В. И. Великодный

Системы программирования

Часть I

УДК 371.385(072)
ББК Ч481.254р30 + Ч481.286р30
П44

Р е ц е н з е н т ы:
О. В. Коровай, к. ф.-м. н, доц. каф. НОиКР ПГУ им. Т. Г. Шевченко
Е. В. Калинкова, ст. пр. каф. ИиВТ ПГУ им. Т. Г. Шевченко

П44 **Великодный В. И.**
Системы программирования. Часть I: Курс лекций. — Тирасполь, 2012. — 68 с.

В пособии даются общие рекомендации по написанию курсовых работ, приведены требования к оформлению курсовых работ и защите. Описан примерный план работы, даются советы по подготовке отчёта и выступления на защите.

Предназначено для студентов специальности «Информатика» и смежных специальностей.

УДК 371.385(072)
ББК Ч481.254р30 + Ч481.286р30

Утверждено Научно-методическим советом ПГУ им. Т. Г. Шевченко

Введение

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam

vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. In hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio placerat quam, ac pulvinar elit purus eget enim. Nunc vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula.

Fusce mauris. Vestibulum luctus nibh at lectus. Sed bibendum, nulla a faucibus semper, leo velit ultricies tellus, ac venenatis arcu wisi vel nisl. Vestibulum diam. Aliquam pellentesque, augue quis sagittis posuere, turpis lacus congue quam, in hendrerit risus eros eget felis. Maecenas eget erat in sapien mattis porttitor. Vestibulum porttitor. Nulla facilisi. Sed a turpis eu lacus commodo facilisis. Morbi fringilla, wisi in dignissim interdum, justo lectus sagittis dui, et vehicula libero dui cursus dui. Mauris tempor ligula sed lacus. Duis cursus enim ut augue. Cras ac magna. Cras nulla. Nulla egestas. Curabitur a leo. Quisque egestas wisi eget nunc. Nam feugiat lacus vel est. Curabitur consectetuer.

Suspendisse vel felis. Ut lorem lorem, interdum eu, tincidunt sit amet, laoreet vitae, arcu. Aenean faucibus pede eu ante. Praesent enim elit, rutrum at, molestie non, nonummy vel, nisl. Ut lectus eros, malesuada sit amet, fermentum eu, sodales cursus, magna. Donec eu purus. Quisque vehicula, urna sed ultricies auctor, pede lorem egestas dui, et convallis elit erat sed nulla. Donec luctus. Curabitur et nunc. Aliquam dolor odio, commodo pretium, ultricies non, pharetra in, velit. Integer arcu est, nonummy in, fermentum faucibus, egestas vel, odio.

Sed commodo posuere pede. Mauris ut est. Ut quis purus. Sed ac odio. Sed vehicula hendrerit sem. Duis non odio. Morbi ut dui. Sed accumsan risus eget odio. In hac habitasse platea dictumst. Pellentesque non elit. Fusce sed justo eu urna porta tincidunt. Mauris felis odio, sollicitudin sed, volutpat a, ornare ac, erat. Morbi quis dolor. Donec pellentesque, erat ac sagittis semper, nunc dui lobortis purus, quis congue purus metus ultricies tellus. Proin et quam. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Praesent sapien turpis, fermentum vel, eleifend faucibus, vehicula eu, lacus.

Глава 1

Основные понятия

1.1. Понятие языка программирования

Современный мир уже немыслим без вычислительных машин. Они проникли практически во все сферы деятельности человека: от огромных автоматических заводов до бытовой техники. Та скорость научно-технического прогресса, которой достигло человечество, была бы невозможна без компьютеров, то есть устройств, способных производить вычисления по заданному заранее сценарию — *программе*.

Наличие программы является принципиальным отличием компьютеров от, например, механического арифмометра. Арифмометр может выполнить лишь одно арифметическое действие за один раз. Как правило, этого недостаточно для решения стоящей перед человеком задачи. Компьютер же способен выполнить все требуемые расчёты без вмешательства человека. Это значительно повышает уровень автоматизации вычислений.

Более того, расчёт по программе не требует жёсткого задания последовательности действий. Компьютер может решать, какая операция будет следующей, в зависимости от результатов предыдущих вычислений, если подобное предусмотрено программой.

Без программы, описывающей решение стоящей перед человеком задачи, компьютер был бы бесполезен. Однако для работы программы необходимо исполняющее устройство, которое выполняло бы предусмотренные ей действия.

Современные вычислительные машины, как правило, представляют собой электронные приборы. Поэтому когда говорят о вычислительных

устройствах, обычно имеют в виду *электронные вычислительные машины* (ЭВМ). Кроме того, часто употребляется термин компьютер (от англ. computer — вычислитель). Существуют и неэлектронные компьютеры: механические, оптические, биологические.

Подавляющее большинство компьютеров строится из электронных компонентов, так как механические и электромеханические вычислительные устройства не позволяют достичь высокой скорости вычислений, громоздки и менее надёжны.

Современные технологии позволяет создавать микроэлектромеханические системы (МЭМС) размером порядка 1 мкм, но это на два порядка больше, чем размеры электронных компонентов микросхем, которые изготавливаются по технологическому процессу 22 нм. Кроме того, скорость распространения сигналов в электронных схемах равна скорости света, чего невозможно добиться в механической системе.

С другой стороны, существуют принципиальные ограничения на быстродействие и уменьшение размеров микросхем, поэтому в последнее время особое внимание уделяют эффективности вычислений. В частности, стремятся при возможности выполнять различные команды не последовательно, а одновременно на нескольких вычислительных устройствах.

Оптические и квантовые компьютеры пока существуют в виде прототипов и ещё не получили широкого распространения. Более того, квантовые компьютеры основаны на совершенно иных принципах вычислений.

Таким образом, можно дать следующее определение компьютера.

Определение 1.1. *Компьютер* — комплекс технических и программных средств, предназначенный для автоматизации подготовки и решения задач пользователя.

Как следует из приведённого выше определения, работа компьютера обеспечивается двумя составляющими: *техническими средствами* (hardware) и *программным обеспечением* (software).

Технические средства (или *аппаратное обеспечение*) — это оборудование, предназначенное для выполнения простейших вычислительных операций. Его работа заключается в выполнении перечня команд, решающих ту или иную задачу.

Существует множество подходов к построению аппаратного обеспечения. Они зависят как от физических принципов, на которых основана работа компьютера, так и от требований к его надёжности, быстродействию, энергопотреблению.

Набор базовых принципов, на которых основано функционирование компьютера, называется *архитектурой*. В настоящее время существует большое количество используемых на практике различных архитектур, но подавляющее большинство из них являются развитием одной следующих:

- *принстонской* (*архитектуры фон Неймана*) и
- *гарвардской*.

В персональных компьютерах обычно используется архитектура фон Неймана, а гарвардская чаще применяется в микроконтроллерах — микрокомпьютерах, изготовленных в виде одной микросхемы и встраиваемых в различные устройства.

Архитектура фон Неймана основывается на следующих принципах, которые были сформулированы ещё в 1946 году:

1. *Принцип двоичного кодирования.* Вся информация, используемая при вычислениях (и программа, и данные) кодируется в виде набора чисел, представленных в двоичной системе. Информация хранится в специальном устройстве — *памяти*. Выбор именно двоичной системы обусловлен техническими особенностями построения компьютеров, хотя, однако, существовали ЭВМ, использующие десятичную и даже троичную системы.

2. *Принцип однородности памяти.* Программы и данные хранятся в одной и той же памяти. (В гарвардской архитектуре используются отдельные память команд и память данных.)

3. *Принцип адресуемости памяти.* Память представляет собой набор пронумерованных ячеек. При этом каждая из ячеек доступна в любой момент времени.

4. *Принцип программного управления.* Программа состоит из набора команд, выполняемых в определенной последовательности.

5. *Принцип жёсткости архитектуры.* Архитектура компьютера в ходе работы не изменяется.

В соответствии с принципом двоичного кодирования программа для компьютера дожна быть представлена в виде последовательнсти числовых кодов. Но запись команд в таком виде неудобна для человека: она

не очень наглядна и требует особой внимательности, заставляет отвлекаться от решения непосредственно задачи.

Кроме того, машинные команды слишком конкретны. Например, они требуют указания адресов ячеек, в которых хранятся данные. К тому же количество команд хоть и может быть большим, но всё же ограничено. Это усложняет программирование, так как подобное описание решения задачи далеко от привычных людям способов. Поэтому программы записывают сначала в человекочитаемом виде, а затем преобразуют в машинные команды.

Как компромисс между требованиями аппаратного обеспечения и лёгкостью описания решения задач, были созданы специальные языки для человеко-машинного взаимодействия — языки программирования.

Определение 1.2. *Язык программирования* — это формальная система, предназначенная для описания решения задачи при помощи компьютера.

Следует заметить, что программы на том или ином языке программирования пишутся людьми для людей, так как компьютеру требуются лишь машинные коды. Об этом следует помнить, и писать программы так, чтобы в них могли разобраться как сам программист, так и другие.

Для того, чтобы описать язык программирования, необходимо определить его основные составляющие — синтаксис и семантику.

Определение 1.3. *Синтаксис* языка программирования — это набор формальных правил, определяющих какие последовательности символов являются допустимыми в этом языке без учёта вложенного в них смысла.

Например, тот факт, что в математических выражениях в языке C# для каждой открывающей скобки должна присутствовать ппарная ей закрывающая, — это требование синтаксиса.

Существуют формальные способы описания синтаксиса языков. Одним из наиболее распространённых является так называемая *форма Бэкуса—Наура*. В ней синтаксис задаётся как набор правил, описывающих отдельные понятия языка через другие понятия или конкретные цепочки символов. Рассмотрим в качестве примера синтаксис целых чисел:

```
число ::= знак цифра+|цифра+
цифра ::= '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'
знак  ::= '+'|'-'
```

Обозначение «::=» используется для отделения определяемого понятия от собственно определения.

Знак «+» означает одно или более повторений. Таким образом, запись «цифра+» соответствует одной или нескольким цифрам.

Символ «|» разделяет альтернативы. То есть, число — это либо последовательность цифр со знаком, либо последовательность цифр без знака.

В апострофах записываются уже не понятия, а конкретные символы из текста программы.

Как видно из этого примера, описание синтаксиса позволяет лишь определить, к какому понятию относится та или иная последовательность символов. При этом не указывается, как это понятие интерпретировать и исполнять вычислительному устройству.

Иными словами, синтаксис позволяет лишь определить корректность какой-либо записи. Например, выражения «12» и «+12» могут быть синтаксически корректны, но эквивалентны они или нет, определить только с помощью синтаксических правил невозможно. Для этого требуется информация о заложенном в них смысле, то есть семантика.

Определение 1.4. *Семантика* языка программирования — это формальное описание значения понятий языка программирования.

Существует три основныых подхода к описанию семантики:

- *операционный* — конструкции описываются как действия, выполняемые некоторым абстрактным компьютером;
- *денотационный* (математический) — конструкции описываются с помощью таких математических понятий, как множество, утверждение и т. д.
- *деривационный* (аксиоматический) — действие конструкции описывается при помощи задания пред- и постусловий.

Не каждый язык обладает достаточными выразительными возможностями для описания произвольных вычислений. Например, рассмотренный выше язык целых описывает лишь числа, но не операции над ними. То есть его практическая ценность минимальна.

Языки, при помощи которых принципиально возможно описать любой возможный вычислительный процесс называются *вычислительно универсальными* или *полными по Тьюрингу*. Полнота по Тьюрингу желательна для языка, который используется на практике, но не обязатель-

на, если он узко специализирован. Также она гарантирует саму возможность запрограммировать любое вычисление, но не простоту записи, поэтому языки программирования стремятся сделать как можно более лаконичными и выразительными.

Важными составляющими любого языка являются также *система типов* и *стандартная библиотека*.

Типы — это средства определения различных разновидностей данных (таких, как целые числа, векторы, матрицы и т. д.), с которыми можно работать при помощи конструкций языка. Очевидно, эта составляющая играет чрезвычайно важную роль при программировании. Развитые языки программирования содержат средства описания новых типов.

Стандартная библиотека — это набор типичных операций над данными, которые, вообще говоря, не являются частью языка, но встречаются очень часто, поэтому присутствуют в любой его реализации.

Первым языком программирования, широко применяемым на практике, был FORTRAN (от англ. formula translator — переводчик формул), разработанный в середине 50-х годов 20-го века. В настоящее время существует несколько тысяч различных языков программирования. На наиболее распространённые языки существуют международные стандарты. Главная причина их появления, объясняющее такое количество, — необходимость в специализированных инструментах, хорошо подходящих для решения определённого класса задач. Использование универсальных языков программирования часто оказывается неоправданным, так как требует написания большего количества программного кода. В то же время специализированные языки обладают выразительными средствами, позволяющими описать нужные действия при помощи всего нескольких строк.

К наиболее распространённым языкам относятся: Java, C, C++, C#, JavaScript, Perl, PHP, Visual Basic, Python, Ruby и другие.

Такое большое количество объясняется тем, что языки часто создаются для решения определённого круга задач. Поэтому спор о том, какой язык лучше, лишён смысла. Каждый язык хорош в своей предметной области.

1.2. Классификация языков программирования

Исторически языки программирования развивались в сторону увеличения уровня абстракции используемых понятий. Первые компьюте-

ры программировались непосредственно в машинных кодах, поэтому программисту приходилось оперировать лишь такими понятиями, как «ячейка памяти», «адрес» и т. д. Набор математических операций тоже был ограничен.

В языке FORTRAN появилась возможность записывать математические формулы в виде, близком к принятому в математике. Эти формулы не могли быть непосредственно вычислены компьютером, поэтому требовался перевод (трансляция) их в машинные команды. Причём одна формула транслировалась сразу в несколько команд.

В дальнейшем языки программирования стали включать в себя возможности работы со сложными структурами данных, такими как списки, очереди, матрицы. При этом программисту уже не требовалось постоянно думать о том, как эти структуры хранятся в памяти, и какие машинные команды их обрабатывают.

Таким образом, первые языки были близки к машине, а новые и более абстрактные близки к человеку. Подобное разделение позволяет ввести следующую классификацию.

1. *Предметно-ориентированные языки* (DSL — domain specific languages) — языки программирования, ориентированные на решение узкого класса задач. Как правило, оперируют непосредственно понятиями предметной области, для которой разрабатывались. Например, в языке матричных вычислений Matlab матрица — одно из базовых понятий, включённых в сам язык. Примеры предметно-ориентированных языков: М-язык системы Matlab, языки пакетов Mathematica, MathCAD, Maple, язык статистических расчётов R, язык табличного процессора Excel и т. д. Предметно-ориентированные языки очень распространены, так как часто проще разработать отдельный язык для решения задачи, чем пытаться решить её с помощью универсального языка.

2. *Языки высокого уровня* — языки, предназначенные для решения широкого круга задач, и включающие конструкции высокой степени абстракции. Иными словами, это универсальные языки близкие к человеку. Примеры: Java, C++, C#, Pascal и т. д.

3. *Языки низкого уровня* — языки, предоставляющие минимальный уровень абстракции от машинных команд. Обычно они содержат средства для прямого доступа к памяти компьютера. С некоторой долей условности можно отнести к низкоуровневым, например, язык C и некоторые другие. Хоть он и достаточно универсален,

многие конструкции в нём требуют для использования знание особенностей архитектуры компьютера.

4. *Язык ассемблера* (иногда просто ассемблер) — группа языков низкого уровня, не абстрагирующихся от машинных кодов. Практически каждая команда на языке ассемблера соответствует одной команде процессора. Название этого класса происходит от термина *ассемблер* (от англ. assembler — сборщик), соответствующего программе, переводящей команды в машинный код. У разных архитектур вычислительных устройств набор команд различается, поэтому для них требуются отдельные ассемблеры.

5. *Машинный код* — набор числовых кодов, соответствующих действиям, которые может выполнять вычислительное устройство. Например, в процессорах архитектуры IA-32 команда завершения работы части программы кодируется числом $C3_{16} = 195_{10}$ (в языке ассемблера этому коду соответствует команда `ret`).

Очевидно, чем выше уровень языка программирования, тем проще на нём писать программы. С другой стороны, за высокий уровень абстракции приходится платить производительностью, так как многие операции (такие как управление памятью, например) приходится автоматизировать. Хоть программа и получается компактной, из-за автоматизации некоторых действий, количество реально исполняемых команд будет больше.

Рассмотрим примеры решения одной задачи на языках различного уровня. Пусть требуется описать функцию вычисления n-го числа Фибоначчи:

$$\begin{aligned} F_1 &= 1, \\ F_2 &= 1, \\ F_n &= F_{n-1} + F_{n-2}. \end{aligned}$$

Это числовая последовательность, в которой первые два элемента — единицы, а каждый следующий элемент равен сумме двух предыдущих.

Предметно-ориентированный язык Mathematica предназначен для решения математических задач, в том числе и связанных с числовыми последовательностями. Поэтому в нём есть соответствующая встроенная функция. Программа будет состоять всего из одной команды.

```
1 Fibonacci[n]
```

Однако эта команда приводит к выполнению процессором большого количества машиных команд, необходимых для обеспечения работы самой системы компьютерной алгебры Mathematica.

На языке высокого уровня Haskell программа описывается в виде, близком к математическому определению.

```
1 fib 1 = 1
2 fib 2 = 1
3 fib n = fib (n-1) + fib (n-2)
```

Необходимо заметить, что хоть такая запись программы и допустима, но она не является самым эффективным решением поставленной задачи. Рекурсия — определение функции через саму себя — в строке 3 приводит к большому расходу памяти.

При программировании на низкоуровневом языке программирования C, часто не удаётся так же кратко указать свойства вычисляемой функции. Возможно использование рекурсии, но если для языка Haskell рекурсия типична, то в C стараются её избегать. Поэтому в примере ниже описаны конкретные действия вычислительной системы, приводящие к результату.

```
1  int fib(int n) {
2      if (n <= 2)
3          return 1;
4      else {
5          int a, b, c;
6          a = 1;
7          b = 1;
8          n -= 2;
9          do {
10             c = a + b;
11             a = b;
12             b = c;
13             n--;
14         } while (n > 0);
15     }
16     return c;
17 }
```

Как видно из примера, количество команд в программе увеличилось. Это объясняется тем, что каждая из них менее абстрактна и представима в виде всего нескольких машинных команд. (В отличие от функции Fibonacci в Mathematica, скрывающей за собой всё решение поставленной задачи.) Упрощение команд приводит к увеличению их количества.

На языке ассемблера для процессора архитектуры IA-32 функция принимает вид (используется синтаксис фирмы Intel):

```
1  fib:
2      mov edx, [esp+8]
3      cmp edx, 2
4      ja l1
5      mov eax, 1
6      ret
7  l1:
8      push ebx
9      mov ebx, 1
10     mov ecx, 1
11 l2:
12     lea eax, [ebx+ecx]
13     cmp edx, 3
14     jbe l3
15     mov ebx, ecx
16     mov ecx, eax
17     dec edx
18     jmp l2
19 l3:
20     pop ebx
21     ret
```

Каждая строка приведённой программы (за исключением строк, заканчивающихся двоеточием, являющихся метками) соответствует единственной машинной команде. Уровень абстракции минимален. Если в программе на языке C ещё можно было использовать переменные, обращаясь к ним по именам (a, b и так далее), то в данном случае необходимо явно указывать, в каких ячейках памяти и каких регистрах процессора размещать данные. Также необходимо знать и учитывать архитектурные особенности процессора и операционной системы.

В то время как программа на C написанная для компьютера с другой архитектурой процессора выглядит точно так же, программа на языке ассемблера, скорее всего будет сильно отличаться из-за различий в наборе команд и других особенностях.

Машинные команды для процессора архитектуры IA-32, решающие поставленную задачу, представляют собой последовательность чисел. Каждое число кодирует ту или иную команду процессора. Обычно последовательность машинных команд записывается в шестнадцатиричной системе счисления, так как запись в двоичной системе чересчур громоздка. Для удобства последовательность разбивают на блоки по 8

цифр (то есть, по 4 байта):

```
8B542408 83FA0277 06B80100 0000C353
BB010000 00B90100 00008D04 0B83FA03
760789CB 89C14AEB F15BC3
```

Как видно, сложность чтения по сравнению даже с программой на языке ассемблера существенно возросла.

Существует и другой способ классификации — по способу описания действий в программе. В соответствии с ней языки программирования классифицируются следующим образом:

1. *императивные* — программа представляет собой явное описание последовательности манипуляций над данными в памяти компьютера;
2. *декларативные* — программа содержит описание цели и средств её достижения. Они в свою очередь делятся на:
    - *функциональные* — программа содержит описание функций, вызывающих друг друга;
    - *логические* — программа состоит из описания фактов, правил и цели.

Большинство языков программирования императивные, так как компьютер по своей сути — это императивное устройство. Императивными являются, например языки C++, C#, Pascal, Visual Basic и другие. При использовании этих языков необходимо явно указать последовательность команд, приводящих к решению задачи.

Примером программы на декларативном (в частности, функциональном) языке программирования является рассмотренная выше программа на языке Haskell. В ней нет явно заданной последовательности действий, строки программы можно менять местами. Но при вычислении конкретного значения функции (например, `fib 10` — в Haskell не используются скобки при перечислении аргументов функции), последовательность вычислений будет восстановлена автоматически.

Другой пример функционального языка — это Excel. В нём каждую формулу в ячейках можно считать функцией, которая зависит от значений в других ячейках. Пользователь явно не задаёт последовательность вычисления, Excel выполняет это самостоятельно.

Из-за отсутствия последовательности действий, многие типичные для императивного языка операции заменяются на функциональные аналоги. Например, вместо цикла приходится использовать рекурсию

(определение функции через саму себя, как в примере с числами Фибоначчи). Это непривычно для человека, привыкшего программировать в императивном стиле, и отчасти мешает широкому распространению функциональных языков.

Функциональные языки получили в последнее время достаточно широкое распространение, так как во многих случаях позволяют более лаконично представить решение задачи. Примеры функциональных языков: Lisp, Haskell, F#, OCaml и т. д.

Логические языки основаны на алгебре логики и, фактически, являются системами логического вывода. Однако не каждую задачу легко представить в виде набора фактов и правил, что ограничивает область примнения логического программирования. Это неудобно, в частности, для численных расчётов. Самым распространённым логическим языком программирования являеся Prolog и его многочисленные диалекты. Существуют и другие логические языки — Mercury, Oz и т. д.

Сравнение рассмотренных классов приведено в таблице 1.1. В настоящее время многие языки сочетают в себе элементы сразу нескольких подходов. Например, F# являясь функциональным содержит такие императивные конструкции, как циклы, изменяемые переменные и т. д.

Кроме принадлежности к императивным или декларативным языкам часто также говорят о поддерживаемых языком парадигмах программирования.

Определение 1.5. *Парадигма программирования* — это система идей, понятий и методов, определяющих способ и стиль написания программы.

Ниже перечислены наиболее распространённые парадигмы.

- *Структурное программирование* (основоположник — Э. Дейкстра). Программа представляется в виде иерархической структуры блоков команд.

- *объектно-ориентированное программирование* (А. Кэй) — программа состоит из описания объектов, присутствующих в задаче, и взаимодействия между ними;

- *событийно-ориентированное программирование* — программа представляется в виде описания реакций на различные события.

Основным методом структурного программирования является *декомпозиция* (разделение) задачи на подзадачи. Каждая из подзадач про-

	Преимущества	Недостатки
Логический	– Простое описание задач, связанных с принятием решений на основе логического вывода. – Упрощение разработки экспертных систем, работающих с фактами и правилами.	– Узость класса эффективно решаемых задач.
Функциональный	– Отсутствие жёстко заданной последовательности действий упрощает автоматическое распараллеливание программ. – Математическая строгость языка позволяет применить более эффективные методы автоматической оптимизации программ. – Во многих случаях программы на функциональных языках более лаконичны и выразительны.	– Сложности, связанные с описанием изменения состояния памяти. – Многие операции, такие как операции ввода-вывода, требуют задания последовательности действий, что вызывает дополнительные сложности. – Отсутствие понятия присваивания требует постоянного автоматического выделения и освобождения памяти.
Императивный	– Близость к архитектуре позволяет получить максимальное быстродействие.	Создатель языка FORTRAN Дж. Бэкус выделил следующие недостатки: – громоздкость, сложный синтаксис, слишком много понятий; – сильная связь с понятием «память»; – сложности в формализации понятия присваивания, что значительно усложняет анализ программ.

Таблица 1.1: Преимущества и недостатки различных классов языков программирования

граммируется отдельно. Получившиеся в результате блоки команд называют *подпрограммами*. Каждая подпрограмма может ссылаться на другие подпрограммы.

Использование подпрограмм позволяет уменьшить дублирование программного кода. Если какая-то часть задачи повторяется, то её описывают как подпрограмму и в дальнейшем используют ссылки на неё. Это уменьшает объём программы и упрощает поиск ошибок.

Процесс разработки ведётся «сверху вниз». Вначале описывается решение задачи со ссылками на ещё ненаписанные подпрограммы. Это увеличивает уровень абстракции, уменьшает количество используемых команд и в результате упрощает программирование. Затем, каждая подпрограмма, на которую была ссылка программируется отдельно. Если подзадача всё ещё сложна, то она также подвергается декомпозиции.

В объектно-ориентированном программировании в задаче выделяются объекты — отдельные сущности, обладающие некоторым набором свойств. Объекты, схожие по своей природе группируются в классы, устанавливаются взаимоотношения между ними, а затем свойства и поведение объектов каждого класса программируется. Само решение задачи представляется как взаимодействие объектов, отправляющих друг другу сообщения.

Событийно-ориентированное программирование обычно используется, когда задача заключается в обработке внешних запросов. Например, при программировании графических интерфейсов. Вначале в задаче выделяют возможные события (например, нажатие на кнопку), и для каждого события пишется обработчик, то есть описывается реакция программы.

Часто языки поддерживают сразу неколько парадигм, то есть являются *мультипарадигменными*.

Когда говорят, что язык поддерживает ту или иную парадигму, имеют в виду, что в языке присутствуют встроенные средства для реализации подходов, её составляющих. Однако можно применять различные парадигмы и в языках, их не поддерживающих. Например, библиотека GLib использует объектно-ориентированный подход, хоть и написана на языке C, не имеющем для этого встроенных средств.

1.3. Алгоритмы и их сложность

Одним из важнейших понятий в математике и компьютерных науках является понятие алгоритма. Этим термином обычно называют метод,

схему решения какой-либо задачи.

Определение 1.6. *Алгоритм* — это точное предписание, определяющее вычислительный процесс, приходящий от варьируемых исходных данных к искомому результату за конечное число шагов.

Из определения следуют свойства, которыми должен обладать любой алгоритм:

- *дискретность* — алгоритм состоит из элементарных дискретных шагов, требующих конечного промежутка времени;
- *детерминированность* — в каждый момент времени известно, какой шаг будет выполняться следующим;
- *понятность* — алгоритм состоят только из шагов, однозначно интерпретируемых исполнителем — вычислительным устройством;
- *конечность* — для допустимого набора входных данных алгоритм должен завершаться за конечное число шагов;
- *универсальность* — алгоритм должен быть применим к различным наборам входных данных;
- *результативность* — алгоритм должен приводить к искомому результату.

Отличие алгоритма от программы заключается в том, что алгоритм — это последовательность шагов сама по себе, идея решения. А программа — это запись шагов алгоритма на каком-то формальном языке. Так как любой алгоритм, чтобы быть выполненным, должен быть записан в виде программы, эти понятия часто отождествляют.

Алгоритм — это описание способа обработки данных. Очевидно, программа для конкретного вычислительного устройства должна кроме алгоритма также содержать описание обрабатываемых структур данных. Эта идея нашла отражение в известной формуле, предложенной Н. Виртом: «алгоритмы + структуры данных = программы».

Важным этапом в разработке любого алгоритма является доказательство его корректности, то есть правильности выдаваемого результата. Не всегда это тривиально. Изучением свойств алгоритмов с математической точки зрения занимается теория алгоритмов, а практическими аспектами преобразования алгоритма в программу — технология программирования.

Для одной и той же задачи может существовать несколько корректных алгоритмов. Например, отсортировать последовательность целых чисел по возрастанию можно перемещая бóльшие числа в конец последовательности, а меньшие — в начало. А можно рассмотреть все возможные перестановки заданной последовательности и выбрать среди них ту, которая уже является отсортированной. Несмотря на то, что оба алгоритма приводят к одному и тому же результату, их трудоёмкость значительно различается, а значит, можно говорить и о различной эффективности этих алгоритмов. Исследование эффективности можно формализовать, если ввести понятие вычислительной сложности.

Определение 1.7. *Вычислительная сложность* — это характеристика эффективности, выражающаяся в зависимости объема требуемых для работы ресурсов от размера входных данных.

Под ресурсами понимают время (как правило) или память. Соответственно рассматривают временну́ю сложность и пространственную (емкостную). Временну́ю сложность часто называют просто сложностью алгоритма.

Сложность алгоритма — это некоторое математическое выражение, выражающее зависимость времени или количества памяти от объёма данных. Но записать это выражение без учёта быстродействия конкретного вычислительного устройства невозможно. Один и тот же алгоритм может на различных компьютерах выполняться за различное время. Поэтому исследуют не абсолютные значения времени, требуемого для получения результата, а характер его увеличения при увеличении объема данных на входе.

Для этого часто используют так называемые *символы Ландау* (или *O-нотацию*). Это специальные математические обозначения, позволяющие скрыть коэффициент пропорциональности, соответствующий быстродействию, в выражениях для сложности.

Чаще всего используют символ O (читается как *«О большое»*).

Определение 1.8. Говорят, что $f(n) = O(g(n))$, если существуют такие константы $C \in \mathbb{R}$ и $n_0 \in \mathbb{N}$, что для любого $n > n_0$ выполняется неравенство $|f(n)| \leqslant |Cg(n)|$.

То есть, приведённое обозначение говорит о том, что f асимтотически ограничена сверху функцией g с точностью до постоянного множителя. Иными словами, символ O «скрывает» константу и медленно растущие слагаемые.

Существуют и другие обозначения подобного рода. Ω — асимптотическое ограничение снизу, Θ — асимтотическое ограничение сверху и снизу.

Например, если время работы алгоритма описывается выажением $5N + 2$, где N — объем входных данных, то говорят, что такой алгоритм обладает сложностью $O(N)$. Здесь символ O скрыл константу и член меньшего порядка.

Аналогично, $2N^2 + 7N + 5 = O(N^2)$ и $5N \log N + 9N + 4 = O(N \log N)$.

Зная зависимость объёма ресурсов от объёма входных данных, можно оценить эффективность алгоритма. Линейная сложность, как правило, лучше, чем, например, квадратическая. Однако, на практике иногда лучше применить алгоритм с квадратической сложностью, так как константа у алгоритма с линейной сложностью, скрывающаяся за символом O, может оказаться значительно больше для всех N, имеющих смысл на практике.

Подобная ситуация возникла например в задаче перемножения матриц. Хоть и известны алгоритмы быстрого перемножения матриц (например, алгоритм Копперсмита — Винограда, имеющий сложность $O(N^{2,3727})$, где N — количество строк и столбцов матрицы), на практике часто пользуются алгоритмом, основанным на определении произведения матриц, так как для малых N он эффективнее.

Поиск алгоритмов с минимальной возможной сложностью для конкретных задач — одна из важнейших проблем современной теории алгоритмов. Эта область науки активно развивается, но в ней до сих пор есть нерешённые вопросы.

Часто встречающиеся сложности в порядке уменьшения эффективности и примеры алгоритмов для каждой из них:

- константная ($O(1)$) — время работы не зависит от задачи — например, получение результата, который уже был предварительно посчитан и сохранён ранее;
- логарифмическая ($O(\log N)$, основание логарифма не важно, так как все логарифмические функции растут пропорционально) — поиск фамилии в телефонной книге методом деления пополам (N — количество абонентов);
- линейная ($O(N)$) — суммирование последовательности чисел (N — количество слагаемых);
- линейно-логарифмическая ($O(N \log N)$) — сортировка последовательности чисел методом Хоара (N — количество сортируемых

Сложность	$N = 10$	$N = 100$	$N = 1000$	$N = 10000$
$O(1)$	1 с	1 с	1 с	1 с
$O(\log n)$	1 с	2 с	3 с	4 с
$O(n)$	1 с	10 с	1,7 мин.	16,7 мин.
$O(n \log n)$	1 с	20 с	5 мин.	1,1 ч.
$O(n^2)$	1 с	1,7 мин.	2,8 ч.	11,6 сут.
$O(n^3)$	1 с	16,7 мин.	11,6 сут.	31,7 года
$O(2^n)$	1 с	$3{,}9 \cdot 10^{19}$ лет	$3{,}3 \cdot 10^{290}$ лет	$6{,}2 \cdot 10^{2999}$ лет

Таблица 1.2: Оценка времени работы алгоритмов различной сложности

элементов);

- квадратическая ($O(N^2)$) — так называемая сортировка вставками;
- кубическая ($O(N^3)$) — перемножение матриц размера $N \times N$ в соответствии с определением матричного умножения;
- экспоненциальная ($O(2^N)$) — алгоритм поиска решения при помощи полного перебора.

Сравнение времени работы алгоритмов с различной сложностью приведено в таблице 1.2. Все алгоритмы предполагаются одинаково быстрыми при $N = 10$.

Как видно, выбор достаточно эффективного алгоритма играет чрезвычайно важную роль. Например, для задачи сортировки большого массива, замена алгоритма сортировки вставками на алгоритм сортировки Хоара может позволить получить результат за намного меньшее время.

С другой стороны не следует жертвовать сложностью. Если быстродействия используемого алгоритма достаточно, то тратить время на его замену было бы нерационально.

Глава 2

Трансляция и выполнение программ

2.1. Трансляция и интерпретация

Как было показано в предыдущей теме, решение одной и той же задачи может быть записано на различных языках программирования. Так как языки программирования — это формальные системы, то задача «перевода» программы с одного языка программирования на другой также поддаётся формализации. Это позволяет создавать автоматические системы, генерирующие текст программы на одном языке по тексту на другом языке. Такие программы называются *трансляторами*, а сам процесс — трансляцией.

Определение 2.1. *Трансляция* — это преобразование программы с одного языка программирования в семантически эквивалентный текст на другом языке.

Как правило, программы пишут для выполнения их на компьютере. Поэтому особый интерес представляет преобразование текста программы в машинный код — компиляция.

Определение 2.2. *Компиляция* — трансляция программы в машинные коды.

Программа, осуществляющая компиляцию называется *компилятором*. Для каждой пары «язык программирования — процессор» требует-

ся отдельный компилятор, так как процессоры различной архитектуры могут иметь различные наборы команд.

Существуют компиляторы, способные генерировать машинный код для процессоров сразу нескольких архитектур. Они называются *кросс-компиляторами*. Они могут использоваться, например, для компиляции программ для микроконтроллеров. Это связано с тем, что запускать компилятор на самом микроконтроллере нецелесообразно, а часто и вовсе невозможно из-за ограниченности ресурсов последнего.

Машинный код отличается не только для различных процессоров, но и для различных операционных систем. Поэтому говорят, о компиляции под конкретную *программно-аппаратную платформу*. Под ней понимают совокупность процессора и операционной системы.

В настоящее время существует большое количество платформ, поэтому становится важной проблема переносимости.

Определение 2.3. *Программно-аппаратная переносимость* — возможность запуска программы на различных платформах.

Компиляторы новых языков программирования часто пишутся на самих этих языках. Сначала на каком-то другом языке реализуется компилятор минимального подмножества нового языка. Затем на этом подмножестве к компилятору дописывается поддержка остальных возможностей языка, пока он не сможет сам скомпилировать себя. Этот процесс называется *раскруткой компилятора*. В нём компилятор выступает и как обработчик исходного текста, и как результат компиляции.

Кроме компиляции существует и другой подход к получению результата работы программы, когда не производится преобразование в машинный код, а каждая команда анализируется и тут же исполняется.

Определение 2.4. *Интерпретация* — это процесс покомандного анализа и исполнения исходного текста программы.

Не следует смешивать понятия компиляции и интерпретации, так как первое — это преобразование программы, а второе — выполнение. Однако очевидно, оба подхода должны приводить к одному и тому же результату для одной и той же программы.

Для интерпретации необходимо приложение, производящее анализ и выполнение команд. Оно называется *интерпретатором*. Если для некоторого языка программирования существует интерпретатор, то для него можно написать компилятор. Верно и обратное. Однако, многие языки программирования традиционно являются интерпретируемыми или компилируемыми.

Например, к традиционно компилируемым языкам относятся C, C++, Fortran и так далее. Интерпретаторы для них не получили широкого распространения. Традиционно интерпретируемыми языками являются Python, Perl, PHP и другие.

Оба рассмотренных подхода обладают как преимуществами, так и недостатками.

К основным преимуществам компиляции относят независимость результатов компиляции от наличия компилятора во время выполнения и, как правило, более высокую скорость исполнения результата компиляции.

Первое объясняется тем, что результат компиляции — это машинный код, который можно исполнять непосредственно. В компиляторе возникает необходимость однократно — лишь в момент компиляции.

Второе преимущество объясняется наличием в полученном машинном коде только команд, решающих поставленную задачу. Кроме того из-за однократности компиляции можно позволить потратить больше времени на автоматическую оптимизацию программы — исключение лишних действий, применение более эффективных конструкций и так далее. Лидерами по быстродействию результирующего машинного кода считаются компиляторы языков C и Fortran. Компиляторы последнего особенно эффективны при программировании задач, связанных с обработкой больших числовых массивов. Однако, в некоторых случаях компиляторы для других языков могут генерировать более быстрый код.

Одним из главных преимуществ интерпретации (и часто решающим) является высокая переносимость между программно-аппаратными платформами. В самом деле, при появлении нового процессора или новой операционной системы нет необходимости в перекомпиляции уже существующих программ. Достаточно написать под новую платформу интерпретатор.

Так, например, одна и та же программа на языке Python может работать на компьютере как под управлением операционной системы Windows, так и под управлением Linux. Сама программа представляет собой текст, сохранённый в файл. Чтобы выполнить её необходимо запустить интерпретатор и указать, какой файл выполнять. Обычно запуск интерпретатора выполняется автоматически.

Для того, чтобы объединить преимущества обоих подходов, в последнее время часто используется гибридная схема. В соответствии с ней, программа сначала компилируется в некоторое промежуточный код, который затем интерпретируется. Схема получения результата ра-

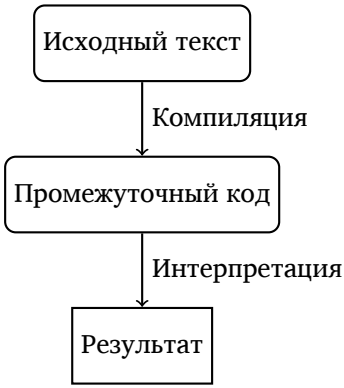


Рис. 2.1: Компиляция в промежуточный код

боты программы в этом случае приведена на рис. 2.1.

Определение 2.5. *Промежуточный код* (или *байт-код*) — промежуточное представление программы на машинно-независимом языке низкого уровня.

Промежуточный код формально является программой на некотором *промежуточном языке*. Для упрощения интерпретатора и повышения скорости его работы, в большинстве случаев промежуточный язык делается как можно проще. Фактически, разрабатывается машинный язык для некоторого абстрактного компьютера. Поэтому интерпретатор промежуточного кода называют *виртуальной машиной*.

Преимущество использования промежуточного кода заключается в том, что часть действий (анализ исходного текста, обнаружение ошибок, оптимизация) выполняется заранее компилятором, что увеличивает скорость работы интерпретатора. При этом сохраняется программно-аппаратная переносимость, так как промежуточный код, хоть и является близким к машинным командам, не привязан к какой-то конкретной архитектуре.

Также следует отметить возможность введения в промежуточный язык более высокоуровневых конструкций по сравнению с машинными командами.

Примерами программных систем, основанных на использовании промежуточного кода, являются .NET и Java. В частности, .NET — это программная платформа, разработанная компанией Microsoft, в основе которой лежит виртуальная машина CLR (от англ. Common Language

Runtime — общеязыковая исполняющая среда), способная выполнять промежуточный код на языке IL (от англ. Intermediate Language — промежуточный язык). Также платформа .NET включает в себя обширную стандартную библиотеку классов и компиляторы с высокоуровневых языков (например, C#) в IL. Платформа .NET стандартизирована и существуют её независимые реализации — Mono, Portable.NET и другие.

Java — это тоже платформа, сочетающая в себе виртуальную машину и компилятор с одноимённого языка. Существуют различные реализации виртуальной машины Java и компиляторы с других языков (Jython, Scala и так далее) в её промежуточный код. Благодаря использованию промежуточного кода, платформа Java используется в самых разных мобильных и встраиваемых устройствах с различным аппаратным обеспечением, так как в этом случае особенно остро стоит проблема переносимости.

При использовании рассмотренной схемы с интерпретацией промежуточного кода, по-прежнему остаётся актуальной проблема быстродействия. По-прежнему приходится выполнять интерпретацию, хоть и более простого промежуточного кода. Решением проблем является так называемая динамическая компиляция.

Определение 2.6. *Динамическая компиляция* или *JIT-компиляция* (от англ. just-in-time — точно в срок) — получение машинного кода во время выполнения программы.

Схема получения результата приведена на рис. 2.2. Компиляция в машинный код с необходимыми оптимизациями производится заранее, а машинный код под конкретную платформу производится в том момент, когда необходимо произвести выполнение программы. При этом машинный код обычно не сохраняется в виде файла, а сразу помещается в оперативную память. Прирост быстродействия обеспечивается тем, что машинный код не требует интерпретации, а выполняется непосредственно компьютером.

Подобный подход применяется, например, в платформе .NET. Для повышения производительности, виртуальная машина CLR выполняет динамическую компиляцию. Также этот подход хорошо зарекомендовал себя в интерпретируемых языках программирования.

Например, V8 — интерпретатор языка JavaScript, применяемого в веб-браузерах, — показывает чрезвычайно высокую производительность именно благодаря использованию динамической компиляции. Для повышения быстродействия разработчики отказались от использования промежуточного кода, так что, фактически, отличие от обычной компиляции заключается только в моменте преобразования кода.

Рис. 2.2: Компиляция в промежуточный код

Использование динамической компиляции приводит к интересным эффектам. Например, код, выполняемый интерпретатором PyPy языка Python, выполняется быстрее, чем код выполняемый CPython, притом, что PyPy сам написан на Python и его код также выполняется CPython. Этот неожиданный результат объясняется тем, что PyPy использует динамическую компиляцию, а CPython — нет.

Написание компиляторов и интерпретаторов — достаточно сложная задача, поддающаяся, однако, автоматизации. Для этого используют так называемые *компиляторы компиляторов* — приложения, генерирующие код компилятора изи его части по описанию языка. Примерами компиляторов компиляторов являются ANTLR, GNU Bison и другие. Они чрезвычайно полезны при разработке собственных предметно-ориентированных языков.

При разработке программ используются также *статические анализаторы* — программы, не выполняющие преобразование в машинный код, но проводящие глубокий его анализ на предмет потенциальных ошибок.

2.2. Этапы трансляции программы

Несмотря на то, что процесс трансляции сильно зависит от языка программирования, в нём можно выделить несколько общих этапов.

Рассмотрим процесс трансляции на примере компиляции *проекта* на языке C++. Большие программы обычно разбивают на несколько файлов, а проектом называют все файлы, относящиеся к конкретной программе. Иногда проекты объединяют в более крупную структурную единицу — *решение*. Решение соответствует приложению целиком, а проекты — отдельным исполняемым файлам, составляющим его.

Проект на языке C++ состоит из файлов с расширениями «.cpp» и «.h». И те, и другие содержат исходный текст программы на языке C++, но компилируются только файлы «.cpp». Файлы «.h» (так называемые заголовочные файлы) используются только для непосредственной подстановки в другие файлы. Это особенность языка C++ и его предшественника — языка C. В языке C#, например, заголовочные файлы не используются и исходный текст хранится только в файлах с расширением «.cs».

Также в проект могут быть включены и другие файлы. Например, с описанием графического интерфейса программы, изображениями, используемыми в ней, и т. д.

На рис. 2.3 приведена диаграмма, описывающая процесс компиляции.

Как видно из рисунка, процесс состоит из трёх этапов.

1. Предварительная обработка
2. Компиляция
3. Связывание

Этап *предварительной обработки* (или *препросессинга*, от англ. preprocessing) предназначен для преобразования исходного текста программы без какого либо его анализа. Часть компилятора, выполняющая предварительную обработку, называется препроцессор.

Предварительная обработка может быть удобна, например, для исключения какой-либо части текста программы или замены некоторого символа другим. Обработка выполняется в соответствии со специальными, командами, называемыми *директивами препроцессора*. Программы на языке C часто содержат большое количество директив, но в языке C++ их количество меньше из-за наличия других средств для получения

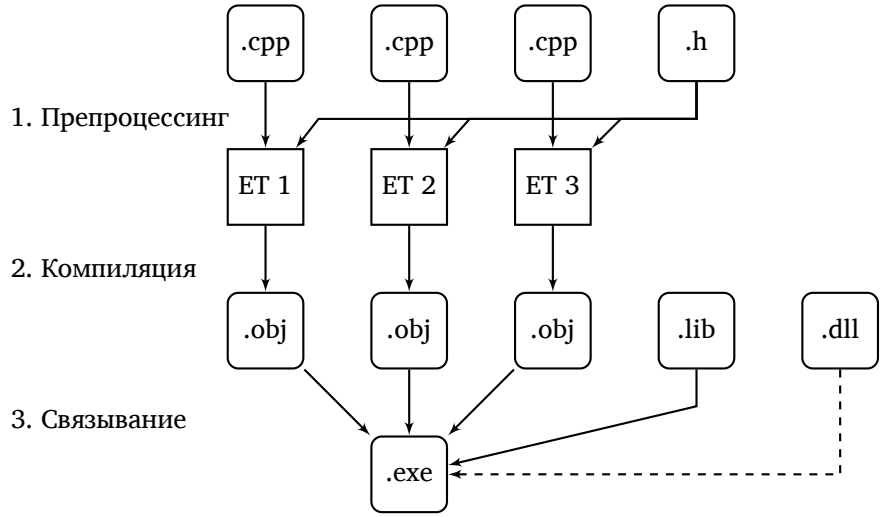


Рис. 2.3: Пример компиляции проекта на языке C++

того же результата. Тем не менее, директивы используются и в программах на C++ (в особенности, директива #include, включающая содержимое одного файла в другой).

В C# также применяется предварительная обработка, причём для управления используются упрощённые аналоги директив языков C и C++. Чаще всего применяют следующие:

- #define символ — определяет указанный символ,
- #if символ … #endif — включает участок кода, заменённый многоточием в программу, если символ определён и т. д.

Например, в приведённой ниже программе строка 3 останется в файле, так как символ DEBUG определён в строке 1 и директива #if включает текст в единицу трансляции. Если же удалить строку 1, то строка 3 в дальнейшем будет проигнорирована компилятором, так как в этом случае символ DEBUG будет уже не определён.

```
1 #define DEBUG
2 #if DEBUG
3 Console.WriteLine("x = {0}", x);
4 #endif
```

Рассмотренный в примере приём может удобен при отладке программы. Однако, злоупотреблять директивами не стоит: так как они не анализируются компилятором, они могут стать причиной труднонаходимых ошибок.

После того, как все директивы препроцессора выполнены, они удаляются из исходного текста программы. Получившиеся после предварительной обработки файлы называются *единицами трансляции*, и именно они подаются на вход компилятору.

Следующий этап — этап *компиляции*, на котором единицы трансляции преобразуются в машинные команды. Этот этап достаточно сложен, поэтому в нём выделяют несколько подэтапов:

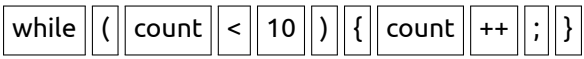
- лексический анализ,
- синтаксический анализ,
- семантический анализ,
- оптимизация и
- генерация кода.

Лексический анализ — это выделение в исходном тексте программы неделимых логических единиц, составляющих программу. Такие единицы называют *лексемами*. Ими являются, например, ключевые слова, имена переменных, символы операций и т. д.

Например, фрагмент программы

```
1   while (count < 10) {count++;}
```

разбивается на лексемы следующим образом:



После выделения лексем компилятор производит поиск синтаксических конструкций, образуемых лексемами — выполняет *синтаксический анализ*. Одним из часто используемых способов представления результата синтаксического разбора потока лексем являются так называемые *абстрактные синтаксические деревья* (*AST*).

Деревом называется иерархическая конструкция, состоящая из точек (узлов), соединённых отрезками (ветвями). Особенностью деревьев является отсутствие циклов, то есть между двумя узлами существует только один путь по ветвям. Узлы, не имеющие связей с узлами на более низком уровне, называются листьями.

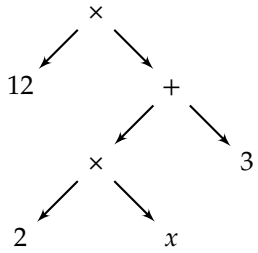


Рис. 2.4: Пример абстрактного синтаксического дерева

Существуют разные способы представления программы в виде дерева. В одном из них в листьях располагаются операнды (то, над чем производятся действия), а в остальных узлах — операции (сами действия). Такое представление позволяет однозначно описать программу. При этом оно отражает также взаимосвязи между лексемами, поэтому деревья — чрезвычайно удобный способ втнутреннего представления программ в компиляторе.

Пример абстрактного синтаксического дерева для выражения

$$12(2x + 3)$$

приведён на рис. 2.4.

В ходе *семантического анализа* для абстрактного синтаксического дерева определяются действия, требуемые для выполнения закодированной в нём программы. Действия также могут иметь какое-то внутреннее представление. Можно также использовать абстрактное синтаксическое дерево, либо какой-то иной способ. Многие компиляторы (например, GCC) для внутреннего представления используют специальный промежуточный код или язык ассемблера.

Зная действия, к которым сводится программа, и их свойства можно провести *оптимизацию*. Это процесс преобразования программы, направленный на уменьшение объёма результирующего кода или уменьшение времени его работы.

Оптимизация заключается обычно в исключении операций, результат которых не используется в дальнейшем, в упрощении и предварительном вычислении арифметических выражений и т. д. Современные компиляторы обладают достаточно мощными средствами для оптимизации программ и часто их результаты оказываются лучше, чем код, оптимизированный вручную человеком.

Последним подэтапом компиляции является *генерация кода*, когда непосредственно определяются последовательности машинных команд или команд промежуточного кода, требуемые для выполнения программы.

Многие компиляторы записывают результаты компиляции каждой единицы трансляции в так называемые *объектные файлы*, которые обычно имеют расширение «.obj» (Windows) или «.o» (Linux).

Объектные файлы могут содержать вызовы подпрограмм, определённых в других единицах трансляции, поэтому их нельзя выполнять непосредственно. Кроме того, в объектных файлах могут содержаться ссылки на стандартные подпрограммы, когд которых расположен в так называемых библиотеках. Поэтому для получения исполняемого файла необходимо объединить объектные файлы и связать их с библиотечным кодом. Это происходит на этапе *связывания*.

Результат трансляции, как правило, — это исполняемый файл, который можно запустить на компьютере, или библиотека, хранящая подпрограммы, которые можно использовать при разработке других программ.

2.3. Исполняемые файлы и библиотеки

Определение 2.7. *Исполняемый файл* — это файл, содержащий программу в виде, в котором она может быть исполнена компьютером.

Исполнителем может быть как непосредственно процессор (если файл содержит машинные команды) или виртуальная машина (если файл содержит промежуточный код).

Исполняемые файлы в операционной системе Windows обычно имеют расширение «.exe». Причём файлы с этим расширением могут содержать как машинный код, так и команды виртуальной машины .NET. У исполняемых файлов в операционной системе Linux расширений как правило нет.

Кроме машинного или промежуточного кода исполняемые файлы также обычно содержат информацию о самой программе: номер версии, её значок и дополнительные сведения. Поэтому исполняемые файлы не могут быть просто запущены на другой программно-аппаратной платформе, если она не поддерживает данный тип исполняемых файлов.

При разработке сложных компьютерных программ иногда приходится использовать часто встречающиеся операции, такие как, например, вычисление наибольшего общего делителя, вывод информации на

экран, работа с матрицами и так далее. Для того, чтобы избежать дублирования программного кода, типичные действия оформляются в виде *подпрограмм* — небольших участков программного кода, предназначенных для выполнения тех или иных действий. Подпрограммы группируются в так называемые библиотеки, которые в дальнейшем можно использовать в других программах.

Определение 2.8. *Библиотека* — это распространяемая независимо коллекция подпрограмм, используемых как части других программ.

Библиотеки могут разрабатываться программистом для своих собственных нужд. Часто использование библиотек оправдано даже если они используются только в одном единственном проекте, так как они позволяют, например, реализовать *принцип модульности*. Модульность означает, что программа разбивается на отдельные, как можно меньше связанные между собой части. Разработать и отладить отдельную часть проще, чем программу целиком, что положительно сказывается на процессе разработки. Сравнительно слабые связи между модулями позволяют достаточно легко модифицировать отдельные части программы.

Многие библиотеки достаточно широко известны. Например:

- Gtk+, Qt — библиотеки для создания графического пользовательского интерфейса;
- OpenGL, DirectX — библиотеки для написания приложений, использующих двумерную и трёхмерную графику;
- MKL, IMSL, LAPACK, GSL, NAG — библиотеки, содержащие подпрограммы для численных расчётов;
- MathGL, PLplot — библиотеки для визуализации численных данных,
- OpenSSL — библиотека для шифрования и безопасной передачи данных по сети и т. д.

В настоящее время существует огромное количество свободно доступных библиотек разного назначения. Например, в дистрибутиве Debian операционной системы Linux их около 4 тысяч. Перед тем, как начать разработку сложного программного продукта, сперва следует изучить существующие библиотеки, чтобы выяснить, не помогут ли они упростить написание программы.

Существует три основных способа организации библиотеки:

	Преимущества	Недостатки
Статические	– программа не требует дополнительных действий по подключению при выполнении.	– необходимость перекомпиляции при обновлении библиотеки; – код библиотеки дублируется в каждом процессе, использующем ей.
Динамические	– возможность обновления библиотеки без перекомпиляции программы; – различные процессы могут совместно использовать один экземпляр загруженной в память библиотеки; – возможность подключать библиотеки на этапе выполнения.	– при выполнении программы требуются действия по загрузке библиотеки в память; – проблема одновременного использования различных версий одной библиотеки.

Таблица 2.1: Сравнение статически и динамически связываемых библиотек

1. в виде исходного текста,
2. статически связываемая библиотека,
3. динамически связываемая библиотека.

Каждый из способов имеет свои преимущества и недостатки. Сравнение двух наиболее распространённых приведено в таблице 2.1.

При распространении библиотеки в виде исходного текста, она представляет собой обычные файлы с текстом программы на том или ином языке программирования. Для того, чтобы использовать такую библиотеку, необходимо просто включить её файлы в свой проект.

Преимуществом подобного подхода является предельная его простота. Для того, чтобы создать библиотеку не требуется выполнения никаких дополнительных действий. Достаточно просто описать требуемые подпрограммы на выбранном языке программирования. Кроме того, программист может изучать исходный текст библиотеки для того, чтобы более эффективно её использовать.

С другой стороны, возникают следующие проблемы. Несмотря на то, что библиотека содержит код, которые не предполагается изменять, его

всё равно необходимо каждый раз компилировать вместе с кодом основного проекта, что может значительно увеличить время трансляции. Если автор библиотеки модифицирует её, то для того, чтобы воспользоваться изменениями, необходимо будет перекомпилировать весь проект целиком. Если у конечного пользователя не будет исходных текстов проекта, то он не сможет обновить часть кода программы, соответствующего тексту библиотеки.

Кроме того, если запустить на компьютере несколько процессов, соответствующих программам, использующим некоторую библиотеку, то код библиотеки будет продублирован в памяти, так как для каждого процесса его код будет загружен в память полностью, несмотря на то, что у них есть общая часть — использованная библиотека.

Указанные недостатки достаточно критичны, поэтому этот способ организации библиотек используется сравнительно редко.

Проблемы связанные с компиляцией можно частично решить, используя *статически связываемые библиотеки*. Они представляют собой заранее скомпилированные подпрограммы, поэтому отпадает необходимость в компиляции библиотеки программистом, использующим её.

Статически связываемая библиотека представляет собой набор объектных файлов, содержимое которых для удобства помещается в один файл-архив. Такой файл обычно имеет расширение «.lib» (Windows) или «.a» (Linux).

На этапе связывания код использованных в программе библиотечных подпрограмм непосредственно помещается в исполняемый файл. Фактически, статически связываемая библиотека — это просто архив с заранее скомпилированными исходными тестами подпрограмм. Таким образом, они не решают проблему дублирования машинного кода в памяти, так как код, хранящийся в исполняемом файле, загружается в память целиком.

Для решения этой проблемы используют *динамически связываемые библиотеки*. Как и статически связываемые, они представляют собой файлы с заранее скомпилированными подпрограммами. Однако, код из них не копируется, а хранится в файле библиотеки. В исполняемый файл вставляется лишь ссылка на подпрограмму в соответствующем файле.

Файлы динамически связываемых библиотек обычно имеют расширение «.dll» (Windows) или «.so» (Linux).

Преимуществом библиотек является то, что они загружаются в память однократно, даже если они используются несколькими процессами.

Другой особенностью динамически связываемых библиотек является то, что их загрузкой в память может управлять сама программа во время выполнения. В ней могут быть команды, загружающие необходимую библиотеку только тогда, когда потребуется подпрограмма, содержащаяся в ней. Более, того, до запуска программы на исполнение может быть вообще неизвестно, какие библиотеки программа будет загружать.

В частности, подобные возможности позволяют реализовать механизм подключаемых модулей (так называемых, плагинов), расширяющих функционал программы. Пользователь устанавливая модули может расширять возможности приложения без модификации его исходного кода. Программа сама найдёт модули на диске загрузит их в память и будет использовать во время работы.

Одним из недостатков динамически связываемых библиотек является проблема версий. Если разные программы используют разные версии одной и той же библиотеки, возникает проблема, заключающаяся в том, что нужно разместить на диске два файла с одинаковым именем. Эта проблема решается различными способами. Например, в операционной системе Linux номер версии добавляется к имени файла, а в Windows на уровне операционной системы существует специальный механизм, позволяющий определить требуемую версию и загрузить в память именно её. Сами библиотеки располагаются в различных подпапках папки WinSxS, что решает проблему конфликта имён.

Кроме того, в операционной системе Windows (в отличие от Linux) нет стандартных мест в файловой системе, где располагаются библиотеки, что иногда создаёт дополнительные сложности.

Практически все популярные языки программирования включают так называемую *стандартную библиотеку*. Она не является частью языка, но содержит описание базовых, часто встречающихся действий и обязательно присутствует в любой реализации языка программирования.

Глава 3

Хранение данных

3.1. Типы данных и литералы

Одним из важнейших понятий в программировании является понятие типа данных.

Определение 3.1. *Тип данных* — это совокупность множества допустимых значений какого-либо набора данных, множества допустимых над ним операций и способа его кодирования в памяти компьютера.

Любые данные, хранящиеся в памяти какого-либо компьютера, относятся к какому-либо типу. Это очевидно следует из того, что любые данные должны быть каким-либо образом закодированы и преобразуются компьютером с помощью некоторого набора операций.

Разнородные данные требуют разные объёмы памяти для хранения, обрабатываются различным образом. Так, например, целые и вещественные числа кодируются в процессорах архитектуры IA-32 различным образом, поэтому для их обработки требуются различные машинные команды. Информация о том, данные какого типа расположены в памяти, может помочь компилятору сгенерировать более эффективный машинный код. В связи с этим, практически во всех распространённых языках программирования в явном виде присутствует концепция типов.

Из-за огромного разнообразия данных, которые требуется обрабатывать, непосредственная поддержка каждого из них на аппаратном уровне, очевидно, невозможна. Поэтому типы данных делят на:

- *базовые* (элементарные), поддержка которых существует в конкретной программно-аппаратной платформе (например, числа),

- *производные* (сложные), которые конструируются из базовых (например, векторы — последовательности чисел).

Базовыми типами для процессора архитектуры AMD64 являются, например, 1-, 2-, 4- и 8-байтовые целые числа, 10-байтовые числа с плавающей точкой. Хотя на самом деле для кодирования любых данных, встречающихся на практике, достаточно было бы только этих типов, для удобства многие другие типы также относят к базовым.

Например, в платформе .NET присутствует следующие встроенные базовые типы (в скобках приведены их названия в языке C#):

- Byte (byte) — 1-байтовые целые со знаком ($-128 \dots 127$);
- SByte (sbyte) — 1-байтовые целые без знака ($0 \dots 255$);
- Int16 (short) — 2-байтовые целые со знаком ($-32768 \dots 32767$);
- UInt16 (ushort) — 2-байтовые целые без знака ($0 \dots 65535$);
- Int32 (int) — 4-байтовые целые со знаком ($-2147483648 \dots 2147483647$);
- UInt32 (uint) — 4-байтовые целые без знака ($0 \dots 4294967295$);
- Int64 (long) — 8-байтовые целые со знаком ($-9223372036854775808 \dots 9223372036854775807$);
- UInt64 (ulong) — 8-байтовые целые без знака ($0 \dots 18446744073709551615$);
- Single (float) — 4-байтовые вещественные числа с плавающей точкой (приблизительно $\pm 1{,}5 \times 10^{-45} \dots \pm 3{,}4 \times 10^{38} \cup \{0\}$, точность — 7 значащих десятичных цифр);
- Double (double) — 4-байтовые вещественные числа с плавающей точкой (приблизительно $\pm 5{,}0 \times 10^{-324} \dots \pm 1{,}7 \times 10^{308} \cup \{0\}$, точность — 16 значащих десятичных цифр);
- Decimal (decimal) — 16-байтовые вещественные числа, отличающееся тем, что представляет числа без округления (приблизительно $\pm 1{,}0 \times 10^{-28} \dots \pm 7{,}9 \times 10^{28} \cup \{0\}$, точность — 28 значащих десятичных цифр);
- Boolean (bool) — логическое значение («ложь» либо «истина»);

- Char (char) — 2-байтовые символы Юникода (Юникод — кодировка, позволяющая представить практически любой символ из существующих систем письменности);
- String (string) — строки (последовательности символов, представляющие фрагменты текста);
- Object (object) — вершина иерархии типов, не используется непосредственно;
- IntPtr, UIntPtr (в C# отсутствуют) — типы, позволяющие хранить адреса ячеек памяти.

Эта система типов в платформе .NET носит название CTS (от англ. Common Type System — общая система типов). Она во многом повторяет систему типов процессоров распространённых архитектур, что сделано для повышения производительности и упрощения работы компилятора.

Многие языки программирования (например, Haskell) имеют развитую систему типов, которые образуют иерархию. Как правило иерархия повторяет вложенность соответствующих множеств в математике. Например, целые числа являются частным случаем вещественных чисел, что может быть отражено в языке программирования. В процессорах архитектуры IA-32, однако, подобные отношения не сохранены: целые и вещественные числа кодируются различным образом и для операций над ними используются различные машинные команды.

В языке C# вершиной иерархии типов является тип object. Сам по себе для хранения данных он не используется, но может быть полезен, если конкретный тип данных неизвестен.

Для того, чтобы работать с данными, должен существовать способ их описания. Для этого используются литералы.

Определение 3.2. *Литерал* — неименованная константа какого-либо типа данных.

Например, число 15 — это неименованная целочисленная константа, то есть целочисленный литерал.

В языке C# можно выделить следующие виды литералов:

1. целочисленные;
2. вещественные;
3. символьные;

4. строковые;

5. логические;

6. литерал null.

Рассмотрим каждый из видов на примерах.

Целочисленные литералы представляют собой последовательность десятичных цифр, перед которой может стоять знак «+» либо «-». По умолчанию литералы относятся к типу int, но можно добавить к числу суффикс «L» для указания на принадлежность к типу long. Также можно использовать суффикс «U» для указания на то, что целочисленный тип беззнаковый.

Пример 3.1. +12 (тип int), 12U (тип uint), 12L (тип long), 12UL (тип ulong) — все эти литералы соответствуют числу двенадцать.

Также существует возможность использовать шестнадцатеричную форму записи чисел. В этом случае необходимо добавить перед литералом «0x».

Пример 3.2. 0x12 ($12_{16} = 18_{10}$), 0x1B ($1B_{16} = 27_{10}$), 0xAUL ($A_{16} = 10_{10}$, тип ulong).

Особенностью *вещественных литералов* является использование точки для отделения дробной части числа. По умолчанию, вещественные литералы принадлежат типу double. Для указания на принадлежность к типу float необходимо добавить суффикс «F», к типу decimal — суффикс «M».

Пример 3.3. 12.3 (тип double), 12.3F (тип float).

Для записи больших чисел можно использовать специальную нотацию вида: «числоEпорядок». Где число — вещественный литерал, а порядок — целочисленный. Эта запись соответствует выражению

Пример 3.4. -12.3E2 ($-12{,}2 \times 10^2 = 1230{,}0$), 1.2E-2 ($1{,}2 \times 10^{-2} = 0{,}012$).

Символьные литералы представляют собой отдельные символы, заключённые в одинарные кавычки. Тип символьных литералов — char.

Пример 3.5. 'A', '%', ' ' — символы «A», «%» и пробел соответственно.

Последовательность	Символ
\'	Одинарная кавычка («'»)
\"	Двойная кавычка («"»)
\\	Обратная наклонная черта («\»)
\0	Символ с кодом 0 (признак конца строки)
\a	Звуковой сигнал
\b	Удаление последнего символа
\f	Переход на следующую страницу
\n	Переход в начало следующей строки
\r	Переход в начало текущей строки
\t	Переход к следующей позиции табуляции
\v	Переход на символ вниз

Таблица 3.1: Специальные символы

Для записи самого символа одинарной кавычки и некоторых других символов применяют экранирование — запись последовательности, начинающейся с символа «\» и соответствующей отдельному специальному символу. Допустимые последовательности приведены в таблице 3.1.

Также можно указать любой символ кодировки Юникод по его коду, используя специальную последовательность вида «\uкод», где код — четыре шестнадцатеричные цифры кода символа.

Пример 3.6. '\u004A' (символ «J»), '\u03BE' (символ «ξ»).

Строковые литералы — это последовательности из нуля и более символов, заключённые в двойные кавычки. Строка из нуля символов называется *пустой строкой*. Строковые литералы относятся к типу string.

Пример 3.7. "" (пустая строка), "abc" (строка из трёх символов), "abc\n" (строка, содержащая специальные символы).

Логические литералы относятся к типу bool и позволяют представить только одно из двух значений: истина (литерал true) либо ложь (false).

Иногда возникает необходимость сохранить в переменной значение, не принадлежащее типу. В этом случае в C# можно использовать *литерал* null. Для того, чтобы объект какого-либо типа мог хранить это значение, необходимо использовать расширенный тип, имя которого состоит из имени исходного типа и знака «?». Например, «int?».

В языке C#, как и во многих других языках, присутствуют различные средства создания производных типов данных. В частности, распространёнными средствами являются:

- перечисления — средство группировки констант по какому-либо признаку;
- массивы — наборы пронумерованных данных одного типа (например, тип, описывающий результаты многочисленных измерений);
- структуры — типы данных, позволяющие группировать объекты различных типов (например, тип, описывающий паспортные данные);
- классы — дальнейшее развитие структур, одно из основных понятий объектно-ориентированного программирования
- и другие.

Производные типы могут быть обобщёнными. Например, тип данных, соответствующий списку значений, можн обобщить, не указывая конкретный тип хранимых элементов, а используя его в качестве параметра T некоторого обобщённого типа «Список элементов типа T». Использование обобщений уменьшает размер программы, так как позволяет лишь один раз описывать схожие действия для различных родственных типов.

Многие типы данных, используемые на практике, допускают взаимные преобразования друг в друга. Так, например, величины типа int могут при необходимости преобразовываться к типу double. Подобные преобразования могут быть удобны, так как у процессора архитектуры IA-32 нет команды для сложения целых и вещественных чисел, только для сложения двух целых и двух вещественных. Поэтому в выражении «1 + 1.0» оба аргумента должны быть приведены к одному и тому же типу.

Преобразования типов могут быть как неявными, выполняемыми автоматически компилятором, так и явными, по указанию программиста.

Неявные преобразования в некоторых случаях могут быть нежелательными, и, кроме того, они усложняют контроль за типами данных в программе. В связи с этим, в некоторых языках неявные преобразования сведены к минимуму или запрещены вовсе. Такие языки называются языками со *строгой типизацией*.

Описание типов данных — один из важнейших этапов работы над программой. Удачно выбранная система типов, хорошо соответствующая предметной области, в которой разрабатывается программа, позволяет значительно упростить процесс программирования и избежать многих ошибок.

3.2. Переменные и константы

Все данные, которые используются в ходе выполнения программы должны располагаться в памяти компьютера. Для доступа к данным в первых программах использовались адреса ячеек памяти, но подобный подход достаточно неудобен. Для упрощения работы с данными была введена абстракция, пришедшая из математики, — понятие переменной.

Определение 3.3. *Переменная* — это объект в памяти компьютера, хранящий значение какого-либо типа.

Для доступа к переменным, им присваиваются *идентификаторы* — имена, записываемые по определённым правилам. Правила записи идентификаторов различаются в различных языках. В некоторых из них разрешено использовать любые последовательности практически любых символов кодировки Юникод, а в некоторых — лишь единственную латинскую букву.

В языке C# упрощённые правила записи идентификаторов выглядят следующим образом. Идентификатор — это последовательность букв, цифр и знаков подчёркивания, начинающаяся не с цифры. Вообще говоря, буквы могут принадлежать любому алфавиту, описанному в Юникод, но рекомендуется использовать только латинские буквы. Во-первых, это упростит чтение и редактирование программы иноязычными разработчиками, во-вторых, некоторые символы в различных алфавитах выглядят одинаково (например латинская «o» и кириллическая «о»), что может вызвать путаницу, и наконец, поддержка нелатинских букв в идентификаторах может отсутствовать в компиляторе.

Регистр символов имеет значение. Поэтому, к примеру, переменные «a» и «A» считаются различными.

Желательно давать переменным имена, отражающие суть хранимых в них данных. Это значительно упрощает чтение программы. Например, для переменной, хранящей количество значений какой-либо величины, имя «NumberOfValues» является более предпочтительным, чем «x».

Любое значение, хранимое в переменной относится к какому-либо типу. Существует несколько подходов к организации связи переменной и типа данных в языке.

При *статической типизации* каждая переменная связывается с типом данных на этапе трансляции программы не может изменять свой тип в дальнейшем. Если указать, что переменная хранит целочисленные значения, то в неё в дальнейшем невозможно будет поместить что-либо

другое. К статически типизированным языкам относятся C++, Pascal, Java и другие.

В языках с *динамической типизацией* тип переменной определяется во время выполнения программы в момент присваивания ей значения. Это позволяет хранить в одной и той же переменной значения различных типов. Динамическая типизация присутствует в таких языках, как Python, PHP, Ruby и т. д.

Язык C# использует смешанный подход. С одной стороны, на этапе трансляции проводятся проверки типов как при статической типизации. С другой стороны, если объявить переменную с использованием ключевого слова dynamic вместо имени типа, то её тип будет определяться динамически по хранимому значению. Однако, эта возможность на практике используется сравнительно редко.

Статическая типизация имеет ряд преимуществ. В частности, она позволяет уже на этапе компиляции программы обнаруживать ряд ошибок, таких как присваивание значения не той переменной. Кроме того, в отличие от динамической типизации, она не требует выполнения проверок во время выполнения программы, что уменьшает время работы.

В C# для того, чтобы связать переменную с типом и выделить под неё место в памяти, необходимо её *объявить*. Все переменные в программе должны быть объявлены. Формат объявления:

тип список переменных;

Здесь список переменных — это перечень идентификаторов через запятую.

Пример 3.8. Переменные a и b после приведённого ниже объявления будут иметь тип int, а переменная c — double.

```
1 int a, b;
2 double c;
```

Переменную при объявлении можно *инициализировать*, то есть присвоить ей какое-либо значение. для этого нужно после соответствующего идентификатора добавить

= значение

После знака «=» может стоять литерал или другая переменная, уже имеющая значение.

Пример 3.9. Переменная g инициализируется значением 9,81, а переменные x и y не инициализированы.

```
1 double x, g=9.81, y;
```

Если переменная инициализируется, то существует возможность не указывать тип переменной, записав вместо него ключевое слово var. В этом случае переменная получит тип, совпадающий с типом присваиваемого значения. При этом сохраняется статическая типизация, так как переменная связывается с некоторым типом, хоть он и не указан явно.

Пример 3.10. Переменная w получает тип double, так как к нему относится литерал 12.3.

```
1 var w = 1.23;
```

После объявления переменной появляется возможность использовать указанный идентификатор, однако не всегда объявление позволяет утверждать, что память под объект выделена.

В языке C# все типы данных делятся на *значимые* и *ссылочные*. К значимым типам относятся целочисленные и вещественные типы, логический тип, структуры и перечисления. Все остальные типы относятся к ссылочным.

Переменная значимого типа хранит непосредственно значение указанного типа, и память под это значение выделяется при объявлении. Идентификатор переменной связывается с выделенным участком памяти и по нему можно получить доступ к хранимому значению.

Переменная ссылочного типа хранит только ссылку на область памяти, где располагается значение. При объявлении память выделяется только для ссылки, а память для значения должна быть выделена отдельно. В языке C# для выделения памяти используется операция new, выделяющая область памяти под объект указанного типа и возвращающая ссылку на неё.

Пример 3.11. Переменная x — ссылка на объект типа System.Drawing.Bitmap(100, 100), соответствующего изображениям размера 100×100 пикселей. Переменная y ссылается на тот же самый объект, так как была инициализирована ссылкой x. В частности, отсюда следует, что изменение x повлечёт за собой изменение y.

```
1 var x = new System.Drawing.Bitmap(100, 100);
2 var y = x;
```

Операции, позволяющей указать, что некоторая область памяти не используется и её можно освободить, в C# нет. Это связано с тем, что виртуальная машина следит за ссылками на каждую выделенную область памяти, и если ссылок на какую-то область больше нет, она освобождается автоматически. Этим занимается специальная подсистема, называемая *сборщиком мусора* (GC — garbage collector). Его поддержка присутствует во многих языках программирования, таких, как Python, Java, Lisp и т. д.

Для повышения быстродействия разработчики языка могут отказаться от сборщика мусора, но в этом случае возникает опасность появления областей, помеченных как занятые, но не используемых, из-за ошибок программиста, забывшего вставить команду на освобождение памяти. Такая ситуация называется *утечкой памяти* (memory leak).

Если значение переменной не должно меняться во время выполнения программы, то её можно объявить как константу, добавив перед объявлением const.

Пример 3.12. Константа $g = 9{,}81$.

```
1 const double g = 9.81;
```

Константы обязательно должна быть инициализированы, так как в дальнейшем присвоить ей какое-либо значение невозможно.

3.3. Виды памяти

Из-за технических ограничений, существующих в настоящее время, память компьютера имеет разнородную структуру. Программы и данные, с которыми работает компьютер, должны храниться в *оперативной памяти*, но она, как правило, работает медленнее, чем процессор. Поэтому процессор проводит вычисления над данными, расположенными в особых ячейках, расположенных внутри него — *регистрах*. Их количество ограничено, но доступ к ним осуществляется за время, сравнимое со временем выполнения одной команды.

Так как регистров мало, то для ускорения доступа к оперативной памяти, часто используемые данные из неё копируются в *кэш процессора*. Этот вид памяти работает быстрее, чем оперативная память, но медленнее, чем регистры. С другой стороны, его объём больше, чем объём данных, которые можно разместить в регистрах.

С другой стороны, оперативная память достаточно дорога и, как правило, требует постоянного электропитания, поэтому большие бъёмы

данных хранят во внешней памяти — жёстких дисках, флеш-памяти, оптических дисках и т. д.

Таким образом, все рассмотренные виды памяти образуют следующую иерархию:

1. регистры (суммарный объём обычно не превышает сотен байт);
2. кэш процессора (до нескольких мегабайт);
3. оперативная память (порядка нескольких гигабайт);
4. внешняя память (объём может значительно отличаться, порядка сотен гигабайт — нескольких терабайт).

Этот список упорядочен по возрастанию объёма и уменьшению быстродействия.

Разница во времени доступа к регистрам и внешней памяти огромна и может отличаться на несколько порядков.

При написании программ обычно не требуется знать, где расположены данные: в регистрах, в кэше или оперативной памяти. Компилятор сам распределяет данные. Можно считать, что все объекты, с которыми работает программа, располагаются в оперативной памяти.

С другой стороны

Глава 4

Управление выполнением

4.1. Операции и операторы

4.2. Подпрограммы и функции

4.3. Исключительные ситуации

Глава 5

Объектно-ориентированное программирование

5.1. Абстракция данных и инкапсуляция

5.2. Наследование

5.3. Полиморфизм

Глава 6

Специальные типы данных

6.1. Массивы

6.2. Контейнерные типы

6.3. Указатели и ссылки

Глава 7

Структуры данных

7.1. Стеки и очереди

7.2. Бинарные деревья

7.3. Хеш-таблицы

Глава 8

Алгоритмы обработки данных

8.1. Сортировки

8.2. Динамическое программирование

8.3. Жадные алгоритмы

Заключение

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam

vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. In hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio placerat quam, ac pulvinar elit purus eget enim. Nunc vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula.

Fusce mauris. Vestibulum luctus nibh at lectus. Sed bibendum, nulla a faucibus semper, leo velit ultricies tellus, ac venenatis arcu wisi vel nisl. Vestibulum diam. Aliquam pellentesque, augue quis sagittis posuere, turpis lacus congue quam, in hendrerit risus eros eget felis. Maecenas eget erat in sapien mattis porttitor. Vestibulum porttitor. Nulla facilisi. Sed a turpis eu lacus commodo facilisis. Morbi fringilla, wisi in dignissim interdum, justo lectus sagittis dui, et vehicula libero dui cursus dui. Mauris tempor ligula sed lacus. Duis cursus enim ut augue. Cras ac magna. Cras nulla. Nulla egestas. Curabitur a leo. Quisque egestas wisi eget nunc. Nam feugiat lacus vel est. Curabitur consectetuer.

Suspendisse vel felis. Ut lorem lorem, interdum eu, tincidunt sit amet, laoreet vitae, arcu. Aenean faucibus pede eu ante. Praesent enim elit, rutrum at, molestie non, nonummy vel, nisl. Ut lectus eros, malesuada sit amet, fermentum eu, sodales cursus, magna. Donec eu purus. Quisque vehicula, urna sed ultricies auctor, pede lorem egestas dui, et convallis elit erat sed nulla. Donec luctus. Curabitur et nunc. Aliquam dolor odio, commodo pretium, ultricies non, pharetra in, velit. Integer arcu est, nonummy in, fermentum faucibus, egestas vel, odio.

Sed commodo posuere pede. Mauris ut est. Ut quis purus. Sed ac odio. Sed vehicula hendrerit sem. Duis non odio. Morbi ut dui. Sed accumsan risus eget odio. In hac habitasse platea dictumst. Pellentesque non elit. Fusce sed justo eu urna porta tincidunt. Mauris felis odio, sollicitudin sed, volutpat a, ornare ac, erat. Morbi quis dolor. Donec pellentesque, erat ac sagittis semper, nunc dui lobortis purus, quis congue purus metus ultricies tellus. Proin et quam. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Praesent sapien turpis, fermentum vel, eleifend faucibus, vehicula eu, lacus.

Index

Оглавление