Кроме перечисленных операций существуют операции с побочным эффектом (изменяющие операнд и вычисляющие значение):

- префиксный инкремент ++x (увеличение на единицу, возвращается новое значение x);
- постфиксный инкремент x++ (увеличение на единицу, возвращается старое значение x);
- префиксный декремент --x (уменьшение на единицу, возвращается новое значение x);
- постфиксный декремент x-- (уменьшение на единицу, возвращается старое значение x);

Присваивание также является операцией:

```
v = e
```

Эта операция вычисляет значение выражения e и присваивает его правой части v. Если типы v и e различные, то транслятор либо вставляет неявное преобразование к типу правой части v, либо выдает сообщение об ошибке (если язык запрещает такое преобразование). Присвоенное значение будет значением операции присваивания. Ассоциативность операции присваивания справа налево, в отличие от операций, приведенных ранее.

Для двуместных операций (таких, как сложение, сдвиги и тому подобных) определена комбинированная операция присваивания, например:

```
v += e
```

Как и обычное присваивание, эта операция вычисляет значение e (возможно с приведением типа), а затем увеличивает значение v на вычисленное значение. Новое значение v будет значением комбинированной операции.

Возникает следующий вопрос: есть ли в этих языках оператор присваивания? Непосредственно такого понятия нет (так как присваивание — это операция, а не оператор). Однако в этих языках есть понятие «оператор-выражение». Оператор-выражение — это любое корректное выражение, после которого поставлена точка с запятой:

```
expr;
```

Поэтому в роли оператора присваивания выступает, например следующая конструкция:

```
i = x + y;
```

Конечно, присваивание может случиться и в других контекстах.

Вещественные типы данных

Вещественные типы данных служат для представления числовых дробных значений.

Большинство языков использует для хранения таких чисел формат с плавающей точкой и основанием 2. Значение числа представляется в виде

$$(-1)^S * M * 2^P$$

где S — бит знака (0 соответствует «+», а 1 — «–»); M — нормализованная мантисса ($1 <= M < 2$); p — порядок.

За счет нормализованности мантиссы число всегда представляется единственным образом (заметим, что нормализованность мантиссы означает, что старший ее бит всегда 1, поэтому эту единицу хранить не надо, т.е. она подразумевается).

Языки C++, C# и Java содержат два плавающих типа: `float` и `double`.

Как и для целых типов, язык C++ не фиксирует конкретный формат представления чисел, можно только сказать, что

```
sizeof(float) <= sizeof(double)
```

Языки C# и Java поддерживают один и тот же формат представления чисел, который определяется международным стандартом IEEE-754 [36]. Этот стандарт определяет базовые форматы представления плавающих чисел и правила выполнения арифметических операций над ними. Несмотря на добровольный характер этого стандарта практически все современные архитектуры (в том числе архитектуры x86 и JVM) поддерживают его требования. Краткое описание стандарта можно найти в приложении к [29]. В языке Java (и спецификации JVM) форматы плавающих типов и операции с плавающими типами полностью соответствуют стандарту.

Стандарт IEEE-754 определяет два базовых формата чисел с плавающей точкой: с одинарной (32 бит) и двойной (64 бит) точностью. Формату одинарной точности соответствует тип `float`, а формату двойной точности — `double`.

В формате одинарной точности один бит занимает знак, 23 бит — мантисса, 8 бит — порядок. В формате двойной точности мантисса занимает 52 бит, порядок — 11 бит.

Два значения порядка (максимальное и минимальное) используются для представления специальных значений.

Легко вычислить минимальное (по абсолютной величине) и максимальное значения типа `float` (учитывая, что старший бит мантиссы всегда 1, и он не хранится):

MAX_FLOAT=$(1-2^{-24})*2^{127}$; MIN_FLOAT=$1/2*2\ 2^{-126}$

Аналогичные значения можно вычислить для двойной точности.

Самое важное, что следует помнить про плавающие типы, это то, что арифметические вычисления над ними неточные, и ошибка вычислений является относительной. Так, для типа `float` точность сложения составляет 2^{-23}, если операнды находятся между 1 и 2, однако для операндов порядка 2^{24} точность уже порядка 1.

Следовательно, задачи с большим объемом вычислений с плавающей точкой требуют применения специальных алгоритмов, устойчивых к погрешностям вычислений и входных данных.

Однако некоторые приложения требуют вычислений с фиксированной точностью, которая не зависит от абсолютной величины операндов (типичный пример — финансовые вычисления). Эти вычисления требуют представления дробных чисел с фиксированным числом знаков в дробной части.

Типы данных с таким представлением называются *фиксированными*.

Языки C++ и Java не содержат в базисе таких типов. Их приходится моделировать с помощью механизма классов.

Язык C# содержит специальный тип `decimal`, который служит для фиксированного представления чисел с дробной частью (также он может представлять целочисленные значения, не представимые целыми типами). Размер значений типа `decimal` 128 бит. В них хранятся бит знака `S`, коэффициент `c` ($0 < c < 2^{96}$) и масштаб `e` (`0 <= e <= 28`). Значение определяется формулой

$$(-1)^S * c * 10^{-e}$$

Таким образом, минимальное (по абсолютной величине) значение составляет 10^{-29}, а максимальное — приблизительно $7.9 * 10^{28}$. Точность представления 28-29 десятичных цифр.

Такой тип подходит и для финансовых расчетов, и для ряда других приложений, требующих абсолютной погрешности в вычислениях.

Операции над вещественными числами включают в себя базовый арифметический набор (сложение, вычитание, умножение и деление), а также операции сравнения (при этом надо помнить, что в силу погрешностей вычислений операция сравнения на равенство может давать неожиданные результаты).

Символьные типы данных

Символьные типы данных служат для следующих представлений:

- символов (литер) алфавитов естественных языков;
- символов, управляющих работой устройств ввода-вывода (например, символов перехода на новую строку, табуляции и т. п.);
- специальных символов (валютного знака).

Каждый символ имеет свое название (например, «возврат каретки», «маленькая кириллическая буква я», «знак копирайта»). Символьные типы основаны на понятии множества символов (character set).

Множество символов определяет, во-первых, набор символов, а во-вторых, кодировку этого набора — отображение набора на диапазон целых чисел. Значения символьного типа — это и есть значения из этого диапазона.

Таким образом, символы могут представляться целым типом данных, что и было принято в ряде языков программирования. Например, в языке C целый тип `char` использовался для представления символов. «По наследству» такое представление перешло и в язык C++. Однако отсутствие специального типа для представления символов снижает надежность программ, обрабатывающих текстовую информацию (а удельный вес таких программ в индустрии программирования постоянно возрастает).

В самом деле, если применение арифметических операций наподобие вычитания кода одного символа из кода другого символа или сложения кода символа с константой вполне допустимо, то перемножение, деление кодов и тому подобные операции, вполне допустимые для целых типов, бессмысленны (а значит, ошибочны) для символьных типов. Поэтому современные языки программирования определяют отдельный символьный тип, несовместимый с целыми типами (хотя значения этого типа, конечно, представляют собой целые числа).

С символьными типами связана проблема представления произвольных множеств символов. Ранние языки программирования (как и виртуальные архитектуры, на которых они основывались) использовали однобайтовые множества символов. В этих множествах число различных символов не превосходило двух с небольшим сотен (а при использовании только английского алфавита хватало и 127 различных символов), поэтому для представления символа хватало одного байта (вспомним, что размер типа `char` в языке C один байт).

Однобайтовые множества символов породили ряд проблем.

Первая проблема состояла в невозможности представления объемных иероглифических алфавитов (китайского, японского и др.).

Вторая проблема заключалась в том, что однобайтовое множество по отдельности покрывает небольшое множество алфавитов. Исторически сложилось так, что почти каждое множество символов содержит общий для всех набор, включающий в себя управляющие символы, символы английского алфавита, цифры, знаки препинания и т. п. Коды символов из этого набора соответствуют американскому стандарту ASCII-7 и принимают значения в диапазоне от 0 до 127. Остальные 128 символов в однобайтовой кодировке заполняются символами других алфавитов. Очевидно, что даже европейские алфавиты предоставляют существенно больший набор символов. Поэтому пришлось разрабатывать множества символов отдельно для западноевропейских языков, кириллицы, турецкого, арабского, греческого и других языков. Можно представить, какие проблемы возникали при создании программ, одновременно обрабатывающих тексты на разных языках (например, на русском, французском и турецком).

Третьей проблемой стало обилие кодировок даже для одного алфавита (для русского набора символов в настоящее время широко используются по меньшей мере четыре однобайтовые кодировки).

Ситуация изменилась после появления в 1991 г. универсального множества символов UNICODE, содержащего символы почти всех реально используемых алфавитов. Символов достаточно много (тысячи), поэтому каждый из них кодируется двухбайтовым числом без знака (промежуток от 0 до 65 535). Для каждого алфавита выделен свой диапазон символов, например, кириллица находится в промежутке от 1040 (большая буква А) до 1298 (буква Ҋ — большая буква Л с крючком).

Почти все языки, появившиеся после 1991 г., используют UNICODE.

Тип `char` в языках Java и C# тоже использует это множество символов. Множество операций над типом включает в себя операции сравнения (в соответствии с кодировкой UNICODE), присваивания и операцию «+» сцепления со строкой. Например, строка `s` получит значение `"line 1"` в следующем фрагменте:

```
char c = '1';
string s = "line " + c;
```

Значения типа `char` могут неявно приводиться к целым типам данных, если это безопасно (например, могут приводиться к `int`, и не могут приводиться к `byte`), поэтому в следующем фрагменте операция «+» обозначает операцию сложения целых чисел:

```
char c = 'x';
int k = x + 1; // то же самое, что и (int)x + 1
```

Обратная операция преобразования (приведение целых значений к символьному типу) допустима только явно:

```
sbyte b = -1;
char c = (char)b;
```

При этом корректность преобразования в языке C# контролируется при включении операции в checked-блок.

В языке C++, как уже говорилось, символьные типы рассматриваются как целочисленные. Кроме типа `char` в языке C++ появился тип `wchar_t` специально для представления символов из UNICODE. Этот тип является целым беззнаковым двухбайтовым типом.

Логические типы данных

Логический тип данных, обозначаемый в языках C++ и C# Java ключевым словом `bool` (`boolean` в Java), состоит их двух значений: `true` (истина) и `false` (ложь).

Набор операций состоит из логического И — «`&&`», логического Или — «`||`» и отрицания Не — «`!`» с обычным математическим смыслом.

Интересной особенностью вычисления логических операций `&&` и `||` является их «ленивость». *Ленивость* исполнения логических операций состоит в том, что они вычисляются слева направо, и если значение левого операнда определяет значение всего выражения (для операции && это `false`, а для операции `||` это `true`), то правые операнды вычисляться не должны.

В языках C# и Java логический тип не может приводиться к целым (да и другим) типам данных (ни явно, ни тем более неявно).

В языке C++ ситуация более сложная. Дело в том, что логический тип был включен в язык не сразу, а сначала C++ унаследовал правила языка C. В этом языке логические операции были (перечисленные ранее), а отдельного логического типа не было. Логические операции возвращали целое значение 1 в качестве «истины» и 0 — в качестве «лжи». Если в каком-то месте от выражения произвольного типа требовалось логическое значение, то это выражение (при необходимости) приводилось к целому типу и вычислялось. При этом ненулевое значение соответствовало «истине», а нулевое — «лжи». Поэтому на языке C (и C++) вполне можно записать следующие фрагменты:

```
while (n-m)
{
   if (n > m)
    n -= m;
   else
    m -= n;
}
```

В языках C# и Java в этом месте компилятор выдаст ошибку, а правильный фрагмент программы имеет вид

```
while (n-m != 0) …
```

То же, конечно, можно записать и на языке C/C++.

Таким образом, тип `bool` в языке C++ неявно приводится к целому типу (`true` — в 1, `false` — в 0), и наоборот, что, конечно, снижает его ценность (по сравнению с логическим типом в C# и Java).

Порядковые типы данных

Порядковые типы данных подразделяются на перечислимые типы и типы диапазона.

Перечислимый тип задается прямым перечислением константзначений. Каждое значение задается своим именем.

Впервые перечислимые типы появились в языке Паскаль и сразу стали популярными как среди разработчиков языков программирования, так и среди программистов. Дело в том, что использование перечислимых типов делает программу более понятной, поскольку все значения имеют имена, которые характеризуют смысл значения.

С точки зрения реализации каждое значение перечислимого типа отображается в целое число, соответствующее его порядковому номеру в определении типа. Этот номер называется *ординалом* значения. По умолчанию нумерация ординалов начинается с нуля, однако некоторые языки позволяют придать свои значения ординалам.

Перечислимые типы реализованы в языках C++, C# и Java (начиная с 2005 г.). Однако в реализации этих типов есть различия.

Начнем с языка C++. Представим себе задачу моделирования светофора. В ней важную роль имеет цвет светофора, который удобно представить перечислимым типом:

```
enum TrafficColors {
    RED, YELLOW, GREEN
};
```

Пусть у нас есть класс `TrafficLights`, представляющий собой светофор, и переменная `light1` этого класса. Тогда следующий вызов не нуждается в дополнительных комментариях:

```
light1.SetSignal(RED);
```

Очевидно, что это переключение светофора в красный цвет.

Цвета нумеруются у нас от 0 до 2 (`RED=0`, `YELLOW=1`, `GREEN=2`). Иногда конкретный номер цвета не важен, но мы можем расширить задачу, потребовав, чтобы числовые значения констант перечислимого типа соответствовали значениям цветов в цветовой системе RGB-24 (в этой системе каждому цвету соответствует целое значение, составленное из трех байтов — значение каждого байта есть интенсивность красного, зеленого и синего цветов в диапазоне от 0 до 255).

Тогда объявление нашего типа изменится (используется шестнадцатеричная форма целых констант — каждая пара цифр соответствует байту со значением 255 или 0):

```
enum TrafficColors {
    RED = 0xFF0000,
    YELLOW = 0xFFFF00,
    GREEN = 0x00FF00
};
```

Теперь значения цветов можно использовать в функциях рисования, требующих указания значения цвета, например:

```
brush.SetBrushColor(YELLOW);
```

Здесь мы предполагаем, что функция `SetBrushColor` имеет один параметр целого типа, означающий цвет в системе RGB-24.

Таким образом, значения перечислимого типа неявно преобразуются в значения типа `int`. Обратное, однако, неверно. Попытка присвоить переменной перечислимого типа какое-либо целое значение рассматривается компилятором как ошибка:

```
TrafficColors cl = 127; // ошибка!!!
```

Если мы хотим придать переменной новое числовое значение, то должны записать явное преобразование:

```
TrafficColors cl = (TrafficColors)-1;
          // теперь формально правильно
```

Однако ясно, что такое преобразование не всегда приводит к ожидаемым результатам (какой реальный цвет будет у переменной `cl?`), поэтому и требуется выписывать явное преобразование.

Однако у реализации перечислимых типов в языке C++ есть ряд недостатков. Первый недостаток состоит в том, что константы перечислимого типа имеют ту же область действия, что и имя перечислимого типа (так называемый *неявный импорт имен*). Это может приводить к конфликту имен. Например, если мы одновременно пытаемся моделировать семафор, то появляется еще одно перечисление:

```
enum SemaColors {
     RED, GREEN
};
```

Это приводит к конфликту. Для учебных программ это не страшно (имена в своих программах поменять легко). Однако в условиях индустриального программирования, когда широко используются библиотеки сторонних производителей (например, одна библиотека моделирует светофор, другая — семафор), такие конфликты доставляют немало хлопот программистам.

Вторым недостатком является фиксированное представление перечислимых типов в языке C++, поскольку они всегда реализуются как целые (на базе основного целого типа `int`). Иногда это неудобно и приводит к напрасной трате памяти (ведь хватило бы и более короткого типа).

В языке C# эти недостатки преодолены. Заметим, что приведенные ранее объявления типов `TrafficColors` и `SemaColors` являются вполне корректными в языке C# (надо только убрать концевые точки с запятой). Однако имена перечислимых констант локализованы в типе и могут использоваться за пределами объявления типа только в виде

```
имя_типа.имя_константы
```

Например:

```
light1.SetSignal(TrafficColors.RED);
```

Кроме того, значения перечислимого типа могут храниться как значения любого целочисленного типа (такой тип называется *базовым* для перечисления). По умолчанию базовым является основной целый тип `int`, но базовый тип легко поменять:

```
enum SemaColors : byte
{
     Red, Green
}
```

В отличие от языка C++ значения перечислимых типов в C# должны преобразовываться в базовый тип только явно:

```
brush.SetBrushColor(TrafficColors.YELLOW);
// в C# - ошибка - SetBrushColor требует целый тип
brush.SetBrushColor((int)TrafficColors.YELLOW);
// теперь правильно!
```

В языке Java перечислимые типы появились относительно недавно (с 2005 г.) и реализованы они наиболее «экзотично»: на самом деле перечислимые типы в Java — это классы. Не вдаваясь в тонкости (о них можно прочитать, например, в [35]), заметим, что объявление и использование перечислений здесь внешне похожи на C#:

```
enum TrafficColors
{
     Red, Yellow, Blue
}
TrafficColors c1 = TrafficColors.Yellow;
```

Неявные преобразования в целый тип и обратно также запрещены, но можно использовать методы перечислимого класса `ordinal()`, возвращающий порядковый номер константы, и `values()`, возвращающий массив констант значений перечислимого типа (индекс в массиве — это номер константы в типе):

```
int colorNumber = TrafficColors Red.ordinal();
    // colorNumber получает значение 0
TrafficColors c =
TrafficColors.values[colorNumber+1];
// c получает значение TrafficColors. Yellow
```

Напоследок заметим, что во всех рассмотренных языках к перечислимым типам применимы операции сравнения (сравниваются ординалы значений).

Скажем несколько слов о диапазонных типах. Они появились в языке Паскаль и позволяли ограничить значения какого-либо целого или перечислимого типа.

Например:

```
type Index = 0..N; // диапазон целого типа
type DaysOfWeek = (Mon, Tue, Wed, Thu, Fri, Sat,
Sun);
// перечислимый тип - дни недели
type WorkDays = Mon..Fri;
// диапазон типа DaysOfWeek
WeekendDays = Sat..Sun;
// другой диапазон DaysOfWeek
```

Чаще всего диапазонные типы используются в качестве типа индекса в массивах. Иногда это удобно, так как позволяет использовать произвольные границы индексов (например, с 1 или с отрицательного числа). В современных языках программирования индексы в массиве

начинаются только с 0 и могут быть только целыми, поэтому область применения типов диапазонов существенно сужается.

В языках C++, Java и C# диапазонных типов нет.

Указательные типы данных

Указатели являются абстракцией понятия машинного адреса. Средства работы напрямую с адресами, с одной стороны, удобны, так как обеспечивают возможности, сравнимые с возможностями программирования на машинном языке (т. е. использование всех возможностей машинной архитектуры без ограничений), а с другой стороны, — опасны (если нет ограничений, то можно сделать все, что угодно, в том числе и ошибки).

Опасности применения указателей перевешивают их удобства, поэтому языки C# и Java не используют понятие указателя (точнее, в C# понятие указателя есть, но оно используется крайне ограниченно и только для работы с библиотеками, написанными на C или других императивных языках; в нашем курсе эти средства не рассматриваются). Вместо указателей в этих языках используется более абстрактный и безопасный тип данных — ссылки.

Указатели есть в языке C++ (где они без изменений заимствованы из языка C). Синтаксис объявления указательной переменной в C и C++ следующий:

```
имя_типа_данных * имя_указателя
```

Например:

```
int * pi; X * pX; char * message = "Hello!";
```

Зачем вообще нужны указатели (другими словами, работа с адресами) в языках программирования? Можно назвать следующие основные причины использования указателей:

- передача адресов объектов данных в подпрограммы;
- работа с объектами из динамической памяти;
- использование адресной арифметики;
- использование адресов подпрограмм как объектов данных (передача параметров — подпрограмм в другие подпрограммы и т. п.).

Рассмотрим подробнее каждую причину.

Передача адресов данных в подпрограммы. Если подпрограмма меняет содержимое своего параметра — объекта данных, то единственный способ это сделать — передать адрес объекта. Кроме того, если параметр — большой объект, то копировать его в подпрограмму накладно, поэтому и передают только адрес объекта.

Для получения адреса объекта служит адресная операция & (не путайте ее с побитовой операцией И, которая имеет два аргумента):

```
&переменная
```

Если переменная имеет тип T, то адресная операция возвращает тип указателя на T (т.е. T*).

Например, функция форматного ввода scanf из стандартной библиотеки языка С вводит (т.е. изменяет) значение целой переменной k, поэтому необходимо передать туда адрес k:

```
scanf ("%d", &k);
```

Первый параметр (формат) "%d" сообщает функции, что она должна ввести целое число, представленное в десятичной системе счисления. Если бы параметром был "%x", то число должно было бы быть представлено в шестнадцатиричной системе.

Как получить доступ к объекту, адрес которого находится в указателе? Для этого служит операция * (операция разыменования):

```
int j = -1;
int * pi = &j; // в pi — адрес j
*pi = 0;//*pi — это j, поэтому j получает значение 0
```

Очевидно, что для любой переменной var верно: var эквивалентно *(&var).

В языке C++ можно не передавать указатели в подпрограммы, а использовать ссылки (как и рекомендуется делать), однако в языке С такой возможности нет.

Работа с объектами из динамической памяти. Это самая важная (и нередко единственная) причина использования указателей.

В языке C++ объект типа T размещается в динамической памяти с помощью операции new имя_типа:

```
T * p = new T;
```

Отведенная динамическая память обязана быть освобождена с помощью операции delete указатель:

```
delete p;
```

Указатель должен содержать адрес, полученный от операции new. Иначе говоря, это адрес объекта, размещенного в динамической памяти. Объект после выполнения delete перестает существовать, и все ссылки на него становятся ошибочными.

Если в памяти необходимо разместить массив (непрерывную последовательность) объектов типа T, то следует использовать другие формы операций new и delete:

```
T * pArr = new T[256]; ... delete [] pArr;
```

Операция new[] возвращает адрес первого элемента размещенного массива. Обратите внимание, что вид объявления указателя не зависит от того, указывает он на единственный объект или на последовательность объектов (массив). Такая неразличимость ука-

зателя на объект и указателя на массив (точнее, на первый элемент массива) — характерная особенность языков C и C++.

Еще одной особенностью языков C и C++ является неразличимость адресов, полученных от адресной операции и от операции `new`.

Размещение объектов данных в динамической памяти — ценная возможность современных языков. Некоторые языки, как уже отмечалось, размещают объекты классов и массивы только в динамической памяти, правда, используя для этого аппарат ссылок, более надежный и абстрактный, чем указатель. Поэтому операция `new` присутствует во всех рассматриваемых здесь языках.

Однако наличие (и требование!) операции явного освобождения памяти (`delete`) становится причиной частых и труднообнаруживаемых ошибок в программах. Две ошибки работы с динамическими объектами, которые могут возникнуть при выполнении (или невыполнении) операции `delete` — это висячие ссылки и мусор.

Висячие ссылки — это адреса уже уничтоженных объектов. Попытка обратиться к объектам по этим адресам ошибочна (по этому адресу, например, могут располагаться совершенно другие объекты, размещенные уже после «смерти» уничтоженного объекта), однако проконтролировать и обнаружить эту ошибку очень трудно.

Пример возникновения висячей ссылки:

```
int * pX = new int; // объект размещен в памяти
int * pY = pX;
// теперь pX и pY ссылаются на один и тот же объект
delete pX; pX = 0;// объект уничтожен
*pY = -1; // ошибка!!! Объект * pY не существует
```

Здесь указатель `pY` становится висячей ссылкой после выполнения `delete pX`. Предсказать, как, когда и где проявится эта ошибка, практически невозможно. Если в примере обнулить не только `pX`, но и `pY`, то висячая ссылка исчезнет, и ошибка немедленно обнаружится при попытке обратиться к нулевому адресу памяти (виртуальные компьютеры современных ОС генерируют в этом случае ошибку защиты памяти).

Конечно, можно легко избежать висячих ссылок, если вообще не уничтожать объекты (не использовать `delete`). Однако это «лекарство» еще хуже «болезни», поскольку приводит к другой ошибке при работе с динамическими объектами — возникновению мусора.

Мусор — это неуничтоженные объекты из динамической памяти, на которые отсутствуют ссылки. Раз нет ссылок на них, то объекты не могут использоваться (они бесполезны), но и не могут быть уничтожены (т.е. зря занимают память) — типичный мусор! Если мусор заполняет всю доступную память, то размещение новых объектов становится невозможным, и программа аварийно завершается. Проблема мусора в том, что эта ошибка проявляется при длительной ра-

боте с программой, иногда уже на стадии эксплуатации. Обнаружить ее еще труднее, чем *висячую ссылку*.

Пример возникновения мусора:

```
int * pX = new int; // разместили первый объект
int * pY = new int; // разместили второй объект
pY = pX; // ошибка!!! Первый объект стал мусорным
```

Как избавиться от мусора и висячих ссылок? Во-первых, не использовать операцию явного освобождения памяти (избавимся от висячих ссылок), а во-вторых, добавить в виртуальную машину языка (точнее, в подсистему времени выполнения) компоненту *автоматический сборщик мусора*. Диспетчер динамической памяти должен отслеживать (запоминать) все ссылки на объекты из динамической памяти. Если все ссылки отслеживаются, то автоматический сборщик мусора может найти неиспользуемые объекты и пометить соответствующие участки памяти как свободные. Как правило, сборщик мусора дополнительно собирает все используемые объекты в непрерывную область памяти (ссылки на них, конечно, изменяются), и этот процесс называется *дефрагментацией памяти*. Когда вызывать автоматический сборщик мусора? Во-первых, при выполнении операции `new`, когда свободной памяти требуемого размера нет. Во-вторых, современные системы могут выполнять автоматическую сборку мусора параллельно с выполнением программы.

Конечно, автоматическая сборка мусора приводит к повышенным накладным расходам, и эффективность программ снижается. Однако повышение надежности программы компенсирует этот недостаток, поэтому современные языки, такие как Java и C#, используют автоматическую сборку мусора.

В языке C++ реализовать эту возможность затруднительно в силу того, что указатели на динамические объекты (адрес получен от `new`) и нединамические объекты (адрес получен путем адресной операции &) не отличаются друг от друга, поэтому отследить все ссылки в программе практически невозможно. Эта особенность языка приводит еще к одной причине сбоев в программах: ошибочному вызыву операции `delete` для адреса нединамического объекта.

Интересно, что реализация языка C++ для платформы .NET использует автоматическую сборку мусора (так как любой язык в .NET должен ее использовать), однако она реализована не для указателей C++, а для переменных специального нестандартного типа, эквивалентного ссылкам языка C# [33].

Использование адресной арифметики. Адресная арифметика — это применение операций сложения и вычитания к указателям.

Например, указатель `p` содержит адрес объекта целого типа `(adr)`. Тогда операция `p+1` возвращает адрес, равный `adr+sizeof(int)`. Аналогично, операция `p-1` возвращает адрес, равный `adr--sizeof(int)`. Пусть в общем случае `p` объявлен как указатель

на тип T (T*p;), ie — выражение целого типа, Addr — значение (машинный адрес), хранящееся в p. Тогда p+ie — это значение (машинный адрес) типа T*, которое равно Addr+ie*sizeof(T). Аналогично p-ie — это значение (машинный адрес) типа T*, которое равно Addr-ie*sizeof(T).

Нетрудно увидеть, что разность между двумя указателями p1 и p2 на тип T есть целое значение, равное разности значений машинных адресов из p1 и p2, деленной на sizeof(T).

Использование адресной арифметики позволяет очень эффективно работать с последовательностями объектов данных одного типа (т.е. с массивами).

Однако непосредственная работа с адресами не может быть проконтролирована ни статически (в момент трансляции), ни динамически (во время выполнения программы). В самом деле, при вычислении адреса никак нельзя гарантировать, что полученный адрес корректен, т.е. указывает на правильный объект требуемого типа. Поэтому программы, использующие адресную арифметику, могут содержать ошибки, которые трудно обнаружить.

Использование адресов подпрограмм. С точки зрения машинного языка имя подпрограммы соответствует машинному адресу первой команды тела подпрограммы. Поэтому в C и C++ имена подпрограмм рассматриваются как указатели особого функционального типа. Подробнее этот тип рассматривается в гл. 6.

Ссылочные типы данных

Только что мы выяснили, что понятие указателя слишком близко к машинному языку, и употребление указателей может привести к хитрым ошибкам в программах. Понятие ссылки тоже является абстракцией адреса, но лишено недостатков указателя.

В языке C++ ссылка — это аналог имени (можно рассматривать ссылку на объект данных как альтернативное имя объекта). Синтаксис объявления ссылки на данные типа T следующий (сравните с синтаксисом указателя):

```
T & имя_ссылки
```

К ссылкам языка C++ применима единственная операция — инициализация.

Как именно выглядит инициализация ссылки зависит от контекста, в котором определена ссылка. Если ссылка объявлена как переменная (локальная или глобальная), то инициализация имеет вид

```
T & имя_ссылки = объект_типа_ T
```

Например:

```
int k = -1; // объект типа int
```

```
int& kk = k; // инициализация ссылки kk
kk++; // это — обращение к k. Теперь k содержит 0
int b = 4;
kk = b; //и это — обращение к k. Теперь k содержит 4
```

После инициализации любое употребление ссылки эквивалентно употреблению объекта, на который она ссылается (из инициализатора).

Другим важным контекстом инициализации ссылки является объявление формального параметра функции как ссылки. В этом случае инициализация формального параметра—ссылки объектом, т.е. фактическим параметром, происходит в момент вызова функции (подробнее см. гл. 6). Во время выполнения функции формальный параметр обозначает объект — фактический параметр, поэтому операции над формальным параметром выполняются на самом деле над фактическим.

В языках C# и Java ссылки — это единственный способ обращения к объектам. В этих языках принята *референциальная объектная модель*. Поясним, что она собой представляет.

В C# и Java выделены референциальные типы — это классы, массивы и интерфейсы. Объекты этих типов располагаются только в динамической памяти, являются анонимными и доступными только через ссылки. Объект создается с помощью операции `new` либо создается копия объекта (клонирование).

Приведем пример на языке C#:

```
X obj = new X(); // создание нового объекта
X obj = obj.Clone(); // создание копии объекта
```

На языке Java операция создания копии имеет вид

```
X obj = obj.clone();
```

Остальные типы в языке C# называются типами-значениями. К ним относятся все простые типы данных, а также структуры (структуры языка C# рассмотрим вместе с классами). В Java термина «типы-значения» нет по той причине, что нереференциальные типы — это простые типы данных (за исключением перечислимого типа, который является классом особого вида). Аналога понятия «структура» в Java нет. Мы будем употреблять термин «типы-значения» для любых нереференциальных типов в этих языках.

Объекты типов-значений могут располагаться в любом классе памяти: в статической, квазистатической (т.е. в стеке) и динамической (в составе объектов референциальных типов).

Пусть `X` — класс. Объявление `X a;` не размещает объект, а объявляет ссылку на `X`. Начальное значение такой ссылки — пустое. Пустая ссылка обозначается ключевым словом `null`. Пустая ссылка совместима с любым классом. Рассмотрим следующий фрагмент (применимый как к C#, так и к Java):

```
X a,b; // это не объекты класса, а ссылки
a = new X(); // объект размещен, а ссылается на него
b = a; // b и a ссылаются на один и тот же объект
a = new X(); // еще один объект
// a ссылается на второй объект
// b ссылается на первый объект
```

Ссылки в C# и Java отличаются от ссылок в C++ тем, что ссылки в C++ «навсегда», т.е. в течение всего времени их жизни, полностью ассоциированы с объектом. Даже если ссылка стоит в левой части операции присваивания, все равно речь идет о присваивании объекту, которым инициализирована ссылка. В C# и Java присваивание ссылке означает новую ассоциацию, поэтому нельзя считать, что ссылки в C# и Java — это альтернативное имя объекта (как в C++).

Однако ссылки в C++ и ссылки в C# и Java похожи в том плане, что после установления ассоциации с объектом ссылка идентична самому объекту, и поэтому не требуется никакая операция разыменования (типа операции * с указателем).

5.2. Составные типы данных

Составные типы данных — это типы, значения которых состоят из подобъектов, т.е. имеют внутреннюю структуру. Самый популярный составной тип данных (и первый составной тип, появившийся в языках программирования) — это массив.

Одномерные массивы

Массив — это непрерывная последовательность элементов одного типа.

Основная операция, применимая к любому массиву, это операция индексирования, обозначаемая чаще всего как [ ]. Операция, записанная в виде `A[I]`, имеет два аргумента: объект-массив `A` и значение `I` некоторого дискретного типа (к дискретным типам относятся целые, перечислимые и диапазоны). Значение `I` называют индексом элемента массива. Индексирование возвращает ссылку на элемент массива:

```
[ ] : A, I->T&
```

Отметим, что возвращаемое значение — это именно ссылка, а не просто значение элемента массива, так как результат операции присваивания может стоять в левой части операции присваивания. Основные требования к операции индексирования — однородность

и эффективность. *Однородность* означает независимость времени выполнения операции от значения индекса, а *эффективность* — быстрое выполнение операции.

Перечислим атрибуты массива:

- базовый тип (T);
- тип индекса (I);
- диапазон индекса (L и R — нижняя и верхняя границы) и связанная с ним характеристика — длина массива.

Не вдаваясь в обсуждение вариантов реализации массивов в различных языках программирования, отметим, что в большинстве современных языков концепция массива упростилась: тип индекса всегда целый, нижняя граница диапазона индексов (L) — всегда 0, верхняя граница (R) — N – 1, где N — длина массива.

Следовательно, основной вопрос, касающийся массивов в современных языках, — это время связывания базового типа T и длины массива N с объектом-массивом.

Базовый тип элементов связывается с массивом всегда статически — в объявлении массива. Длина массива N в C и C++ тоже статическая (и тоже задается в объявлении), а в C# и Java — динамическая и задается при размещении объекта в динамической памяти (т.е. при обращении к операции `new`).

Одномерные массивы в языке C++. Объявление массива имеет вид

```
T att[LEN];
```

Здесь `T` — произвольный тип данных, а `LEN` — статическое (т.е. вычисляемое компилятором) выражение. В зависимости от контекста объявления массив размещается либо в статической, либо в динамической памяти. Примеры размещения массива в динамической памяти см. в подразд. 5.1.

Массив в C++ не является «полноценным» объектом данных. Это проявляется, например в том, что один массив нельзя присвоить другому:

```
int a1[256]; int a2[256];
a1 = a2; // ошибка !!!
```

Операция индексирования в языке не контролирует корректность индекса:

```
for (int k = 0; k < 300; k++)
    a1[k] = a2[k];
```

В нашем примере индекс, начиная со значения 256, выходит за верхнюю границу, но ни при компиляции, ни во время выполнения никакой диагностики выдано не будет. Поведение программы при таких ошибках непредсказуемо и зависит от реализации.

Имя массива трактуется в C и C++ как указатель, т.е. адрес первого элемента массива (напомним, что индекс первого элемента массива — 0), а операция индексирования тесно связана с адресной арифметикой, т.е. для любого массива `array` и целого значения `i` верны следующие тождества:

```
&(array[i]) ≡ array+i
```

или в другом виде

```
array[i] ≡ *(array+i)
```

Как уже отмечалось, корректность адресной операции не может быть проконтролирована, поэтому и операция индексирования не контролируется, даже в очевидных случаях типа `array[-1]`.

Отличие имени массива типа `T` от указателя типа `T*` состоит в том, что имя массива трактуется как указатель-константа, поэтому этому имени ничего нельзя присвоить (что вполне резонно, так как иначе доступ к элементам массива может исчезнуть). Более того, любой указатель можно трактовать как имя массива и применять к нему операцию индексирования:

```
int * p; ... p[0] = 0; // то же самое, что *p=0;
```

Поэтому когда операция `new[]` (размещение массива в динамической памяти) возвращает указатель, это вполне согласуется с правилами языка.

Таким образом, понятие указателя перекрывает понятие массива, поэтому программисты на C и C++ нередко предпочитают работать с массивом через указатель (что, как правило, более эффективно).

Приведем пример как можно суммировать элементы массива на C и C++:

```
int arr[N];
int * pCurr = arr;
int * pEnd = arr+N;// pEnd указывает ЗА конец массива
int sum = 0;
while (pCurr < pEnd)sum += *pCurr++;
```

или более короткая запись:

```
for (int *pCurr = arr, *pEnd=pCurr+N, sum =0;
                                  pCurr < pEnd; pCurr++)
        sum += *pCurr;
```

Заметим, что C++ — действительно выразительный язык!

Фактически понятие одномерного массива в C и C++ используется в основном как шаблон для размещения последовательностей в статической или квазистатической памяти. Еще одно отличие

имени массива от указателя состоит в том, что операция `sizeof` для указателя возвращает размер указателя (т. е. машинного адреса), а для имени массива — размер всей последовательности элементов, составляющей массив.

Одномерные массивы в языках C# и Java. Объявление массива в этих языках имеет вид

```
T [] array;
```

Напомним, что `array` — это только ссылка, а собственно массив размещается в виде

```
array = new T[Len];
```

Здесь `Len` — произвольное целочисленное выражение (необязательно статическое).

Можно использовать и явную инициализацию:

```
int [] arr;
arr = new int [] {1, 3, 5, 7, 9};
```

Длина массива является *квазидинамическим* свойством, т. е. она задается динамически в момент создания, но измениться уже не может. Узнать длину массива во время выполнения программы можно с помощью свойства `Length` в C# и свойства `length` в Java. Приведем пример суммирования элементов массива:

```
double [] arr;
double sum = 0;
for (int i = 0; i < arr.Length; i++) sum += arr[i];
```

Операция индексирования контролируется при каждом обращении к ней. Если индекс выйдет за пределы диапазона `0... Length-1`, то генерируется исключительная ситуация (исключения описаны в гл. 9): в Java — `ArrayIndexOutOfBoundsException`, а в C# — `IndexOutOfRangeException`.

В отличие от C и C++ массивы в C# и Java — полноправные объекты, принадлежащее к классам специального вида. Оба языка предоставляют большой набор операций над массивами, включающий в себя копирование элементов, поиск значений, слияние и разбиение и т. п. В Java эти операции сосредоточены в классах `System` и `Arrays`, а в C# — в классе `Array`.

Еще раз отметим, что длина массива не может измениться после его размещения. В случае если требуется массив с переменной длиной, можно использовать класс `ArrayList` из стандартной библиотеки (он даже называется одинаково для обоих языков). Этот класс позволяет добавлять и удалять элементы из массива. В C++ соответствующий класс из стандартной библиотеки STL называется `vector`.

Многомерные массивы

Многомерные массивы в большинстве языков рассматриваются как массивы массивов (единственное исключение, пожалуй, составляет Фортран), например двумерная матрица рассматривается как одномерный массив из одномерных массивов. Рассмотрим объявление матрицы на языках C и C++:

```
float matr[N][M];
```

Это одномерный массив длиной N, каждый элемент которого (т.е. строка матрицы) — одномерный массив длиной M. Такой многомерный массив все равно остается непрерывной последовательностью элементов, вначале располагаются элементы первой строки, за ними — второй и т.д. Если последовательно пробегать по элементам многомерного массива, то индексы меняются справа налево (и быстрее всех самый правый индекс), как в следующем примере суммирования элементов трехмерного массива:

```
float m [N1][N2][N3], sum = 0.0;;
for (int i = 0; i < N1; i++)
     for (int j = 0; j < N2; j++)
          for(int k = 0; k < N3; k++)
               sum += m[i][j][k];
```

Такие массивы называют прямоугольными.

Однако в языках с референциальной моделью возникает следующая проблема. Массивы в таких языках представляются ссылками, поэтому, например, двумерный массив вместо прямоугольного представляет собой массив из ссылок на массивы-столбцы матрицы. Такие массивы уже не обязаны иметь одну и ту же длину и называются ступенчатыми. В C и C++ подобные массивы можно построить, используя массив из указателей.

Приведем пример ступенчатого массива в языке Java (он же является примером на C#):

```
int [][] jagged;
jagged = new int [4][];
// создали массив из ссылок на массивы int
for (int i = 3; i>=0; i--)
     jagged[i] = new int[i+1];
// создали треугольную матрицу
...
int det = 1;
for (int i = 0; i < 5; i++)det *= jagged[i][i];
// вычислили определитель треугольной матрицы
```

В следующем примере массивы имеют вид прямоугольных (длина столбцов одинаковая), но это все равно ступенчатые массивы:

```
double [][]a = new double [5][5];
// создали квадратную матрицу 5x5
int [][]b = new int [][] {{1,2,3},{3,4,5}};
// создали и инициализировали массив 2x3
```

В языке Java все многомерные массивы ступенчатые, создать прямоугольный массив, элементы которого образуют в памяти непрерывную последовательность, в этом языке нельзя. Однако заметим, что доступ к элементам ступенчатого массива менее эффективен, чем к элементам массивов в C и C++ (объясните почему). По этой причине в язык C# добавлен еще один вид многомерных массивов — прямоугольные:

```
int [,] a = new int [5,6];
int sum = 0;
for (int i = 0; i < 5; i++)
     for (int j = 0; j < 6; j++)
          sum += a[i,j];

double [,] x = new double[4,]; // ошибка!!!
int [,] b = {{1,2},{3}}; // ошибка!!!
int [,] c = {{1,2},{3,4},{5,6}};
// создан прямоугольный массив 3x2
```

Необходимость добавления прямоугольных массивов обусловлена двумя главными причинами: эффективностью и возможностью обработки массивов, совместимых с моделью данных языков типа C.

Динамические строки

Динамическая строка (далее будем называть ее просто строкой) — это последовательность символов произвольной длины. Длина строки определяется при размещении в памяти и далее не меняется. На первый взгляд, строка может быть реализована (по крайней мере, в языках C# и Java) как массив из символов, однако это не так. Языки C# и Java содержат специальный встроенный тип (точнее, класс) `String` (в C# можно употреблять вместо этого имени ключевое слово `string`). В C++ встроенного типа строки нет, но зато есть класс `string` из стандартной библиотеки STL.

Необходимость введения специального типа обусловлена несколькими причинами. Во-первых, набор операций для строк существенно шире и специфичней набора операций для обычных массивов. Этот набор включает в себя много вариантов поиска (символа, подстроки, с учетом регистра), сравнения (с учетом разных правил упорядочения) и других подобных вариантов, редко применяемых для массивов.

Кроме того, строки должны интегрироваться со встроенным в язык понятием строковой константы (`"literal constant"`). Сравните инициализацию строки и символьного массива (C#, Java):

```
char [] strArr = {'l','i','n','e'};
String str = "line";
```

Однако главным аргументом в пользу специальной реализации строкового типа является то, что строки реализуются как *неизменяемый объект*. Это означает, что содержимое строки после ее размещения и инициализации рассматривается как константа. Любые операции над строками (даже те, которые манипулируют содержимым строки) не меняют ее содержимое, а вырабатывают новое строковое значение. Операция индексирования (`s[i]`) применима к строкам, но в отличие от массивов она возвращает не ссылку на символ, а значение символа, поэтому для строки `s` нельзя писать `s[0] = ' '`, так как значение не может быть левой частью операции присваивания.

Для обычных массивов модификация отдельных элементов является обычной операцией так же, как и сортировка, меняющая порядок элементов и т. п.

Зато ряд операций над строками в силу их неизменяемости можно реализовать существенно эффективней, чем операции над массивом символов.

Отметим, что строки — это полноценные классы, обладающие богатым набором операций. Самая частая операция над строками — конкатенация (сцепление) строк, обозначаемая как «+»:

```
String s1 = "Hello", s2 = "world!", s;
s = s1 + ", " + s2; // получилось: Hello, world!
```

Правда, следует отметить, что использовать операцию конкатенации для конструирования новых строк следует с осторожностью. Рассмотрим пример на языке C#, который строит строку из входного текста, удаляя из нее все концы строк:

```
string s="", curr;
while ((curr = Console.ReadLine()) != null)
       s += curr;
Console.WriteLine(s);
```

Здесь мы воспользовались тем фактом, что функция `ReadLine()` из класса `Console` не включает символ конца строки в возвращаемое значение, поэтому этого символа нет в выдаваемом тексте. На Java можно написать аналогичную программу, используя класс `System`.

Проблема в том, что мы используем операцию конкатенации для построения постоянно разрастающейся строки. Если входной текст достаточно длинный, то возникает эффект фрагментации памяти: мы требуем все более длинные куски памяти, освобождая куски, которые короче. Конечно, автоматический сборщик мусора исправит ситуацию, но производительность программы существенно упадет. Неслучайно и C#, и Java содержат стандартный класс `StringBuilder`,

который специально спроектирован для конструирования строк возрастающей длины. Он работает с памятью более оптимально, чем реализация операции «+». После завершения конструирования строки достаточно вызвать функцию `ToString` (в Java `toString`) для получения сконструированной строки:

```
string curr;
StringBuilder sb = new StringBuilder();
while ((curr = Console.ReadLine()) != null)
       sb.Append(curr);
Console.WriteLine(s);
```

Записи (структуры)

В традиционных императивных языках программирования, таких как Паскаль или С, записи (в С они называются структурами) наряду с массивами используются очень широко.

Запись — это совокупность объявлений переменных, которые объединены в отдельный объект. Эти переменные называются *полями записи* и доступны с помощью операции «точка». Приведем пример структуры на языке C++:

```
struct Complex
{
    double Re, Im;
}; // объявление структуры с двумя полями
Complex ImagOne = {0.0, -1.0};
    // объявление переменной типа структуры
    // с одновременной инициализацией
Complex c1, c2; // две переменных типа структуры
c1.Re = 1.0; c2.Im = 0.0; // Обращение к полям
c2 = c1; // структуры можно присваивать
```

Понятие класса в объектно-ориентированных языках заведомо шире и универсальней понятия записи, поэтому в современных языках структуры либо упразднены (как в Java), либо являются частным случаем класса (как в C++). В C# есть понятие структуры, но мы рассмотрим его вместе с понятием классов C#.

Другие составные типы данных

Ряд языков программирования содержит другие составные типы данных, отличные от массивов и записей. Например, язык Паскаль включает в себя понятия файла и множества.

Что касается типов (и операторов), абстрагирующих ввод-вывод, то большинство языков программирования не содержат в базисе со-

ответствующие конструкции. Весь ввод-вывод отнесен к стандартным или специализированным библиотекам. Причина этого — разнообразие и эволюция устройств ввода-вывода. Фиксация каких-либо концепций ввода-вывода в конструкциях языка неизбежно приведет к неадекватности этих конструкций в более поздних реализациях. Например, файлы языка Паскаль отлично подходили к самым популярным устройствам хранения данных в 1960-х гг. (это ленточные устройства). Однако концепция файлов языка Паскаль совершенно не соответствовала интерактивному вводу-выводу, который стал стандартом, начиная с 1980-х гг. Поэтому ряд диалектов Паскаля (например, ТурбоПаскаль) игнорировал эти конструкции, вводя вместо них свои понятия.

Сменить библиотеку значительно проще, чем реализацию.

То же самое относится к введению в базис языка различного рода контейнеров, т.е. структур данных, предназначенных для хранения и выборки данных. Современные языки индустриального программирования содержат единственный контейнер — массив. Более специализированные контейнеры предоставляются стандартными библиотеками.

Это связано с тем, что не существует универсального и абсолютно лучшего для любых применений способа реализации контейнеров (это относится и к алгоритмам). Поэтому в стандартную библиотеку включают реализации контейнеров, подходящие для большинства приложений. Если же стандартная реализация не подходит, то можно использовать вместо стандартной специализированную библиотеку с тем же стандартизованным интерфейсом.

5.3. Управление последовательностью действий: операторный базис языков программирования

Оператор — конструкция языка программирования, основной целью которой является изменение состояния выполняемой программы. Оператор — это абстракция понятия машинной команды, поэтому понятие оператора характерно для языков с императивной парадигмой. Изменение состояния выполняемой программы называется *побочным эффектом*. Таким образом, цель оператора — побочный эффект.

По видам побочного эффекта различают следующие категории операторов:

- оператор присваивания;
- операторы управления последовательностью вычислений;
- операторы ввода-вывода.

Отметим, что операторный базис современных языков наиболее стабилен по сравнению с другими базисными конструкциями. Операторы одного языка похожи на операторы другого. Далее мы будем рассматривать операторы только из языков C++, Java и C#.

Как отмечалось ранее, эти языки не содержат непосредственно понятия «оператор присваивания», его роль может выполнять оператор-выражение, т.е. выражение, за которым следует точка с запятой:

```
выражение; // обратите внимание на ";" !
```

Поскольку выражение — это суперпозиция операций, то такое понятие имеет смысл только при наличии операций с побочным эффектом. Как мы уже знаем, рассматриваемые языки обладают богатым набором операций с побочным эффектом, главная из которых — операция присваивания.

Несмотря на свою гибкость (а может быть благодаря ей), оператор-выражение может иметь очень странный вид:

```
5;
```

Другой пример (C и C++):

```
void proc(); // это объявление процедуры
proc; // это не вызов proc, но и не ошибка
proc();// а вот это - вызов proc
```

Так как имя подпрограммы в C и C++ — это указатель (так же, как имя массива тоже указатель), то вторая строка данного примера содержит корректное выражение (состоящее только из указателя) без побочного эффекта.

Операторы ввода-вывода также отсутствуют в рассматриваемых языках, их роль выполняют классы и функции из стандартных библиотек.

Так что самый широкий класс операторов — это операторы передачи управления. Вспомним, что нормальный порядок выполнения команд — последовательный. Поэтому и операторы императивной программы выполняются последовательно, пока не встретится какой-либо оператор управления.

Блоки. Блок — это последовательность операторов и объявлений, заключенная в фигурные скобки `{}`. Строго говоря, блок не меняет последовательность выполнения, но он позволяет сгруппировать последовательность операторов в один оператор — блок. Блок может появляться везде, где может стоять одиночный оператор. Некоторые конструкции требуют наличия блока: тело функции, try-блок, переключатель.

Блок является областью действия переменных, объявления которых он содержит. После выхода управления из блока эти переменные перестают существовать (напомним, что такое поведение мы называем квазистатическим).

Оператор перехода. Оператор перехода имеет вид

```
goto метка;
```

Оператор, помеченный меткой, будет выполняться следующим после оператора перехода. Метка — это идентификатор. Оператор перехода *локален*, т. е. он может располагаться только в том же блоке, где и метка — в теле функции. Кроме того, переходы внутрь блоков запрещены.

Сейчас общепризнано, что употребление оператора перехода ведет к появлению непонятных и потенциально ошибочных программ. Программы должны строиться только из структурных управляющих конструкций — блоков, ветвлений, циклов (возможность этого доказана математически). Методология построения таких программ называется *структурным программированием* [15]. Употребление оператора перехода нельзя признать удачным почти ни в каком случае. Операторы `return`, `break`, `continue` специально введены для минимизации употребления переходов.

Оператор перехода есть в языках C++ и C#. Язык Java не имеет оператора `goto` вообще, хотя слово `goto` зарезервированное. Интересно, что первый язык программирования Фортран имел пять вариантов оператора перехода!

Операторы ветвления. Операторы ветвления бывают простые и многовариантные. Простой оператор ветвления имеет вид

```
if (условие)
        оператор1
else
        оператор2
```

Если логическое `условие` истинно, то выполняется `оператор1`, а в противном случае — `оператор2`.

Если `оператор2` пуст, то можно использовать укороченную форму:

```
if (условие)
    оператор1
```

Например:

```
if (a < b)
    min = a;
else
    min = b;
```

Наличие укороченной формы может приводить к неоднозначности:

```
if (a < b) if (a < c) x = 0; else x = 1;
// плохой стиль — нужно использовать отступы!
```

Это можно интерпретировать, как

```
if (a < b)
     {if (a < c) x = 0; else x = 1;}
```

либо как

```
if (a < b)
     {if (a < c) x = 0;}
else
     x = 1;
```

Обе интерпретации равноправны, но по соглашению, которого придерживаются все языки (а не только рассматриваемые), выбирается первая. Говорят, что `else` приклеивается к ближнему `if`.

В любом случае нужно использовать отступы для облегчения чтения.

Многовариантное ветвление в общем виде представляет собой суперпозицию операторов `if`, поэтому особого оператора для этого не предусмотрено, но обратите внимание на расположение операторов:

```
if (условие1)
     оператор1
else if (условие2)
     оператор2
...
else if (условиеi)
     операторi
else
     оператор
```

Последняя `else`-часть может отсутствовать.

Существует частный случай многовариантного ветвления, для которого введен специальный оператор — *оператор переключателя*:

```
switch (выражение) блок
```

`Выражение` должно быть целого или перечислимого типа. `Блок` может содержать метки специального вида `case N:`. Также может присутствовать в одном экземпляре метка `default:`. Эффект оператора переключателя состоит в том, что вычисляется значение `выражения`. Пусть его значение есть `i`. Если в блоке переключателя есть метка `case i:`, то происходит переход на эту метку. Если этой метки нет, но есть метка `default:`, то происходит переход на метку `default:`. В противном случае ничего не происходит, и все операторы внутри блока игнорируются. Для выхода из середины блока переключателя используется оператор `break;`. Например:

```
switch (color)
```

```
{
     case RED:
          colorString = "красный";
          break;
     case GREEN:
          colorString = "зеленый";
          break;
     case YELLOW:
          colorString = "желтый";
          break;
     default:
          colorString = "неверный цвет";
          break;
}
```

Мы рассмотрели семантику переключателя в C, C++ и Java. Основной недостаток переключателя в языке C — это использование перехода. Что будет, если мы забудем расставить операторы `break;`? Ничего хорошего: при любом значении переменной `color` произойдет переход на какую-либо метку, а дальше операторы в блоке будут выполняться последовательно таким образом, что переменная `colorString` всегда получит значение `"неверный цвет"`, поскольку соответствующий оператор последний в блоке.

В языке C# требуется, чтобы каждая последовательность операторов после метки `case` или `default` завершалась оператором `break;` или `return;`. Следовательно, компилятор не позволит сделать описанную ранее ошибку.

Операторы цикла. Есть три вида операторов цикла:

- *цикл с предусловием* (`цикл-while`);
- цикл с постусловием (`цикл-do`);
- `цикл-for`.

Циклы while и do. Эти циклы имеют вид

```
while (условие) S      // цикл-while
do S while (условие); // цикл-do
```

Оператор `S` называется телом цикла. Тело выполняется, пока логическое условие истинно.

Циклы отличаются тем, что в первом случае условие повторения проверяется перед первым выполнением тела, а во втором — после первого его выполнения.

Например (линейный поиск элемента `X` в массиве `A`):

```
i = 0;
while (i < A.length && X != A[i])i++;
```

Попробуйте ответить на вопросы: на каком языке написан фрагмент (C++, C# или Java) и что будет, если условие записать в виде

```
X != A[i] && i < A.length
```

Цикл-for. Этот очень мощный оператор, который может моделировать семантику других видов цикла, имеет вид

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

Выражения `e1`, `e2` и `e3` могут быть опущены. Выражение `e1` может содержать объявления переменных. Область действия этих переменных — оператор цикла (т.е. выражения `e1`, `e2`, `e3` и тело цикла `S`). Выражение `e2` должно быть логическим.

Сначала вычисляется выражение `e1`, дальше вычисляется `e2` и пока `e2` истинно выполняется тело цикла `S`, причем после выполнения `S` обязательно вычисляется `e3`. Например, предыдущий пример цикла `while` можно переписать с помощью цикла `for`:

```
int i;
for (i = 0; i < A.length && X != A[i]; i++);
```

Тело цикла — пустой оператор.

Если выражение `e2` отсутствует, то цикл выполняется без проверки условия. Для выхода из тела цикла можно использовать оператор `break;`.

Рассмотрим пример цикла, игнорирующего пустые строки входной последовательности:

```
for (;;)
{
     string s = Console.ReadLine();
     if (s == null) break;
     if (s.Length != 0) Console.WriteLine(s);
}
```

Здесь выход из цикла происходит, если закончилась входная последовательность.

Кроме `break;` есть еще один оператор, нарушающий последовательный порядок выполнения — `continue;`. Этот оператор игнорирует остаток тела и сразу переходит на проверку условия (при этом выражение `e3` все равно выполняется). Операторы `break;` и `continue;` могут появляться и в телах других операторов цикла.

В языках C# и Java есть еще одна форма оператора `for` — цикл-foreach, который в C# имеет вид

```
foreach (T v in Coll) S
```

Здесь `T v` — это объявление переменной, а `Coll` — объект-контейнер (например, массив). Тип данных `T` должен быть типом элементов контейнера. Переменная `v` последовательно принимает значение элементов контейнера. Например, суммирование элементов массива `double [] arr` можно выполнить следующим образом:

```
double sum = 0;
foreach (double x in arr) sum += x;
```

Такая форма не использует операцию индексирования, поэтому более эффективна, чем пример из подразд. 5.2. Правда, оператор цикла `foreach` не может использоваться для модификации контейнера и его элементов. В таких случаях надо использовать другие циклы.

В языке Java синтаксис цикла `foreach` немного отличается, но семантика остается такой же:

```
for (T v: Coll) S
```

Трансляторы C# и Java определяют понятие «контейнер» (в этих языках контейнеры носят название *коллекция*) как класс, реализующий специальный *интерфейс* (в C# — `IEnumerable`, а в Java — `Iterable`). Понятие интерфейса будет рассмотрено в подразд. 8.3, а здесь отметим лишь, что этот интерфейс реализуют массивы и многие классы-коллекции из стандартной библиотеки (например, `ArrayList`). Важно, что программисты могут создавать свои классы-коллекции, которые будут эффективно обрабатываться оператором цикла `foreach`.

Глава 6
ПРОЦЕДУРНЫЕ АБСТРАКЦИИ

6.1. Подпрограммы. Передача управления в подпрограммах

Подпрограмма — это абстракция последовательности команд. Мы можем дать некоторой последовательности операторов имя и использовать это имя в программе вместо указания всей последовательности.

Подпрограммы подразделяются на процедуры и функции.

Основное назначение функции — вычисление одного значения. Таким образом, функция — это обобщение понятия «операция» и мощное средство для расширения базисного набора операций. Для указания того, какое значение вычисляет функция, используется *оператор явного возврата значения*:

```
return выражение;
```

Значение `выражения` и будет значением функции.

Целью процедуры является исключительно побочный эффект — изменение состояния выполняемой программы (например, изменение значений переменных, ввод-вывод и т. п). Таким образом, процедура — это обобщение понятия «оператор» и средство для расширения базисного набора операторов.

Подпрограммы — это первое средство развития, которое появилось в языках программирования. Понятие подпрограммы очень хорошо согласуется с методологией структурного программирования. В соответствии с этой методологией программа должна строиться из структурных блоков, имеющих одну точку входа и одну точку выхода. Проектирование программы начинается с больших блоков (например, три блока «подготовить — обработать — завершить» хорошо подходят под многие задачи обработки данных). Далее каждый блок детализируется в виде суперпозиции более подробных блоков, пока не доходят до уровня операторов языка. Средствами суперпозиции могут служить только управляющие конструкции языка (именно поэтому нельзя использовать оператор перехода). Если оформлять каждый структурный блок (или хотя бы наиболее крупные блоки) в виде подпрограмм, то решение задачи значительно проще понять и разработать.

С современной точки зрения подпрограмма — это «черный ящик», имеющий один вход и один выход. Программисту, использующему

подпрограмму, неважно какова структура этого ящика так же, как неважно из какой точки подпрограммы осуществляется возврат.

Понятие подпрограммы подразумевает наличие двух конструкций:

- определение подпрограммы;
- вызов подпрограммы.

К сожалению, терминология языка С, перешедшая в C++, Java и C#, не совпадает с традиционной. В языке С все подпрограммы называются функциями. Процедура реализуется в С как функция, у которой нет возвращаемого значения. В процедуре нельзя использовать оператор явного возврата значения. Если требуется выйти из середины процедуры, то можно использовать оператор возврата в форме

```
return;
```

В обычных функциях использовать эту форму оператора `return` нельзя.

Определение функции имеет вид

```
тип имя(список_формальных_параметров)тело
```

Тип — это тип возвращаемого значения. Для процедур используется специальное ключевое слово — `void`, сигнализирующее об отсутствии возвращаемого значения. В списке формальных параметров специфицируются имена и типы параметров (другое название параметра — аргумент). Тело — это блок, выполняемый при вызове функции.

Пример определения функции:

```
int GCD (int a, int b)
{
     while (a != b)
     {
          if (a < b)
               b -= a;
          else
               a -= b;
     }
     return a;
}
```

Пример определения процедуры (C++):

```
void swap(T& x, T& y)
{
          T temp = x;
          x = y; y = temp;
}
```

Вызов функции имеет вид

```
имя (список_фактических_параметров)
```

Вызов процедуры может быть только оператором-выражением:

```
swap(a,b);
```

Вызов функции, возвращающей значение, может появиться везде, где может стоять значение соответствующего типа:

```
x = GCD(a,b);
cout << "Наибольший общий делитель 980 и 24="
<< GCD(980,24);
```

При вызове функции выполнение начинается с первого оператора тела функции и завершается выполнением оператора возврата. В процедурах оператор возврата может отсутствовать, тогда выполнение завершается по выходу из блока тела процедуры.

6.2. Передача параметров в подпрограммах

Рассмотрим вопрос параметризации подпрограмм. Для каждой подпрограммы указывается набор формальных параметров.

Можно рассматривать формальные параметры как локальные переменные тела подпрограммы.

При вызове подпрограммы указывается список фактических параметров. Соответствие между фактическими и формальными параметрами выполняется по позиции в списке: первый фактический параметр соответствует первому формальному параметру и т.д. Такой способ называется *позиционным*. Язык C#, начиная с версии 4, предусматривает альтернативный — *ключевой* способ отождествления, в котором используются имена формальных параметров, но мы не будем его рассматривать. Очевидно, что при позиционном способе отождествления количество формальных параметров должно совпадать с количеством фактических.

Фактические параметры представляют собой выражения или переменные (частные случаи выражения).

Рассмотрим подробнее связывание фактических и формальных параметров. Это связывание всегда динамическое, так как происходит в момент вызова подпрограммы.

Существует три вида формальных параметров:

- входные параметры (параметры, от которых требуется только значение). Мы используем только значения фактических параметров, которые не меняются при выходе из тела функции;
- выходные параметры (эти параметры не обязаны иметь начальное значение, но могут быть изменены в теле функции);

- изменяемые параметры (требуется и исходное значение, и возможность его изменения).

С входным параметром может связываться произвольное выражение, а выходным или изменяемым — только объекты, которые могут стоять в левой части оператора присваивания.

В большинстве языков программирования вместо указания вида параметра указывается способ (механизм) связывания параметра, называемый *способом передачи параметра*.

Существует два основных способа передачи параметров: по значению и по ссылке.

Передача параметров по значению. Формальный параметр есть некоторая локальная переменная. Место для локальных переменных отводится в стеке. При вызове подпрограммы значение фактического параметра копируется в соответствующий формальный параметр. Все изменения формального параметра связаны с изменением локальной переменной и не сказываются на фактическом параметре. Перед копированием может потребоваться приведение типа, если типы фактического и формального параметров не совпадают.

Передача параметров по ссылке. Фактически этот способ есть передача ссылки по значению. Формальный параметр — это ссылка на объект. В момент вызова происходит инициализация ссылки фактическим параметром. Преобразования типов в этот момент не происходит: типы формального и фактического параметров должны совпадать. Поскольку ссылка после инициализации отождествляется с объектом, то любые изменения формального параметра подразумевают изменения фактического параметра.

Очевидно, что способ передачи по значению соответствует семантике входных формальных параметров.

По ссылке можно передавать выходные и изменяемые параметры.

В языке C существует только способ передачи параметров по значению (ссылочный тип в языке C отсутствует). Если требуется выходной или изменяемый параметр, то программист должен передавать адреса, тем самым «вручную» моделируя передачу параметра по ссылке. Например, процедуру `swap` из предыдущего пункта можно переписать на C и C++ следующим образом:

```
void swap(T* x, T* y)
{
     T temp = *x;
     *x = *y; *y = temp;
}
```

Вызов этой процедуры имеет вид

```
T a,b;
swap(&a,&b);
```

В языке C это единственный вариант реализации семантики изменяемых параметров, но в языке C++, безусловно, предпочтителен вариант передачи параметров по ссылке.

В языке Java, как и в C, параметры передаются только по значению. При этом указателей и операции взятии адреса в языке нет, поэтому написать аналог процедуры `swap` нельзя. Однако в этом языке есть референциальные типы, поэтому объявление параметра референциального типа и означает передачу самого объекта по ссылке, а выходные и изменяемые параметры типов-значений использовать в Java нельзя.

В языке C# объекты любых типов можно передавать как по значению, так и по ссылке. Для реализации выходных и изменяемых параметров существуют ключевые слова `ref` и `out`.

По умолчанию формальные параметры передаются по значению. Однако перед выходным формальным параметром в списке должно стоять ключевое слово `out`, а перед изменяемым — `ref`. Например:

```
void swap(ref int x, ref int y)
{
     int tmp = x; x = y; y = tmp;
}
void genarray(out byte[] a, int len)
{
     a = new byte[n];
}
```

Вызовы при этом имеют вид

```
int a = 0; b = -1;
swap(ref a, ref b);
byte [] array;
genarray(out array, 256);
```

Обратите внимание на наличие слов `ref` и `out` как в объявлении функций, так и в вызове.

Способы передачи параметров касаются механизма реализации связывания, но не семантики входных и выходных параметров. Понятно, что семантике входных параметров соответствует передача по значению, но при этом способе происходит копирование содержимого фактического параметра, что неприемлемо для значений большого размера. Их лучше передавать по ссылке.

Приведем пример программы передачи параметров на языке C++

```
struct Large {
     char body[100000];
     ... // другие поля
};
```

```
void BadProc(Large s); // не лучший способ!!!
void BetterProc(Large& s); // уже лучше
```

Однако передача по ссылке позволяет менять фактический параметр произвольным образом. Семантика входного параметра при этом теряется. Для того чтобы сохранить (и проконтролировать) неизменяемость ссылочных параметров, в языке C++ можно использовать ссылку на константу:

```
void BestProc(const Large& s);
```

Теперь при попытке изменить содержимое формального параметра `s` компилятор C++ выдаст сообщение об ошибке.

Перегрузка имен

Перегрузка имени означает, что в одной области видимости имеется несколько определяющих вхождений одного и того же имени. В языках программирования определяющим вхождениям обычно соответствуют объявления имен. В этом случае перегрузка имени означает, что в одной области видимости находится несколько разных объявлений одного и того же имени.

Обычно такая ситуация расценивается как ошибка. В самом деле, каждое объявление вводит некоторую сущность (например, объект данных, тип данных и т. п.). Иметь разные сущности, названные одним именем, неудобно, а иногда и невозможно, если их нельзя различить по контексту. Однако это не относится к именам подпрограмм. Имена подпрограмм в большинстве индустриальных языков программирования можно перегружать. Однако если хотя бы одно из перегруженных имен не является именем подпрограммы, то имеет место *конфликт имен*, который рассматривается как ошибка.

Разрешение иметь одно имя для разных вариантов подпрограммы связано с тем, что подпрограмма — это абстракция операции (функция) или действия с побочным эффектом (процедура), а операции (действия) могут иметь разные варианты реализации, оставаясь содержательно одной и той же сущностью. Более того, большинство стандартных операций и процедур в языках программирования уже перегружено. Например, операция сложения в любом языке с несколькими числовыми типами имеет несколько вариантов (по одному для каждого типа). То же самое относится к стандартным процедурам ввода-вывода (например, в языке Паскаль): для каждого простого типа есть свой вариант процедуры `write` или `read`. Естественно распространить перегрузку и на имена пользовательских процедурных абстракций.

Таким образом, в случае перегруженных имен подпрограмм мы имеем дело не с разными одноименными сущностями, а с одной

сущностью (операцией или действием), имеющей несколько вариантов реализации.

Такая ситуация (одна сущность — несколько форм сущности) называется *полиморфизмом*. Языки программирования имеют несколько видов полиморфизма. Полиморфизм при перегрузке часто называют статическим, поскольку выбор формы (варианта реализации) происходит статически, во время трансляции. Подробнее варианты полиморфизма обсуждаются в гл. 8 и 10.

Перегрузка имен подпрограмм сводится к тому, что транслятор должен связать вызов подпрограммы с вариантом ее реализации (объявлением). Какая информация требуется для того, чтобы этот выбор был однозначным?

В языках C++, C# и Java перегрузка имен функций (напомним, что термина «подпрограмма» в этих языках нет) происходит на основе профиля параметров функции. *Профиль функции* (называемый иногда *сигнатурой*) — это упорядоченный список типов формальных параметров из объявления функции. Имена параметров и тип возвращаемого значения не входят в профиль. Так как типы фактических параметров вызова известны компилятору, то вызову функции соответствует список типов фактических параметров (*профиль вызова*). Следовательно, для нахождения требуемого варианта реализации (объявления функции) нужно найти объявление с профилем, соответствующим профилю вызова. Самый простой вариант соответствия — совпадение профилей функции и вызова. Таким образом, понятно, что разные объявления одноименной функции в одной области видимости должны иметь разные профили. Например:

```
void f();
void f(int);
void f(const char *);
f(); // вызов первой функции
f(0); // вызов второй функции
f("overloading sample"); // вызов третьей функции
```

Конкретный механизм сопоставления профилей может быть достаточно сложным, так как точное совпадение — это частный случай. При поиске соответствия профилей необходимо учитывать стандартные неявные преобразования, пользовательские преобразования (если они разрешены) и другие аспекты. Подробности правил перегрузки рассматриваются в конкретных руководствах [2, 17, 28].

Некоторые языки программирования (например, C# и C++) позволяют перегружать не только имена подпрограмм, но и некоторые стандартные операции, трактуя их как функции. Особенно богатые возможности перегрузки стандартных операций предоставляются языком C++. Подробнее это рассматривается в гл. 7.

6.3. Подпрограммные типы данных

Как уже говорилось в подразд. 5.1, имена функций в C и C++ рассматриваются как указатели. В результате можно рассматривать имена функций как константы некоторых указательных типов данных. В C и C++ такие типы можно назвать функциональными (а в других языках подпрограммными).

Функциональные типы необходимы, например для передачи функций как параметров других функций.

Приведем пример переменной функционального типа:

```
double (*pFunc) (double);
pFunc = cos;
```

Такое описание означает, что `pFunc` — это указатель на функцию от одного параметра типа `double`, возвращающую значение тоже типа `double`. Далее переменной `pFunc` присваивается значение адреса функции `cos` (косинус). Указатели функционального типа необязательно разыменовывать. Вызывает функцию, присвоенную `pFunc`, следующий оператор:

```
double x = pFunc(1.57);
```

Опишем заголовок функции, вычисляющей определенный интеграл от функции `F` на интервале `[A, B]` с точностью `eps`:

```
double Integrate (double (*F)(double), double A,
double B, double eps);
```

Вызов этой функции может иметь вид

```
double t = Integrate(sin, 0.0, 1.57, 1E-6);
```

Заметим, что несмотря на наличие функционального типа, функции нельзя считать полноценными объектами данных в языке C++ (по сравнению с функциональными языками). Главная причина этого заключается в том, что множество значений функциональных типов состоит только из функциональных констант (имен функций). Нельзя динамически (в процессе выполнения программы) строить и исполнять новые функции.

В целом понятие указателя провоцирует появление ошибок в программах и является слишком низкоуровневым. Поэтому язык Java не содержит понятия функционального типа, а следовательно, в нем нельзя передавать функции, как параметры. Однако объектно-ориентированный стиль программирования позволяет обойтись без такой концепции.

Язык C# тем не менее включает в себя понятие «делегаты», аналогичное функциональному типу, но не основанное непосредственно на концепции указателя и являющееся более надежным.

Делегаты обладают достоинствами функционального типа данных, и в то же время не имеют его недостатков. Внешне объявление делегата очень похоже на объявление функционального типа. Объявление делегата начинается с ключевого слова `delegate`, за которым идет спецификация функции:

```
delegate void Handler (Object o, EventArgs e);
// Handler - имя делегата
```

Делегат `Handler` можно рассматривать как функциональный тип, значениями которого являются списки функций, каждая из которых имеет указанный в объявлении делегата прототип. Экземпляр делегата объявляется как переменная такого типа:

```
Handler hndl;
```

К экземпляру делегата можно применять следующие операции:

- присваивание (=);
- добавление функции (+=);
- удаление функции(–=);
- вызов.

Представим, что у нас есть несколько функций, удовлетворяющих объявлению делегата `Handler` (значение ключевого слова `static` объясняется в гл. 7, сейчас его можно проигнорировать):

```
static void MyHandler1(Object o, EventArgs args)
     { ... }
static void MyHandler2(Object o, EventArgs args)
     { ... }
```

Тогда следующее присваивание сотрет старое содержимое экземпляра делегата и создает в нем список из одной функции `MyHandler1`:

```
hndl = MyHandler1;
```

Можно добавить к списку новую функцию `MyHandler2`:

```
hndl = new Handler(MyHandler2);
```

Теперь операция вызова делегата будет вызывать по очереди все функции из списка, передавая каждой функции аргументы из вызова:

```
EventArgs a = new EventArgs();
hndl(this,a); // вызов MyHandler1 и MyHandler2
hndl -= MyHandler1;
// удаление MyHandler1 из списка
hndl(this,a); // вызов MyHandler2
```

Делегаты могут использоваться и для передачи функций как параметров другим функциям (если описать параметр функции как экземпляр делегата).

Глава 7
ОПРЕДЕЛЕНИЕ НОВЫХ ТИПОВ ДАННЫХ

7.1. Класс как тип данных

В современных языках программирования типы данных рассматриваются как совокупность множества значений и множества операций над типом. Множество значений определяется структурой данных, а множество операций — это набор подпрограмм, имеющих формальные параметры соответствующего типа. Механизм классов отражает эту концепцию, позволяя объединить структуру данных типа и набор его операций в единое понятие, которое и называется классом.

Автор языка C++ Б.Страуструп вводит понятие класса как типа данных, определяемого пользователем [28].

Класс расширяет понятие записи (структуры) в том смысле, что структуры обладают только данными (полями), а класс может обладать и данными, и функциями.

Класс представляет собой набор определений *членов класса*, которому присвоено имя. Членами класса могут быть переменные (аналогично полям структуры), называемые *членами-данными*, функции, называемые *методами доступа* (или просто методами), и другие классы, называемые вложенными. Объявление класса имеет следующий вид:

```
class имя_класса {
        список_объявлений_членов_класса
}    //  в  C++  после  объявления  класса  нужно
        // поставить точку с запятой
```

Объявление члена-данного идентично объявлению переменной, объявление метода (члена-функции) — это объявление функции. В C++ в объявлении класса можно указать как полное определение метода, так и только прототип метода (т.е. заголовок без тела), а в C# и Java требуется полностью определить функцию. Объявление вложенного класса — это обычное объявление класса. Перед объявлением члена класса могут стоять *модификаторы*, смысл которых будем определять по мере введения модификаторов.

Имя класса — это имя типа. Можно объявлять объекты (или ссылки на объекты) так же, как это делается с базисными типами данных:

```
имя_класса имя_объекта
```

Как уже отмечалось, объекты классов в C++ размещаются в любом виде памяти (все зависит от контекста объявления), а в C# и Java — только в динамической памяти при выполнении операции `new`.

Члены класса подразделяются на статические и нестатические. Начнем с рассмотрения статических членов.

Приведем пример законченной программы на Java:

```
public class Sample { // это пример на Java
     public static void main(String[] args)
     {
          Hello.Greet();
     }
}
class Hello {
     static final String msg = "Hello, world!";
     static void Greet()
     {
          System.out.println(msg);
     }
}
```

Если запустить эту программу, то она выдаст строку `Hello, world!`. Обратим внимание на класс `Hello`. Он содержит два члена, перед обоими стоит модификатор `static`, превращающий их в статические (по умолчанию эти члены нестатические). Перед членом-данным `msg` стоит также модификатор `final`, который сообщает транслятору о том, что присвоенное переменной значение не может измениться (финальное). Другими словами, `msg` — это константа.

Статический член-данное — это переменная, которая существует в единственном экземпляре для всех объектов класса. Статический член размещается в памяти при загрузке программы (точнее при загрузке класса) независимо от размещения (или неразмещения) объектов класса.

Статический метод — это функция, которую можно вызывать, не используя объекты класса.

Таким образом, статические члены — это члены, никак не зависящие от конкретных объектов класса. В языке SmallTalk (первом «чистом» объектно-ориентированном языке) используется довольно удачная терминология: статические члены-данные называются переменными класса, а нестатические — переменными экземпляра.

Обращение к статическим членам класса происходит через имя класса:

```
Hello.Greet();
System.out.println(Hello.msg);
```

Таким образом, класс — это область непосредственной видимости для имен его членов (это относится не только к статическим членам). Поэтому, чтобы обратиться к статическому члену класса, следует указать имя класса. Говорят, что имена членов класса необходимо «уточнять» именем класса (полная аналогия с обращением к константам перечислимого типа в C# и Java). Однако преимущество функций-членов состоит в том, что они могут обращаться к членам своего класса без уточнений, поэтому функция `Greet` обращается к члену `message` своего класса `Hello` без уточнения. Такая возможность делает текст функций-членов более выразительным и компактным.

Зачем нужны статические члены? В языке C++ статические члены можно уподобить глобальным переменным и глобальным функциям. Отличие (в пользу членов класса) состоит в том, что имена методов локализованы в классе, их надо уточнять (в C++ синтаксис уточнения имени отличается: вместо точки должен стоять знак «`::`», например, `Hello::Greet()`), и поэтому их проще выбирать по сравнению с глобальными именами, т. е. меньше вероятность конфликта имен. Еще одно важное отличие в пользу статических членов состоит в возможности функций-членов обращаться к закрытым (инкапсулированным) членам класса (см. подразд. 7.3). Однако если в C++ еще можно (хотя и не рекомендуется) обходиться без статических членов, то в C# и Java никакая программа без них не обходится. Дело в том, что в этих языках программа состоит только из классов (правда, можно еще объявлять перечисления и интерфейсы, но это не меняет дела). Кроме того, в этих языках отсутствует понятие глобальной функции и глобальной переменной. Любая функция и любой объект данных должны принадлежать какому-либо классу. Такая ситуация вообще характерна для объектно-ориентированных языков.

В приведенном примере на Java есть только два описания класса, но нет ни одного их экземпляра (ведь в программе нет ни одного обращения к операции `new`). По правилам языка статическая функция с прототипом `static void main(String[] args)` в одном из классов (любом) играет роль функции `main` в C++, которая вызывается при запуске консольной программы (аргументы командной строки передаются как элементы параметра `args`). Такой класс называется главным, и перед ним, а также перед методом `main` должен стоять модификатор `public`. Этот модификатор означает разрешение использования класса и соответствующего метода в любом месте программы. Понятие статической функции позволяет обойтись без создания каких-либо экземпляров классов при запуске программы.

В языке C# действует такое же правило для консольных программ, только функция называется `Main` (с прописной буквы M).

Необходимость и смысл статических функций можно увидеть на примере класса `Math` в языках C# и Java. Этот класс содержит определения математических функций (тригонометрических, экспоненты

и др.) и констант (PI, E). Очевидно, что все они реализованы как статические члены. Такие классы являются контейнерами операций и не содержат никаких нестатических членов данных. Объекты таких классов, как Math, не имеет смысла размещать в памяти.

В языке C# есть специальное понятие статического класса. Если перед определением класса стоит модификатор static, то такой класс может иметь только статические члены, и создавать объекты такого класса запрещено. Примером стандартного статического класса, конечно, является Math.

Приведенный пример можно переписать и на C#, для чего надо только переименовать main, изменить модификатор final на const и заменить вывод на System.Console.WriteLine(Hello.msg);.

Однако статические члены, конечно, играют вспомогательную роль. Основными в классе являются нестатические члены класса.

Нестатические члены-данные играют роль, аналогичную полям записи (структуры). Каждый экземпляр класса имеет свой набор членов-данных, поэтому к нестатическому члену-данному нельзя обращаться через имя класса: необходимо уточнять, какому именно объекту принадлежит член. В языках C# и Java обращение к нестатическому члену происходит только через ссылку на конкретный объект (ведь сами объекты анонимные). В C++ для обращения к члену используется имя объекта. Например (считаем, что класс X содержит нестатический целый член k):

```
// фрагмент на C# и Java
X a = new X();
a.k = 0;
X b = new X();
b.k = a.k;
// фрагмент на C++
X obj; // компилятор C++ сам разместит объект
obj.k = -1;
X * pX = new X();
(*pX).k = -1;
pX -> k = -1; // другая форма записи
```

Поясним последнюю строчку. Если указатель pX ссылается на объект какого-либо класса X (другими словами, имеет тип X*), для обращения к члену k этого класса вместо громоздкой записи (*pX).k можно использовать операцию «->»:

```
pX -> k.
```

Теперь рассмотрим нестатические методы класса.

Каждый нестатический метод класса X содержит неявный формальный параметр — ссылку на объект класса X, для которого вызывается метод. Эта ссылка обозначается ключевым словом this (в C++ this не ссылка, а указатель). Через эту ссылку метод класса может

обращаться ко всем членам класса (для краткости будем опускать слово «нестатический», так как члены во всех языках по умолчанию нестатические). Передача этой ссылки производится неявно транслятором, а сама ссылка берется из обращения к методу. Вспомним, что обращение к члену должно быть через ссылку на объект, которая и будет ссылкой `this`.

Рассмотрим пример на Java:

```
class Simple
{
     int x = 0, y = 1;
     void SwapMembers()
     {
          int tmp = this.x;
          this.x = this.y; this.y = tmp;
     }
     void Out()
     {
          System.out.println(this.x);
          System.out.println(this.y);
     }
}
public static void main(String[]args)
{
     Simple s = new Simple();
     s.Out();
     s.SwapMembers();
     s.Out();
}
```

При вызове `main` будут напечатаны сначала 0 и 1, а затем 1 и 0 (каждое значение на новой строке).

Однако использование `this` внутри функций-членов необязательно (как необязательно было использование имени класса для статических членов внутри статической функции). В приведенном примере можно опустить `this` (транслятор сам подставит эту ссылку для имен членов класса). Имена членов класса непосредственно видимы внутри методов класса, поэтому код тел методов получается более компактным и выразительным. Далее мы будем явно употреблять `this` только там, где это необходимо.

Приведем более содержательный пример класса на языке Java, в котором реализуется тип данных стек. Стек — это контейнер, хранящий данные по принципу «последний поступивший выдается первым». Аналогом может служить стопка тарелок (первой берем верхнюю тарелку, которую положили последней). Следует иметь в виду, что стек — очень популярная структура данных, поэтому каждый индустриальный объектный язык включает класс-стек в стан-

дартную библиотеку (в Java класс `Stack` находится в библиотечном пакете java.util). Так что пример, конечно, носит модельный характер, но при этом демонстрирует возможности Java по созданию новых типов данных.

Стек реализуется в виде массива. Индекс первого свободного элемента обозначен как `top`. Функция `Pop()` извлекает элемент из стека, а функция `Push(x)` заносит элемент в стек. Функции `IsEmpty()` и `IsFull()` проверяют стек на пустоту и полноту соответственно. Например:

```
class Stack
{
     int [] body;
     int top;
     Stack(int size)
     {
          body = new int[size];
          top = 0;
     }
     int Pop() { return body[--top]; }
     void Push(int x) { body[top++] = x; }
     boolean IsEmpty() { return top ==0; }
     boolean IsFull() {
          return top == body.length;
     }
}
```

Поясним, что представляет собой первая функция-член класса. Она имеет «странный» заголовок: `Stack(int size)`, странность которого заключается в отсутствии указания возвращаемого типа и в том, что имя метода совпадает с именем класса. Если мы попробуем проделать такое с другим методом или членом-данным, то программа не оттранслируется.

Эта «странная» функция называется *конструктором*. Конструктор автоматически вызывается при размещении объекта соответствующего класса в памяти. Фактические параметры вызова конструктора указываются в операции `new` в виде `new Stack(128)`. При этом в конструктор будет передано значение 128 — длина массива, где будут храниться данные стека. Конструктор относится к категории специальных функций-членов. Подробнее эти функции рассматриваются в подразд. 7.2.

Конечно, приведенный стек обладает рядом недостатков, главным из которых является отсутствие инкапсуляции и неустойчивость к ошибкам, а точнее, неадекватные сообщения в случае возникновения ошибок (эти недостатки будут рассматриваться в подразд. 7.3 и в гл. 9), здесь же отметим, что несмотря на компактность кода, созданный тип данных вполне работоспособен.

Приведем пример использования нашего стека: выдадим элементы массива в обратном порядке:

```
public static void main(String[]args)
{
     Stack s = new Stack(128);
     int [] x = {1,2,3,4,5,6,7};
     for (int k : x) s.Push(k);
     while (!s.IsEmpty())
          System.out.println(s.Pop());
}
```

Еще раз напомним, что в языках C# и Java объекты размещаются в динамической памяти. Иногда это приводит к некоторой неэффективности. Рассмотрим небольшой класс `Point` на языке Java, который реализует понятие точки на плоскости (в целочисленной системе координат), которой может служить, например экран монитора.

```
class Point
{
     int x,y;
     // координаты точки, других членов данных нет
     Point(int x, int y)
          { this.x = x; this.y = y; }
     void Move (int dx, int dy)
          { x+= dx; y += dy; }
     // другие методы ...
}
```

Пусть у нас есть объект `screen`, имеющий экранную функцию рисования линий `DrawLine`, которая принимает массив точек и вырисовывает ломаную линию, задаваемую массивом (каждые два соседних элемента массива задают отрезок, а весь массив — ломаную), в соответствии со следующим алгоритмом:

```
static final int MAX_POINTS = 1024;
void DrawSample()
{
     Point [] points = new Point[MAX_POINTS];
     int i = 0;
     while (есть_еще_точки) {
          int x,y;
          ввести координаты очередной точки x и y
          Point p = new Point(x,y);
          points[i++] = p;
          // проверка на переполнение массива
          if (i == MAX_POINTS) break;
```

```
    }
    screen.DrawLine(points);
}
```

Проблема в том, что массив `points` содержит не точки, а ссылки. А объекты-точки надо еще разместить в памяти, что и происходит в цикле `while`. Общий максимальный расход памяти определяется в виде

```
MAX_POINTS *(sizeof (ссылка)+sizeof(Point))
```

Тут важно то, что сам объект класса `Point` невелик: всего два целых значения. Следует понимать, что методы не занимают память объекта, поэтому расход памяти под ссылки сравним с общим объемом памяти для хранения точек. К тому же автоматический сборщик мусора должен сначала утилизировать память под объекты-точки, а затем только освободить массив из ссылок.

Язык Java не позволяет как-либо обойти эту проблему, но в языке C# существует специальное понятие, позволяющее уменьшить накладные расходы на размещение объектов, которое называется *структурой*. Структура языка C# — это средство определения новых типов с урезанной по сравнению с классами функциональностью. Внешне определение структуры имеет вид определения класса, только здесь вместо ключевого слова `class` используется `struct`:

```
struct Point
{
    int x,y;
    public Point(int x, int y)
          { this.x = x; this.y = y; }
    public void Move (int dx, int dy)
          { x+= dx; y += dy; }
    // другие методы ...}
}
```

Однако структура — это не класс, а тип-значение. Объекты структур размещаются так же, как и переменные простых типов данных, в любом виде памяти — в зависимости от контекста объявления. Переменная структурного типа — это объект, а не ссылка (в этом смысле структуры C# ведут себя, как объекты классов в C++). Поэтому при выполнении операции `new Point[MAX_POINTS]` создается массив объектов, а не ссылок (так же, как и в C++).

Цикл ввода точек имеет вид

```
while (есть_еще_точки) {
          int x,y;
          ввести координаты очередной точки x и y
          points[i].x = x;  points[i].y = y;
          // проверка на переполнение массива
```

```
        if (++i == MAX_POINTS) break;
}
```

В этом случае меньше расход памяти, меньше работы у сборщика мусора: ему необходимо только утилизировать память от массива.

Однако структуры нельзя использовать в качестве базовых и производных классов, поэтому они применимы только для небольших и простых типов данных. Отметим, что структуры языка C++ при внешней похожести на структуры языка C# являются полноценными классами без какого-либо ограничения функциональности.

Перегрузка стандартных операций

Языки C++ и C# позволяют перегружать некоторые стандартные операции, которые рассматриваются как функции языка. Количество аргументов в этом случае должно совпадать с количеством аргументов стандартной операции.

В C++ можно перегружать почти все операции (исключение составляет доступ к члену: «точка» и ряд других), включая присваивание, индексирование, вызов (круглые скобки) и другие достаточно «экзотические». Все они имеют один или два аргумента.

В C# набор перегружаемых операций существенно уже и содержит только арифметико-логические операции. Кроме того, аргументы перегружаемых операций в C# могут быть только ссылками на пользовательские классы. В C++ допустимы любые типы аргументов перегружаемых операций.

Синтаксис перегружаемой операции в C++ имеет вид

```
тип operator символ (аргументы) тело-функции
```

Здесь `тип` — это тип возвращаемого значения, `символ` — это символ операции («+», «[ ]», «–>» и др.).

Любые операции можно перегружать с помощью функции-члена. В этом случае первый (или единственный) аргумент операции рассматривается как объект класса, в котором перегружена операция. Адрес этого объекта передается в функцию-член как `this`, а второй аргумент (если он есть) передается в функцию-член как единственный аргумент:

```
class X
{
     X operator + (const X& second); // двуместный +
     X operator + ();// одноместный +
};
```

В этом примере обе формы операции «+» (одноместная и двуместная) перегружены как функции-члены. Транслятор трактует обращение к этим операциям следующим образом:

```
X a,b,c;
c = a+b; // эквивалентно c = a.operator+(b);
a = +b; // эквивалентно a = b.operator+();
```

Такая форма подходит не для всех операций (подробнее это рассматривается в подразд. 7.3). Некоторые операции (арифметико-логические, включая комбинированные присваивания) можно перегружать внешней функцией (либо глобальной, либо функцией-членом другого класса):

```
X operator + (const X& a, const X& b); // двуместный+
X operator + (const X& a); //одноместный +
X a,b,c;
c = a+b; // эквивалентно c = operator+(a, b);
a = +b; // эквивалентно a = operator+(b);
```

Однако компилятор не может различить по виду `a+b`, какой вариант имеется в виду. Он ищет либо внешнее определение, либо функцию-член. Если присутствуют оба варианта, то это ошибка. Программист должен определить (для каждой формы операции отдельно), следует ли вообще перегружать эту форму операции, а если следует, то каким образом (метод либо внешняя функция).

Наличие возможности перегрузки (вместе с механизмом конструкторов и других специальных функций-членов) позволяет определять классы, почти неотличимые от базисных типов, например тип комплексных чисел:

```
struct complex {
    double re, im;
    ...
    complex& operator +=(complex x) {
    re += x.re, im += x.im);
    return *this;
    }
};
complex operator +(complex x, complex b){
complex t = x;
return x += b;
}
ostream& operator << (ostream& s, complex c)
{
    return s << '(' << c.re + << ',' << c.im << ')';
}
istream& operator >> (istream& s, complex& c)
{
    return s >> c.re >> c.im;
}
```

Последние перегрузки (операций сдвига) демонстрируют подход языка C++ к реализации операций стандартного ввода-вывода. В стандартной библиотеке определены классы потоков ввода (istream) и вывода (ostream). Для этих классов и всех базисных типов перегружены операции сдвига вправо (ввод) и сдвига влево (вывод) как глобальные функции. Операции всегда принимают первый аргумент, т.е. ссылку на поток, эта же ссылка является результатом операции. За счет ассоциативности операции выполняются слева направо. Для новых классов операции ввода-вывода перегружаются аналогично с использованием уже определенных операций для членов-данных. Компилятор всегда находит по типу правого параметра операций перегруженного сдвига требуемый вариант операции, обходясь без адресов, списков параметров переменной длины и прочих особенностей библиотеки стандартного ввода-вывода языка C.

Другие примеры перегрузки операций (например, присваивания и индексирования) можно найти в следующем пункте.

В C# нет понятия глобальной функции, а операции можно перегружать только статической функцией-членом класса-операнда операции:

```
struct Complex
{
     public double Re, Im;
     ...
     public static Complex operator +
          (Complex l, Complex r)
     { return new Complex(l.Re + r.Re, l.Im + r.Im);
     }

}
```

Подробнее перегрузка операций в C++ и C# рассматривается в [23, 28].

7.2. Специальные функции-члены

Многие объектно-ориентированные языки программирования содержат специальные функции-члены. Особая роль этих функций состоит в том, что компилятор выделяет их среди обычных методов и может вставлять обращения к ним в нужных местах программы.

К специальным функциям относятся:

- конструкторы (C++, C#, Java);
- деструкторы (C++) и финализаторы (C#, Java);
- операторы преобразования (C++, C#).

Специальные функции-члены введены для решения следующих проблем, возникающих при использовании новых типов данных:

- инициализация объектов типа;
- уничтожение объектов типа;
- копирование объектов типа;
- неявные преобразования объектов одного типа в другой.

Проблема инициализации состоит в том, что после размещения объекта в памяти требуется выполнить действие (нередко нетривиальное) по его инициализации (например, в стеке из подразд. 7.1 необходимо обнулить индекс `top` и разместить тело стека). В большинстве языков программирования без классов программисту необходимо не забыть явно запрограммировать соответствующую операцию, причем транслятор никак не контролирует соблюдение этого правила.

Аналогичная проблема возникает при уничтожении объектов. Нередко объект захватывает какие-либо системные ресурсы в процессе инициализации и(или) функционирования, например размещает в памяти свои члены-данные (как это делал объект-стек), открывает внешние файлы, создает графические объекты типа курсоров. Вычислительные ресурсы не безграничны, поэтому все захваченные ресурсы необходимо освободить при уничтожении объекта. В языках со сборкой мусора динамическая память освобождается автоматически (поэтому стек в Java и C# не нуждается в дополнительных действиях по зачистке памяти), но забота об остальных видах ресурсов ложится на программиста.

Объектно-ориентированные языки поддерживают автоматическое или полуавтоматическое выполнение операций по инициализации и освобождению ресурсов. Конструкторы отвечают за инициализацию объектов. Проблема освобождения захваченных ресурсов в *C++ решается с помощью деструкторов*, а в *Java и C# — с* помощью финализаторов и ряда дополнительных конструкций и интерфейсов.

Проблема копирования состоит в том, что объекты могут содержать ссылки на другие объекты. При копировании таких ссылок возникает дилемма: копировать либо только ссылку, либо полностью содержимое объекта, на который указывает ссылка. Первый вид копирования называется *поверхностным*, а второй — *глубоким*. Ответ на вопрос, какую копию делать (поверхностную или глубокую), не всегда тривиален. Например, при копировании стека необходимо полностью копировать содержимое его тела. А при копировании динамических строк можно обойтись копированием ссылки, так как строки являются неизменяемыми объектами (это одна из причин высокой эффективности реализации динамических строк).

В общем случае транслятор не в состоянии выбрать нужную семантику копирования, поэтому в языках программирования, которые мы рассматриваем, принята следующая схема: по умолчанию реализуется поверхностное копирование, но программисту предоставля-

ются средства для указания пользовательской (глубокой) семантики копирования. В C++ этими средствами являются переопределение конструктора копирования и операции присваивания, в Java и C# — наследование и реализация специального интерфейса.

Конструкторы. Конструктор — это специальная функция, которая не имеет типа возвращаемого значения (не путайте с процедурой, которая реализуется как void-функция), а имя конструктора совпадает с именем класса:

```
class X{
    X(аргументы конструктора) { тело конструктора}
    ...
}
```

Конструктор класса вызывается автоматически при размещении объекта в памяти. Конкретный момент вызова зависит от класса памяти:

- конструкторы статических объектов вызываются до входа в функцию main;
- конструкторы квазистатических объектов выполняются при входе в блок, точнее в момент, когда управление достигает объявления соответствующих объектов;
- конструкторы динамических объектов выполняются при вызове операции new.

Заметим, что первые две ситуации могут встречаться только в C++, поскольку в Java и C# объекты размещаются только в динамической памяти.

Пусть у нас есть класс X с конструкторами следующего вида:

```
class X{
    X() {...}
    X(int i) {...}
    X(int i, double k) {...}
}
```

В языке C++ аргументы вызова конструктора указываются следующим образом:

```
X a; // специальный случай — вызов X()
X b(1); // вызов X(int)
X c(0, 1.5); // вызов X(int, double)
X * pX = new X(); // вызов X()
pX= new X(-1); // вызов X(int)
int a;
pX = new X(a, 0.0); вызов X(int)
```

Замечание. В первой строке здесь более логичным кажется написать X a(), как в четвертой строке. Однако по синтаксису языка C, перешедшему по наследству в C++, конструкция X a() синтаксически является объ-

явлением прототипа некоторой функции a, не имеющей параметров и возвращающей значение типа X, поэтому пришлось вводить специальный случай X a.

В языках Java и C# аргументы вызова конструктора указываются в операции new:

```
X a = new X(); // вызов X()
X b = new X(1); // вызов X(int)
X c = new X(0, 1.5); // вызов X(int)
```

В случае если транслятор не находит конструктор с нужным прототипом, то выдается сообщение об ошибке.

Поскольку конструктор предназначен для автоматического вызова, то на него накладываются достаточно очевидные ограничения:

- конструктор не может быть вызван как обычная функция;
- нельзя получить адрес конструктора.

Заметим, что в C++ существует еще один контекст вызова конструктора, который иногда ошибочно принимают за явный вызов: при создании временного объекта. Временные объекты создаются компилятором либо неявно (например, при выполнении неявных преобразований объектов классов), либо явно, например при выполнении оператора возврата:

```
X MakeX(int i)
{
     return X(i); // вызов X(int)
}
```

Память под временные объекты отводится компилятором.

Особенность конструкторов состоит в том, что часть его кода генерируется автоматически транслятором. Эту часть кода называют *системной*. Кроме системной может быть и пользовательская часть, представляющая собой тело конструктора из объявления. Системная часть конструктора всегда выполняется перед пользовательской частью. Необходимость системной части вызвана тем, что класс может наследовать какому-то базовому классу, поэтому необходимо вызвать конструктор базового класса. Кроме того, в языке C++ класс может содержать подобъекты (т.е. члены-данные, которые сами являются объектами других классов). В Java такая ситуация невозможна (там частью объекта может быть только ссылка, но не сам объект), а в C# объект может содержать структуры. Системная часть содержит вызовы конструктора базового класса (во всех языках) и конструкторов — членов подобъектов (подструктур) в языках C++ и C#.

Конечно, системная часть может быть пустой (если класс не является ничьим наследником и не содержит подобъектов). Примером такого класса является рассмотренная структура Complex на языке C++. «Умный» компилятор, конечно, игнорирует пустую системную часть.

В языках C# и Java системная часть включает в себя еще выполнение инициализаторов членов-данных. Например, в примере со стеком можно было указать инициализатор члена `top`:

```
int top = 0;
```

Инициализирующим выражением может быть любое корректное выражение (необязательно константное).

Особая роль конструктора умолчания. Конструктором умолчания называется конструктор без параметров: `X()`. Его особая роль проявляется при генерации системной части: если не указано, какой именно конструктор (базового класса, подобъекта) вызывается, то генерируется вызов конструктора умолчания.

В силу важности конструктора умолчания он может быть либо описан явно, либо сгенерирован автоматически. *Конструктор умолчания генерируется автоматически* (такие конструкторы называют *неявными*), *если в классе нет ни одного явно объявленного конструктора*. Откуда, кстати, следует, что любой класс имеет хотя бы один конструктор (либо сгенерированный неявный конструктор умолчания, либо явно объявленный конструктор).

В классе `Stack` мы могли бы обойтись без явных конструкторов, если бы убрали конструктор `Stack(int size)` и вставили инициализаторы:

```
int top = 0;
int []body = new int[DefaultStackSize];
static int DefaultStackSize;
```

`DefaultStackSize` — это статический член класса `Stack`. Его можно объявить как константу (тогда все стеки будут иметь один и тот же размер, что явно не лучший вариант) либо разрешить менять динамически перед вызовом сгенерированного конструктора умолчания:

```
Stack.DefaultStackSize = 64;
Stack s1 = new Stack(); // размер - 64
Stack s2 = new Stack(); // размер - 64
Stack.DefaultStackSize = 128;
Stack s3 = new Stack(); // размер - 128
```

В данном случае лучше, конечно, использовать первоначальный вариант с явным конструктором.

В правиле генерации неявного конструктора есть только одно исключение: в структурах языка C# конструктор умолчания всегда неявный независимо от наличия или отсутствия других конструкторов (его нельзя определять явно). Сгенерированный конструктор будет присваивать членам-данным умолчательные значения (в языках C# и Java у всех типов значений есть значения по умолчанию, а у ссылок — это `null`).

А что делать, если конструктора умолчания нет? Например, для стека придумать умолчательное значение размера трудно (чем значение 128 лучше или хуже значения 32?), поэтому лучше обойтись без конструктора умолчания, заставив программиста — пользователя стека указать размер явно при вызове `new`.

В этом случае необходим механизм явного указания, какие конструкторы (и с какими параметрами) требуется вызвать. В C++ следует указывать вызовы конструкторов базовых классов и подобъектов, в C# и Java — только вызовы конструкторов базового класса.

В C++ для указанных целей существует конструкция *список инициализации*, в которой указаны параметры вызовов конструкторов. Список инициализации указывается через двоеточие после заголовка конструктора и непосредственно перед блоком-телом конструктора:

```
заголовок : список_инициализации тело
```

Пусть класс `D` является наследником класса `B`, у которого есть конструктор `B(int)`. Пусть также `D` содержит член-данное `subobj` — подобъект класса `S`, а у класса `S` есть конструктор `S(double)`:

```
class B { ... B(int); ...};
class S { ... S (double); ...};
class D : public B {
      int k;
      S subobj;
      D ();
      ...
};
```

Тогда конструктор `D()` должен включать в себя список инициализации для базового класса `B` и подобъекта `subobj`, например:

```
D::D() : subobj(1.0), B(-1) { ... }
```

В список инициализации можно включать и инициализаторы для простых типов данных:

```
D::D() : subobj(1.0), B(-1), k(0) { ... }
```

Это особенно удобно, так как C++ не позволяет инициализировать члены-данные в объявлении класса, а для членов-ссылок это вообще единственно возможный способ инициализации.

В языке C# список инициализации «вырождается» в инициализатор конструктора. Он может содержать только либо вызов конструктора базового класса вида `base(аргументы)` (здесь `base` — это не имя, а ключевое слово!), либо вызов альтернативного конструктора вида `this(аргументы)`:

```
class D : B
{
```

```
D(int i): base(i) { ... }
D() : this(0) { ... }
```

И, наконец, в Java списка инициализации (или инициализатора конструктора) нет, а вместо него действует следующее правило: первым оператором конструктора может быть либо вызов конструктора базового класса вида `super(аргументы)` (ключевое слово `super` в Java является аналогом `base` в C#), либо вызов альтернативного конструктора вида `this(аргументы)`. Например, конструктор умолчания в классе `Stack` мог бы иметь вид

```
Stack() { this(DefaultStackSize); }
```

Обсудим вопрос об инициализации статических членов-данных. Статические члены должны быть инициализированы не в конструкторе объекта, так как по определению статические члены класса существуют независимо от конкретных экземпляров класса.

В C++ программист должен разместить и инициализировать статические члены-данные где-либо в программе по своему усмотрению (главное, чтобы такое определение было в программе в единственном экземпляре):

```
class S {
      static int counter;
      ...
};
int S::counter = 0;
```

В языке C# статические члены можно инициализировать при объявлении члена. Для нетривиальной инициализации существует понятие статического конструктора:

```
class X
{...
      const int N = 64;
      static byte [] sqrs;
      static X()
      {
          sqrs = new byte[N];
          for (int i = 0; i < N; i++)
                sqrs[i] = i*i;
      }
}
```

В языке Java инициализировать в объявлении можно только константы, а для остальных случаев есть аналог статического конструктора — статический блок:

```
class X
{     ...
```

```
        static final int N = 64;
        static int [] sqrs;
        static // статический блок
        {
             sqrs = new int[N];
             for (int i = 0; i < N; i++)
                  sqrs[i] = i*i;
        }
}
```

Деструкторы и финализаторы. Деструкторы и финализаторы — это специальные функции-члены класса, автоматически вызываемые при уничтожении объекта класса. Роль деструкторов и финализаторов — освобождение захваченных объектом вычислительных ресурсов. Если объект не захватывает ресурсы или ресурсы освобождаются автоматически, то нужды в деструкторе нет.

Деструктор класса `X` в языке C++ имеет вид

```
~X();
```

Деструкторы бывают только без параметров.

В языке Java деструкторов нет, их роль (до определенной степени) играет метод `void finalize()` — *финализатор*.

В языке C# формально деструкторы есть (точнее есть функции, название и синтаксис которых совпадают с деструкторами C++), однако их поведение аналогично поведению метода `finalize()` в языке Java, поэтому будем также называть их финализаторами.

Деструкторы по смыслу обратны конструкторам, поэтому поведение и порядок вызова деструкторов обратны поведению и порядку вызова конструкторов соответствующих объектов. Так, например, в языке C++ момент вызова деструкторов обратен моменту вызова конструкторов:

1) деструкторы статических объектов вызываются после выхода из функции `main`;

2) деструкторы квазистатических объектов выполняются при выходе из блока;

3) деструкторы динамических объектов выполняются при вызове операции `delete`.

Как уже отмечалось, в C++ существуют также временные объекты. По правилам языка они уничтожаются при выходе из конструкции, в контексте которой создаются. Например, временные объекты, созданные при выполнении вызова функции, уничтожаются (и вызывается соответствующий деструктор) сразу после завершения вызова.

Как и конструкторы, деструкторы имеют пользовательскую (тело объявленного деструктора) и системную (сгенерированный код) части. Системная часть выполняется после тела деструктора. В системную часть входят вызовы деструкторов подобъектов (в порядке, обратном конструированию) и вызовы деструкторов базовых классов.

Если в классе не объявлен явный деструктор, то компилятор генерирует неявный деструктор класса, состоящий только из системной части.

Итак, программист на языке C++ может достаточно точно установить момент вызова деструкторов объектов. На этом основана простая и эффективная техника гарантированного освобождения локально захваченных ресурсов, называемая *«захват ресурса — это инициализация»* (английская аббревиатура — RAII). В соответствии с этой техникой для работы с ресурсом разрабатывается класс, в конструкторе которого происходит захват ресурса. Освобождение ресурса происходит в деструкторе объекта. Для работы с классом в текущем блоке объявляется локальная (в нашей терминологии квазистатическая) переменная, при создании которой происходит захват ресурса. Далее работаем с переменной, а после выхода из блока освобождение происходит автоматически.

Многие классы из стандартной библиотеки поддерживают эту технику, например классы `ifstream` и `ofstream`, представляющие собой внешние файлы ввода и вывода. В конструкторах этих классов происходит открытие файла, в деструкторах — закрытие. Схема задачи по обработке информации из внешнего файла может иметь следующий вид:

```
{// текущий блок
      ifstream in("inputfile"); // открыли ввод
      ofstream out("outfile");// открыли вывод
      // обработали информацию из in
      // и скопировали в out
} // при выходе из блока файлы автоматически
  // закрылись
```

Программист, использующий эту технику, гарантированно освобождает ресурсы. Подробнее эта техника рассматривается в [17].

Конечно, «утечка» ресурсов происходит и в программах на C++, но происходит она, как правило, в объектах из динамической памяти.

В языках C# и Java вместо деструктора предлагается реализовать метод-финализатор. Проблема состоит в том, что финализатор вызывается сборщиком мусора, поэтому предсказать момент вызова финализатора невозможно. Более того, возможна ситуация, когда памяти хватает, и финализатор вообще не будет вызван. Поэтому для классов, захватывающих ресурсы, программист должен реализовать обычные методы, которые освобождают эти ресурсы (например, метод `close()` для закрытия открытых файлов). На программиста-пользователя возлагается ответственность за своевременный вызов таких методов. Кроме того, необходимо предусмотреть вызов «очищающих» методов в финализаторе, если программист-пользователь по каким-то причинам сам их не вызвал. Таким образом, наличия автоматической сборки мусора недостаточно для написания на-

дежных программ, корректно работающих с вычислительными ресурсами.

Копирование объектов. По умолчанию языки предоставляют средства для поверхностного копирования (иногда такое копирование называют *побитовым*).

В языке C++ копирование объектов происходит при выполнении операции присваивания и при инициализации объекта копированием из другого объекта такого же типа. Первая ситуация очевидна, а вторая требует пояснений. При инициализации объекта класса всегда вызывается конструктор. В случае инициализации копированием вызывается специальный *конструктор копирования*, т. е. конструктор, имеющий прототип:

```
X (const X&), либо X (X&)
```

Таким образом, объект, из которого копируются данные, передается в конструктор по ссылке.

Конструктор копирования вызывается в следующих контекстах.

1. Объявление объекта:

```
X a; // работает конструктор умолчания X()
X b = a; // работает конструктор копирования X(a)
```

Заметим, что эта запись полностью эквивалентна следующей записи:

```
X b(a); // работает конструктор копирования X(a)
```

2. Передача параметра функции по значению (конструктор работает при инициализации формального параметра фактическим параметром):

```
void f(X a);
X b;
f(b); /* работает конструктор копирования X(b) для
формального параметра a */
```

3. Возврат значения функции (конструктор работает при инициализации временного объекта возвращаемым значением):

```
X MakeX()
{
     X a;
     return a;
}
X b;
b = MakeX(); /* работает конструктор копирования
X temp(a) для временного объекта, далее операция
присваивания b = temp; */
```

4. При возбуждении исключения типа X (см. гл. 9):

```
X a;
throw a;
```

Каждый класс имеет операцию присваивания и конструктор копирования. Если в классе нет явно определенных операции присваивания и конструктора копирования, то они генерируются компилятором. Сгенерированные компилятором присваивание и конструктор копирования осуществляют поверхностное копирование. Генерация происходит почленно: для членов-данных базисных типов копирование побитовое, для членов-подобъектов вызывается соответствующая операция (конструктор копирования), которая может быть сгенерированной, а может быть и явной.

Программист-разработчик класса сам определяет, какая семантика копирования требуется для класса. Если стандартное поверхностное копирование не работает, то программист должен явно определить в классе операцию присваивания и конструктор копирования.

Заметим, что компилятор требует единого подхода к реализации копирования: операция присваивания и конструктор копирования должны одновременно либо генерироваться, либо реализовываться явно.

Приведем пример простого класса, реализующего динамический вектор (разумеется, это модельный пример, в реальной жизни следует использовать векторы из стандартной библиотеки). Вектор содержит в себе указатель на динамически размещаемый массив и требует глубокого копирования:

```
class Vector {
     int *body;
     int size;
public:
     Vector (int sz)
     {
          body = new int[size = sz];
     }
     Vector (const Vector & v)
     {
          body = new int [size = v.size];
          for (int i = 0; i < size; i++)
               body[i] = v.body[i];
     }
Vector &operator = (const Vector& v)
{
          if (this != &v) {
               delete [] body;
               body = new int [S size = v.size];
               for (int i = 0; i < size; i++)
                    body[i] = v.body[i];
```

```
        }
        return *this;
    }
    ~Vector() { delete [] body; }
    int& operator[] (int index) { return
    body[index]; }
};
```

Обратите внимание на реализацию операции присваивания, которая делает то же, что и конструктор копирования, но предварительно она должна «очистить» объект, при этом проверив, нет ли присваивания в себя.

Попробуйте ответить на вопрос, что будет, если убрать явные операции присваивания и конструктор копирования, т.е. оставить стандартную семантику копирования.

В языках C# и Java операция присваивания применяется к типам значений и ссылкам. Поэтому понятий конструктора копирования и переопределения операции присваивания там нет. Для копирования содержимого объекта следует определить метод класса `Clone()` (в Java — `clone()`), который возвращает ссылку на копию объекта. Реализация метода, конечно, должна учитывать семантику копирования для объекта класса. Для реализации поверхностной семантики язык Java предоставляет вариант метода `clone()`, реализующего простое копирование всех членов-данных, а язык C# — метод `MemberwiseClone()` с аналогичной семантикой.

Преобразование объектов классов. Для реализации явных преобразований объектов классов вполне хватает средств, которые мы уже рассмотрели. Пусть `X` — это новый класс, а `T` — произвольный тип данных (отличный от `X` и необязательно класс). Тогда конструктор с прототипом `X(T)` можно рассматривать как функцию явного преобразования из типа `T` в класс `X`:

```
T t;
X a = new X(t); // Java или C#
X b(t); // C++
b = X(t); // C++
```

Аналогично нестатическая функция-член класса `X` с прототипом

```
T MakeT()
```

может рассматриваться как функция явного преобразования из класса `X` в тип `T`.

Однако приведенные примеры таких преобразований явные, т.е. явно указанные программистом.

При этом проблема состоит в возможности сделать неявными преобразования, определяемые программистом (называемые

пользовательскими), т.е. заставить компилятор вставлять вызовы соответствующих функций, как это происходит, например с преобразованиями арифметических типов в C++.

Язык Java запрещает любые неявные преобразования между объектами классов (исключение составляют только неявные преобразования к стандартному типу `String`, разрешенные в некоторых контекстах).

Языки C++ и C# разрешают неявные преобразования для классов, определяемых пользователем.

В C++ преобразования определяются специальными функциями-членами: *конструкторами преобразования* и *функциями преобразования*.

Конструктор преобразования имеет прототип вида

```
X(T)  (можно еще X(T&) и X(const T&))
```

При наличии такого конструктора допустимо неявное преобразование из `T` в `X`:

```
T t;
X b(t); // это явный вызов конструктора!
b = t; /* а вот это - уже неявное преобразование:
транслятор вставляет такой код: b = X(t); */
void f(X a);
f(t); // еще неявное преобразование: f(X(t));
void g(const X& a);
g(t); // аналогично: g (X(t));
```

Заметим, что в процессе неявного преобразования могут появляться временные объекты.

Функция преобразования имеет вид

```
class X {
      operator T(); };
```

У функции преобразования нет типа возвращаемого значения (оно определяется именем функции) и нет параметра (кроме неявного параметра `this`).

При наличии такой функции допустимы неявные преобразования:

```
X a;
T t;
t = a; // t = a.operator T();
void f(T t);
f(a); // f(a.operator T());
void g(const T& t);
g(a); // g(a.operator T());
```

Неявные преобразования можно вызывать и явно, используя обычный синтаксис:

```
X a, b; T t;
a = (X)t;
t = (T)b;
```

Ключевое слово `explicit`. Неявное преобразование, осуществляемое конструктором (преобразования), может иметь неприятное следствие: иногда конструктор имеет синтаксис конструктора преобразования случайно, а само преобразование семантически бессмысленно с точки зрения типа данных. Например, в рассмотренном ранее классе `Vector` конструктор `Vector(int)` является конструктором преобразования, поэтому следующий пример вполне корректен, хотя особого смысла не имеет:

```
Vector v(20);
v = 1; // v = Vector(1); - скорее всего, ошибка
// старое содержимое v уничтожено,
// v заменен новым вектором длины 1
```

Для подавления неявного вызова конструктора преобразования, если такое действие может привести к ошибке, конструктор преобразования необходимо объявлять с использованием ключевого слова `explicit` (явный):

```
explicit Vector (int sz);
```

Теперь компилятор выдаст ошибку в приведенном примере. Если же ошибки нет, то программист должен выразить свои намерения явно:

```
v = Vector(1);
```

Неявные преобразования в C#. В языке C# область применения пользовательских преобразований у́же, чем в языке C++. Можно определять свои преобразования только между двумя классами, нельзя определять преобразования в типы значений или из них.

Преобразование из класса `X` в класс `Y` реализуется с помощью специального метода — функции преобразования:

```
static operator Y (X x) { ... }
```

Функция преобразования может быть только статическим методом либо класса `X`, либо класса `Y`.

Если такая функция преобразования есть, то она вызывается с использованием обычного синтаксиса преобразований: `(Y)x`.

Компилятор вставляет неявное преобразование из `X` в `Y` только, если соответствующая функция преобразования снабжена модификатором `implicit`:

```
static implicit operator Y (X x) { ... }
```

Если же используется модификатор `explicit`, то функция преобразования может вызываться только явно. По умолчанию принимается модификатор `explicit`, что снижает вероятность случайной ошибки.

7.3. Инкапсуляция. Абстрактные типы данных

Инкапсуляция — это языковой механизм, позволяющий ограничить доступ к отдельным членам класса. Иногда инкапсуляцию называют *упрятыванием*, или *защитой*, информации. Инкапсуляция позволяет скрыть внутреннее представление объекта, обеспечивая его целостность, за счет того, что пользователь не может «испортить» внутренние данные объекта.

Например, недостатком класса `Stack` из подразд. 7.1 является то, что его данные не инкапсулированы. Поэтому можно, например изменить значение переменной `top`, присвоив ей нулевое значение и «опустошив» тем самым весь стек. Также можно непосредственно модифицировать тело стека (массив `body`) и т. п.

Рассмотрим, как реализована инкапсуляция в объектно-ориентированных языках программирования.

Во-первых, в языках с классами возможно ограничение доступа к отдельным членам класса. Единицей инкапсуляции (или единицей защиты) является член класса. При этом правила защиты применяются единообразно ко всем членам, будь то данные, методы, специальные функции-члены, вложенные классы.

Во-вторых, атомом защиты является весь класс целиком, т. е. правила защиты применяются единообразно ко всем экземплярам класса. Нельзя устанавливать отдельные правила доступа к экземпляру класса, отличные от правил доступа к другим экземплярам этого класса.

Инкапсуляция в языке C++. В этом языке все члены класса относятся к одной из следующих *областей доступа*:

- `public` — открытая, доступная любым функциям;
- `protected` — защищенная, доступная только собственным методам и методам производных классов;
- `private` — закрытая, доступная только собственным методам.

Члены класса, находящиеся в закрытой области (`private`), недоступны для использования со стороны внешних функций. Напротив, члены класса, находящиеся в открытой секции (`public`), доступны для использования из любых функций, в том числе и внешних. При объявлении класса каждый член класса помещается в одну из перечисленных выше областей доступа следующим образом:

```
class имя_класса {
    private:
```

```
определение_закрытых_членов_класса
     public:
          определение_открытых_членов_класса
     protected:
          определение_защищенных_членов_класса
...
};
```

Порядок следования областей доступа и их количество в классе произвольны.

Ключевое слово, определяющее первую область доступа, может отсутствовать. Умолчание зависит от того, с какого ключевого слова начинается объявление класса.

Объявление класса может начинаться с ключевого слова `class` (как ранее) или `struct`.

Класс C++ отличается от структуры C++ только определением по умолчанию первой области доступа в их описании (а также определением по умолчанию способа наследования — см. подразд. 8.1):

- для структур умолчанием является открытый доступ (`public`);
- для классов умолчанием является закрытый доступ (`private`).

Различия в умолчаниях связаны с обеспечением совместимости программ на языке C и на языке C++, что позволяют рассматривать «старые» структуры в программах на C как классы C++ без функций-членов и с открытыми данными (т.е. «плохие», но все-таки классы).

Модель управления доступом, основанная на трех уровнях доступа (только для класса, только для иерархии классов, для любых классов и функций), является достаточно простой. Однако эта модель иногда является слишком ограничительной, так как не позволяет различать права доступа для внешних функций и классов. «Чужаки» (т.е. внешние классы и функции) либо все сразу имеют доступ (к открытым членам), либо все сразу не имеют доступа (к закрытым и защищенным членам). На практике выясняется, что иногда некоторые внешние классы или функции нельзя рассматривать как «чужаков».

Для примера рассмотрим класс `String`, инкапсулирующий структуру динамической строки. Все ли операции с этим классом имеет смысл реализовывать через его методы? Например, удобно определить операцию конкатенации (сцепления) двух строк и использовать для нее перегрузку стандартной операции сложения «+».

Есть два варианта перегрузки операции «+». Во-первых, можно перегрузить ее как функцию-член:

```
class String {
public:
     String operator + (const String & S);
     ... // другие члены, в том числе закрытые
};
```

а во-вторых, можно перегрузить ее как внешнюю функцию:

```
String operator + (const String &S1, const String
&S2);
```

Обращение к операции в обоих случаях имеет вид

```
String m1, m2, m3;
m1 = m2 + m3; //
```

Однако компилятор трактует вызов операции «+» по-разному:

```
m1 = m2.operator + (m3); // функция-член
m1 = operator + (m2, m3); // внешняя функция
```

При этом допустима только одна интерпретация, которую должен выбрать программист.

В первом варианте первый параметр сложения играет выделенную роль, а во втором варианте оба параметра равнозначны. Второй вариант лучше отвечает общепринятой семантике сложения, а это важно при перегрузке стандартных операций (если перегруженная операция не отвечает общепринятой семантике, то это верный путь к ошибкам в программе).

Кроме того, второй вариант более универсален, например он позволяет единообразно трактовать неявные преобразования. Поясним это на примере класса `String`.

Предположим, что класс `String` имеет конструктор преобразования из вещественного типа, который переводит число в текстовое представление:

```
String(double d); // слово explicit отсутствует!
```

Такой конструктор аналогичен методу `toString()` в языке Java. Этот метод применим ко всем объектам данных (даже простых типов). Если контекст использования объекта требует строкового значения, то компилятор Java автоматически подставляет вызов метода `toString()` для объекта. Таким образом, можно считать, что метод `toString()` реализует неявное преобразование объекта данных языка Java в строку (и это единственное неявное преобразование, допустимое в Java). Итак, в Java можно записать:

```
String s = "Line ";
String ss = s + 6; // ss получает значение "Line 6";
```

Однако теперь (при наличии конструктора преобразования) такой код верен и для класса `String` в языке C++ (и это достигнуто чисто языковыми средствами). Компилятор подставляет цепочку из неявных преобразований, т.е. одно пользовательское, второе — стандартное:

```
String ss = s + (String)(double)6;
```

Теперь рассмотрим, как перегружена операция сложения для класса `String`. Если она перегружена функцией-членом, то приведенный ранее пример работает корректно:

```
String ss = s.operator+((String)(double)6);
```

а вот перестановка операндов приводит к ошибке:

```
String ss = 6 + s;
```

Если же перегрузить операцию сложения внешней функцией, то преобразование будет работать корректно в обоих случаях:

```
String ss = operator+(s, (String)(double)6);
String ss = operator+((String)(double)6, s);
```

Таким образом, можно привести общую рекомендацию: двуместные «симметричные» операции, в которых оба аргумента равноправны, лучше перегружать как внешние функции, а «асимметричные» операции с выделенным левым операндом (к такому типу относятся, например, комбинированные операции присваивания типа «+=») — функциями-членами.

Однако если выбрать первый вариант, то мешает правило инкапсуляции: внешняя операция не может получить доступ к закрытым переменным. Это пример функции, которая, с одной стороны, должна быть внешней по отношению к классу, а с другой стороны, должна иметь доступ ко всем членам класса. Таким образом, модель языка C++ нуждается в расширении: некоторые внешние функции должны иметь «привилегированный» доступ.

Такое расширение реализовано в C++ посредством *друзей класса*.

Друг класса `X` — это внешняя функция (глобальная либо член другого класса), имеющая такие же права доступа к членам `X`, как и у членов этого класса. Понятие друга декларируется самим классом (друзей не навязывают), понятие друга не является транзитивным (друг моего друга необязательно мой друг), друзья класса не наследуются (если производный класс хочет иметь в друзьях друзей базового класса, то он должен явно их объявить).

Объявление друга должно содержаться внутри объявления класса, объявляющего друга. Объявление друга имеет три модификации:

```
friend прототип_глобальной_функции;
friend прототип_функции_члена_класса;
friend имя_класса;
```

Например:

```
class X
{
   friend void f(X&);
   friend void Y::AccessX(X&);
```

```
    friend class Z; // все функции-члены Z — друзья X
};
class String
{ ...
    friend String operator + (const String &, const
String &);
    ...
};
```

Инкапсуляция в языках C# и Java. В языках C# и Java имеются такие же уровни доступа (и такие же ключевые слова), как и в языке C++: `public`, `protected`, `private`. Кроме того, модель управления доступом дополнена другими уровнями, которые исключают необходимость в друзьях.

Дело в том, что в C# и Java можно набор логически связанных классов объединить в некоторую сущность и использовать эту сущность как целое (например, распространять, импортировать из нее классы и т. п.). Эта сущность играет роль библиотеки (классов). Конечно, оттранслированные классы на C++ тоже можно объединять в библиотеки, но понятие «библиотеки» на языковом уровне нет ни в C, ни в C++. Приходится использовать соответствующее понятие из виртуального компьютера операционной системы.

В языке C# роль библиотеки играет понятие сборки (`assembly`), а в Java — понятие пакета (`package`).

Исходные тексты программ как на C#, так и на Java хранятся в текстовых файлах (расширение .cs и .java соответственно). Напомним, что программы состоят только из определений типов (классов, перечислений, интерфейсов). Принадлежность класса сборке (C#) определяется при трансляции в командной строке, вызывающей компилятор C# (интегрированная среда разработки сама генерирует командную строку из установок проекта). В Java есть специальная конструкция `package`, которая указывает, какому пакету принадлежит класс (классы) из файла программы. Эта конструкция должна быть первой в файле. Форма конструкции очень проста:

```
package имя-пакета;
```

Например:

```
Package ru.soft-company.common.graphics;
```

Поскольку классы из сборок (пакетов) должны быть логически связаны, то они имеют привилегированный доступ друг к другу по сравнению с классами из внешних сборок (пакетов).

В результате появляется еще один уровень доступа (промежуточный между закрытым и открытым). В языке C# этот уровень доступа называется внутренним (и обозначается ключевым словом `internal`). В Java аналогичный уровень называется пакетным, он

является умолчательным и не имеет ключевого слова. Внутренний (пакетный) уровень доступа означает разрешение доступа со стороны всех классов, входящих в сборку (пакет).

Внутренний (пакетный) доступ позволяет обойтись без понятия «друг класса». Полного аналога друга класса в C# и Java нет, поскольку закрытый доступ в этих языках запрещает доступ любым другим классам. Однако если в сборке (пакете) есть классы, которые нуждаются во взаимном доступе (например, функции преобразования в C#), следует обеспечить внутренний (пакетный) доступ к членам классов.

Все классы в сборке (пакете) считаются логически связанными друг с другом (отсюда и особый уровень доступа). Здесь возникает тонкий момент, связанный с понятием наследования. Являются ли производные классы логически связанными с классами из «родительской» сборки (пакета)? Понятно, что производный и базовый классы тесно связаны, поэтому и возникает (во всех языках) защищенный уровень доступа. Однако если классы в сборке логически связаны с базовым классом, то они должны быть связаны и с производными классами, но насколько тесно? Язык Java разрешает всем классам из пакета иметь доступ к защищенным членам классов из этого же пакета (тем самым ослабляя понятие защищенного уровня по сравнению с C++). Понятие защищенного доступа в языке C# аналогично C++, но появляется еще один (пятый) уровень доступа, эквивалентный защищенному уровню Java: внутренний защищенный (`protected internal`), который определяется как «внутренний или защищенный».

Теперь окончательно уточним определения уровней доступа.

В языке C# это следующие уровни:

- `public` — открытый (неограниченный), доступный методам любых классов из любых сборок;
- `internal` — внутренний, доступный методам всех классов из этой же сборки;
- `protected internal` — внутренний, или защищенный, доступный только собственным методам, методам производных классов (из любых сборок) и методам всех классов из этой же сборки;
- `protected` — защищенный, доступный только собственным методам и методам производных классов (из любых сборок);
- `private` — закрытый, доступный только собственным методам.

Примечание. Строго говоря, можно было бы ввести и еще один уровень доступа — внутренний и защищенный, т. е. доступный только собственным методам и методам производных классов из этой же сборки. Такой уровень есть в промежуточном языке платформы .NET и в реализации C++/CLI для этой платформы, но авторы C# не стали включать его в язык.

Уровни доступа в языке Java следующие:

- `public` — открытый (неограниченный), доступный методам любых классов из любых пакетов;
- (не имеет ключевого слова) — пакетный, доступный методам всех классов из этого же пакета;
- `protected` — защищенный, доступный только собственным методам, методам производных классов из любых пакетов и методам всех классов из этого же пакета;
- `private` — закрытый, доступный только собственным методам.

Синтаксически в C# и Java нет областей доступа, поэтому модификатор доступа распространяется только на член, перед которым он стоит. При отсутствии модификатора доступа в C# подразумевается `private` (как для классов, так и для структур), а в Java — пакетный уровень.

В заключение отметим, что в C# и Java есть возможность управлять доступностью классов из сборки (пакета), отсутствующая в C++.

Если перед классом стоит модификатор `public`, то он доступен из любой сборки (пакета). Отсутствие модификатора означает доступность класса только изнутри сборки (пакета).

Абстрактные типы данных

Вспомним, что в современных языках программирования тип данных определяется как пара: множество значений и множество операций.

Если тип данных рассматривать как класс, то множество значений определяется набором членов-данных, а множество операций — набором методов класса.

Абстрактный тип данных (АТД) — это тип, в котором внутренняя структура данных полностью инкапсулирована. Другими словами, с точки зрения пользователя абстрактный тип данных представлен только множеством операций. Класс является абстрактным типом данных, если открытыми членами являются только методы.

Говорят, что совокупность открытых членов класса составляет *интерфейс* класса, поэтому интерфейс АТД представлен только операциями.

Конечно, некоторые методы могут быть закрытыми. Эти методы называют вспомогательными. Главное, чтобы структура класса (члены-данные) была скрыта.

Объектно-ориентированная парадигма подразумевает широкое использование АТД. Некоторые языки, например SmallTalk, вообще запрещают открытые члены-данные, тем самым любой класс является в этом языке абстрактным типом данных.

Языки C++, C#, Java позволяют открывать члены-данные, но это считается «дурным тоном» с точки зрения объектно-ориентирован-

ного стиля. Даже если некоторые операции сводятся к присваиванию и(или) считыванию значения некоторой переменной — члена класса, то и в этом случае предлагается использовать не открытый доступ к члену-данному, а методы класса, называемые *селекторами*. Функции-селекторы — это пара функций, одна из которых (функция `get`) считывает значение члена-данного, а другая (функция `set`) присваивает новое значение. Отсутствие одной из функций селекторов означает запрещение соответствующей операции. Например, класс `Vector` на C++ (см. подразд. 7.2) может содержать `get`-селектор, возвращающий длину вектора, но `set`-селектора в классе нет, поскольку длина вектора не меняется в процессе его жизни:

```
class Vector
{
     public:
          ...
          int GetLength() { return size; }
          ...
};
```

Функции-селекторы подчеркивают дуализм данных и операций, поскольку позволяют абстрагироваться от того, как именно реализована сущность: как данное или как операция. Важно, что функции-селекторы могут реализовываться не только как считывание-запись некоторого данного, но и как полноценные операции, содержащие нетривиальные вычисления. Внутреннее устройство селекторов недоступно и неинтересно пользователю.

Язык C# поддерживает абстракцию функций-селекторов, вводя конструкцию «свойство» (`property`).

Свойство синтаксически выглядит, как член-данное класса. Обращение к свойству неотличимо от обращения к члену-данному (за некоторыми исключениями). Объявление свойства выглядит, как объявление члена-данного, сразу за которым следует объявление `get`- и `set`-селекторов в фигурных скобках. При присваивании свойству значения вызывается `set`-селектор, в его теле присваиваемое значение доступно через идентификатор `value` (в контексте тела селектора `value` — это ключевое слово, а в других контекстах — идентификатор). При считывании свойства вызывается `get`-селектор, который должен вернуть значение свойства.

Приведем простой пример:

```
class PropSample
{
     private int _datum = 0;
     public int Datum // выглядит как член-данное
     {    // но это свойство
          get { return _datum; } // чтение
```

```
            set { _datum = value; } // запись
    }
}
PropSample ps = new PropSample();
ps.Datum = -1; // вызов set-селектора с value=-1
Console.WriteLine(ps.Datum); // вызов get-селектора
```

Часто свойства доступны только для считывания (не содержат `set-селектора`), что позволяет сохранить целостность объекта. Приведем пример класса `Stack` из подразд. 7.1 на языке C# (обратите внимание на свойства):

```
public class Stack
{
      int [] body;
      int top = 0;
      public Stack(int size)
      {
            body = new int[size];
      }
      public int Pop() { return body[--top]; }
      public void Push(int x) { body[top++] = x; }
      public bool Empty
      {
            get { return top ==0;}
      }
      public bool Full
      {
            get {return top == body.Length;}
      }
      public int Length
      {
            get { return body.Length; }
      }
}
```

В заключение отметим, что понятие АТД выходит за рамки объектно-ориентированного подхода и широко используется и в других парадигмах. Еще один подход к реализации понятия АТД с объектно-ориентированной точки зрения, представленный интерфейсами, обсудим в следующей главе.

Глава 8

ОБЪЕКТНО-ОРИЕНТИРОВАННЫЕ МЕХАНИЗМЫ В СОВРЕМЕННЫХ ЯЗЫКАХ ПРОГРАММИРОВАНИЯ

8.1. Наследование

Объектно-ориентированный стиль отличается использованием следующих четырех основных механизмов:

- абстракция;
- инкапсуляция;
- наследование;
- полиморфизм.

Здесь мы обсудим понятие наследования и связанное с ним понятие динамического полиморфизма.

Наследование — отношение между классами, при котором класс-наследник обладает всеми членами родительского класса (или множества родительских классов в случае множественного наследования). При этом класс-наследник может добавлять новые и переопределять унаследованные члены.

В документации по языкам C++ и C# родительский класс называется *базовым*, а класс-наследник — *производным*. Авторы языка Java придерживались терминологии из SmallTalk: родительский класс — *суперкласс*, класс-наследник — *подкласс*. Мы для единообразия будем использовать терминологию из C++, т. е. базовый и производный.

Сосредоточимся на одиночном наследовании (множественное наследование в полном объеме реализовано только в C++).

В объявлении производного класса указывается базовый класс, дополнительные и переопределяемые члены класса (объявление унаследованных членов повторять не надо).

Рассмотрим одиночное наследование в языке C++. Синтаксис объявления производного класса следующий:

```
class имя-производного-класса :
      модификатор-доступа имя-базового-класса
      { объявления-новых-и-переопределенных-членов}
```

Например:

```
class Derived: public Base
{
      int new_imember;
```

```
    public:
        Derived ();
        int new_method();
};
```

Вместо ключевого слова `class` может стоять слово `struct` (как и в определении класса без наследования).

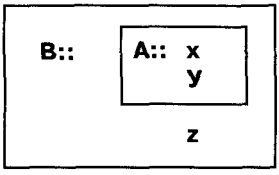
Модификатор доступа может отсутствовать, тогда для `class` подразумевается `private`, а для `struct` — `public`.

Поясним смысл модификатора при имени базового класса. В C++ производный класс может усилить (но не уменьшить!) ограничения на доступ к унаследованным членам. Если модификатор `public`, то права доступа не изменяются (самый частый случай), если модификатор — `protected`, то унаследованные публичные члены становятся защищенными в производном классе. В случае когда модификатор `private`, то все унаследованные члены становятся закрытыми в производном классе. Закрытое наследование полезно, например для реализации интерфейсов.

Рассмотрим следующий пример:

```
struct A {
    int x,y;
};
struct B: A {
    int z;
};
A a1;
B b1;
b1.x = 1;
b1.y = 2;
b1.z = 3;
a1 = b1;
```

Объект производного типа `B` содержит внутри себя члены базового типа `A`:



При наследовании наследуются не только члены-данные, но и методы. Однако существуют исключения. В языке C++ не наследуются:

- конструкторы;
- деструктор;
- операция присваивания.

Из подразд. 7.2 известно, что конструкторы и деструкторы содержат системную часть (генерируемую компилятором), в которой происходит, в частности, вызов конструктора (деструктора) базового класса.

При создании объекта производного типа `B` сначала вызывается конструктор базового типа `A` (в системной части). При этом если конструктору базового типа требуются параметры, то его необходимо вызывать явно в списке инициализации конструктора. Затем вызывается пользовательская часть — тело конструктора производного типа `B`.

Деструкторы вызываются в обратном порядке. При разрушении объекта производного типа сначала вызывается деструктор этого типа, а затем деструктор базового типа.

Совместимость производного типа с базовым. Наследование существенно изменяет взгляд на систему типов языка. В чисто императивных языках типы данных не пересекаются: каждый объект данных принадлежит одному и только одному типу. Это один из главных недостатков императивных языков. В языках с наследованием объекты производного типа принадлежат одновременно и своему, и базовому типу. С точки зрения системы типов базовый тип включает в себя все объекты производных типов. Все операции, применимые к базовому типу, применимы и к производному. Поэтому везде в программе, где могут стоять объекты данных базового типа, на их месте могут появляться объекты производного типа.

Таким образом, существует неявное преобразование из объектов производного типа в объекты базового типа, которое рассматривается чисто формально, ведь никаких действий (при одиночном наследовании) при таком преобразовании не происходит.

Пусть `X` — базовый класс, а `Y` — производный.

Рассмотрим следующий фрагмент:

```
X x; Y y;
x = y; // неявное преобразование из Y в X: x = (X)y;
// объект y рассматривается как объект типа X
// работает операция присваивания из базового
// класса X
void f(X x);
f(y); // ~ f((X)y);
// неявное преобразование из производного типа
// в базовый работает конструктор копирования
// из базового класса X
void g(X& x);
g(y);
```

Заметим, что тип объекта базового класса при присваивании ему объекта производного класса не меняется. Это же касается и формального параметра функции `f` при передаче параметра по значению.

Копируется часть объекта производного типа в объект базового типа, а часть объекта, специфичная для производного класса, просто игнорируется.

Совершенно меняется ситуация при рассмотрении ссылок или указателей на объекты базового и производного классов. Тип ссылки (указателя) на производный класс неявно преобразуется к типу ссылки (указателя) на базовый класс (т. е. к указателям и ссылкам применимо то же правило преобразования, что и к объектам собственно классов). Однако при присваивании указателю адреса объекта производного класса тип объекта, на который ссылается указатель, меняется:

```
X x; Y y;
X *pX = &x;  // типы левой и правой частей
             // присваивания совпадают,
             // pX указывает на объект типа X
pX = &y;     // типы левой и правой частей
             // присваивания не совпадают
             // тип левой части (Y*) неявно
             // преобразуется в X*. Сейчас pX
             //  указывает на объект типа Y
```

То же самое относится и к ссылкам: если инициализировать ссылку на базовый класс объектом производного класса, то тип объекта, обозначаемого ссылкой, — это производный класс:

```
X x; Y y;
X& xx = x; // xx обозначает объект класса X
X& xxx = y; // xxx обозначает объект класса Y
```

Таким образом, указатели и ссылки на объекты классов обладают статическим и динамическим типом. Статический тип указателя (ссылки) — это тип из объявления указателя (ссылки). Динамический тип — это тип объекта, адрес которого в настоящий момент содержится в указателе (ссылке). Динамический тип указателя может меняться при присваивании. Динамический тип ссылки определяется при инициализации ссылки и не меняется в течение жизни ссылки. Динамический тип либо совпадает со статическим, либо является производным (прямо или косвенно через цепочку наследования).

Объекты классов не обладают динамическим типом, их тип либо задается в объявлении, либо указывается при выполнении операции `new` (для динамических объектов).

Понятие динамического типа используется при динамическом полиморфизме.

Классы как области видимости. Добавленные и переопределенные при наследовании члены производного класса образуют отдельную область видимости, вложенную в область видимости базового класса.

Вложенность областей видимости определяет алгоритм поиска, определяющего вхождения имени члена класса. Рассмотрим такой алгоритм для обращения к члену класса извне методов класса, т.е. через объект или указатель:

```
Y a;
a.f();
```

Разберемся, как происходит поиск определяющего вхождения имени при обработке обращения `a.f`. Компилятор по имени объекта находит его тип из объявления `Y a;`. Далее анализируется объявление класса `Y`. Если в классе `Y` нет объявления имени `f`, то поиск продолжается во вложенной области видимости, т.е. в базовом классе и далее по цепочке наследования до тех пор, пока объявление имени `f` не будет найдено либо цепочка наследования не закончится (в этом случае выдается ошибка). Учтите, что при поиске игнорируются права доступа (закрытые объявления также просматриваются) и контекст использования имени (т.е. факт, что `f` вызывается как функция, игнорируется). Права доступа и корректность использования имени анализируются только после того, как найдено объявление имени. При этом поиск заканчивается и не возобновляется даже в случае ошибки (например, имя `f` недоступно или `f` не является функцией).

Рассмотрим следующий пример:

```
class X
{
     public: int a;
};
class Y : public X
{
 public:
     void f();
     void g();
     void g(int i);
private:
     int a;
};
class Z : public Y
{
 public:
     int f,i;
     void g(int i);
};

Z z; Y y;
z.f(); // ошибка: имя Z::f не является функцией
y.i = 0; // ошибка: имени i нет ни в Y, ни в X
```

```
y.f(); // правильно: Y::f();
z.a = -1; // ошибка: имя Y::a недоступно
z.g(); // ошибка: вызов Z::g(int) без параметра
y.g(); // правильно: Y::g();
y.g(0); // правильно: Y::g(int);
z.g(0); // правильно: Z::g(int);
```

Заметим, что если подобная иерархия встречается в реальной программе, то это свидетельствует о плохом проектировании. Проблема здесь в том, что имеет место скрытие имен. Скрытие имени происходит, если во вложенной области видимости имя переопределяется. В нашем примере имеет место сразу несколько скрытий:

- объявление `Y::a` скрывает `X::a`;
- объявление `Z::f` скрывает `Y::f`;
- объявление `Z::g` скрывает `Y::g` (оба объявления).

Скрытие имен провоцирует появление ошибок в программах, поэтому его следует избегать.

Заметим, что в примере есть два определяющих вхождения одного и того же имени (`Y::g`) в одной области видимости (класс `Y`). Такая ситуация называется перегрузкой и, как уже отмечалось, она допустима только для имен функций. Если хотя бы одно из перегруженных имен не является именем функции, то возникает ситуация *конфликта имен*. Заметим, что при описании имен во вложенных областях конфликта имен нет, а есть скрытие одного объявления другим. Также заметим, что объявления перегруженных имен методов должны находиться в одной и той же области видимости, иначе имеет место скрытие, даже если прототипы методов разные (поэтому метод `Z::g(int)` скрывает оба перегруженных метода `Y::g()` и `Y::g(int)`).

Если обращение к имени происходит внутри тела метода класса, то алгоритм поиска определяющего вхождения работает аналогично, но следует иметь в виду, что здесь возникает еще одна вложенная область видимости, а именно блок-тело метода. В теле метода могут объявляться локальные переменные, к которым относятся также и формальные параметры метода. Поэтому поиск начинается с наиболее вложенной области — блока, и только потом он достигает области объявления членов класса. Кроме того, если имя не найдено ни в теле метода, ни в классе (в том числе базовом), тогда поиск продолжается в области видимости, в которой находится объявление класса (например, в глобальной области видимости).

Например:

```
void f();
void g();
class X
{ ...
 protected:
    void f();
```

```
};
class Y : X
{
     int i;
     void h(int k)
     {
          int i = 0; // скрывает член i
          f(); // X::f(); - допустимо
          g(); // глобальная g();
          Y::i = k; // допустимо
     }
};
```

Одной из причин того, что скрытие все-таки допустимо (хотя его следует избегать), является то, что до скрытых имен можно добраться, используя уточнение именем класса. Например, внутри метода `Y::h` скрытый член `i` становится доступен через уточнение: `Y::i`.

В приведенном примере (с иерархией классов `X-Y-Z`) ошибок, связанных со скрытием, можно избежать, используя уточнения:

```
z.Y::f(); // допустимо
z.X::a = -1; // допустимо
z.Y::g(); // допустимо
```

Аналогично к именам из глобальной области видимости можно обратиться, используя синтаксис:

```
::f();
```

Однако к именам локальных переменных в блоке правило уточнения не относится. Так, в следующем примере формальный параметр `n` недоступен внутри вложенного блока:

```
void f(int n)
{ ....
     if (a < b) {
          double n; // плохой стиль!!!
          // нет никакой возможности использовать
          // здесь формальный параметр n
          ...
     }
}
```

Особенности реализации наследования в языках Java и C#

Многое из того, что говорилось о наследовании в языке C++, относится и к языкам Java и C#. Сосредоточимся на их отличиях.

Синтаксис объявления производного класса имеет вид

```
class имя extends имя-базового-класса { // Java
     объявления-новых-и-переопределенных-членов
}
class имя : имя-базового-класса { // C#
     объявления-новых-и-переопределенных-членов
}
```

Модификация доступа при наследовании не разрешена, все унаследованные члены сохраняют свои модификаторы доступа.

Фундаментальное отличие Java и C# от C++ состоит в том, что все классы имеют общего предка — класс `Object`. Если класс объявляется без базового, то по умолчанию он наследуется непосредственно от `Object`.

Класс `Object` играет особую роль в этих языках. Он не обладает членами-данными, зато имеет большой набор методов (в том числе и статических). Ранее уже упоминались методы `toString()`, `clone()` в языке Java, которым соответствуют методы `ToString()` и `MemberwiseClone()` в языке C#. Далее будем учитывать, что по неформальному соглашению имена методов в Java начинаются со строчной буквы, а в C# — с прописной. Программисты не обязаны следовать этому соглашению, но авторы стандартной библиотеки неукоснительно его соблюдают. Поэтому далее иногда не будем уточнять, к объекту какого языка относится метод, поскольку это можно понять по имени.

Наличие единого предка имеет ряд важных следствий.

Во-первых, все объекты располагают единым набором операций, отвечающим, в частности, за копирование, сравнение, финализацию, перевод в текстовое представление. Переопределение этих операций позволяет гибко управлять поведением объектов.

Во-вторых, появляется возможность создания универсальных классов, в частности контейнеров, которые могут взаимодействовать с любыми объектами. Например, если в определении класса `Stack` из гл. 7 изменить тип хранимых объектов с `int` на `Object`, то стек будет в состоянии хранить значения любых объектов. Это возможно, поскольку в соответствии с правилами совместимости производного и базового классов ссылке на базовый класс можно присвоить значение ссылки на любой производный класс. Следовательно, ссылке на `Object` можно присвоить любую другую ссылку.

Правда, что делать потом с этими ссылками? Ведь функция `Pop()` тоже вернет ссылку на `Object`. Пусть у нас есть два класса: `X` и `Y`. Рассмотрим пример работы с классом `Stack`, объекты которого хранят ссылки на `Object`:

```
Stack s = new Stack(32);
s.Push(new X()); s.Push(new Y());
```

```
Y y = s.Pop(); // ошибка — несоответствие типов
X x = s.Pop(); // ошибка — несоответствие типов
```

Ошибка возникает, так как нет неявного преобразования от базового класса к производному. Причем проблема не в том, что такое преобразование не поддерживается, а в том, что такое неявное преобразование небезопасно.

Преобразовать ссылку базового класса `Base` в ссылку на производный класс `Derived` можно только, если динамический тип ссылки `T` ковариантен производному классу `Derived`.

Класс `T1` ***ковариантен*** классу `T2`, если `T1` совпадает с `T2` или является его наследником. Аналогично `T1` ***контравариантен*** классу `T2`, если `T2` ковариантен `T1`. Классы `T1` и `T2` ***инвариантны***, если они не ковариантны и не контравариантны одновременно.

В языках C# и Java есть операция явного преобразования ссылок из класса `X` в класс `Y` (`X` и `Y` различны):

```
(Y)X
```

Эта операция имеет содержательный смысл (т. е. выполняется во время работы программы), если `Y` ковариантен `X`. Если `Y` контравариантен `X`, то по определению `Y` является базовым для `X`, а любая ссылка на производный класс неявно преобразуема в базовый, поэтому компилятор проигнорирует это преобразование. Наконец, если `X` и `Y` инвариантны, то компилятор выдаст сообщение об ошибке.

Важнейшей особенностью операции преобразования ссылок в Java и C# является то, что она контролирует корректность преобразования. В случае если преобразование ссылок невозможно, то генерируется исключение (см. гл. 9). Заметим, что в C++ операция преобразования ничего не контролирует. Есть вариант контролируемой операции, но он применим только к полиморфным классам (см. подразд. 8.2).

По определению операция преобразования ссылки `x` типа `X` в контравариантный тип `Y` корректна, если динамический тип `x` ковариантен `Y`.

Пустая ссылка `null` приводима в любой тип.

Итак, чтобы пример со стеком компилировался, следует вставить операции преобразования:

```
Y y = (Y)s.Pop();
X x = (X)s.Pop();
```

Оба языка содержат большой набор классов-контейнеров (коллекций), способных хранить любые объекты. В Java они находятся в пакете `java.util`, в C# — в пространстве имен `System.Collections`.

Однако вспомним, что наряду с классами в этих языках есть массивы и типы значений. Если бы они не были совместимы с классом

Object, то полезность последнего была бы ограничена. Заметим, что в некоторых объектно-ориентированных языках (например, SmallTalk) любое значение принадлежит некоторому классу (т. е. даже значения простых типов являются экземплярами соответствующих классов). Такой подход упрощает язык, но приводит к неэффективности операций со значениями простых типов. Поэтому в C# и Java принят компромиссный подход: для эффективности выделены типы значений, а для общности (и удобства) введены понятия классов-оболочек и операций преобразования между типами значений и классами-оболочками. Эти операции называются упаковкой и распаковкой (boxing-unboxing).

Классы-оболочки. Упаковка и распаковка

Для каждого простого типа данных в стандартной библиотеке имеется класс-оболочка. Этот класс полезен в двух отношениях. Во-первых, он содержит набор методов, облегчающих работу с данными простых типов. Например, класс оболочка Int32 для типа int в языке C# содержит методы, преобразующие значение в различные виды текстового представления и обратно (Parse, TryParse, ToString).

Во-вторых, компилятор умеет преобразовывать значения простых типов в экземпляры класса-оболочки (эта операция называется упаковкой) и обратно — из классов-оболочек в значения (эта операция называется распаковкой).

В языке C# операция упаковки никак не обозначается и выполняется неявно:

```
int i = 25;
Object obj = i; // произошла упаковка
// Чтобы узнать, в какой объект упаковано значение,
// напечатаем имя типа объекта и текстовое
// представление
Console.WriteLine(obj.GetType().FullName  +  ":"  +
obj.ToString());
```

В этом случае будет напечатано

```
System.Int32:25
```

Распаковка имеет вид явной операции преобразования к типу значения (продолжение предыдущего примера):

```
int j = (int)obj;
Console.WriteLine(j);
```

В этом случае будет напечатано 25.

В языке Java операция упаковки выглядит как создание объекта класса-оболочки, а операция распаковки — как в языке C#, т.е. вид явного приведения к типу оболочки:

```
int i = 25;
Object obj = Integer(i); // произошла упаковка
// Чтобы узнать, в какой объект упаковано значение,
// напечатаем имя типа объекта и текстовое
// представление
System.out.println(obj.getClass().getName() + ":" +
obj.toString());

int j = (Integer)obj;
System.out.println (j);
```

В этом случае будет напечатано

```
java.lang.Integer:25
25
```

Начиная с 2005 г. в Java поддерживаются операции автоупаковки и автораспаковки (в C# они были введены с самой первой версии). Автоупаковка означает, что операция упаковки выполняется автоматически (иначе говоря, то, что в Java назвали автоупаковкой, в C# всегда называлось просто упаковкой):

```
Object obj = 30;  // вставлено: obj = new Integer
//(30);
```

Автораспаковка — это автоматическая распаковка из ссылки, объявленной как класс-оболочка, в значение соответствующего простого типа.

В языке C# эти операции имеют вид

```
Int32 refInt = 32; // (авто)упаковка
int i = refInt; // автораспаковка
```

а в языке Java

```
Integer refInt = 32; // автоупаковка
int i = refInt; // автораспаковка
```

В языке C# классы-оболочки существуют для всех типов значений, в том числе для перечислений (`System.Enum`). Структуры (они ведь тоже типы значений) тоже могут упаковываться и распаковываться в объекты в динамической памяти в контексте, требующем ссылки.

Классы-оболочки совместно с операциями автоупаковки и распаковки позволяют трактовать абсолютно все значения в программе как объекты, но без потери эффективности при выполнении операций с простыми типами данных.

Например, напишем пример работы с объектом класса `Stack` на языке Java:

```
Stack s = new Stack();
int [] x = {1,2,3,4,5,6,7};
for (int k : x) s.Push(k);
while (!s.IsEmpty()) System.out.println(s.Pop());
double [] xx = {1.1,2.2,3.3,4.4,5.5,6.6,7.7};
for (double k : xx) s.Push(k);
while (!s.IsEmpty()) System.out.println(s.Pop());
```

Видно, что стек действительно может хранить значения произвольных типов.

Правда, упаковка и распаковка требуют определенных накладных расходов, однако появление обобщенных типов позволило исправить эту проблему без ущерба для гибкости языка.

Запрещение наследования

В языках C# и Java есть возможность запрета наследования. Если класс в языке C# помечен модификатором `sealed` (в Java для этой цели служит модификатор `final`), то этот класс не может быть базовым ни для какого класса. Например, классы-оболочки (см. подразд. 8.2), структуры, встроенный класс `String` не могут наследоваться. Проектирование полиморфных иерархий — достаточно нетривиальное дело, поэтому некоторые авторы рекомендуют явно закрывать свои классы для наследования, если они не проектируются специально как иерархии (см. подразд. 8.3).

Заметим, что закрытие класса для наследования может в некоторых случаях увеличить эффективность вызова методов (см. подразд. 8.2).

8.2. Динамический полиморфизм

Понятие полиморфизма подразумевает наличие нескольких вариантов реализации одной сущности. Виды полиморфизма различаются по виду сущности и по способу связывания сущности с вариантом реализации.

В рассматриваемых языках (C++, C#, Java) воплощена разновидность статического полиморфизма, называемая перегрузкой функций. При перегрузке имеется сущность — операция. Сущность выражается именем операции (например, операция «+» или метод `println`). Набор вариантов ее реализации — различные варианты одноименных функций в одной области видимости, различающиеся

профилем параметров. Связывание вызова функции с вариантом ее реализации происходит статически (транслятором) при сопоставлении количества и типов фактических и формальных параметров в вызове и реализации функции.

При *динамическом полиморфизме* сущностью является операция или действие, выражаемое профилем метода класса, а вариантами реализации — заместители метода в производных классах.

В C++ и C# введено понятие *виртуальных функций* (*методов*). Механизм виртуальных методов заключается в том, что при вызове виртуального метода через указатель или ссылку связывание вызова метода с конкретной реализацией метода зависит от динамического типа указателя или ссылки и выполняется во время работы программы. Для невиртуальных методов связывание вызова с реализацией зависит от статического типа (поэтому неважно, как вызывается невиртуальный метод: через ссылку или объект) и происходит статически.

Тип данных (класс), содержащий хотя бы одну виртуальную функцию, называется *полиморфным типом* (*классом*), а объект этого типа — *полиморфным объектом*.

В языке Java нет термина «виртуальный метод», но только потому, что все методы в Java являются виртуальными, поэтому любой класс языка Java полиморфен.

В языке C# есть как виртуальные, так и невиртуальные методы, при этом класс `Object` содержит виртуальные методы (например, `ToString()` и `Finalize()`), поэтому обычные классы C# также полиморфны (за исключением статических классов).

В C++ и C# виртуальность функции определяется описателем `virtual` перед объявлением метода. В языке C++ во всех классах-наследниках наследуемая виртуальная функция остается виртуальной. Таким образом, все типы-наследники полиморфного типа являются полиморфными типами.

В C# возможна ситуация, когда наследуемая функция перестает быть виртуальной (или начинает новую цепочку виртуальных методов), однако использовать эту ситуацию настоятельно не рекомендуется.

Механизм виртуального вызова (т.е. вызова виртуального метода через ссылку или указатель) основан на понятии *замещения* (`overriding`) виртуального метода. Пусть есть (базовый) класс `Base`, имеющий виртуальный метод `M` (*замещаемый* метод). *Заместитель* виртуального метода — это метод, определенный в производном классе и имеющий тот же прототип (т.е. тот же список формальных параметров и тот же тип возвращаемого значения), что и метод `M`. Допускается следующее единственное ослабление требования совпадения прототипа: заместитель может иметь тип возвращаемого значения, ковариантный возвращаемому типу замещаемого метода `M`.

В языке C# заместитель должен быть обязательно снабжен описателем `override`. При его отсутствии метод не считается заместителем. Как уже отмечалось, это не является хорошей практикой, поэтому компилятор выдает по поводу «псевдозаместителя» предупреждение. Подавить это предупреждение можно только указанием описателя `new` перед объявлением этого метода.

При вызове виртуального метода `M` через ссылку или указатель на объект класса `Base` определяется динамический тип `T` ссылки или указателя (т.е. тип объекта, адрес которого хранится в ссылке или указателе). Напомним, что тип `T` должен быть ковариантен `Base` (т.е. должен совпадать с `Base` или быть его наследником). Далее вызывается ближайший по цепочке наследования от `T` к `Base` заместитель метода `M`.

Другими словами, если метод `M` замещен в `T`, то вызывается этот заместитель, если же `M` не замещен в `T`, то отыскивается заместитель в родительском классе и т.д. до тех пор, пока заместитель не найдется (в крайнем случае вызывается метод из `Base`).

В рассматриваемых нами языках реализация виртуального вызова достаточно эффективна и не будет перебирать родительские классы, однако в таких языках как SmallTalk или JavaScript вызов виртуального метода может привести к циклу просмотра всей иерархии родительских классов.

Приведем пример вызова виртуального метода на языке C++:

```
#include <iostream>
using namespace std;

class A {
     public:
          virtual void f (int x)
          {
               cout << "A::f" << '\n';
          }
};

class B: public A{
     public:
          void f (int x)
          {
               cout << "B::f" << '\n';
          }
};

int main(){
     A a;
     A* pa;
     B b;
```

```
    B* pb ;
    pb = & b;
    pb -> f (1); // B::f
    pa = pb;
    pa -> f (1); // B::f
    pa = &a;
    pa -> f (1); // A::f
    return 0;
}
```

Аналогичный пример на языке C# будет иметь вид

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

То же самое можно написать и на языке Java, надо только убрать модификаторы `override` и `virtual` (и, конечно, изменить вывод).

Заметим, что несмотря на то, что мы говорим о виртуальных методах, виртуальность — это свойство вызова, которое проявляется

только при вызове через ссылку или указатель. Если в C++ вызвать виртуальный метод через имя объекта, то никакого виртуального вызова не будет (ведь тип собственно объекта известен во время трансляции и меняться не может).

Если вызывать виртуальную функцию класса `X` из функции-члена класса `X` или его производного класса, то это тоже будет виртуальный вызов (ведь он осуществляется через ссылку (указатель) `this`).

Однако виртуальность вызова можно отменить, уточнив имя метода именем конкретного класса.

Например, в фрагменте программы вызова виртуального метода можно статически привязать вызов к функции `A::f(int)`, если поставить уточнение:

```
pc = & c1;
pc -> A::f (1); // A::f
pa = pc;
pa -> A::f (1); // A::f
pa = &a1;
pa -> A::f (1); // A::f
```

«Снятие» виртуальности вызова часто применяется при написании заместителей. Заместители виртуальных методов можно условно разделить на две категории: первые целиком берут на себя ответственность за реализацию сущности, а вторые (каскадная) выполняют только часть работы, вызывая реализацию этого же метода из базового класса (обычно либо в начале, либо в конце вызова). В этом случае следует явно указать, какая функция вызывается, иначе произойдет рекурсивный вызов.

В C# для каскадного вызова можно воспользоваться ключевым словом `base` (например, `base.handler(args)`), а в Java — `super` (например, `super.handler(args)`).

Заметим, что в ряде случаев «умный» компилятор может сам отменить виртуальность вызова, но только если динамический тип объекта ссылки можно предугадать. Например, в конце подразд. 8.1 отмечалось, что в языках C# и Java можно запретить наследование класса («запечатать» класс) с использованием модификаторов `sealed` и `final`. Если вызов виртуального метода происходит через ссылку на запечатанный класс, то очевидно, что динамический тип этой ссылки не может измениться (ведь производных классов уже не может быть), поэтому вызов можно привязать к реализации метода статически.

Также можно запретить и замещение виртуального метода. Если в Java перед объявлением метода `M` в классе `C` стоит модификатор `final` (вспомним, что все методы в Java виртуальные), это означает, что замещение метода в производных классах запрещается. При вызове «запечатанного» метода `M` через ссылку на `C` виртуальность тоже можно снять.

В C# не все методы виртуальные, поэтому модификатор `sealed` может стоять только перед методом-заместителем и должен сопровождаться модификатором `override`. Как и в Java, это означает запрещение замещения метода в производных классах и может использоваться для повышения эффективности вызова метода.

Реализация динамического полиморфизма

Для иллюстрации того, как может быть реализован динамический полиморфизм, рассмотрим реализацию механизма виртуальных функций в случае одиночного наследования в языке C++. Языки C# и Java используют схожие методы реализации.

Для реализации механизма виртуальных функций используется специальный (связанный с полиморфным типом) массив указателей на виртуальные методы класса. Такой массив называется *таблицей виртуальных методов* (ТВМ). В каждый полиморфный объект компилятор неявно помещает указатель на соответствующую ТВМ, хранящую адреса виртуальных методов, который условно обозначается

```
vtbl* pvtbl,
```

Строки ТВМ содержат адреса заместителей виртуальных методов: если метод замещен в классе, то записывается его адрес, а если не замещен, то заимствуется адрес заместителя из таблицы для базового класса.

Рассмотрим пример иерархии из трех классов в следующем фрагменте программы на языке C++:

```
class A {
   public:
        virtual void f() {cout << "A::f ";}
        virtual void u() {cout << "A::f ";}
        void g() { cout << "A::g ";}
};
class B : public A {
   public:
        void f() {cout << "B::f ";}
        void g() { cout << "B::g "; }
        virtual void h() { cout << "B::h "; }
};
class C : public B {
   public:
        void f() { cout << "C::f "; }
        void u() { cout << "C::u "; }
        void h() { cout << "C::h "; }
        virtual void w() {cout << "C::w ";}
```

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

Результат работы приведенного фрагмента программы следующий:

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

Для рассмотренного примера с виртуальными функциями созданные таким образом структуры приведены на рис. 8.1.

В ТВМ типа-наследника имеющиеся адреса одинаковых методов замещаются, а новые — дописываются в конец. Так, ТВМ класса `A` содержит две записи (адрес метода `f()` и `u()`), ТВМ класса `B` — три записи (адреса `f()`, `u()` и `h()`), а ТВМ класса `C` — четыре (адреса `f()`, `u()`, `h()`, `w()`). Важно то, что каждому методу соответствует фиксированный индекс в ТВМ. Метод `f()` всегда имеет индекс `0` (и будет его иметь в любой ТВМ, производной от `A` класса), метод `u()` — индекс 1 и т.д.

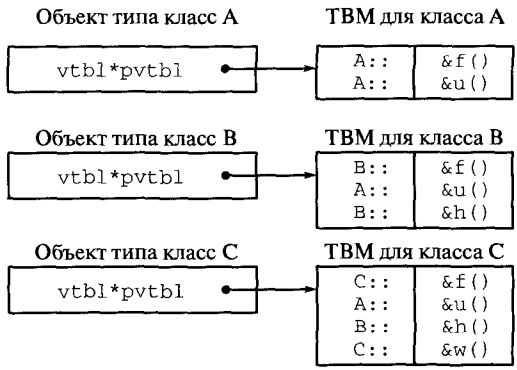


Рис. 8.1. ТВМ для иерархии классов

Так как указатель на ТВМ находится в самом начале объекта, то он доступен всегда, каким бы ни был тип указателя на объект. Конечно, при этом из ТВМ могут быть выбраны только те методы, которые имеются в объявлении класса — статического типа указателя. Слндовательно, как показано в рассматриваемом примере, если объект производного типа обрабатывается через указатель базового типа, то из ТВМ данного объекта можно вызывать только виртуальные методы, перечисленные в базовом типе. Поэтому в процедуре `P()` метод `w()` класса `C` не вызывается (там нет статически объявленных ссылок на `C`).

Обратите внимание, что невиртуальный метод `g()` никак не отражен в структуре таблицы виртуальных методов. Несмотря на то что все реализации этого метода имеют один профиль, они не замещаются, так как замещение имеет место только для виртуальных методов. В нашем случае имеется скрытие имени `A::g` именем `B::g`.

Мы видим, что реализация виртуального вызова не включает в себя никаких циклов и достаточно эффективна, хотя надо иметь в виду, что использование виртуальных методов всегда влечет за собой издержки. К таким издержкам относятся дополнительные команды по извлечению адреса метода при виртуальном вызове, дополнительная память на ТВМ (правда, для каждого класса требуется единственный экземпляр ТВМ) и дополнительная память для указателя на ТВМ в каждом полиморфном объекте (что может оказаться существенным). Поэтому объект полиморфного класса всегда не пуст, даже если и не содержит членов-данных, как в приведенном примере.

Иерархия классов в данном примере имеет по меньшей мере два недостатка. Во-первых, там есть скрытие имени, а во-вторых, там нет виртуального деструктора.

Полиморфные объекты, как правило, должны иметь виртуальный деструктор, даже если деструктор в базовом классе имеет пустое тело. Этот деструктор необходим для резервирования места в ТВМ.

Дело в том, что если указатель типа базового класса ссылается на объект производного класса, то при удалении объекта с использованием данного указателя в случае невиртуальности деструкторов сработает деструктор того типа, который был использован при объявлении указателя (т.е. базового класса). В нашем примере при вызове `delete pa` будет работать сгенерированный деструктор класса `A`. Однако сгенерированные деструкторы не виртуальны. Для приведенного примера это не важно, но представим, что какой-либо производный класс захватывает ресурсы и корректно освобождает их через деструктор, а этот деструктор не может быть вызван через указатель на `A`.

Заметим, что накладными расходами на виртуальность деструктора полиморфного объекта можно пренебречь, поскольку ссылка на ТВМ уже имеется, метод вызывается единственный раз за время жизни,

следовательно, все расходы — это дополнительная строка в ТВМ для каждого производного класса.

Итак, деструктор полиморфного объекта должен быть виртуальным.

8.3. Интерфейсы и абстрактные классы

При проектировании полиморфных иерархий классов часто возникает понятие абстрактного класса.

Абстрактный класс — это класс, который предназначен исключительно для того, чтобы быть базовым классом. Экземпляры абстрактного класса нельзя создавать, но можно (и нужно) использовать ссылки (указатели) на абстрактный класс. Интерфейс абстрактного класса используется для работы с объектами производных классов, на которые указывают ссылки абстрактного типа. Абстрактные классы тесно связаны с полиморфными иерархиями.

В объектно-ориентированных языках абстрактные классы реализуются следующими тремя способами:

- перед классом ставится модификатор `abstract`. Такой способ используется в языках C# и Java;
- класс содержит хотя бы один абстрактный метод. Абстрактный метод — это виртуальный метод без тела, и он должен быть обязательно замещен в одном из классов-наследников. В языках C# и Java абстрактный метод указывается модификатором `abstract` перед объявлением метода (модификатор тогда следует поставить и перед объявлением класса). В C++ абстрактные методы называются чистыми виртуальными функциями;
- если в классе, производном от абстрактного класса с абстрактными методами или интерфейса (интерфейсы рассматриваются далее), не замещен хотя бы один абстрактный метод, то класс тоже является абстрактным. В C# и Java незамещенные абстрактные методы должны быть явно объявлены как абстрактные.

В языке C# абстрактными могут быть и свойства:

```
abstract int Length { get;}
```

Это означает, что соответствующая функция-селектор (в данном случае только `get`-селектор) является абстрактным методом.

Поясним понятие абстрактного метода, поскольку именно наличие абстрактных методов чаще всего приводит к появлению абстрактных классов. В C# и Java возможно объявление абстрактных классов без абстрактных методов, но это используется относительно редко (а в C++ абстрактных классов без абстрактных методов вообще нет).

При проектировании иерархий классов для некоторой проблемной области полезно соблюдать следующее правило: следует выби-

рать базовый класс (вершину иерархии) максимально обобщенным (в рамках проблемной области). Иногда возникают настолько общие классы, что невозможно указать реализацию некоторых их методов. Эти методы имеют смысл для конкретных классов-наследников, но не для абстрактного класса базового в иерархии. Такие методы и являются абстрактными.

Пример проблемной области, хорошо иллюстрирующей понятие абстрактных классов и методов, — экранные графические объекты.

Определим, какими свойствами должен обладать каждый графический объект (для краткости будем называть графический объект фигурой). Все эти свойства необходимо вынести в базовый класс иерархии. Например, к этим свойствам относится точка привязки, которая есть у каждой фигуры. Точку привязки будем обозначать целочисленными координатами. Также общим является поведение фигуры: способность каждой фигуры отрисовывать себя, менять размеры, перемещаться по экрану и т.д. Поведение должно реализовываться следующими методами:

```
void Draw();// отрисовать объект
void Resize(); // изменить размер
void Move (int dx, int dy); // сместиться на (dx, dy)
```

Как можно реализовать эти методы для произвольной фигуры? Понятно, что методы `Draw()` и `Resize()` невозможно сформулировать в терминах произвольной фигуры, поскольку они слишком специфичны (одна реализация для точки, вторая — для отрезка, третья — для окружности и т.д.).

А вот универсальная реализация метода `Move()` в принципе возможна, например, можно изменить координаты точки привязки и послать уведомление об изменениях (чтобы прикладная программа, использующая фигуры, перерисовала содержимое экрана).

Итак, метод `Move()` — пример конкретного метода. Его можно заместить (если приведенная выше реализация не подойдет), но у него есть своя осмысленная реализация, которую можно использовать.

Для методов `Draw()` и `Resize()` невозможно привести универсальную и осмысленную реализацию, поэтому эти методы удобно сделать абстрактными.

При добавлении конкретной фигуры в иерархию необходимо заместить все абстрактные методы класса (т.е. написать конкретную реализацию методов `Draw()` и `Resize()`). Если этого не сделать, то производный класс останется абстрактным.

Невозможно вызвать несуществующую реализацию `Draw()` и `Resize()` для базового класса, поскольку невозможно создать объект абстрактного класса (транслятор контролирует это статически). Таким образом, механизм абстрактных классов хорошо защищен.

Приведем пример иерархии классов фигур (рис. 8.2) на языке Java:

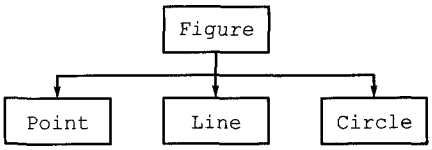


Рис. 8.2. Иерархия классов фигур

```
public abstract class Figure
{
    int x,y;
    ...
    public void Move (int dx, int dy)
    {
        // реализация метода
    }
    public abstract void Draw(); // тело отсутствует
    public abstract void Resize(); // тело отсутствует
    ...
}
class Circle extends Figure {
     void Draw() { /* конкретная реализация */ }
     void Resize(){ /* конкретная реализация */ }
     ...
}
class Point extends Figure {
     void Draw() { /* конкретная реализация */ }
     void Resize(){ /* конкретная реализация */ }
     ...
}
... // другие классы
```

На языке C++ абстрактные классы реализуются как классы с чистыми виртуальными методами. Чистый виртуальный метод (ЧВМ) — это метод с объявлением:

```
void Draw() = 0; // тело отсутствует
```

Компоновщик не требует наличия тела у ЧВМ (для всех остальных виртуальных методов реализация обязательна). Так же, как и для других объектно-ориентированных языков, компилятор запрещает создание экземпляров абстрактных классов.

Интерфейсы

С понятием абстрактного класса тесно связано понятие интерфейса. Интерфейс можно рассматривать как абстрактный класс, доведенный до «абсолюта». Интерфейс состоит только из абстрактных методов. Поскольку в нем нет реализации методов (как нет и не-

виртуальных методов), интерфейс не имеет нестатических членов (статические члены, например константы, допустимы).

Интерфейс представляет собой «чистый» контракт. Производный класс, наследуя интерфейс, «подписывается» под контрактом. Производный класс, наследующий интерфейс и замещающий все его методы, называют *реализацией интерфейса*.

Удобство понятия интерфейса состоит в том, что он не накладывает никаких ограничений на реализацию. Например, абстрактный класс `Figure`, приведенный ранее, этим свойством не обладает и накладывает определенные ограничения на то, как надо реализовывать производный класс (учет точки привязки, например). А вот интерфейс может быть реализован любым классом, главное, чтобы все абстрактные методы замещались реализацией. Поэтому программы, в которых классы взаимодействуют друг с другом только посредством явно объявленных и документированных интерфейсов, являются очень гибкими и открытыми для расширений. Например, современные текстовые процессоры объявляют некоторый набор интерфейсов, который позволяет любой программе, корректно реализующей эти интерфейсы, вставлять в текст документа произвольные объекты, манипулировать ими и визуализировать их.

Рассмотрим, как интерфейсы реализованы в современных языках.

Реализация интерфейсов в C++. Собственно конструкции «интерфейс» в Си++ нет, но его можно смоделировать. Интерфейс на Си++ — это класс, в котором нет никаких нестатических данных, все операции являются публичными чистыми виртуальными функциями. Статические члены могут присутствовать в классе-интерфейсе, поскольку не имеют отношения к экземпляру класса. Существует единственное исключение из правила, что все методы интерфейса абстрактные, а именно деструктор. Мы уже обсуждали необходимость виртуального деструктора в полиморфных объектах. Заметим, что деструктор в C++ не может быть абстрактным (попробуйте объяснить, почему). В классе-интерфейсе деструктор — это виртуальная пустая функция, необходимая только для размещения строки в ТВМ.

Рассмотрим пример класса-интерфейса C++, описывающего «контракт» понятия «множество» объектов типа `T` (предполагаем, что тип `T` доступен в точке объявления интерфейса):

```
class Set{
  public:
       virtual void Incl(T & x) = 0;
       virtual void Excl(T & x) = 0;
       virtual bool IsIn(T & x) = 0;
           // другие абстрактные методы
       virtual ~Set() {} // деструктор
};
```

Для реализации интерфейса следует выбрать структуру данных для внутреннего представления множества. Пусть, например, мы выбрали класс «линейный список» SList для представления множества. В C++ есть, по крайней мере, две возможности реализации интерфейса. Первая — это использование «композиции объектов»: класс-реализация SetImpl содержит объект класса SList как член-данное. Вторая — множественное наследование, допустимое в C++, причем класс-интерфейс наследуется открытым образом, а класс SList — закрытым. Закрытое наследование делает недоступным в классе SetImpl внутреннее представление множества, что необходимо с точки зрения инкапсуляции:

```
class SetSListImpl : public Set, private SList
{
   public:
      void Incl(T & x);
      void Excl(T & x);
      bool IsIn(T & x);
      // объявление заместителей других абстрактных
      // методов
      ~ SetSListImpl ();
};
```

Теперь надо только написать функцию-генератор:

```
Set * MakeSet ()
{
        return new SetSListImpl ();
}
```

Чтобы совсем инкапсулировать реализацию класса, можно объявить конструктор умолчания класса SetSListImpl приватным, а функцию-генератор — другом класса SetSListImpl. В этом случае обращение к MakeSet() — это единственная возможность создать объект-множество. При этом внешние классы вообще ничего не знают о структуре реализации интерфейса. Если реализация по каким-то причинам не подходит, например по причинам эффективности, то можно создать новую реализацию и сменить функцию-генератор:

```
class SetBitScaleImpl : public Set, private
BitScale
{
  private:
  SetBitScaleImpl() {}
// только для запрещения несанкционированного
// создания
friend Set * MakeSet ();
  public:
```

```
        void Incl(T & x);
        void Excl(T & x);
        bool IsIn(T & x);
      // объявление заместителей других абстрактных
      // методов
        ~ SetBitScaleImpl ();
};
Set * MakeSet ()
{
        return new SetBitScaleImpl();
}
```

Исходные тексты классов-клиентов (пользователей интерфейса `Set`) менять не надо (не надо даже перетранслировать), достаточно только скомпоновать их объектный код с объектными модулями, содержащими новую реализацию. Реализация интерфейса полностью инкапсулирована.

Таким образом, понятие интерфейса очень хорошо сочетается с понятием абстрактного типа данных (см. подразд. 7.3).

Отметим, что за счет множественного наследования класс C++ может реализовывать произвольное количество интерфейсов.

Реализация интерфейсов в C# и Java

Эти языки в отличие от C++ явно поддерживают интерфейсы на языковом уровне. Одной из причин этого является то, что в них не реализовано множественное наследование в полном объеме.

C# и Java поддерживают одиночное наследование класса и множественное наследование интерфейсов. Не вдаваясь в детали, отметим, что множественное наследование классов с членами-данными и виртуальными методами создает некоторые проблемы при реализации. Этих проблем нет, если наследуемые множественным образом классы могут иметь только (абстрактные) методы.

Таким образом, в C# и Java есть конструкция «интерфейс».

Интерфейс состоит только из объявлений прототипов методов. Модификаторов доступа нет, так как скрывать надо только реализацию, а методы интерфейса обязаны быть открытыми. Из данных допускаются только статические члены-константы. Допускаются также объявления вложенных интерфейсов (для «композиции» интерфейсов).

Приведем примеры объявлений интерфейсов:

```
interface IEnumerable // стандартный интерфейс C#
{
     IEnumerator GetEnumerator();
}
```

```
interface IEnumerator // стандартный интерфейс C#
{
     object Current { get; }
     bool MoveNext();
     void Reset();
}
interface Iterable // стандартный интерфейс Java
{
     Iterator iterator();
}
interface Iterator // стандартный интерфейс Java
{
     boolean hasNext();
     Object next();
     void remove();
}
```

Приведем также примеры объявлений классов, реализующих интерфейсы:

```
class MyCollection : IEnumerable // C#
{ ....
     public IEnumerator GetEnumerator()
     { .... /* реализация метода */ }
}
class MyCollection implements Iterable // Java
{ ....
     public Iterator iterator()
     { /* реализация метода */ }
}
```

Это примеры интерфейсов (из стандартной библиотеки C# и Java), которые объявляют «контракт» итераторов, позволяющих последовательно перебирать элементы коллекции.

Класс может реализовывать произвольное количество интерфейсов:

```
class Sample extends Base implements I1, I2, I3, I4
{...}
class SampleCS : Base, I1, I2, I3, I4 {...}
```

Объектов-интерфейсов нет, и это понятно почему: ведь интерфейс — обобщение понятия «абстрактный класс», а объекты абстрактного класса создавать нельзя. Однако можно получать ссылку на интерфейс (точнее, на реализацию интерфейса) из ссылки на класс, реализующий этот интерфейс. Ссылки на интерфейс можно использовать для вызова методов интерфейса.

Таким образом, ссылка на интерфейс имеет синтаксис и ведет себя как ссылка на объект базового класса. Этой ссылке можно при-

сваивать ссылку на объект класса, реализующего интерфейс (что похоже на неявное преобразование из производного класса в базовый). Также ее можно явно (и только явно) преобразовывать к ссылке на объект класса, реализующего интерфейс (это похоже на явное преобразование из базового в производный класс).

Приведем следующий пример:

```
void RemoveLongStrings (MyCollection coll, int max-
Len)
  Iterable obj = coll;
  // ссылку на коллекцию поместили в интерфейс
  Iterator i = obj.iterator();
  while (i.hasNext())
  {
        String s = (String)i.next();
        if (s.length() > maxLen)
             i.remove();
  }
}
```

Эта функция удаляет из коллекции `coll` все строки длиной больше заданного значения. Если коллекция содержит нестроковые объекты, то преобразование `(String)i.next()` сгенерирует исключение (см. гл. 9).

Интеграция интерфейсов в язык программирования

Как уже отмечалось, интерфейсы — мощное средство интеграции программ, настолько мощное, что его можно использовать и для интеграции механизмов языка и пользовательских классов. Языки C# и Java поддерживают ряд стандартных интерфейсов, позволяющих интегрировать семантику языковых конструкций и классы, реализующие эти интерфейсы.

Например, приведенные ранее интерфейсы итераторов интегрируют классы-коллекции, реализующие эти интерфейсы, с циклом `foreach`:

```
MyCollection coll;
...
int sum = 0;
for (int i : coll)
     sum += i;
```

Компилятор языка Java вставит обращения к интерфейсам `Iterable` и `Iterator`, а также распаковку из `Object` в `int`.

Другим примером является интерфейс `IDisposable`, используемый в `using`-блоке языка C# (см. гл. 9).

Язык Java использует понятие пустых интерфейсов (интерфейсов-маркеров), которые специально предназначены для интеграции с транслятором. Интерфейсы-маркеры не содержат никаких методов, их смысл зафиксирован в документации и воплощен транслятором. Например, если класс реализует интерфейс-маркер Cloneable, то это означает, что класс будет реализовывать открытый метод `clone()` создания своей копии:

```
class CloneAttack implements Cloneable
{
     public Object clone()
     {
          // возвращает свою копию
     }
}
```

Следует отметить, что любой класс уже содержит версию метода `clone()`, унаследованную от класса `Object`. Этот метод возвращает поверхностную копию объекта. Однако проблема состоит в том, что этот метод — защищенный, следовательно, может использоваться только из производных классов или из классов своего пакета. Чтобы позволить копировать себя внешним классам, используется интерфейс `Cloneable`. Более подробно подходы к копированию объектов в Java рассматриваются в [2].

Глава 9

БЕЗОПАСНОСТЬ И ОТКАЗОУСТОЙЧИВОСТЬ ПРОГРАММ

9.1. Надежность программ. Подходы к обеспечению отказоустойчивости программ

Интуитивно надежность программы — это свойство программы работать без отказов. Мера надежности программы — вероятность безотказной работы в заданном окружении в течение заданного промежутка времени.

Причина отказов программы — ошибки. Можно условно выделить два основных подхода к проектированию «безошибочных» (надежных) систем: математический и инженерный.

При математическом подходе программа рассматривается как преобразователь предикатов. Есть предусловие — предикат, описывающий входные данные и окружение, и постусловие — предикат, описывающий требуемый результат работы программы. Если корректно построить программу как преобразователь предусловия в постусловие, то можно математически доказать ее правильность.

Математический подход весьма сложен и не до конца разработан. Сложность построения предикатов сопоставима со сложностью разработки соответствующей программы (а нередко и превосходит ее). Кроме того, встает вопрос о том, правильно ли сформулированы предикаты и о правильности собственно доказательства правильности. Современная индустрия программирования не использует доказательства правильности программ, хотя при разработке ряда особо критичных систем используются элементы математического подхода.

Инженерный подход рассматривает разработку надежных программ как искусство построения надежной системы из потенциально ненадежных компонент.

Надежная система должна быть готовой к появлению ошибок и реакции на них. В случае некритичных ошибок система должна восстанавливать свое состояние и продолжать свое **нормальное** функционирование, а в случае серьезных сбоев система может деградировать, но обязана продолжать работу в режиме ограниченной функциональности.

Различают два основных источника ошибок (точнее, сбоев или отказов):

- не выявленные до текущего времени ошибки в программе;

- проблемы внешнего окружения (сбои оборудования, отказы связи и т. п.).

В любом случае программа должна адекватно реагировать на любые отказы (насколько это возможно). Отказоустойчивые программы обязаны проверять результат выполнения каждого потенциально сбойного участка программы. Например, системные вызовы (обращения к сервису виртуального компьютера ОС) всегда возвращают некоторое значение, которое сигнализирует об ошибке. Также должны быть устроены и другие компоненты надежного кода.

Основная проблема состоит в том, как именно реагировать на обнаружение ошибки. В подразд. 9.3 будут рассмотрены два основных подхода к обработке ошибок: семантика продолжения и семантика завершения.

В современных языках для обработки ошибок введено понятие исключительной ситуации. *Исключительная ситуация* (или *исключение*) ассоциируется с ошибкой. Важно подчеркнуть, что использование исключительных ситуаций для других целей влечет за собой и резкое снижение как эффективности программы, так и ее отказоустойчивости. Следует придерживаться следующего принципа: исключительная ситуация — всегда «авария».

Далее мы рассмотрим следующие аспекты обработки исключительных ситуаций (ИС):

- определение ИС;
- возникновение ИС;
- распространение ИС;
- реакция на ИС.

Классическая техника обработки ошибок в императивном языке подразумевает, что при каждом аварийном происшествии программа приобретает примерно следующий вид:

```
if (проверка-возникновения-ИС) {
     нормальный-ход-выполнения
} else { // ошибка
     пытаемся-исправить
     if (не-получается)
          return error_code;
}
return SUCCESS;
```

В такой программе нормальный ход выполнения и собственно код, который обрабатывает ошибки, переплетены между собой, что весьма неудобно.

Механизм обработки ситуаций, чтобы быть управляемым и контролируемым, должен поддерживать принцип разделяемости (отдельно код для нормальной ситуации и отдельно — для исправления ошибок). Причем часто реакция на ошибку находится совсем в другом месте относительно места возникновения ошибки. Далее

рассмотрим средства обработки ИС в языках программирования C++, Java и C#. Причем отметим, что базовые механизмы обработки ИС в этих языках практически идентичны.

9.2. Определение исключительной ситуации

Идея определения ИС состоит в том, что ошибке (исключительной ситуации) сопоставляется тип данных. В языке C++ разрешается связать с исключительной ситуацией любой тип данных (как базисный, так и определенный пользователем класс). Информация о произошедшей «аварии» записывается в экземпляр типа данных. Простейший случай — целочисленный тип (ошибке сопоставляется целое значение — код ошибки) или строка (`const char*` — ошибке сопоставляется текст сообщения).

В реальных программах информации, содержащейся в значениях базисных типов, недостаточно для адекватной реакции на ошибку. В современных программах рекомендуется использовать для определения ИС специальный класс `exception` из стандартной библиотеки C++. Отметим, что современный компилятор вставляет код по генерации исключений этого класса, так что полезно «собственные» ИС делать производными от `exception` классами, что позволяет единообразно обрабатывать «собственные» и стандартные ИС.

Языки Java и C# используют для определения ИС специальные классы. В Java тип любой ИС должен быть производным от класса `Throwable`, а в C# — от `Exception`.

Все классы ИС делятся на два вида: пользовательские и системные. Возбуждение пользовательских ИС происходит в коде пользователя, а системных ИС — в коде, генерируемом компилятором. Пользовательские ИС рекомендуется делать производными от специальных классов: в C++ —от `exception`, в C# — от `ApplicationException`, в Java — от `Exception` (подробнее см. подразд. 9.3).

Возбуждение ИС происходит с помощью специального *оператора возбуждения исключения* `throw`:

```
C++ : throw выражение;
C#, Java : throw new X();
```

Здесь `выражение` — произвольное выражение C++, а `X` — допустимый класс ИС. Значение, которое вырабатывается оператором возбуждения ИС, рассматривается как объект, называемый *текущим исключением*.

Система времени выполнения содержит специальную область памяти для хранения текущего исключения. В C++ текущее исключение копируется в выделенную область памяти (при копировании объекта класса будет работать конструктор копирования). В других языках операция `new` сразу размещает объект в выделенной памяти.

Как только возникает ИС, начинается процесс ее распространения.

9.3. Распространение и обработка исключительных ситуаций

Исключения распространяются по *принципу динамической ловушки*. Этот принцип заключается в том, что выбирается реакция на ИС (ловушка), ближайшая по динамической цепочке вызовов. Вспомним, что единицей области действия переменных является блок. В соответствии с принципом динамической ловушки блок, в котором возникло исключение, является аварийным и завершается (аварийно). Все локальные переменные, объявленные в аварийном блоке, уничтожаются. Процесс уничтожения переменных в аварийном блоке называется *сверткой стека*. В C++ во время свертки стека для локальных переменных—объектов работают деструкторы. Если в этот момент в деструкторе возникнет новое исключение, то места для него уже нет (оно занято текущим исключением). Программа в этом случае завершается аварийно.

Если аварийный блок содержит ловушки (обработчики), то происходит поиск подходящей ловушки. Если ловушка не найдена (или ловушек вообще не было), то блок, из которого вызван аварийный блок, становится в свою очередь аварийным (получается, что вызов этого блока стал эквивалентен выполнению оператора возбуждения ИС) и процесс распространения повторяется уже для этого блока. Так происходит до тех пор, пока либо найдется ловушка, либо произойдет выход из последнего блока (в `main`). В последнем случае программа завершается аварийно с выдачей соответствующей диагностики.

Рассмотрим объявление ловушек и процесс поиска ловушек.

В некоторых языках ловушки можно размещать в любых блоках, но в рассматриваемых языках для размещения ловушек служит специальный блок, называемый *блоком с ловушками*, или *try-блоком*. Вид блока с ловушками следующий:

```
try
блок
список-ловушек
```

Список ловушек упорядочен. Каждая ловушка в списке имеет вид

```
catch (тип имя) блок
```

или

```
catch (тип) блок
```

или

```
финальная-ловушка
```

Поиск ловушек при выходе из аварийного блока происходит следующим образом. Пусть тип текущей ИС — `T`. Тогда ловушки перебираются по порядку и тип из ловушки сопоставляется с типом `T`. Единственное преобразование, которое можно применять, — преобразование из производного класса в базовый. Таким образом, ловушка базового класса подходит исключению любого производного класса. Ловушка класса `Throwable` в Java или `Exception` в C# гарантированно «ловит» любую ИС.

В Java и C# правила отождествления крайне просты (поскольку речь идет о сопоставлении ссылок на классы из одной иерархии), в C++ эти правила посложнее (учитывая, что тип ИС из ловушки может быть ссылочным, указательным, значением и т. п.), но базовый принцип поиска прост — отыскиваются типы, которым принадлежит тип текущего исключения (вспомним, что все объекты производного типа одновременно принадлежат базовому).

Последней может быть финальная ловушка. Она подходит для любого типа.

Финальная ловушка в C++ имеет вид

```
catch (...) блок
```

Финальная ловушка есть и в C# (хотя без нее можно было обойтись):

```
catch блок
```

Обработка исключения

Как только ловушка найдена, начинается обработка ИС, т. е. выполнение блока ловушки. Если у параметра ловушки есть имя, то текущее исключение ассоциируется с параметром ловушки (по правилам передачи параметров подпрограмм). Если имя опущено, то это означает, что информация об ИС игнорируется в блоке ловушки.

Если блок ловушки завершился нормально, то ИС считается обработанной (т. е. исправленной), аварийное состояние завершается и объект текущего исключения уничтожается (в C++ сработает деструктор).

Вопросы, как именно следует располагать ловушки и какой код они должны содержать, весьма и весьма сложные и выходят за рамки данного курса [17, 23].

Ловушки могут нормально завершаться как через обычный выход из блока ловушки, так и оператором возврата. При обычном выходе из блока ловушки выполнение продолжается с первого оператора, следующего за `try`-блоком с ловушками. Оператор возврата в блоке ловушки рассматривается как возврат из функции, внутри которой находится `try`-блок с ловушками.

Возможна, однако, ситуация, когда и блок ловушки завершается ненормально, и процесс распространения исключения продолжается.

Это может быть в двух следующих случаях:

- возбуждение новой ИС (переупаковка);
- частичная обработка.

При возбуждении новой ИС внутри блока ловушки располагается оператор возбуждения ИС с новым объектом — текущим исключением. Старый объект уничтожается и замещается новым объектом. Распространение ИС продолжается «вверх» по цепочке вызовов с новым текущим исключением. Такая ситуация возникает довольно часто, когда «авария» не исправляется, но необходимо изменить информацию об исключении, например перевести на другой язык, изложить сообщение об ошибке в других терминах. Часто необходимо «переупаковать» ИС из «чужого» класса в класс ИС своего приложения и т.д.

При частичной обработке исключение также считается необработанным, но текущий объект остается прежним. Частичная обработка завершается выполнением укороченной формы оператора возбуждения ИС:

```
throw;
```

После выполнения этого оператора распространение ИС продолжается «вверх» по цепочке вызовов со старым текущим исключением.

9.4. Особенности реализации исключений в языках C++, C# и Java

Язык C++ не требует обязательной обработки ИС. Так, некоторые программы или хотя бы некоторые функции не требуют защиты. Особенно это касается научно-исследовательских программ, для которых эффективность важнее надежности. Не использовать механизм обработки ИС легко — надо просто не пользоваться `try`-блоками. В этом случае возникновение ИС (например, стандартной) немедленно завершает программу аварийным образом.

В C# и Java любая программа поддерживает обработку ИС. В случае возбуждения необработанной ИС программа также завершается аварийно, но печатается трассировочная информация по всей динамической цепочке вызовов.

Спецификация ИС

В C++ и Java существует концепция спецификации ИС.

Спецификация ИС обусловлена очевидным фактом: для того чтобы адекватно реагировать на ошибку, следует понимать, какие ошибки могут возникать при вызове какой-либо функции. Как уже отмечалось, вызов функции (т.е. выполнение ее тела-блока) может привести к возникновению (точнее, распространению) необработанной ИС. С точки зрения вызывающего блока совершенно все равно, то ли ИС класса `X` возникла внутри вызова функции `f()` и распространилась за пределы ее тела, то ли вместо вызова `f()` стоит оператор `throw X();`. Эффект в обоих случаях одинаковый.

Язык C++ позволяет описать, какие именно ИС могут возникать при вызове функции. Это делается с помощью спецификации ИС:

```
прототип-функции спецификация-ИС тело-функции
```

Спецификация ИС имеет вид

```
throw(список-имен-типов-ИС)
```

Спецификация ИС объявляет, что при вызове функции могут возникнуть только ИС из списка. Это не значит, что внутри функции не могут возникать другие ИС (это возможно, просто функция объявляет, что все ИС не из списка либо не возникнут внутри нее никогда, либо будут в ней корректно обработаны). А вот исключения из списка не только могут возникать, но и не будут обрабатываться этой функцией.

Пустой список ИС говорит о том, что функция декларирует, что она не пропустит ни одной ИС. Вызов такой функции максимально безопасен.

Например:

```
void f() throw (X) { ... }
int MyClass::getLength() throw (MemoryErr, FileErr)
{ ... }
void g(int k) throw() {}
```

Если программист вызывает функцию со списком исключений, то у него возникают две альтернативы по каждой ИС из списка. Во-первых, он может обработать эту ИС (как говорят, подавить ИС), во-вторых, он может добавить неподавленную ИС в свой список ИС (например, используя функцию `f` из предыдущего примера):

```
void suppress() throw ()
{
    try
    {
        f(); // здесь может возникнуть ИС типа X
    }
    catch (X ex)
    {
        // успешная обработка ИС типа X
```

```
    }
}
void pass () throw (X)
{
    f(); // здесь может возникнуть ИС типа X
}
```

Правда, возможен еще один (недостойный) вариант — проигнорировать ситуацию:

```
void ignore ()
{
    f(); // здесь может возникнуть ИС типа X
}
```

Компилятор C++ не считает это ошибкой, поскольку спецификация ИС необязательна.

Зачем специфицировать ИС, если делать это необязательно? Можно задокументировать этот факт без всяких спецификаций.

Дело в том, что если при вызове функции со списком ИС все-таки произойдет неспецифицированное исключение, то программа будет немедленно аварийно завершена с выдачей адекватной диагностики. Действительно, ожидать, что неспецифицированное исключение будет обработано, не следует (ведь никто и не подозревал, что оно произойдет), а нарушение спецификации говорит о серьезной ошибке в функции, так что лучше завершить программу сразу, чем продолжать распространение ИС, потеряв информацию о месте возникновения реальной проблемы.

Язык Java тоже содержит конструкцию спецификации ИС, только эта конструкция обязательна.

Это означает, что функция `ignore` из предыдущего примера не будет успешно откомпилирована:

```
class X extends Exception { ... }
class Sample
{
    void f throws X { .... throw new X(); .... }
    void suppress (){
        // нет списка – значит ничего не выбрасывает
        try
        {
          f(); // здесь может возникнуть ИС типа X
        }
        catch (X ex)
        {
          // успешная обработка ИС типа X
    }
        void ignore ()
```

```
        {
            f(); // ошибка: необработанное исключение X
        }
}
```

Наличие обязательной спецификации ИС повышает надежность программ на Java, заставляя программиста тщательнее продумывать вопросы обработки ошибок. Однако с языковой точки зрения обязательность ИС породила некоторую проблему.

Проблема возникает вследствие того, что существуют ИС, появление которых возможно всюду. Это относится к стандартным ИС, а точнее, к ИС, которые возбуждаются исполняющей системой виртуальной Java-машины (JRTE).

Исключения, порождаемые JRTE, относятся к одному из двух классов (точнее, являются производными от этих классов): `Error` и `RuntimeException`. Все ИС, порождаемые JRTE, не надо специфицировать. Рассмотрим подробнее иерархию классов ИС.

```
class Error extends Throwable {...} — ошибки функ-
ционирования Java-машины и JRTE;
class Exception extends Throwable {...} — базовый
класс для пользовательских ИС
class RuntimeException extends Exception {...} —
базовый класс для ошибок типа выхода за границу ин-
декса массива, нехватки памяти и так далее.
```

Очевидно, что ошибки типа `Error` могут непредсказуемо возникать везде, и обрабатывать их не надо. Как можно исправить на Java ошибки работы JVM?

Ошибки типа `RuntimeException` могут появляться тоже почти везде (например, везде, где порождаются новые объекты, и т.д.). Требование повсеместной проверки таких ошибок невозможно. Однако эти ошибки отличаются от первой категории (и потому имеют общий базовый класс с пользовательскими ИС) тем, что программист может по своему усмотрению ставить ловушки на отдельные ИС из этого класса. При этом компилятор не будет требовать обязательной спецификации или реакции на эти ИС.

Таким образом, обязательно специфицировать или обрабатывать (подавлять) только ИС, являющиеся наследниками класса `Exception`, но не `RuntimeException`. Именно к этому виду должны относиться пользовательские ИС.

Конечно, программист может «обмануть» транслятор (точнее, самого себя), выведя свои ИС из `Error` или `RuntimeException`. Авторы языка Java указывают: «Расширение типов `RuntimeException` или `Error` и использование их объектов в методах конкретных классов свидетельствует о неверном понимании механизмов обработки исключений и затрудняет практическое применение методов про-

граммистами, которые вправе надеяться на наличие исчерпывающего описания набора возможных исключений в составе предложений `throws`» [2].

Конструкция `try-finally`

Как уже отмечалось, в процессе свертки стека в языке C++ гарантируется выполнение деструкторов всех локальных объектов. Если программист грамотно использует методики типа RAII (см. подразд. 7.2), то он может быть уверен, что локально захваченные ресурсы будут освобождены.

Однако в языках с автоматической сборкой мусора уничтожение объектов (и, как следствие, выполнение финализаторов) носит недетерминированный характер. Существуют классы, объекты которых необходимо уничтожать (освобождая захваченные ими ресурсы) по запросу (например, классы для работы с файлами и т. п.). Работа с такими объектами проходит по следующей схеме:

```
создание объекта;
работа с объектом;
уничтожение объекта;
```

Однако если в процессе работы с объектом произойдет выброс необработанной ИС, то объект уничтожен не будет, и произойдет утечка ресурса.

Необходимо иметь механизм гарантированного исполнения некоторого кода в блоке независимо от того, завершился блок нормально или аварийно.

Такой механизм уже есть в C++ (деструкторы локальных объектов), а в C# и Java эту роль играет конструкция `try-finally`:

```
try {
     блок кода
}
finally {
     // этот код выполнится сразу после любого
     // выхода из try-блока
}
```

Блок `finally` может появляться также после всех ловушек try-блока, если они есть. Тогда работа с явно уничтожаемым объектом может выполняться по следующей схеме:

```
создание-объекта;
try
{
     работа-с-объектом;
```

```
}
ловушки
finally
{
     уничтожение-объекта;
}
```

В языке C# есть специальный интерфейс для работы с явно уничтожаемыми объектами:

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

Явно уничтожаемые объекты должны реализовать этот интерфейс. Процедура `Dispose()` должна освобождать ресурсы, занятые объектом (кроме памяти, разумеется, поскольку это забота сборщика мусора).

Примером класса, реализующего этот интерфейс, является стандартный класс `Image` из библиотеки .NET, инкапсулирующий работу с растровыми изображениями. Схема работы с объектами типа `Image` следующая:

```
Image img = Image.FromFile("picture.jpg");
try
{
     работа-с-картинкой-в-img
}
finally
{
     ((IDisposable)img).Dispose(); // освободили
     // картинку
}
```

Интерфейс `IDisposable` является примером интерфейса для языковой интеграции. Этот интерфейс интегрирован с `using`-блоком языка C#. Оператор `using`-блок позволяет автоматически для заданной переменной (типа ссылки на объект, реализующий `IDisposable`) гарантированно выполнить процедуру `Dispose()`. В примере `using`-блока выполняется та же работа, что и в предыдущем примере:

```
using (Image img = Image.FromFile("picture.jpg"))
{
     работа-с-картинкой-в-img
}
// img гарантированно освобожден
```

Глава 10

СТАТИЧЕСКАЯ ПАРАМЕТРИЗАЦИЯ. ПОНЯТИЕ ОБ ОБОБЩЕННОМ ПРОГРАММИРОВАНИИ

10.1. Параметрический полиморфизм

Современные объектно-ориентированные языки поддерживают три вида полиморфизма: статический, динамический и параметрический.

Статический полиморфизм в форме перегрузки подпрограмм и динамический полиморфизм в форме динамического связывания виртуальных методов поддерживаются всеми объектно-ориентированными языками. Параметрический полиморфизм в форме механизма шаблонов реализован в языке C++ с 1990-х гг., а в языках C# и Java — относительно недавно (2003—2005 гг.).

Первые две формы полиморфизма мы рассмотрели ранее (см. гл. 6, 7 и 8). Отличительной чертой этих видов полиморфизма является то, что связывание сводится к выбору одного варианта из некоторого конечного набора. При статическом полиморфизме по профилю вызова подпрограммы статически выбирается вариант ее перегрузки, а при динамическом — в момент вызова виртуального метода по ссылке (указателю) динамически выбирается вариант замещения этого метода.

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

При параметрическом полиморфизме полиморфными сущностями являются параметризованные типы (классы) и подпрограммы (методы). В C++ параметризованные сущности называются *шаблонами* классов и функций, а в C# и Java — *обобщенными* (или *настраиваемыми*) классами и методами, или *обобщениями*.

Два основных понятия параметрического полиморфизма — это объявление параметризованной абстракции (шаблона или обобщения) и конкретизация этой абстракции (т.е. ее использование).

Связывание конкретизации и объявления всегда происходит статически (во время трансляции).

В C++ при конкретизации шаблона происходит создание новых непараметризованных сущностей — классов и функций (конечно, повторного создания одних и тех же сущностей не происходит, транслятор за этим следит). Во время конкретизации происходит статиче-

ский контроль корректности (соответствия типов) и генерация кода. Конкретизация происходит статически во время трансляции.

В C# при конкретизации обобщения проверяется корректность (соответствие типов). Генерация машинного кода для создаваемых непараметризованных сущностей происходит при запуске сборки (JIT-компилятором среды .NET) на основе промежуточного кода, созданного при трансляции объявления обобщения, и информации о типах — аргументах конкретизации.

И, наконец, в языке Java при конкретизации обобщения только проверяется корректность конкретизации (соответствие типов). Можно рассматривать конкретизацию как процесс создания новых непараметризованных сущностей (классов и методов), однако вся информация об этих сущностях существует только во время трансляции и *стирается* при генерации объектного кода для JVM. Фактически во всех конкретизациях используется одна версия обобщенного класса или метода, код которого генерируется в контексте объявления.

Каждый из этих подходов обладает своими достоинствами и недостатками. Механизм шаблонов C++ позволяет создавать очень мощные и одновременно эффективные абстракции. Выразительные средства языка C++ не имеют аналогов среди современных индустриальных языков. На основе статической параметризации разработаны библиотеки шаблонов STL (часть стандартной библиотеки C++), Loki и Boost, значительная часть которой вошла в последний стандарт C++.

Правда, платой за мощность и выразительность при этом стала повышенная сложность шаблонов C++.

Обобщения в языке C# позволяют создавать мощные и достаточно эффективные параметризованные библиотеки, распространяемые в виде сборок.

Обобщения в языке Java также позволяют создавать удобные параметризованные классы, однако выигрыша в эффективности по сравнению с непараметризованным представлением в этом случае нет. Зато средства статической параметризации языка Java обеспечивают полную совместимость с уже разработанными пакетами классов. Это очень полезное свойство, учитывая, что обобщения появились в языке только в 2005 г. после 10 лет активного программирования на Java, в процессе которого были созданы тысячи программ и библиотек пакетов.

Несмотря на существенные различия, механизмы параметрического полиморфизма во всех языках позволяют обеспечить повышенную надежность создаваемых абстракций за счет статического контроля соответствия типов.

Наиболее естественное и популярное применение параметрического полиморфизма — это создание параметризованных классов коллекций, способных хранить объекты одного типа.

Далее рассмотрим основные понятия механизмов статической параметризации в языках C++, C# и Java. Более подробно эти средства для C++ рассматриваются в [1, 9, 28], для C# — в [23], а для Java — в [35].

10.2. Шаблоны языка C++

Для описания шаблонов используется ключевое слово `template`, после которого указываются формальные параметры шаблона, заключенные в угловые скобки. Формальные параметры шаблона (или просто параметры) перечисляются через запятую. Формальный параметр шаблона может быть именем типа (вид параметра `class T` или `typename T`) или именем переменной (вид параметра `тип X`):

```
template <список параметров> объявление_функции_
или_класса
```

Имя класса (функции) в объявлении шаблона класса (функции) называется именем шаблона (не путайте с идентификатором шаблона, рассматриваемым далее).

Хорошим кандидатом на статическую параметризацию является класс `Vector`, рассмотренный ранее. Очевидно, что его следует параметризовать типом хранимого элемента, тогда получится универсальный массив для хранения данных практически любого типа. Менее очевидным является второй параметр шаблона — размер вектора. Если сделать его статическим параметром, то тело вектора не надо располагать в динамической памяти, не надо хранить длину вектора, а вектор в этом случае упрощается: он становится «плоской» структурой, для которой подходят неявные побитовые присваивание и конструктор копирования. При этом также не требуется деструктор. Все это, правда, за счет некоторой потери гибкости (ведь параметр обязан быть константным выражением). Например:

```
template <typename T, int size> class Vector
{
      T body[size];
  public:
      T& operator[](int index)
      {
         if (index <0 || index >= size)
            throw std::exception("Bad vector size");
         return body[index];
      }
      T& getAt(int index) { return body[index];}
};
```

Получившаяся абстракция не уступает по эффективности и надежности встроенным массивам языка Паскаль: использует плоскую память и проверяет индекс массива на корректность во время выполнения. Если же программист уверен в корректности индекса, то он может использовать более эффективную функцию `getAt()`.

Явная конкретизация шаблона класса (в C++ используется термин «инстанцирование») имеет вид

```
имя_шаблона <аргументы шаблона>
```

Такая конструкция называется в документации по C++ идентификатором шаблона. Аргументы (или фактические параметры) шаблона — это имена типов для параметров-типов и константные выражения для параметров-переменных. Конкретизация шаблона приводит к порождению нового типа. Его именем является идентификатор шаблона. Идентификатор шаблона полностью эквивалентен имени типа. При порождении нового типа может происходить генерация кода для функций — членов нового типа.

Разумеется, компилятор (и компоновщик) должен следить за тем, чтобы новые типы (и генерируемый код) не дублировали друг друга. Это нетривиальная задача (учитывая, что конкретизации шаблона могут быть разбросаны по разным модулям программы), которую каждая реализация решает по-своему.

Так, следующие конкретизации шаблона `Vector` обозначают один и тот же тип во всех модулях программы:

```
Vector<int, 32> v;
typedef Vector <int, 32> IntVec32;
IntVec32 x;
const int n = 16;
Vector <int, 2*n> z = v;
```

Переменные `v`, `x`, `z` — относятся к одному типу, их можно присваивать друг другу и т.д.

Если тип `Vector` кажется недостаточно гибким, то можно использовать определение вектора из подразд. 7.2, превратив его в шаблон и заменив все вхождения типа элемента на `T`:

```
template <typename T> class DynVector
{
      T * body;
      int size;
public:
      explicit DynVector(int sz) { ...}
      // реализация других методов
      ...
      ~ DynVector() { delete [] body; }
};
```

Полезно дополнить класс дополнительными методами, например:

```
void swap(DynVector&);
// обменивает тела без копирования всего массива
// значений
```

Очевидно, что механизм шаблонов классов — это мощное средство развития, позволяющее дополнить язык новыми типами данных. Практически любой содержательный класс, приведенный в данном учебнике (например, `complex`) можно параметризовать. Эту идею воплотили в жизнь создатели стандартной библиотеки C++, недаром большая часть этой библиотеки называется STL (Standard Template Library — библиотека стандартных шаблонов).

Шаблоны функций объявляются аналогично шаблонам классов. Приведем пример функции вычисления квадрата числового значения:

```
template <typename T> T square (T x)
{
     return x*x;
}
```

Этот шаблон функции может породить семейство конкретных функций. Порождение может происходить в результате явной конкретизации (с указанием идентификатора шаблона):

```
int a = square<int>(127);
```

В момент конкретизации происходит порождение новой функции (с генерацией объектного кода).

Заметим, что порожденные функции можно рассматривать как семейство перегруженных функций (считая, что имя порожденной функции совпадает с именем шаблона функции). Тогда возникает интересная возможность вызова порожденной функции без явного указания параметров шаблона:

```
double d = square(127.0); // square<double>
int a,x = 1;
a = square(x); // square <int>
```

Поскольку компилятор знает профиль вызова (например, `(int)`), то он может сопоставить его с профилем шаблонной функции (`(T)`) и отождествить (`int<->T`). Это процесс называется ***выводом типа***. Идея вывода типов проста, но конкретные правила, как и правила поиска правильной перегрузки, довольно сложны, например, см. [9].

Заметим, что процесс вывода приводит к (неявной) конкретизации шаблона функции. Возможность неявной конкретизации очень

полезна, например, при использовании шаблона операции. Пусть мы хотим создать шаблон функции скалярного произведения двух векторов и перегрузить ее как операцию умножения:

```
template <typename T, int size> T operator *
          (Vector<T,size>&v1, Vector<T,size>&v2)
{
     T sum = 0;
     for (int i=0; i < size; i++)
          {
          sum+=v1.getAt(i)*v2.getAt(i);
     return sum;
}
```

Очень важно понимать, что для конкретизации необходимо иметь полный текст шаблона (как функции, так и класса, включая определения всех его членов). Порождение новой сущности происходит во время конкретизации с учетом контекста конкретизации и полного текста шаблона. Например, определение шаблона операции скалярного произведения требует от типа-параметра `T` наличия и доступности четырех операций: наличия преобразования из нуля, операций «+=» и «*», а также конструктора копирования. В контексте определения шаблона непонятно, есть ли эти операции, как их вызывать и т.д. Поэтому все решения по связыванию компилятор может принять, только имея полную информацию как о шаблоне, так и о контексте конкретизации. Именно поэтому библиотеки шаблонов распространяются в исходных текстах (иначе их использовать нельзя). Это абсолютно приемлемо для сторонников открытых исходных текстов, но неприемлемо для поборников авторских прав.

В частности, поэтому в языках Java и C# (корпоративные разработки) применяется более слабая модель связывания при статической параметризации, позволяющая использовать обобщения в оттранслированном виде.

Такие же соображения применимы даже к простому шаблону вектора. Тип элементов этого шаблона требует наличия доступного конструктора умолчания (без этого нельзя завести массив объектов типа). Если его в аргументе шаблона нет, то при конкретизации компилятор выдаст ошибку.

Доступность полной информации как о шаблоне, так и о контексте конкретизации позволяет реализовать такие понятия, как перегрузка шаблонов функций, явная специализация шаблонов и частичная специализация шаблонов классов. В результате хорошо спроектированная библиотека шаблонов позволяет генерировать машинный код, сравнимый по эффективности, а то и превосходящий код, выдаваемый оптимизирующими компиляторами императивных языков типа C и Фортрана.

Перегрузка шаблонов функций

Рассмотрим новый вариант шаблона функции возведения в квадрат, который использует операцию скалярного произведения:

```
template <typename T, int N> T square (Vector<T,N>& v)
{
     return v*v;
}
```

Приведем пример использования разных вариантов `square`:

```
Vector <double, 64> v;
double d = square(1.5); // square <double>
d = square(v);  // square <Vector<double,64>>
```

Явная специализация шаблонов

Для явной специализации необходим общий шаблон. Обычно такой шаблон реализует универсальный алгоритм или структуру. Однако может возникнуть ситуация, в которой для конкретного варианта аргументов шаблона он либо неэффективен, либо вообще неправильно работает. В этом случае можно использовать явную специализацию.

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

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

```
template <> объявление_обычной_функции
```

Впрочем, `template <>` можно опускать, тогда явная специализация будет иметь вид объявления функции (это сделано для совместимости со старыми компиляторами).

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

Примером является функция вычисления максимума:

```
template <typename T> T max(T a, T b)
{
    return a > b ? a : b;
}
```

Однако для строк языка C (и строковых литералов), т.е. типа `const char *`, шаблон работает неверно: он использует операцию сравнения указателей, которая бессмысленна в данном контексте для строк. В этом случае следует написать конкретный вариант для строк языка C, использующий операцию сравнения из его стандартной библиотеки (которая, кстати, является и частью стандартной библиотеки C++):

```
template <> const char * max (const char * a, const
char * b)
{
    return strcmp(a,b) > 0 ? a: b;
}
int q = max(1,2); // шаблон
const char * p = max ("line2", "line1");
  // специализация
```

Аналогичный вид имеет и явная специализация шаблона класса, только в ней уже нельзя опускать `template <>` и требуется явно указать аргументы шаблона:

```
template <typename T> class Container // общий шаблон
{
   // реализация контейнера для любых типов
};
template <> class Container<const char *>
// специализация
{
   // оптимизированная реализация для строк
};
```

Интересно, что интерфейс общего шаблона может отличаться от интерфейса специализированного шаблона (другой профиль методов и даже другой набор методов).

Частичная специализация шаблонов классов

Частичная специализация также хороша тогда, когда общий шаблон не подходит по каким-либо причинам. Частичная специализация порождает новый вариант параметризованного шаблона, который является частным случаем общего шаблона. Набор аргументов, подходящих для частичной специализации, подходит и для общего

шаблона, но является «более точным». Например, общий шаблон использует просто имя любого типа, а частичная специализация — тип указателя, тип массива, функциональный тип и т. п. Другим вариантом специализации является меньшее число параметров, чем в общем случае, и т. д. Точное определение понятия «более точный» дано в стандарте C++, и здесь мы не будем вдаваться в детали.

Пример шаблона, нуждающегося в частичной специализации, — шаблон класса `Stack` (другой пример — класс `DynVector`), содержащий ряд нетривиальных операций. Как уже отмечалось, для любого нового типа хранимых элементов порождается новый набор операций. Причем для ряда типов эти операции делают одно и то же. Например, для указателей все операции над стеком неразличимы. Однако для каждого указательного типа порождается свой набор, что приводит к недопустимому разрастанию машинного кода, который производит одни и те же операции.

Рассмотрим специализацию стека для указателей. Она будет реализована как частный случай (т. е. как конкретизация шаблоном самого себя) — стек, хранящий указатели типа `void*`:

```
template <typename T> class Stack; // общий шаблон
template <typename T> class Stack <T*>
// специализация
     : private Stack <void*> {
     // наследует конкретизацию общего шаблона
     // закрытым образом
     typedef Stack <void*> BASE;
     public:
     explicit Stack(int sz) : BASE(sz) {}
     T * Pop() { return (T*)BASE::Pop();}
     void Push (T * p) { BASE::Push(p);}
     void Swap () {BASE::swap();}
     // ... переадресация других методов
};
Stack <int> s(256); // общий шаблон
Stack <const char *> strs(64);
// частично специализированный
strs.Push("sample");
```

10.3. Особенности реализации параметрического полиморфизма в языках C# и Java

В языках C# и Java синтаксис обобщенных типов проще, чем в языке C++, поскольку параметром обобщения здесь может быть только тип (поэтому указывать этот факт не надо):

```
class имя-обобщенного-типа <список-параметров>
{ объявления-членов }
```

Конкретизация в C# и Java имеет вид такой же, как и в C++, и рассматривается как имя нового типа:

```
class X <T> { ... }    // обобщенный тип
class Y { ... }        // обычный тип
X<Y> s = new X<Y>(); // создан экземпляр конкретного
// класса
X <Y> s1 = s; // две ссылки на объект конкретного
// класса
```

Рассмотрим пример обобщенного типа стек на C# и Java (используя код из гл. 7), который несмотря на простоту, демонстрирует характерные различия между подходами к параметрическому полиморфизму в этих двух языках (хотя общего в этих подходах больше по сравнению с шаблонами C++):

```
class Stack<T>
{
     T [] body;
     int top;
     public Stack(int size)
     {
          body = new T[size];
          top = 0;
     }
     public T Pop() { return body[--top]; }
     public void Push(T x) { body[top++] = x; }
     public bool IsEmpty {
          get { return top ==0; }
     }
     public bool IsFull {
          get { return top == body.Length; }
     }
}
```

Приведем пример использования созданного стека:

```
Stack<int> s = new Stack<int>(16);
int [] x = {1,2,3,4,5,6,7};
foreach (int k in x) s.Push(k);
while (!s.IsEmpty) Console.WriteLine(s.Pop());
Stack <double> sd = new Stack <double>(32);
double [] xx = {1.1,2.2,3.3,4.4,5.5,6.6,7.7};
foreach (double k in xx) sd.Push(k);
while (!sd.IsEmpty) Console.WriteLine (sd.Pop());
```

Обратим внимание на два момента.

Во-первых, нельзя использовать объекты стеки для хранения разнотипных данных. Универсальный стек (из `Object`) мог быть использован для работы с обоими массивами (`int[]x` и `double[]xx`), но в данном примере следует создавать два стека. Попытка записать в `Stack<int>` целое число будет пресечена транслятором. Таким образом, созданный стек безопасен с точки зрения типов. Работая с таким стеком, можно быть уверенным, что никаких исключений, связанных с неверным преобразованием ссылок, не будет.

Во-вторых, получившийся вариант более эффективен вследствие отсутствия неявных операций упаковки в объект типов значений. Ранее эта операция незаметно происходила при обращении к `Push()`. Сейчас мы имеем свой вариант `Push()` для каждого вида стека. В конечном счете компилятор генерирует отдельный код для конкретизации типом значений, а также для конкретизации ссылочными типами. Таким образом, если в программе есть конкретизации `Stack <double>`, `Stack<int>`, `Stack<String>`, `Stack<char[]>`, то будет сгенерировано три набора функций членов: по одному набору для каждого типа значений (`double, int`) и один набор для ссылочных типов (`String, char[]`). Отметим, что возможность генерировать один и тот же код для типов-ссылок связана с тем, что с точки зрения стека типы отличаются только размером. Если размер типов одинаковый, то код функций, копирующих значения этого типа, тоже одинаковый.

Каждый набор сгенерированных методов не использует функции упаковки-распаковки и проверяемые преобразования типов (правильность соответствия типов уже проверена компилятором), что также повышает эффективность по сравнению с универсальным типом.

Теперь рассмотрим реализацию обобщенного типа на Java:

```
Stack<Integer> s = new Stack<Integer>(32);
int [] x = {1,2,3,4,5,6,7};
for (int k : x) s.Push(k);
while (!s.IsEmpty()) System.out.println(s.Pop());
Stack <Double> sd = new Stack<Double>(12);
double [] xx = {1.1,2.2,3.3,4.4,5.5,6.6,7.7};
for (double k : xx) sd.Push(k);
while (!sd.IsEmpty()) System.out.println(sd.Pop());
```

По сравнению с предыдущим вариантом имеем два важных изменения.

Во-первых, в стеке хранятся ссылки типа `Object` (как в универсальном типе `Stack`), а не объекты типа параметра `T`, как в C#. Это связано с тем, что в Java нельзя создавать массивы объектов типа параметра, как нельзя создавать и объекты типа параметра (`new T()` запрещено) внутри обобщенного класса.

Во-вторых, вместо `Stack<int>` (`Stack<double>`) используется `Stack<Integer>` (`Stack<Double>`). Это связано с тем, что аргументами конкретизаций могут быть только ссылочные типы, поэтому необходимо пользоваться классами-обертками.

Эти ограничения продиктованы уже упоминавшимся в подразд. 10.1 обстоятельством: компилятор Java не сохраняет информацию о конкретизированных типах в программе (она используется только на этапе трансляции). В отличие от C#, где для каждой разновидности типа генерируется свой набор функций-членов, в Java используется только код, сгенерированный по объявлению обобщенного типа. В объявлении никакой информации о типе нет, поэтому код функций членов генерируется, как для ссылок на тип `Object`. Более того, обобщенный тип стек можно использовать и как обычный универсальный тип (хотя делать это настоятельно не рекомендуется). В нашем случае универсальный тип `Stack` (см. пример его использования подразд. 7.1) будет прекрасно без всяких изменений работать и с обобщенным типом `Stack`.

Зачем же тогда обобщенные типы в Java? Как и в C#, для надежности: использование параметризованных контейнеров позволяет гарантировать, что они хранят объекты требуемого типа и не возбуждают исключения, связанные с неверными преобразованиями. Очевидно, что повышенная безопасность — это основная цель добавления в язык механизма обобщений.

Основной принцип современного индустриального программирования заключается в том, что безопасность важнее эффективности.

Рассмотрим еще одну конструкцию, связанную с обобщениями, — ограничения.

Зададимся вопросом, что компилятору необходимо знать о типе-параметре, чтобы скомпилировать объявления обобщенного типа. В языке C# необходимо знать, является ли тип типом значений, и если является, то каков его размер. Про остальные типы известно, что это ссылки. Данной информации компиляторам Java и C# достаточно, чтобы скомпилировать объявления, в которых значения типа-параметра только присваиваются и передаются как параметр (строго говоря, можно вызывать операции для класса `Object`, поскольку каждый тип гарантированно их поддерживает).

Однако ни про какие другие операции компилятор ничего не знает. Поэтому вызов «нестандартных» операций изнутри обобщений приводит к выдаче ошибки, т.е. компилятор не знает, поддерживается ли эта операция, поэтому и выдает ошибку, ведь гарантировать правильность вызова операции нельзя.

Ограничение — это конструкция, сообщающая компилятору о том, что тип параметра поддерживает некоторые специфичные операции, которые можно применять к значениям этого типа. В момент конкретизации компилятор проверит, правилен ли тип-аргумент, т.е. поддерживает ли он перечисленные в ограничении операции.

Как сообщить о наличии у типа некоторых операций, отличных от операций над `Object`, и откуда вообще берутся такие операции? Конечно же, новые операции появляются в языках C# и Java при объявлении производного класса (от `Object` или от другого типа), когда программист определяет эти операции в объявлении класса. Других способов определения новых операций нет. Вот и решение проблемы: сообщить компилятору о том, что тип-параметр должен совпадать или быть производным от типа, где определяется эта операция.

В C# синтаксис ограничений следующий:

```
where список-ограничений
```

Ограничение появляется после имени обобщенного класса. Ограничение может быть:

- требованием наследоваться от определенного класса `BaseClass` (`T where T:BaseClass`);
- требованием быть типом значений (`where T: struct`);
- требованием иметь доступный конструктор умолчания (`where T: new`).

Например:

```
class Gen <T,X> where T:struct, IComparable, X:new()
{ ... }
```

В Java допускается только требование наследования от определенного типа (`T extends BaseClass`), поэтому ограничение ставится внутри списка параметров:

```
class Gen <T extends Comparable, X > { ... }
```

Рассмотрим пример контейнера, про элементы которого предполагается, что они являются производными от некоторого класса `Good`, содержащего операцию `int getPrice()`. Это необходимо для реализации операции вычисления суммарной цены всех элементов контейнера. Конечно, можно описать контейнер в следующем виде:

```
class Container <T extends Good>
{
     int getOverallPrice()
     {
     ... T x; ... x.getPrice(); // корректно
     }
}
```

Однако такое решение не универсально, поскольку от любого пользователя контейнера требуется `обязательно` наследовать от класса `Good`, что слишком ограничительно (может мне ничего больше и не надо кроме контейнера, а мне подсовывают еще и класс

в нагрузку). Лучшее решение (и самое общее) — ввести интерфейс, декларирующий метод `getPrice()`, и использовать его:

```
interface IPricedItem
{
     int getPrice();
}
class Container <T extends IPricedItem>
{
     int getOverallPrice()
     {
     ... T x; ... x.getPrice(); // корректно
     }
}
```

Теперь любой класс, реализовавший интерфейс `IPricedItem`, может использовать обобщенный контейнер.

Теперь рассмотрим пример контейнера, использующего упорядочение элементов. Чтобы сделать его максимально общим, следует использовать стандартный интерфейс `IComparable`:

```
class MyDict <T> where T: IComparable
{
    // теперь можно использовать операцию сравнения
    // элементов контейнера: CompareTo(Object x)
}
```

Еще лучше использовать обобщенный стандартный интерфейс (для каждого подобного универсального класса и интерфейса существует обобщенный вариант):

```
class MyDict <T> where T: IComparable<T>
{
    // теперь можно использовать операцию сравнения
    // элементов контейнера: int CompareTo(T x)
}
MyDict<String> stringDict = new MyDict<String>();
// корректно: строки поддерживают сравнение
MyDict<int> intDict = new MyDict<int>();
// тоже корректно: целые поддерживают сравнение
MyDict<Object> intDict = new MyDict< Object >();
// ошибка компиляции — объекты в общем случае нельзя
// сравнивать
```

Ясно, что главная цель введения ограничений — повышение надежности обобщенных классов и методов.