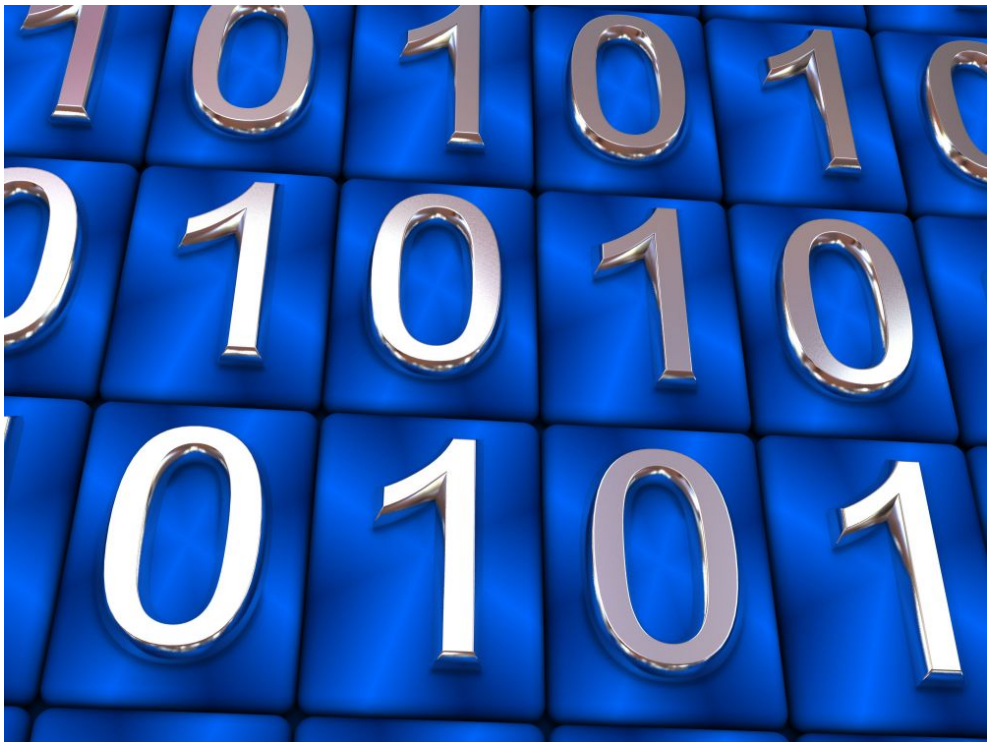
УНИВЕРСИТЕТ ИТМО

А.А. Тюгашев

ОСНОВЫ ПРОГРАММИРОВАНИЯ

Часть I



Санкт-Петербург
2016

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

УНИВЕРСИТЕТ ИТМО

А.А. Тюгашев

ОСНОВЫ ПРОГРАММИРОВАНИЯ

Учебное пособие. Часть I.

Санкт-Петербург

2016

А.А. Тюгашев. Основы программирования. Часть I. – СПб: Университет ИТМО, 2016. – 160 с.

Учебное пособие содержит теоретический материал и лабораторный практикум для изучения дисциплины «Основы программирования». Представлен панорамный взгляд на предметную область, с представлением не только традиционной императивной, но и функциональной, и логической парадигм программирования, исторической ретроспективы и связи с другими областями информатики. Приводится сравнение программирования на языках высокого и низкого уровней (ассемблер). Несмотря на обзорный характер, после прочтения и прохождения входящего в книгу лабораторного практикума студент будет способен писать программы средней сложности на языках C/C++. Книга содержит и специальные главы, посвященные жизненному циклу программных средств современной ИТ-индустрии, проблеме ошибок в программах и методах верификации программного обеспечения, стилю программирования.

Учебное пособие адресовано студентам, обучающимся в ИТМО на кафедре КОТ по направлению 09.03.02 «Информационные системы и технологии»; преподавателям, ведущим теоретические и лабораторные занятия по курсу «Основы программирования». В то же время издание может представлять интерес для школьников, студентов средних специальных заведений и широкого круга читателей, заинтересованных в освоении основ программирования.

Рекомендовано к печати Ученым советом факультета КТиУ 08.12.2015 г., протокол №10.

Университет ИТМО – ведущий вуз России в области информационных и

фотонных технологий, один из немногих российских вузов, получивших в 2009 году статус национального исследовательского университета. С 2013 года Университет ИТМО – участник программы повышения конкурентоспособности российских университетов среди ведущих мировых научно-образовательных центров, известной как проект «5 в 100». Цель Университета ИТМО – становление исследовательского университета мирового уровня, предпринимательского по типу, ориентированного на интернационализацию всех направлений деятельности.

Оглавление

Введение

Данное пособие — не учебник по одному из популярных языков программирования. Прочитав его, Вы не станете профессионалом в C# или Java, использующим полученные навыки для поиска наиболее выгодных предложений на рынке труда. Книга не предназначена также для обучения методологии программирования на уровне, превышающем начальный. В ней нет описаний методов написания эффективных алгоритмов, построения пользовательских интерфейсов, доступа к базам данных и пр., хотя косвенно эти темы в ней освещаются. Цель – освещение базовых принципов современного программирования, с примерами на языках Си и C++ и небольшим введением в функциональное (Лисп), логическое (Пролог) и визуальное программирование.

В настоящее время насчитывается около восьми тысяч языков программирования, причем одни не похожи на другие.

Во введении можно долго рассуждать о об исторической ретроспективе предмета, о его связи со смежными дисциплинами, значимости для жизни современного общества и т. д. Все эти аспекты важны, но, как представляется автору, в самом начале лучше погрузить читателя в суть того, что ему предстоит изучать. Получить представление о предмете может помочь набор примеров — семантически эквивалентных программ (подробнее о том, что такое синтаксис и семантика, будет рассказано далее), которые делают одно и то же, будучи исполненными на ЭВМ, оснащенной соответствующими средствами. Выглядят программы на этих языках по-разному. Следуя примеру Лоуренса Теслера [1, стр. 76], используем для иллюстрации не банальный пример «Здравствуй, мир!», а программу, имеющую (условно) прикладное значение — подсчитывающую сумму нечетных чисел, входящих в последовательность целых чисел. Итак, перейдем к примерам.

Программа на языке BASIC:

```
10 DIM T(100)
20 INPUT N
30 FOR I=1 TO N
40 INPUT T(I)
50 NEXT I
60 GOSUB 110
70 PRINT "СУММА НЕЧЕТНЫХ=" S
80 GOTO 200
110 REM подпрограмма
120 S=0
130 FOR I=1 TO N
```

```
140 IF NOT ODD(T(I)) THEN GOTO 160
150 S=S+T(I)
160 NEXT I
170 RETURN
200 END
```

Программа на языке COBOL:

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01 NUMERIC-VARIABLES USAGE IS COMPUTATIONAL.
       02 NUMBERS PICTURE 9999 OCCURS 100 TIMES INDEXED BY I.
       02 N PICTURE 999.
       02 SUM PICTURE 99999.
       02 HALFC PICTURE 9999.
       02 MODC PICTURE 9.
PROCEDURE DIVISION.
EXAMPLE
       MOVE 23 TO NUMBERS (1)
       MOVE 34 TO NUMBERS (2)
       MOVE 7 TO NUMBERS (3)
       MOVE 9 TO NUMBERS (4)
       MOVE 11 TO NUMBERS (5)
       MOVE 5 TO N
       PERFORM SUMNECH.
SUMNECH
       MOVE 0 TO SUM
       PERFORM ANALIS-1 VARYING I FROM 1 BY 1 UNTIL I>N
ANALIS-1
       DIVIDE 2 INTO NUMBERS (I) GIVING HALFC REMAINDER MODC
       IF MODC IS EQUAL TO 1 ADD NUMBERS(I) TO SUM.
```

Программа на языке APL:

```
∇ СУМ←СУМНЕЧЕТ ЧИСЛА
∇ СУМ←+(2|ЧИСЛА)/ЧИСЛА

вызов: СУМНЕЧЕТ 2 3 3 4 7 9
```

Программа на языке Форт:

```
: СУМНЕЧЕТ
0 SWAP 0
DO
  SWAP DUP 2 MOD
    IF +
   ELSE DROP
   THEN
LOOP

Вызов: 2 3 3 4 7 9 СУМНЕЧЕТ
```

Программа на языке Лисп:

```
(DEFUN СУМНЕЧЕТ(ЧИСЛА)
  (COND
    ((NULL ЧИСЛА) 0)
    ((ODD (CAR ЧИСЛА)) (+ (CAR ЧИСЛА)(СУМНЕЧЕТ(CDR ЧИСЛА))))
    (T (СУМНЕЧЕТ (CDR ЧИСЛА))))))
```

Программа на языке ассемблера микропроцессора Motorola 68000:

```
СУМНЕЧЕТ MOVE.L   (A7)+,A2 Адрес возврата из стека в A2
      MOVE.L   (A7)+,A1 Адрес первого числа => A1
      MOVE.W   (A7)+,D1 Заслать n в D1
      CLR.W D2    Обнулить D2
      JMP      СЧЕТЧИК  Перейти в конец цикла n=0?
ЦИКЛ     BTST     0,1(A1)  Если число по адресу A1 четное…
      BEQ.S СЛЕД     …перейти к метке СЛЕД
      ADD.W (A1),D2  …иначе прибавить число к D2
СЛЕД     ADDQ.W   #2,A1 Взять в A1 адрес следующего числа
СЧЕТЧИК  DBF      D1,ЦИКЛ  Уменьшить D1,пока не -1 => на ЦИКЛ
      MOVE.W   D2,-(A7) Занести сумму нечетных в стек
      JMP      (A2)     Перейти по адресу возврата
```

Программа на языке Пролог:

```
sumnech([X|Xs],S):-odd(X),sumnech(Xs,S1),S is S1+X.
sumnech([X|Xs],S):-sumnech(Xs,S),\+ odd(X).
sumnech([],0).
odd(X):-integer(X),X rem 2 =:= 1.
```

Программа на визуальном языке программирования российской разработки HiAsm (рис. 1).

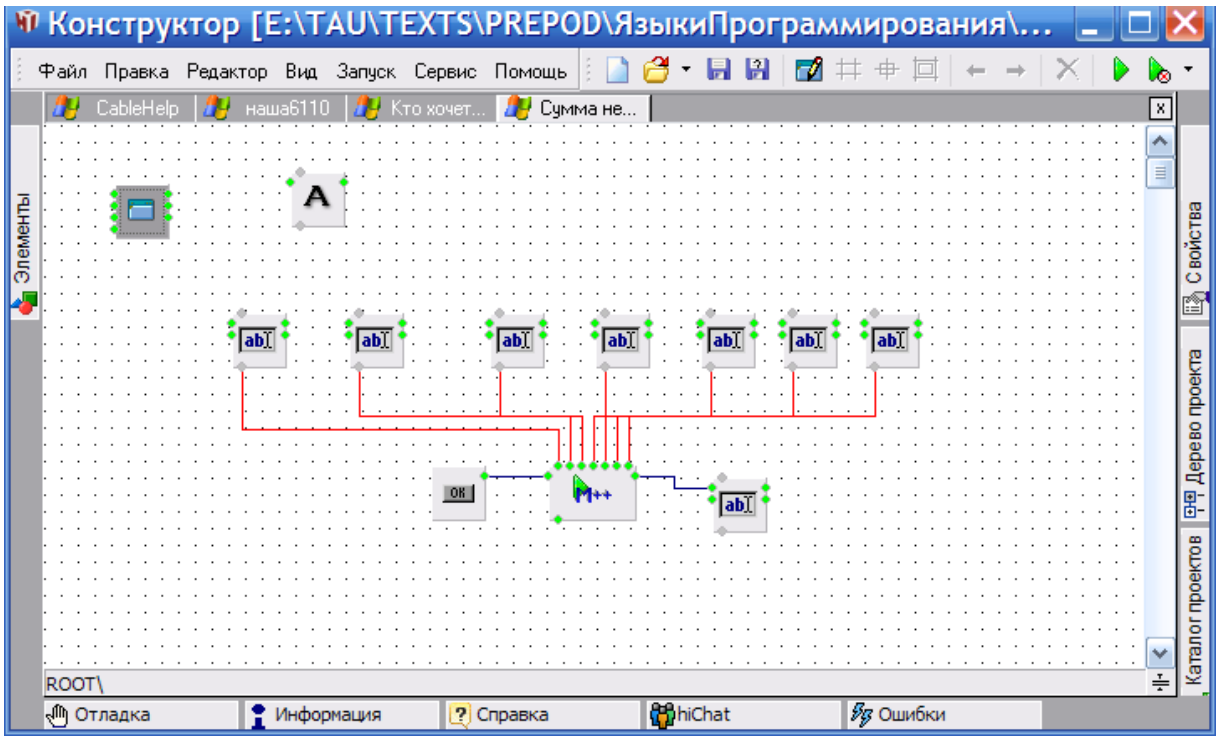


Рис. 1

Автор надеется: читатель не без интереса просмотрел приведенные программы и обратил внимание на то, что они заметно различаются по длине, стилю и внешнему виду вообще…

Целью настоящей книги является дать читателю представление о «ландшафте» предметной области, относящейся к программированию ЭВМ, описать некоторые базовые понятия, обрисовать историю данной предметной области и ее перспективы. Есть даже приложение о *эзотерических языках*.

ЗАМЕЧАНИЕ

Во введении и далее по тексту книги приняты следующие обозначения. *Ключевые понятия*, на которые стоит обратить внимание, набраны курсивом. Для примеров программ используется `моноширинный шрифт`. То, что выдает машина в ходе исполнения программ, набрано **`жирным моноширинным шрифтом`**.

Базовые понятия

Цзы Лу спросил: «Вэйский правитель намеревается привлечь вас к управлению государством. Что вы сделаете прежде всего»?

> Учитель ответил: «Необходимо начать с исправления имен».
>
> Цзы Лу спросил: «Вы начинаете издалека. Зачем нужно исправлять имена?»
>
> Учитель сказал: «Как ты необразован, Ю! Благородный муж проявляет осторожность по отношению к тому, чего не знает. Если имена неправильны, то слова не имеют под собой оснований. Если слова не имеют под собой оснований, то дела не могут осуществляться. Если дела не могут осуществляться, то ритуал и музыка не процветают. Если ритуал и музыка не процветают, наказания не применяются надлежащим образом. Если наказания не применяются надлежащим образом, народ не знает, как себя вести. Поэтому благородный муж, давая имена, должен произносить их правильно, а то, что произносит, правильно осуществлять. В словах благородного мужа не должно быть ничего неправильного»
>
> Легенда о Конфуции

Базовые понятия данного, казалось бы, технического узкоспециального курса принуждают обратиться к проблемам весьма глубоким и иллюстрируемым далекими от техники примерами, могущими показаться забавными.

Но обращаясь к основам — чтобы понять, что такое язык программирования, нужно выяснить сначала, что такое язык и что такое программирование.

ЗАМЕЧАНИЕ

> Парадоксальным образом базовое понятие курса — язык — определяется само через себя, образуя «странную петлю» [2].

Что такое программирование? Прежде всего отметим, что у данного слова есть разные значения (о том, что такое *значение*, речь пойдет в дальнейшем), их можно найти в толковых словарях. В частности, утверждается, что вершители судеб современности, определяющие, что именно мы будем иметь счастье лицезреть по телевизору, занимаются *программированием* (подготовкой программы передач). В математике есть разделы с названиями *линейное программирование*, *динамическое*

программирование, стохастическое программирование. Все они не имеют непосредственного отношения к тому, о чем мы станем говорить . Но вернемся к программированию, являющемуся предметом рассмотрения в настоящей книге. В толковых словарях можно найти такие его определения:

> ***«раздел прикладной математики и вычислительной техники, разрабатывающий методы составления программ для ЭВМ»;***
>
> ***«вид деятельности, необходимый для организации решения различных задач на ЭВМ…»;***
>
> ***«процесс создания компьютерных программ».***

Мы будем использовать следующее рабочее определение: «Программирование — процесс создания или модификации программ для ЭВМ». Слово «рабочее» означает, что определение не претендует на абсолютную истину (ее знает лишь Бог), но вполне пригодно для целей нашего рассмотрения.

Обратим внимание на то, что ключевым понятием здесь является ЭВМ (электронная вычислительная машина), или, используя англицизм, — компьютер.

ЗАМЕЧАНИЕ

> При этом имеются в виду машины с дискретными состояниями [3], или цифровые вычислительные машины — ЦВМ.

Разобравшись немного с тем, что такое программирование, попробуем перейти к понятию *язык*. Что же это — язык? Вновь заметим, внеся в рассмотрение долю здорового юмора, что в русском языке слово «язык» имеет несколько значений. Это часть тела человека или животного (недурного вкуса при правильном приготовлении). И элемент колокола. И пленный враг, имеющий важные сведения, во время войны. Но нас интересует иное значение. Что говорится по этому поводу в толковых словарях? Язык — это:

> ***«система знаков (звуков, сигналов), передающих информацию»;***
>
> ***«система фонетических, лексических, грамматических средств, являющихся орудием выражения мыслей, чувств, волеизъявлений, служащая важнейшим средством общения людей».***

К сожалению, мы сталкиваемся здесь с отсылками к таким важнейшим и глубоким понятиям, как *знак* (*символ*) и *информация*. Более того, можно заметить, что приведенные определения небезупречны. В самом деле, инопланетяне, обладающие разумом, — пресловутые зеленые человечки, — разве мы вправе отказать им в наличии собственного языка, служащего средством общения? Или можем ли мы отрицать наличие у животных неких видов сигналов, служащих орудием выражения их чувств?

В определениях есть отсылка к понятию *информация*. Что мы можем считать информацией? Если мы знаем, какая будет завтра погода, дает ли это нам возможность не промокнуть и не замерзнуть, правильно одевшись? Видимо, дает, и это — информация, полезная и ценная. Получить ее мы можем, прослушав прогноз по радио (устная речь) или прочитав на экране коммуникатора (письменная речь).

Заметим, однако, что если у передачи не будет слушателя, обладающего разумом и целенаправленным поведением, — назовем его *интеллектуальным агентом*, — она останется чередой звуков разной частоты и вряд ли заслуживает полноценного права зваться *информацией*.

ЗАМЕЧАНИЕ

> Вопрос о сути понятия «интеллект» остается открытым, пока неизвестно, возможно ли построить интеллект искусственный, не уступающий человеческому.

Итак, язык возникает, когда один интеллектуальный агент прибегает к некой системе сигналов для передачи информации другому интеллектуальному агенту.

Примеры языков могут быть весьма разнообразными и даже неожиданными. Перечислим некоторые из них.

- сигнальные флаги и огни на флоте;
- бой часов на Спасской башне Кремля;
- иероглифы у древних египтян и современных китайцев;
- языки математических и химических формул;
- язык электрических принципиальных схем;
- пароль и отзыв у разведчиков;
- язык нот для записи музыкальных произведений;
- язык жестов у глухих;
- азбука Брайля для слепых и слабовидящих, использующих тактильные ощущения, и т. д.

Согласно преданию, индейцы в древности передавали друг другу вести, зажигая костры на вершинах гор. Вероятно, читатели слышали о языке цветов (имеются в виду не краски, а цветы, из которых формируют букеты, — здесь мы опять сталкиваемся с проблемой значения слов).

Языки можно разделить на *естественные* (возникшие и развивающиеся без некоего целенаправленного замысла или проекта) и *искусственные*. Примером искусственного (созданного человеком, причем используемого в технических устройствах) языка может служить азбука Морзе.

ЗАМЕЧАНИЕ

Безусловно, к искусственным относятся и языки программирования.

Понятие языка неразрывно связано с понятием символа. Значительная часть языков,, в том числе языков программирования, используют в качестве своих базовых составляющих *цепочки символов* (идущих последовательно один за другим знаков). Что же это — *знак*? Рассмотрим в качестве примера знак Р.

Что это? Русская буква «эр»? Латинская буква «пэ»? Греческая буква «ро»? Или обозначение давления (если, например, символ встретился в записи физического закона)? Или обозначение фосфора при записи химической реакции? Автомобилисты могут вспомнить о столь дефицитной в больших городах стоянке. Мы начинаем чувствовать нечто важное, глубокое, характеризующее символ и язык вообще. А именно, формальный (синтаксический), содержательный (семантический) и прагматический аспекты этого феномена.

Внешний вид символов и способы их сочетания при записи послания образуют первый — формальный — уровень языка. Но есть еще и значение символа. Говорят, что знак обозначает *денотат*.

А есть еще и смысл записанного на языке послания для получателя!

Лучше понять это позволит следующий пример. Предположим, вы — зеленый человечек, брат по разуму из системы Эпсилон Эридана. Высадившись на Земле, вы находите листок бумаги — записку на русском языке: «Даша любит Петю». Что можно извлечь из этой записки? Безусловно, вы понимаете, что планета населена разумными существами, обладающими письменностью. Далее можете отметить наличие в записке линейной последовательности символов разного размера, разделенных пробелами. Некоторые символы повторяются, другие — нет. Видимо, это все. Проведен анализ послания с *синтаксической*, или *формальной*, точки зрения. Далее. Предположим, что записку находит кто-то из читающих по-русски. Какие он сделает выводы? Существует некая особа женского пола по имени Даша, и она неравнодушна к мужчине или мальчику по имени

Петя. Это уже понимание значения символов, или *семантический* анализ. После этого он без особых эмоций отложит записку в сторону или выбросит.

И лишь один-единственный Ваня, найдя эту записку в определенном месте, плача, рвет на себе волосы. А вот это уже — *прагматический* аспект, или *смысл* послания для получателя.

Вернемся к *значению* символов. Предположим, мы имеем дело с записью

$$2 \cdot 2=$$

Какой символ (или символы) уместно поставить в конце? 4? А может быть, 10? Будет ли запись $2 \cdot 2=10$ правильной? Зависит ли это от используемой системы счисления?

А может ли быть «правильной» запись $2 \cdot 2=1022$ или это исключено? Представим себя на месте приказчика на небольшом свечном заводике (или менеджера, выражаясь по-современному). Предположим, мы хотим записать в блокнот наблюдение, что двое рабочих за две смены изготавливают 1022 свечи. Становится ли в этом случае приведенная запись осмысленной (допустимой)?

Вернемся, однако, к языкам программирования. Ясно, что на этих языках записывают не произвольную информацию, а целенаправленные предписания, направленные на решение некоторой задачи. Подобного рода предписания называют еще *алгоритмами* (слово происходит от прозвища древнеарабского математика Аль-Хорезми, жившего в городе Хорезме и описавшего в том числе правила производства арифметических действий над числами в индийской — привычной нам — записи). Программа представляет собой алгоритм решения задачи, записанный на понятном ЭВМ языке. Поэтому языки программирования некоторое время назад иногда называли *алгоритмическими языками*. Собственно, название одного из широко известных языков — ALGOL — получено как сокращение от ALGOrithmic Language (по-русски — алгоритмический язык, Алгол).

ЗАМЕЧАНИЕ

Метод решения задачи может быть записан разными способами.

Помимо того, что язык служит средством общения и передачи информации, он является и механизмом *мышления*. Многие полиглоты отмечали, что в зависимости от того, на каком языке они думают в данный момент, они формулируют идеи по-разному, идут к выводам несколько разными путями и даже получают различные умозаключения. Дуглас Хофштадтер написал: «Я обнаружил, что, когда я «думаю по-французски» мне в голову приходят совсем иные мысли, чем когда я «думаю по-

английски»! Мне захотелось понять, что же главнее, язык или мысли?» [2]. Карлу V Габсбургу, императору Священной Римской империи, приписывают высказывание: «Если бы я хотел говорить с мужчинами, я говорил бы по-французски, если бы я хотел говорить с женщинами, я использовал бы итальянский, если бы я хотел говорить с моей лошадью, я бы говорил на немецком, если бы я хотел говорить с Богом, я бы говорил по-испански».

Физики, математики, химики, ботаники и даже астрологи для описания своих задач и методов их решения используют специфичные языки, удобные в каждом конкретном случае. Жан-Луи Лорьер отмечает [4], что привычный всем со школы современный язык математики, кажущийся столь логичным и естественным, рождался в муках на протяжении тысячелетий. В современном виде он существует совсем недавно, причем в некоторых разделах науки (скажем, в математической логике, где можно для обозначения одного и того же действия встретить запись и $\rightarrow$, и $\supset$, и $\Rightarrow$) до сих пор не существует единой общепринятой системы обозначений! Вычисления в Древнем Египте или Вавилоне были гораздо более громоздкими и трудоемкими просто в силу используемых форм записи. Для примера Лорьер приводит два математических выражения — в записи Франсуа Виета (1540–1603):

$$\left\{\begin{array}{l} H \text{ in } B \\ -F \text{ in } D \\ \hline -F+D \end{array}\right\} \text{ aequebitur } A$$

и принятой сейчас:

$$\frac{ay-by}{b+y}=x.$$

Контрольные вопросы и упражнения

1. Что такое программирование?
2. Что такое язык?
3. В чем заключается разница между естественным и искусственным языками?
4. Какие языки, используются в искусстве, технике, других областях? Приведите свой пример.
5. Что такое знак (символ)?
6. В чем заключается синтаксический аспект языка?
7. В чем заключается семантический аспект языка?

8. В чем заключается прагматический аспект языка?
9. Каковы функции различных языков?

История развития языков программирования

При формулировании понятия «программирование» мы упоминали о том, что здесь подразумевается подготовка программ для ЭВМ, или компьютера. Попробуем рассказать подробнее, что под этим понимается. Компьютер — некая машина, способная выполнять различные действия в соответствии с заложенной в нее *программой*. Именно программа определяет возможность полезного применения ЭВМ, без нее она остается практически ненужной грудой «железа» — проводов, микросхем, пластмассы. Благодаря возможности менять исполняемые программы, закладываемые в *память* машины без изменения ее электронной схемы, мы получаем удивительное и не характерное для ранее созданных человеком машин свойство — *универсальность*, то есть способность одной и той же машины выполнять разные функции.

Программа представляет собой некую систематизированную совокупность инструкций (команд), которые входят в перечень доступных (исполнимых) для конкретной ЭВМ. Набор доступных команд (*система команд*) современного компьютера обычно довольно примитивен и состоит из элементарных действий, подобных сложению или пересылке данных, производимых над содержимым ячеек памяти ЭВМ. Потрясающе, но именно выполнение большого (действительно большого!) количества этих элементарных действий в нужном порядке позволяет современным ЭВМ вычислять траектории полета космических аппаратов к Юпитеру, весьма правдоподобно имитировать сражение на Курской дуге, обыгрывать чемпиона мира по шахматам, анализировать тайны мира элементарных частиц и выполнять иные столь же сложные задачи.

Программист — человек, занятый программированием, иными словами — маг и чародей, который способен заставить компьютер выполнять все перечисленное (и еще то, что пока не заставили выполнить компьютер, но замысел чего уже появился в голове читателя). Но, конечно, любой маг должен знать нужные заклинания. В нашем случае заклинания — это языки программирования, на которых составляются программы.

К какому моменту можно отнести появление первого *программируемого* компьютера (подчеркнем это слово, поскольку машины для автоматизации счета известны с глубокой древности — вспомним абак, но управление процессом вычислений производилось вручную)? Первым программистом в истории человечества часто называют леди Лавлейс, составившую для так называемой аналитической машины Чарльза Бэббиджа несколько предписаний, в том числе самое сложное, с вложенными циклами (причем

Ада Лавлейс, фактически, и ввела такое фундаментальное понятие программирования, как цикл!), по расчету чисел Бернулли. Справедливости ради отметим, что аналитическая машина существовала лишь в проекте. Основывалась она не на электричестве, а на механических узлах и должна была оснащаться паровым двигателем (да, именно это было вершиной техники в 1830-е годы, когда создавался этот выдающийся проект). Однако если бы машина была построена, она стала бы первым программируемым компьютером. Для ввода программы в машину предполагалось использовать карты с отверстиями (перфокарты), подобно тому, как программируются узоры, создаваемые на ткацком станке.

Фактически, программа Ады Лавлейс (кстати, в честь этой хрупкой женщины назван язык программирования, разработанный по заказу министерства обороны США) представляет собой набор карт, а не некую запись в виде текста.

Заметим, что к числу гениальных прозрений леди Лавлейс следует отнести и то, что она предвидела возможность машины оперировать не только числами, но и произвольными объектами, выраженными, например, алгебраически в символьной форме (то есть то, что ныне называется символическими вычислениями, или компьютерной алгеброй), и даже звуками, представляемыми нотами.

Тем не менее машина Бэббиджа не была построена. В какой же стране и когда появился действительно функционировавший компьютер? Был он электронным или механическим? И какой язык программирования применялся? Из одной книги по информатике в другую кочует укоренившийся миф, гласящий, что страна — США, машина — электронная, а год — 1946-й. Это не соответствует действительности. Еще в 1938 году немецкий инженер Конрад Цузе построил Z1 — первую из трех модификаций своей вычислительной машины. Элементной базой служили списанные с телефонной станции реле — фактически, машина была электромеханической. Программа должна была задаваться с помощью ленты с отверстиями — перфоленты.

Однако Цузе, как и леди Лавлейс за век до него, увлекла идея создания универсальной машины, способной решать задачи не только вычислительные, но и из других областей. Он назвал этот свой проект логической машиной. Жаль, что она, как и аналитическая машина Бэббиджа, так и не была построена (правда, по чертежам и записям Бэббиджа базовые части аналитической машины были реализованы энтузиастами позже, уже в XX веке).

В 1945 году, по окончании Великой Отечественной войны, Цузе переехал из Берлина в Баварские Альпы, где и занялся определением языка для будущей логической машины. Язык, названный им Plankalkül (по-русски

произносится «планкалкюль», приблизительно означает «планирующее исчисление»), фактически, представляет собой языком программирования высокого уровня с такими современными чертами:

- наличие повторно вызываемых функций;
- встроенные типы данных;
- массивы;
- аналог условного оператора;
- аналог оператора `assert`;
- исключения;
- цикл `ПОКА`.

Аналога оператора безусловного перехода `GOTO` в языке не было — в полном соответствии с разработанными спустя десятилетия строгими принципами структурного программирования!

Весьма любопытной особенностью языка Plankalkül являлась *двумерная* запись программы, что роднит его с некоторыми современными так называемыми эзотерическими языками (BeFunge и пр.) и, вероятно, может рассматриваться как признак его более высокого уровня (!), но весьма усложняет построение транслятора. Однако, по крайней мере, для подмножества (причем *линеаризованного*) оригинального языка — Plankalkül 2000, энтузиастами компилятор (на машинный код современной ЭВМ) был создан, как и эмулятор машины (см. http://www.zib.de/zuse/Inhalt/Programme/Plankalkuel/Plankalkuel-Report/Plankalkuel-Report.htm).

Далее приводится пример программы на этом языке (сортировка):

```
P6 sort (V0[:6.8.0]) => R0[:6.8.0]

W1[0](4)[
V0[i0:8.0] => Z0[i0:8.0]
1 => Z4[:32.0]
W1[1](i0)
[
(V0[i0:8.0] < Z0[i1:8.0]) & (Z4[:32.0]=1) →
[
i0-i1 => Z1[:32.0]
W1[2](Z1[:32.0])
[
i0 – i2 – 1 => Z3[:32.0]
i0 – i2 => Z2[:32.0]
```

```
Z0[Z3[:32.0]:8.0] => Z0[Z2[:32.0]:8.0]
]
V0[i0:8.0] => Z0[i1:8.0]
0 => Z4[:32.0]
]
]
]
END
```

Цузе и сам предполагал создать автоматический преобразователь (аналог современных трансляторов с языков высокого уровня в машинный код) программы на Plankalkül в исполнимую на его предполагаемой машине перфоленту, но это не было им воплощено в реальность. Однако Цузе составил 49 страниц программ на Plankalkül для оценки шахматных позиций.

На заре вычислительной техники программы вводились в память ЭВМ с пульта, перемычками или тумблерами. Программа писалась в машинных кодах — а память ЭВМ и тогда, и сейчас может хранить лишь двоичные представления. Пример программы в машинном коде (это настоящий код архитектуры PDP-11/CM ЭВМ, предполагающий прибавление 64 к содержимому первого регистра процессора):

```
0110010111000001
0000000000000100
0000000000000000
```

К сожалению, машинный язык весьма неудобен для человека, которомуприходилось запоминать, какие команды кодируются какими последовательностями нулей и единиц, в ячейках с какими номерами хранятся промежуточные результаты вычислений и пр. Неудивительно, что программирование было занятием непростым, доступным немногим, в программы по невнимательности вносилось большое количестве ошибок.

ЗАМЕЧАНИЕ

> Машинные коды можно считать языками программирования первого поколения.

Итак, имелась проблема: пропасть между сложностью решаемых задач и примитивностью машинных команд [5]. Забегая вперед, заметим, что с развитием языков программирования проблема эта окончательно не исчезла, она лишь перешла на другой уровень.

Потом возникла идея: использовать при записи вместо неудобных и трудных для запоминания кодов мнемонические обозначения команд, после чего специальные люди-кодировщики по таблицам соответствия переводили программу в машинный код. Большой скачок в

программировании произошел, когда машине доверили выполнение такого перевода с помощью специальной системной программы (ассемблера). Кроме того, стало возможно обозначать ячейки не конкретными адресами в памяти (фактически, номерами ячеек), а символическими именами и потом ссылаться на них в программе. Саму технологию и используемый язык программирования назвали *автокодом*, или тоже *ассемблером*. Термин *ассемблер* произошел от английского слова *assembler*, так называли сборщик частей в одно целое, в данном случае — сборщик программы. Процесс сборки называется ассемблированием. Слово «автокод» можно расшифровать как «автоматическое кодирование в машинных кодах».

ЗАМЕЧАНИЕ

Уже этот подход стали называть автоматическим программированием [5]. Та же ситуация повторилась при появлении языков третьего поколения.

Соответственно, язык ассемблера отнесем ко второму поколению языков программирования.

Представленная ранее в машинных кодах программа на языке ассемблера выглядит значительно понятнее (по-английски `ADD` означает «сложить», `HALT` — «стоп»):

```
ADD   #100, R1
HALT
```

При переходе от машинных кодов к языкам ассемблера получили сразу несколько полезных эффектов. Во-первых, за счет удобства для человека повысилась производительность труда, сократились сроки разработки. Во-вторых, повысились надежность и качество создаваемых программ за счет меньшего количества возможностей внесения ошибок в программу.

ЗАМЕЧАНИЕ

Ассемблеры можно отнести к языкам программирования второго поколения.

Был ли минус при переходе от машинного кода к ассемблеру? Да. Дело в том, что программа на автокоде непонятна самой машине и для ее использования необходимо наличие специальной программы-переводчика — ассемблера. Данная программа способна построить программу в машинном коде по представленному на ее вход тексту с мнемоническими обозначениями команд.

В то же время при решении задачи программист все равно принужден мыслить в терминах машины. Иными словами, ассемблер, как и машинный код, остается языком так называемого *низкого* уровня. Неудивительно, что уровень машины человек — с высоты своего положения царя зверей —

считает низким! Впрочем, если серьезно, дело еще и в *уровне абстракции*. Следует отметить еще одно важное обстоятельство. Программы на ассемблере не были *переносимыми* — при смене используемой ЭВМ (например, покупке у другого производителя, более мощной и надежной, с иным набором машинных команд) программы, разработанные с использованием команд предыдущей машины, становились практически бесполезными и все надо было переписывать заново!

Возникает резонный вопрос: а почему машины и их языки столь примитивны? Ответ прост: проще, дешевле и надежнее сделать их именно такими. Впрочем, неоднократно делались попытки создать машину с аппаратной поддержкой более развитого и сложного языка. Так, созданный в СССР компьютер «Эльбрус» в качестве ассемблера использовал язык высокого уровня Эль-76.

ЗАМЕЧАНИЕ

Архитектура суперЭВМ «Эльбрус» поддерживала указание на уровне машины типа хранимого в ячейке памяти значения.

Микропроцессор Intel 432 — еще один пример, когда усложнялся машинный язык, в частности, команды поддерживали обработку сложных структур данных. Существовали и Лисп-машины с аппаратной реализацией данного функционального языка, ориентированного на обработку списков. Тем не менее широкого распространения данный подход не получил, и «центр тяжести» в согласовании сложных задач и примитивных машинных команд остается в области программной реализации.

Очередная революция в языках программирования произошла, когда появились языки программирования *высокого* уровня, или третьего поколения. В них характерные для машины понятия, такие как ячейки памяти или простейшие операции суммирования чисел в ячейках памяти, заменялись абстрактными переменными и довольно сложными выражениями, похожими на используемые в математике формулы (неудивительно, что первый язык программирования высокого уровня так и назвали — Фортран, от слов «формульный транслятор»).

ЗАМЕЧАНИЕ

Весьма любопытно, что Фортран жив и успешно используется до сих пор, например, в научных расчетах — конечно, развиваясь и вбирая новшества в программировании (были выпущены стандарты Фортран 77 и Фортран 90). Объясняется это, в частности, большим объемом наработанных эффективных библиотек программ, которые нет смысла переписывать на другие языки, но не только этим — например, в язык изначально встроена поддержка комплексных чисел, чего нет во многих современных языках.

При создании языков высокого уровня делалась попытка выразить в языке суть алгоритма решения задачи, абстрагированного от мелких деталей реализации на той или иной ЭВМ, что нашло свое отражение в названии еще одного знаменитого языка программирования — Алгол (от англ. algorithmic language — алгоритмический язык).

Более того, по крайней мере в некоторой степени (поскольку все же нюансы реализации на разных платформах различались) программы на языках высокого уровня стали переносимыми. Для того чтобы программу можно было использовать на некоторой ЭВМ (с произвольной системой команд), достаточно было наличия для нее транслятора (программы-переводчика) с соответствующего языка высокого уровня в машинный код. При наличии трансляторов для разных архитектур теоретически становится возможным написать и отладить программу на одной ЭВМ, а затем перенести ее на другую и там использовать.

Естественно, к минусам данной технологии можно отнести необходимость предварительной разработки транслятора. При этом он может относиться к одной из двух разновидностей: *компиляторы* и *интерпретаторы*. Разница между ними заключается в следующем. Компилятор принимает на вход всю программу на языке высокого уровня целиком и в результате процесса трансляции строит так называемый объектный модуль, содержащий машинный код, понятный процессору целевой ЭВМ. Интерпретатор переводит программу на языке высокого уровня построчно, при этом для каждой строки исходной программы создает некоторое внутреннее представление на специальном *промежуточном* языке, которое направляется специальной *машине времени исполнения*, или *виртуальной машине* (*Virtual Machine*). Эта машина немедленно исполняет полученное предписание.

И у компиляторов, и у интерпретаторов есть достоинства и недостатки. К достоинствам компилятора относится в первую очередь то, что полученный машинный код может непосредственно исполняться на данной ЭВМ, при этом наличия компилятора в момент выполнения не требуется. Выполнение же происходит на максимально возможной скорости. Однако в случае необходимости внесения в программу изменения, даже минимального, потребуется произвести перекомпиляцию исходного текста полностью, даже если он насчитывает десятки тысяч строк!

Интерпретатор переводит программу построчно, поэтому он удобен для отладки программы, когда постоянно происходят уточнения и изменения и нужно быстро увидеть, как поведет себя откорректированная программа. К недостаткам интерпретатора относится то, что, поскольку программа исполняется интерпретатором, а не непосредственно процессором, во-

первых, скорость выполнения программы существенно ниже, чем при компиляции (обычно в 10–20 раз), а во-вторых, требуется наличие машины времени выполнения в памяти.

ЗАМЕЧАНИЕ

> В настоящее время существуют так называемые «компиляторы-на-лету» (Just In Time — JIT), например, в Java-программировании. Они позволяют сократить потери времени из-за использования режима интерпретации с порядков до двух-трех раз.

В то же время скомпилированная программа в машинном коде — аналогично программе на ассемблере — может исполняться лишь на архитектурно совместимой ЭВМ. А для интерпретируемого языка (например, Java) достаточно установки на данный компьютер машины времени исполнения (в случае Java — JVM (Java Virtual Machine)), после чего на ней теоретически могут быть исполнены любые программы на данном интерпретируемом языке (лозунг Java — Write Once Run Everywhere — «написав раз, запускаешь везде!»). Среди языков программирования высокого уровня Бейсик, Лисп, Java, Питон, Форт обычно реализуются как интерпретаторы, а Фортран, Си, C++, Паскаль, Модула-2 — как компиляторы, хотя это правило не является обязательным.

Программирование на языках высокого уровня гораздо удобнее для человека, чем в терминах машины. В результате появления Фортрана и Кобола программирование стало доступным широкому кругу специалистов. Неудивительно, что языки программирования высокого уровня стали бурно развиваться. В настоящее время известно **более 8000 языков программирования.**

Несмотря на наличие такого большого количества языков, лишь некоторые из них массово применялись, породили многочисленных потомков или оказали существенное влияние на систему понятий и дальнейшее развитие предметной области. Наиболее известными и/или оказавшими влияние на теорию и практику развития программирования языками являются (приведены язык-родоначальник и созданные затем на его основе версии и языки-потомки):

- Фортран — Фортран IV — Фортран 77 — Фортран 90;
- Кобол;
- Алгол — Алгол 60 — Алгол 68;
- Симула — Симула-67;
- Smalltalk;
- PL/I — PL/M;

- BASIC — GW-Basic — Turbo Basic — Quick Basic — Visual Basic;
- Паскаль — Модула — Модула-2 — Оберон — Active Oberon — Компонентный Паскаль — Zonnon;
- Ада — Ада 83 — Ада 95 — Ада 2012;
- (BCPL — B) — C — Objective C — C++ — Java — C# — C11;
- APL — K — J;
- Лисп — Scheme — Common Lisp — Clojure — AutoLisp;
- ML — Standard ML — Ocaml — F# — LazyML — Miranda — Haskell — Curry;
- Planner — QA4 — Popler — Conniver — QLisp;
- Пролог — Parlog — Mercury — Oz — Fril — P#;
- Форт;
- Клу.

Большинство известных языков программирования используют в качестве основы для наименования ключевых слов английский язык (впрочем, недавно появились сообщения о создании языка программирования, основанного на арабской письменности с ее записью справа налево). Для русскоязычного читателя немаловажным может оказаться вопрос: существуют ли языки программирования, ориентированные на русскую лексику? Весьма серьезным предубеждением против нашей родины является почему-то имеющееся у некоторых программистов мнение, что их не существует.

Нет, естественно, они существуют. Отвлечемся при этом от простого перевода известных языков, изначально основанных на английской лексике, на русский (а подобный перевод производился не раз, например, для таких языков, как Алгол, Ада или Пролог). Поговорим об изначально разработанных на основе русских слов языках. Во-первых, существовали (и существуют, естественно, не столь широко известные, но используемые в специальных и военных приложениях — и, если подумать, становится ясно, что иначе просто нельзя) русские автокоды для отечественных ЭВМ. А их было создано немало, в том числе выдающихся, например БЭСМ-6, имеющих оригинальную архитектуру и превосходящих иностранные аналоги своего времени.

Если говорить о языках высокого уровня, то еще в начале 1960-х годов появился ЛЯПАС (логический язык для представления алгоритмов синтеза), трансляторы с которого были созданы практически для всех активно используемых в СССР ЭВМ («Урал», БЭСМ, серий ЕС ЭВМ и СМ ЭВМ). Уникальная разработка (фактически, первый в мире персональный

компьютер) «МИР», произведший сенсацию на международной выставке в Лондоне и закупленный IBM в 1967 году, программировался на языках Алмир-65 и Аналитик. Они были реализованы аппаратно, но фактически представляли собой языки высокого (или даже сверхвысокого) уровня за счет поддержки абстрактных типов данных, вычислений в произвольных алгебрах, аналитических преобразований. Весьма интересна диалоговая система программирования ДССП, ориентированная на стек и словарную организацию, подобно языку Форт, но с рядом отличий, ведущих свою родословную от первой в мире троичной ЭВМ «Сетунь».

Замечательным оригинальным языком, который обладает весьма интересными возможностями, а по своему стилю был даже выделен Н. Н. Непейводой [6] наряду с Прологом в группу «языков сентенциального программирования», является созданный В. Турчиным в СССР в 1966 году РЕФАЛ (Рекурсивных Функций Алгоритмический язык).

Специально для поддержки преподавания информатики в средних школах в свое время был создан весьма интересный и мощный язык РАПИРА. Очень интересна и система КуМир, включающая не просто язык, но целую учебную среду разработки программ. Ведутся работы над основанными на русских ключевых словах языках программирования и в настоящее время. Назовем всего несколько: встроенный язык системы управления предприятием 1С, языки Глагол, Пифагор, Фактор.

Помимо языков программирования, предназначенных для последующего перевода и исполнения вычислительной машиной (процессором), в литературе выделяют в обособленную группу так называемые *скриптовые языки*, или *языки сценариев*. К этой группе относят, например, такие популярные языки, как Perl, PHP, JavaScript, BASH, и другие. Под сценарием подразумевается некий набор действий, исполняемых операционной системой, браузером или иным подобным программным компонентом. При этом скрипт (программа на языке сценариев) интерпретируется соответствующим компонентом. Типичный пример – язык командной строки, с помощью которого можно задать операции над файлами в указанной папке.

У пытливого читателя, возможно, уже возник вопрос: что пришло — и пришло ли — на смену языкам программирования высокого уровня, или языкам третьего поколения? Многие создаваемые в настоящее время языки их создатели относят к четвертому поколению (а наиболее нескромные — даже к пятому, но это не является общепринятой точкой зрения).

К направлениям исследований и разработок в области языков программирования четвертого поколения могут быть отнесены:

- проблемно-ориентированные языки программирования;

- декларативный подход к программированию;
- языки визуального (графического) программирования;
- использование в программировании подмножеств естественного языка.

Проблемно-ориентированные (или *предметно-ориентированные*) языки программирования (англ. DSL — Domain Specific Language) подразумевают, что в языке наряду с универсальными управляющими конструкциями и типами данных присутствуют встроенные средства для описания понятий, характерных для конкретной предметной области, для решения задач которой предназначается данный язык. Предметно-ориентированные языки программирования используются в различных сферах — атомной энергетике, космических исследованиях, радиотехнике и пр. Далее приводится пример программы на проблемно-ориентированном языке, транслятор с которого был разработан автором в ходе автоматизации программирования бортовых алгоритмов управления реального времени для космических аппаратов:

```
t11:=(a003=0)=>f005+(a003=1)=>f200
t12:=(a004=0)=>f006+(a004=1)=>f201
t9:=t11CHt12
t13:=f003CHf004
t111:=(a003=0)=>f015+(a003=1)=>f210
t112:=(a004=0)=>f016+(a004=1)=>f211
t91:=t111CHt112
t10:=t13CHt91
t8:=(a002=0)=>t9+(a002=1)=>t10
t6:=f101->t8
t18:=f009CHf010
t19:=((f011->f012)->f014)
t17:=t18->t19
t15:=f103->t17
t16:=f007CHf008
t14:=f102->t16
t7:=t14CHt15
t5:=t6CHt7
t4:=f002CHt5
t3:=f111CHf100
t2:=t3->t4
endtxt
f200=f220
f005=f015
f006=f016
end
```

Весьма популярны встроенные проблемно-ориентированные языки в мощных информационных системах. Яркий пример — системы автоматизации управления предприятием, в которых поддерживаются такие понятия, как документ, бухгалтерский счет, проводка и пр. Встроенный язык программирования системы SAP/R3 называется ABAP, язык белорусской системы «Галактика» — VIP, есть свой язык в известном в нашей стране пакете 1С. Система автоматизации проектирования AutoCAD позволяет писать дополнительные приложения на специально адаптированной версии языка программирования Лисп — AutoLiSP. В системе управления базами данных Oracle для написания программ применяется язык PL/SQL. Все это позволяет значительно быстрее и удобнее создавать прикладные программы и повысить качество разработки.

Декларативный подход к программированию означает, что с программиста снимается обязанность подробного инструктирования ЭВМ, как именно решать задачу (пошагового описания алгоритма), вместо чего ему необходимо лишь выполнить постановку задачи некоторым формальным образом, задав существующие ограничения, то есть описать, что требуется получить в качестве результата. Происходит переход от «Как?» к «Что?». Декларативный подход является попыткой воплощения идеальной технологии программирования, в качестве которой может быть рассмотрена такая технология, когда по некоторому довольно неформальному описанию задачи автоматически генерируется синтаксически и семантически корректная программа решения. Поиск решения при этом возлагается на встроенную в систему программирования «машину вывода». Ярким примером декларативного подхода являются и языки семейства SQL – языки запросов к базам данных, в которых описывается, что нам надо извлечь из базы, а система управления базой данных сама осуществляет все необходимые для этого действия.

ЗАМЕЧАНИЕ

> В некотором смысле можно провести аналогию между исполнителем (в качестве которого выступает процессор, виртуальная машина, иногда — робот) в императивных языках и «машиной вывода», являющейся более «интеллектуальной» версией исполнителя.

Еще одно достоинство декларативных языков — пригодность для формальных рассуждений, которые, в частности, могли бы поднять методы обеспечения надежности и безошибочности в программировании на новый уровень (известно, что тестирование может помочь обнаружить ошибки в программах, но не может гарантировать их отсутствия). К сожалению, существующие методы формального анализа программ чрезвычайно трудоемки. Много усилий было направлено на создание более простых и

строгих языков, а также совершенствование методов анализа, но результаты не обнадеживают.

Декларативная программа обладает также следующим преимуществом. Она может быть подвергнута логическому анализу в своих собственных понятиях, для чего не требуется привлечение дополнительных формализмов, таких как предикаты на состояниях. Более того, возникает возможность создания инструментальных средств, позволяющих автоматизировать процессы анализа, преобразования и синтеза программ. Появление таких средств может в корне изменить программирование. При программировании на традиционном (императивном) языке у программиста нет иного выбора, кроме как вникать в низкоуровневые подробности выполнения программ. В декларативных языках логика и управление отделены друг от друга. Декларативную программу проще написать и, что немаловажно, проще понять, чем эквивалентную императивную программу. Помимо этого возникает возможность переложить ответственность за реализацию порядка выполнения операций на компьютер. Более того, становится возможным использовать нетривиальные алгоритмы планирования вычислений, такие как недетерминированные, «ленивые» и параллельные подходы.

К сожалению, универсального решения, позволяющего решать подобным образом произвольные задачи, не существует, иначе отпала бы необходимость в программировании и программистах. Тем не менее в конкретных предметных областях при наличии заранее подготовленных программных модулей, из которых может быть «собрана» требуемая программа решения, декларативный подход возможен. Довольно успешными реализациями декларативного подхода можно считать язык логического программирования Пролог, основанный на выводе в логике предикатов, а также, как уже говорилось, SQL — язык запросов к реляционным СУБД.

Идея *визуального программирования*, также называемого графическим, сводится к тому, что написание программы как текста заменяется в том или ином масштабе ее изображением в виде графической диаграммы (рисованием). В современных системах программирования просматривается тенденция к развитию средств, позволяющих программисту при создании программы оперировать не текстовыми, синтаксическими конструкциями, а графическими образами. Традиционный термин «писать программу» при этом трансформируется в «построить, проектировать программу». Описываться при этом могут самые разные аспекты программного комплекса. Такое богатство средств визуального представления обусловило возникновение некоторых трудностей при определении понятия визуального, или графического программирования. Существуют следующие варианты: «использование

графических средств разработки и визуальной метафоры при создании программного обеспечения» (Microsoft), «программирование, предусматривающее создание приложений с помощью наглядных средств», «графический язык программирования — любой язык программирования, в котором программы задаются путем графического манипулирования элементами взамен текстового манипулирования ими» (Википедия) и пр. Визуальное представление обладает рядом значительных преимуществ перед текстовым представлением, в частности, высокой наглядностью и удобством для человека, что позволяет достичь ряда целей, в том числе сокращения трудоемкости разработки, повышения качества и надежности создаваемых программ. В настоящем учебнике визуальным языкам программирования посвящена отдельная глава.

Попытки использования естественного языка в программировании имеют довольно долгую историю. С самого появления ЭВМ весьма привлекательной является идея общения с компьютером на обычном человеческом языке — русском, английском, японском и т. д. Действительно, как было бы здорово, придя с занятий, сказать домашнему компьютеру: «Рассчитай мне курсовую по физике!» — и получить готовый результат. Однако все усилия в этом направлении, несмотря на постоянно появляющиеся заявления вроде «еще чуть-чуть», «через пять лет» и т. д., не принесли решающего успеха. Это не кажется удивительным, если вспомнить, что, как показано еще в классической работе А. Тьюринга [3], создание компьютера, способного общаться на произвольные темы на не специально усеченной версии естественного языка, эквивалентно созданию искусственного интеллекта в полном смысле слова — машины, не уступающей разуму человека. Да и вопрос о самой принципиальной возможности создания искусственного интеллекта остается открытым.

К сожалению, естественный язык неполон, неточен, избыточен, и неоднозначен [4]. Тем не менее предпринят ряд успешных попыток применения некоторых ограниченных подмножеств естественного языка в определенных предметных областях (запросы к базам данных, искусственные миры, к примеру, мир кубиков [2]). Как правило, в таких случаях предложения должны строиться по заранее предписанным правилам, а передаваемый смысл — соответствовать некоторой точно определенной системе понятий.

Контрольные вопросы и упражнения

1. В чем заключаются особенности программируемых ЭВМ по сравнению с другими машинами?
2. Что называется системой команд компьютера?
3. Опишите возможности машины Беббиджа и программы Ады Лавлейс.

Когда была написана первая программа?

4. В какой стране и когда был построен первый компьютер?
5. Что представляет собой язык Plankalkül, разработанный К. Цузе?
6. Сколько языков программирования насчитывается сегодня? Многие ли активно используются? Выскажите свое мнение о причинах.
7. Перечислите преимущества и недостатки — программирования в машинных кодах.
8. Перечислите преимущества и недостатки — программирования на ассемблере.
9. Перечислите преимущества и недостатки программирования на языках высокого уровня.
10. Существуют ли основанные на русской лексике языки программирования? Приведите примеры таких языков.
11. В чем разница между интерпретатором и компилятором? Перечислите их преимущества и недостатки.
12. Каковы основные направления исследований в области языков программирования четвертого поколения?
13. Что такое предметно-ориентированные языки программирования? Перечислите их достоинства и недостатки.
14. Сколько поколений языков программирования известно по состоянию на 2013 год?
15. В чем состоит сущность декларативного подхода к программированию?
16. В чем состоят основная идея и преимущество визуального языка?
17. В чем заключаются особенности естественных языков? Что такое неполнота, неоднозначность, избыточность?
18. Перспективен ли естественный язык для постановки задач ЭВМ? Почему? Обоснуйте свое мнение.

Императивное программирование

После рассмотрения предыстории перейдем к более подробному описанию существующих и используемых сейчас языков программирования. Заметим, что они довольно сильно различаются между собой в зависимости от используемых базовых теоретических положений — так называемой *модели вычислений*.

Большинство языков традиционно принадлежат к так называемой *императивной модели*. В силу этого их иногда называют *традиционными*. Не совсем точным термином, однако используемым в сходном значении, является *процедурное программирование*.

Рассмотрим императивную модель более подробно. Она тесно связана с внутренним устройством большинства современных ЭВМ — так называемой фон-неймановской архитектурой.

ЗАМЕЧАНИЕ

В основе названия лежит исторический казус. Математик Джон фон Нейман не является изобретателем данной архитектуры — документ, в котором она описана, был издан под его научной редакцией. Впоследствии это стало даже предметом судебного разбирательства в США, в результате чего были защищены авторские права Эккерта и Моучли — конструкторов ЭВМ UNIVAC.

Описание фон-неймановской архитектуры

Устройство обычного современного компьютера представлено на рис. 2.

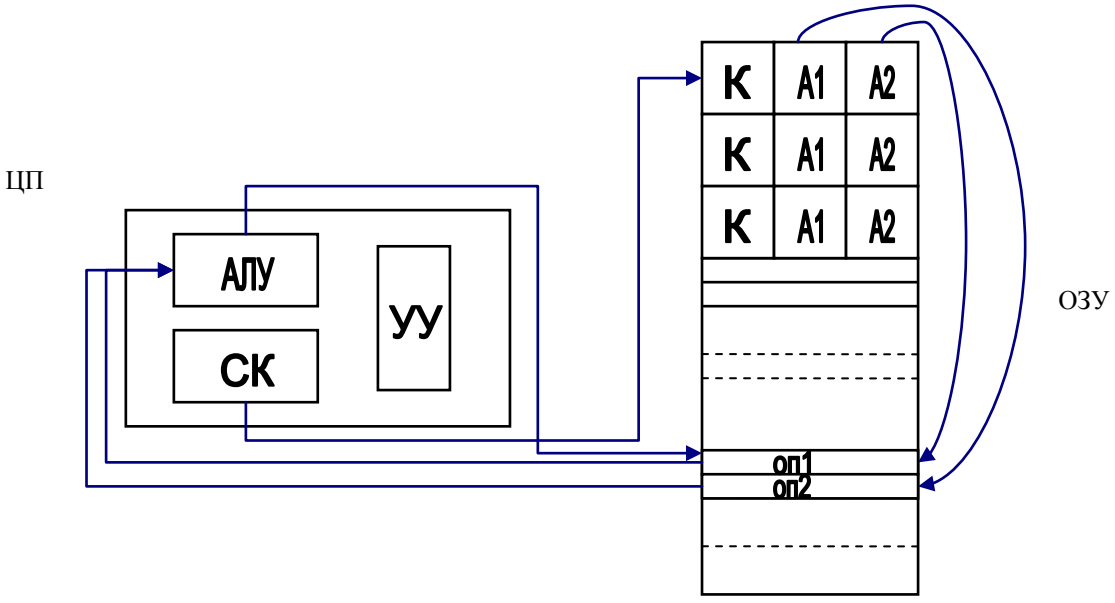


Рис. 2

Его ключевыми элементами являются:

- память — оперативное запоминающее устройство (ОЗУ);
- центральный процессор (ЦП);
- арифметико-логическое устройство (АЛУ), способное выполнять базовые операции преобразования данных;
- устройство управления (УУ);
- счетчик команд (СК);
- устройства ввода-вывода.

Память компьютера представляет собой линейный набор однородных пронумерованных ячеек. АЛУ и УУ — важнейшие узлы центрального процессора (ЦП), представляющего собой «мозг» ЭВМ. Еще один принципиально важный элемент ЦП — так называемый счетчик команд — специальный регистр, находящийся в процессоре, который содержит адрес (номер) ячейки памяти, хранящей следующую команду, подлежащую исполнению.

Команды и данные кодируются двоичными кодами и хранятся в одной и той же памяти. Универсальное двоичное представление означает, что по содержимому ячейки априори нельзя сказать, хранится ли в ней число, символ или команда. Команда может содержать несколько битов, отведенных под код действия — К (например, 110 — сложение), адрес первого операнда (оп1), например, слагаемого — А1, и адрес второго операнда (оп2) — А2. Данная особенность архитектуры позволяет создавать самомодифицирующиеся программы, изменяющие собственные команды — преобразующие их подобно данным (примером являются так называемые полиморфные вирусы или некоторые задачи «олимпиадного программирования»). Но у этой архитектуры есть и недостаток с точки зрения безопасности — программа при выполнении может непреднамеренно или целенаправленно (вредоносные программы) разрушить саму себя или другие программы, находящиеся в памяти! От подобного недостатка избавлена, например, архитектура отечественных суперЭВМ «Эльбрус» [7], в которой каждая ячейка снабжается специальным тегом, содержащим тип хранимой информации (кстати, в этом компьютере аппаратура поддерживала язык программирования высокого уровня Эль-76).

Все вычисления машины с фон-неймановской архитектурой выполняются централизованно — в АЛУ под управлением УУ. Рассмотрим цикл работы, связанный с выполнением очередной команды.

Команда извлекается из памяти, передается в процессор, где декодируется (дешифрируется) устройством управления. Из памяти с использованием

адресов, указанных в команде (например, A1 и A2), извлекаются операнды. УУ побуждает АЛУ выполнить операцию в соответствии с дешифрированным кодом операции К. Результаты операции пересылаются в память — по адресу, указанному в команде. После этого значение счетчика команд автоматически увеличивается — на следующем цикле извлечению из памяти подлежит следующая команда. Процесс повторяется.

В случае необходимости программа может продолжиться с другой команды, используются особые команды *переходов*, изменяющие значение счетчика команд принудительно. Команда перехода может быть безусловной или условной. Условная команда перехода модифицирует значение счетчика команд лишь в случае наличия того или иного выставляемого АЛУ флага, говорящего о том, что результат предыдущей операции, например, равен нулю или отрицателен.

В архитектуре фон Неймана есть узкое место — канал обмена процессора с памятью, о чем говорил в своей лекции при вручении премии Тьюринга ее лауреат, создатель языка программирования Фортран Джон Бэкус. Чем больше число объектов, с которыми оперирует программа, — данных и разнообразных операций над ними, — тем больше времени требуется, чтобы найти решение: приходится на каждом шаге передавать информацию от процессора в память и обратно. Так что какими бы скоростными не были процессор и память, общее быстродействие будет зависеть от возможностей канала обмена.

Анализируя работу фон-неймановской машины, подчеркнем еще раз следующие важнейшие моменты.

- Команды программы выполняются одна за другой в единственном центральном процессоре — это принципиально последовательная архитектура.
- После выполнения предыдущей команды автоматически выполняется следующая, если только предыдущая команда не была командой, изменившей счетчик команд (командой безусловного или условного перехода).
- Хранение команд и данных в одной и той же памяти, с одной стороны, дает определенную гибкость, с другой — создает проблемы с безопасностью и замедляет работу ЭВМ в целом.

ЗАМЕЧАНИЕ

Описанная организация вычислительного процесса — вовсе не единственно возможная. Так, в процессорах ARM, применяемых в смартфонах и планшетных ЭВМ, у команд — не только команд условного перехода — имеется набор флагов, дающих возможность проверить на каждом шаге

состояние вычислительного процесса и обусловить этим выполнение или невыполнение данной команды. Аналог подобного подхода на более высоком уровне программирования — таблицы решений [8]. Спроектированный и построенный в 1946 году в Великобритании Аланом Тьюрингом компьютер ACE включал в команду адрес следующей подлежащей исполнению команды.

Все же можно с полным основанием назвать описанную модель вычислений традиционной — именно так работает подавляющее большинство современных ЭВМ и именно на нее ориентированы большинство из существующих языков программирования.

Контрольные вопросы и упражнения

1. Что входит в понятие модели вычислений?
2. Почему архитектура большинства современных компьютеров носит имя фон Неймана? Заслуженно ли это?
3. Опишите основные элементы «фон-неймановской» архитектуры ЭВМ. В чем ее недостатки?
4. Процедурная, императивная и традиционная парадигмы программирования — это одно и то же?
5. Из каких этапов состоит цикл работы «фон-неймановского» компьютера?
6. Существуют ли альтернативные архитектуры ЭВМ? В чем их преимущества и недостатки?
7. Представьте себя инженером-конструктором компьютера. Как бы вы построили ЭВМ?

Базовые понятия и конструкции императивных языков

Программа на императивном языке содержит последовательность предписаний, или инструкций, которые должен выполнить компьютер. Эти предписания в ассемблере принято называть командами, а в языках высокого уровня — *операторами*. Фрагмент программы может выглядеть следующим образом:

<*оператор*$_1$>
<*оператор*$_2$>
<*оператор*$_3$>
…
<*оператор*$_n$>

В некоторых языках программирования операторы нумеруются.

Пример на языке Бейсик:

```
10 INPUT "Введите A и B";A,B
20 D=A+B
30 PRINT "Сумма введенных чисел равна",D
```

Данная программа может быть исполнена интерпретатором Бейсика, позволяя подсчитать сумму двух введенных чисел. Номер строки позволяет ссылаться (продолжить исполнение с нее) на строку с определенным номером.

Пример на языке Бейсик:

```
GOTO 20
```

Сейчас, однако, нумерация строк — скорее анахронизм. Вместо них используется аппарат *меток*, являющихся необязательным, но допустимым префиксом строки программы, что позволяет нарушить естественный ход выполнения программы, выполнив *переход* к метке.

Пример на языке Си:

```
goto label;
```

Пример на языке Java:

```
break label;
```

Пример на ассемблере PDP-11:

```
BR MET
```

ЗАМЕЧАНИЕ

> В некоторых из современных языков высокого уровня переходы считаются злом, метки в них могут принципиально отсутствовать.

Приведенная программа выглядит вполне лаконично и исчерпывающе. Однако в большинстве современных языков, включая языки ассемблера, помимо собственно перечня выполняемых действий программа в обязательном порядке содержит еще несколько секций, или разделов (имеющих свое предназначение блоков текста).

Часто требуется указать имя программы, после чего используются ключевые слова наподобие `begin` и `end`.

Пример на языке Паскаль:

```
PROGRAM MYPROGRAM;
BEGIN
  (* программа *)
END MYPROGRAM.
```

При этом программа может структурироваться на меньшие части — подпрограммы, которые называются функциями, процедурами или

классами в зависимости от языка. Каждая подпрограмма, в свою очередь, имеет свое собственное имя, начало и конец.

Современные программы, как правило, являются частью некоторого программного комплекса и взаимодействуют с другими его элементами. Даже простая программа обычно использует стандартные так называемые библиотечные модули, позволяющие решать типовые задачи, например вычислять синус, без необходимости каждый раз изобретать велосипед. В связи с этим в программе чаще всего есть блок, в котором описано, каким образом программа взаимодействует с другими (какие библиотечные модули импортируются и откуда, какой интерфейс имеет программа). В интерфейсной части обычно описывается, какие данные поступают на вход и какие получаются в результате выполнения программы или подпрограммы.

Описание обрабатываемых данных является важнейшей частью программы во многих языках программирования. Известно высказывание известного швейцарского ученого Никлауса Вирта: «Программы = Алгоритмы + Структуры данных» [9]. Поэтому помимо собственно действий, отражаемых алгоритмической частью (она еще может называться процедурной), программа содержит объявление данных, которые подлежат обработке. В простейшем случае это перечисление всех встречающихся в дальнейшем переменных с указанием их типов.

Перечисленные непроцедурные, или декларативные, части программы чаще всего находятся в ее начале, а также в начале каждой подпрограммы. Иногда декларативная часть выносится (не полностью) в исходные файлы специального типа.

Но вернемся к процедурной части. Именно она содержится внутри функций и процедур и называется *телом* функции, чем подчеркивается отличие от заголовка подпрограммы, включающего имя и список входных и выходных данных.

Несмотря на то что изначально в некоторых языках допускалось использование строго одного оператора в строке и значение имела даже позиция (колонка), в которой он начинался, а в языках ассемблера до сих пор принят принцип «одна строка — одна команда», большинство современных языков программирования допускают запись в одной строке нескольких операторов:

<оператор$_1$>; <оператор$_2$>;
<оператор$_3$>
…
<оператор$_n$>

В этом случае необходимо понять, где в данной строке заканчивается один

оператор и начинается другой. Точка с запятой в примере использована не случайно — это весьма популярный символ, используемый во многих современных языках программирования для отделения одного оператора от другого.

Независимо от того, один оператор записан в строке или несколько, в данном случае мы имеем дело с так называемой *линейной программой*, или *линейным участком*. Графически он может быть представлен следующим изображением (рис. 3).

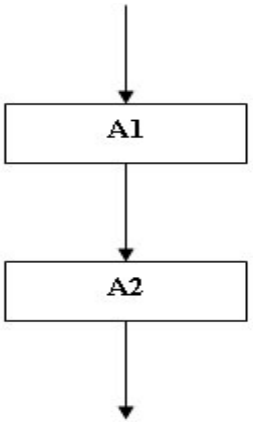


Рис. 3

При выполнении каждого встреченного в строке императивной программы предписания некоторым образом меняется состояние ЭВМ (содержимое ячеек памяти). При этом можно сказать, что мы имеем *фокус управления*, находящийся при выполнении программы в том или ином известном месте и определяющий, какое действие будет выполняться следующим.

Какие виды операторов встречаются в программах? Фундаментальное значение в императивной парадигме программирования имеет так называемый оператор присваивания, меняющий значение некоторой переменной в памяти ЭВМ. Приведем примеры операторов присваивания из разных языков программирования.

Пример на языке Бейсик:

```
LET X=X+1
```

Пример на языке Фортран:

```
A=B+C
```

Пример на языке Си:

```
b=sin(x)+cos(x)/(-2)*y;
```

Пример на языке РАПИРА:

```
К*СУММАНАЛОГА → М;
```

Оператор присваивания имеет две части. В одной из них содержится имя

переменной, принимающей в результате выполнения оператора присваивания некоторое значение. В другой содержится константа, имя другой переменной или записанное в соответствии с правилами языка выражение, значение которого вычисляется и присваивается. Как видно из примеров, в выражении допускаются вызовы функций.

Помимо исполнения оператора в программе может встретиться другой вид действия, а именно вызов подпрограммы с указанием ее имени или номера строки, которой нужно передать управление. Подразумевается, что при его исполнении произойдет передача управления иным функции, процедуре, классу нашей программы или же библиотечной подпрограмме. В вызванной функции, в свою очередь, должен обязательно встретиться оператор возврата управления, по выполнении которого вызывающая программа продолжится со следующей за вызовом подпрограммы строки.

Условный оператор и оператор выбора

Если бы все программы были линейными (их операторы выполнялись строго в однажды заданном порядке, без альтернатив), их возможности были бы сильно ограниченными. Несомненно, реальные задачи могут содержать множество альтернатив, и хочется, чтобы программа могла обрабатывать разные случаи и ситуации (вспоминая русские сказки: «направо пойдешь — коня потеряешь», «налево пойдешь…» и пр.). В языках программирования с этой целью используются *условный переход, условный оператор* и *оператор выбора*.

Условный оператор — конструкция, позволяющая поставить выполнение тех или иных действий в зависимость от выполнения некоторых условий. Различают несколько форм условного оператора. Обычный вариант, содержащий записываемоме по правилам языка программирования условие, имеет две разновидности *полную* и *сокращенную*. В полной форме указывают два действия — первое, подлежащее выполнению, если условие выполняется, и второе — выполняемое, если условие ложно.

Пример на языке Паскаль:

```
IF D>=0 THEN VYCHKORNI()
ELSE WRITELN('Действительных корней у данного уравнения нет');
```

В некоторых языках (например, Nemerle, какой-то из форм, сокращенной или полной, может не быть).

Графически условный оператор принято изображать в виде ромба, в котором записывается условие, а из боковых углов исходят линии, соответствующие ветвям выполнения ДА и НЕТ (рис. 6). На блок-схеме видно условие P (предикат) и два действия, S1 и S2, подлежащие выполнению при истинности и ложности условия.

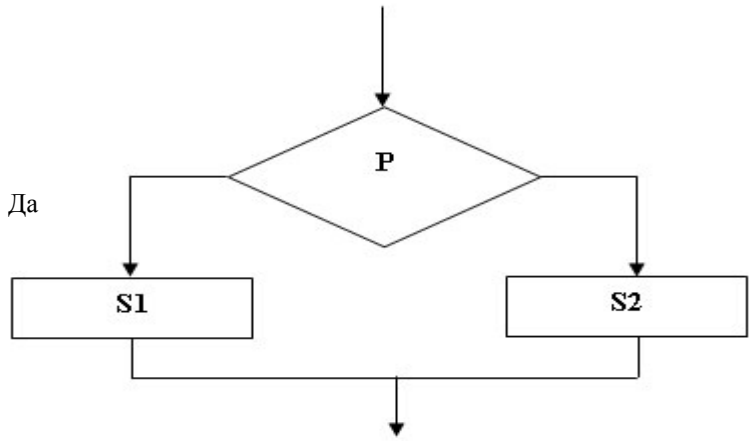


Рис. 4

В сокращенной форме условного оператора приводится лишь действие, которое необходимо выполнить в ситуации, когда условие истинно. В противном случае просто ничего не выполняется, лишь происходит передача управления следующему по порядку записи программы оператору.

Пример на языке РАПИРА:

```
ЕСЛИ ОСТДЕНЕГ>0 ТО ВЫДАТЬ_ДЕНЬГИ();
```

В качестве условия P могут выступать переменная специального логического типа или некоторое выражение, могущее быть истинным или ложным, например, в случае сравнения двух величин.

Условие в операторе может быть и сложным, в нем могут присутствовать логические связки И, ИЛИ, НЕ.

В некоторых языках программирования применяется *условный переход*. В этом случае действием является передача управления оператору с заданным номером или помеченному указанной в операторе условного перехода меткой. В следующем примере в качестве действия, выполняемого при истинности условия, использован оператор перехода `goto`.

Пример на языке Си:

```
if (summa<0) goto oshibka;
```

Любопытен так называемый *арифметический* условный оператор. В ранних версиях языка Фортран имелся лишь данный вариант условного оператора, который затем был дополнен описанными выше разновидностями. Выглядит он следующим образом: `IF (E) Mneg , Mzero , Mpos`. Здесь `E` – некоторое выражение, значение которого вычисляется (должно быть числом). Затем перечислены три метки программы, переход к которым происходит соответственно, если результат

меньше нуля, равен нулю и больше нуля. Ниже приводится пример фрагмента программы на Фортране, вычисляющей корни квадратного уравнения

```
      DN = B*B - 4*A*C
      IF (DN) 90,10,10
   10 D = SQRT(DN)
      X1 = (-B + D) / (2*A)
      X2 = (-B - D) / (2*A)
```

В языках уровня ассемблера обычно не предусматривается разветвление программы по условию одной командой. В них подобное действие разбивается на два:

1. Установка флагов процессора - либо автоматически арифметическими и логическими командами при равенстве результата нулю, переносе в следующий разряд, получении отрицательного значения, и в других подобных ситуациях, или с помощью использования специальных команд тестирования ячейки и сравнения двух величин.
2. Переход по значению заданного условия (флага).

В случае если условие не выполняется, происходит переход к следующей команде.

Пример на языке ассемблера PDP-11:

```
BIT  #1,A ; проверка первого разряда A
BEQ  KZ1; переход, если равен нулю
```

или

```
CMP  NOB,#33 ; сравнение NOB с числом 33
BLT  KZ1; переход, если NOB < 33
```

На практике встречаются случаи, когда возможных ситуаций не две, а больше (вспомним камень на развилке из сказки, обычно там предусмотрены три варианта). Во многих языках программирования существует специальный оператор для поддержки подобных случаев — *оператор выбора*, или переключатель. Как правило, для ветвления используется набор допустимых значений некоторой переменной (заметим, что переменная должна быть *перечислимого*, например целого, типа), однако возможен и более универсальный вариант с возможностью указания произвольного условия для каждого варианта, подлежащего обработке. Каждому указанному в операторе выбора значению переменной сопоставляется своя ветвь выполнения программы. Еще одна ветвь может присутствовать для случая, если значение переменной, по которому производится выбор, не соответствует ни одному из указанных.

Пример на языке Паскаль:

```
case NUM of
  0: writeln ('Нуль');
  1: writeln ('Один');
  2: writeln ('Два');
  3: writeln ('Три');
  4: writeln ('Четыре');
end;
```

Пример на языке РАПИРА:

```
ВЫБОР ИЗ
      А>Б: А-Б→А !
      А=Б: 0 → Б !
      А<Б: Б-А→А
      ИНАЧЕ ?"ТАК НЕ ДОЛЖНО БЫТЬ"
ВСЕ
```

Повторное исполнение — рекурсия и итерация

В программах может возникать необходимость повторяющегося выполнения некоторых действий. Для этого можно использовать два механизма — *итерацию* и *рекурсию*.

Рекурсия означает, что некоторая подпрограмма — прямо или косвенно через другие подпрограммы — вызывает на выполнение саму себя, реализуя принцип змеи, кусающей себя за хвост. Многие математические понятия, например факториал, определяются с использованием рекурсии: факториал `N` — это умноженный на `N` факториал `N-1` для `N>0`, факториал нуля принимается равным единице.

ЗАМЕЧАНИЕ

> Чтобы рекурсия не продолжалась вечно, необходима проверка так называемого условия останова рекурсии, или граничного условия.

Пример рекурсивного вычисления факториала на языке Си:

```
/* Рекурсивное вычисление факториала */
long int fac(int N)
{
  if (N==0) return 1;
  if (N>18) {printf("Слишком большое число!\n\a");return -1;};

  return N*fac(N-1);
}
```

Через рекурсию определяются знаменитые числа Фибоначчи, менее известные числа Люка и многие другие замечательные математические

объекты, что позволяет писать для их вычисления лаконичные и прозрачные по смыслу программы.

ЗАМЕЧАНИЕ

> Часто рекурсивные программы весьма лаконичны, непосредственно передают смысл задачи, изящны по сравнению с решающими ту же задачу итеративными.

Любопытно, что формула для прямого вычисления чисел Фибоначчи и Люка тесно связана с понятием *золотого сечения*, по мнению многих, являющегося воплощением красоты и гармонии. Не связано ли это магическим образом с красотой рекурсивных программ?

Следует отметить, что вызов подпрограммы — это некоторое количество дополнительных действий, причем на каждый рекурсивный вызов для запоминания точки возврата требуется некоторое количество оперативной памяти. При чрезмерно большой глубине рекурсии может наступить ошибка типа «переполнение стека вызовов». В отличие от этого поддержка итерации в ЭВМ архитектуры фон Неймана более проста и естественна. В некоторых языках программирования (Кобол, Бейсик, Фортран ранних версий) рекурсивные вызовы запрещены.

В других языках, напротив, отсутствует поддержка итерации, и реализовать повторное выполнение одних и тех же действий можно только с использованием рекурсии. Большинство современных языков программирования допускает применение как итерации, так и рекурсии.

Конструкции, обеспечивающие итерацию, называются циклическими конструкциями. Примером может служить цикл ПОКА, обеспечивающий выполнение S, пока выполняется логическое условие P. Цикл ПОКА представлен на рис. 5.

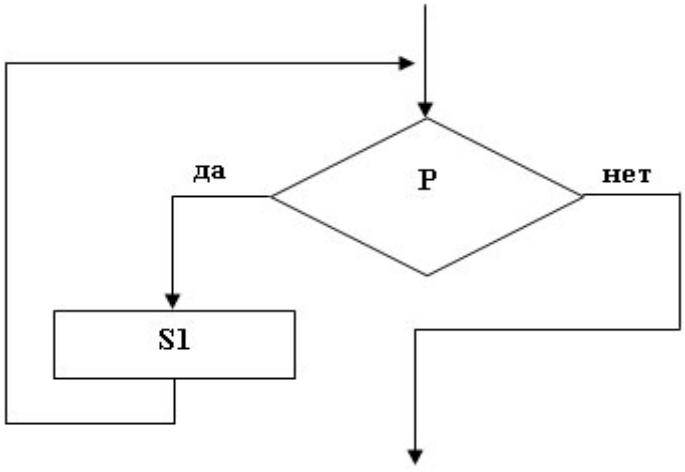


Рис. 5

Действие S1 называют *телом цикла*. ПОКА еще называют циклом с предусловием. Обратим внимание на следующие важные моменты:

- Цикл с предпроверкой может быть не выполнен ни разу — в случае, если условие P изначально ложно.
- При выполнении цикла программа может войти в бесконечное исполнение в случае, если условие P изначально истинно, а выполнение тела цикла не влияет на истинность условия. Эта ситуация называется *зацикливанием*.

Пример на языке Си:

```
/* Умножение на два и печать значения, пока не дойдем до тысячи */
while (a<1000)
{
  a*=2;printf("уже дошли до a=%d!\n",a);
}
```

Пример вычисления факториала с помощью итерации на языке Си:

```
/* Итеративное вычисление факториала */
long int fac(int N)
{
  int i;
  long F;

  if (N==0) return 1;
  if (N>18) {printf("Слишком большое число!\n\a");return -1;};

  i=F=1;
  while (i<=N) F*=i++;

  return F;
}
```

В языках программирования встречаются и другие виды циклических конструкций. Пример — цикл с постусловием, или цикл ДО. Графически он может быть изображен таким, как на рис. 6.

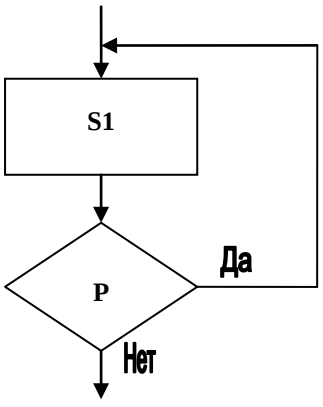


Рис. 6

Пример на языке Си:

```
/* ввод чисел и суммирование до тех пор, пока не встретим ноль */
do {
  scanf("%d",&a);s+=a;
}
while (a!=0);
```

В некоторых языках встречается цикл с прямым указанием количества повторений.

Пример на языке РАПИРА:

```
ПОВТОР 5 РАЗ :: ?"ЭТО ПРАВДА?!" ВСЕ
```

Пример на языке Лого (русская версия):

```
ПОВТОРИТЬ 5
      { x=x+1}
```

Весьма популярной конструкцией является цикл с параметром (цикл ДЛЯ, цикл со счетчиком), связанный с такой широко используемой в программировании структурой данных, как массив. При этом используется некоторая переменная, называемая *переменной цикла*. До первой итерации она инициализируется некоторым первоначальным значением. По окончании каждой итерации переменная цикла меняется заданным образом, после чего полученное значение сравнивается с указанным граничным. Если граничное значение не достигнуто, цикл повторяется еще раз, в противном случае — завершается. Популярное начальное значение — единица, она же является типичным приращением. Некоторые языки программирования допускают произвольные приращения, некоторые (например, Паскаль) — лишь +1 и –1 (инкремент и декремент). При обработке массива переменная цикла используется в качестве индекса обрабатываемого элемента массива.

Пример на языке программирования Бейсик:

```
10 FOR I=1 TO 10 STEP 2
20 K[I]=1
30 NEXT I
```

В этом примере единица присваивается элементам массива K через один.

Современные языки программирования, например C# и Java (а также, например, язык РАПИРА), имеют конструкцию цикла ДЛЯ … ИЗ, которая перебирает все элементы некоторой структуры данных и применяет к ним заданное действие без необходимости использовать специальную переменную цикла.

Пример на языке C#:

```
foreach (int element in fibarray)
{
  element += 11;
}
```

Пример на языке РАПИРА:

```
ДЛЯ БУКВА ИЗ КНИГА ::
       ЕСЛИ БУКВА="А" ТО СЧ+1->СЧ ВСЕ
ВСЕ;
```

Структурное программирование

В данной главе активно использовались рисунки для иллюстрирования различных операторов языков программирования. Графические обозначения соответствуют языку блок-схем алгоритмов и программ. Обратите внимание на то, что действия представлены прямоугольными блоками. А также на то, что у каждой иллюстрирующей использование оператора схемы был только один вход и один выход. Представим теперь, что мы эту схему обвели прямоугольной рамкой. У рамки будет одна входящая линия сверху и одна выходящая — снизу. Если теперь заменить прямоугольный блок построенным блоком в рамке, вы поймете идею так называемого строгого *структурного программирования*.

Структурное программирование запрещает использование произвольных переходов к метке, за счет чего программа приобретает более упорядоченный характер, что, с точки зрения апологетов данного подхода, позволяет создавать более качественные и надежные программы. Программы, изобилующие командами безусловного перехода `GOTO`, в логике которых из-за этого практически невозможно разобраться, получили в среде сторонников структурного программирования обидное прозвище спагетти.

Теоретической основой структурного программирования принято считать принципы, изложенные в классической работе Бома и Джакопини. Эта работа на итальянском языке была опубликована в 1965 году, а в английском переводе — в 1966 году. В соответствии с так называемой структурной теоремой, сформулированной и доказанной в работе, всякая программа может быть построена с использованием только трех основных типов блоков — линейного, условного и циклического.

ЗАМЕЧАНИЕ

> Существует также строгое теоретическое доказательство того, что цикла ПОКА и последовательной композиции достаточно для написания программы с произвольной по сложности логикой.

У структурного программирования есть и критики, в основном среди практикующих программистов. Они (например, Кнут [10]) указывают на то, что с использованием операторов безусловного перехода `goto` во многих случаях удается писать более эффективные программы. Типичная, хотя и не единственная ситуация — необходимость обеспечить выход из глубины вложенных один в другой циклов.

ЗАМЕЧАНИЕ

> Существует строгое теоретическое доказательство того, что последовательной композиции и условного перехода достаточно для написания программы с произвольной по сложности логикой.

Разумно, видимо, не доходя до фанатизма, для получения более упорядоченных и понятных программ применять как идеи структурного программирования, так и в оправданных случаях — условные и безусловные переходы.

Мы рассмотрели основные виды операторов современных императивных языков программирования высокого уровня:

- оператор присваивания;
- условный оператор и оператор выбора;
- операторы циклов.

Особняком стоит вызов подпрограммы (процедуры, функции). В некоторых языках для этого используется специальный оператор, в других — просто записывается имя функции или процедуры со списком параметров.

Какие еще операторы встречаются в программах? *Пустой* оператор, который ничего не делает, зато может быть помечен и на него может быть передано управление. Во многих языках имеются операторы *ввода-вывода*, позволяющие обмениваться информацией с оператором. В примерах использован оператор языка Бейсик `PRINT`, выводящий данные на экран (некоторые языки, например Си, не содержат встроенных операторов ввода-вывода, для обмена информацией применяются библиотечные функции, в частности `printf()`).

Упоминали мы также операторы безусловного и условного перехода к метке. Наряду с операторами цикла, условным оператором и оператором выбора они являются *управляющими*, то есть используемыми для изменения порядка выполнения действий в программе.

Использование лишь конструкций, допустимых с точки зрения структурного программирования, вынуждает применять вспомогательные операторы. Так, имеется оператор прерывания текущей итерации цикла и

переход к следующей, не доходя до конца тела цикла в тексте (`continue` в Си). Есть оператор, прерывающий выполнение текущего структурного блока и вызывающий продолжение исполнения программы с места начала следующего блока в тексте программы (`break` в Си).

Исключения

Одной из популярных современных идей, относящихся к механизмам управления выполнением императивных программ, являются так называемые *исключения*. При выполнении программы возможны различные нештатные ситуации (в вычислениях — попытка деления на ноль, переполнение и пр., при работе с файлами — запись на защищенный диск, при поиске данных — отсутствующая запись и т. д.). Подобные случаи традиционно отслеживались с помощью условного оператора и оператора выбора. В наихудшем варианте, если программист не учел возможности возникновения непредвиденной ситуации, происходит «вылет» из программы в операционную систему, часто с невразумительным сообщением об ошибке, да еще и на английском языке. Механизм исключений предназначен специально для выявления нештатных вариантов исполнения. Блок с исключениями выглядит при этом следующим образом:

```
Попытайся
Начало-блока-с-исключениями
{
<Действия при выполнении которых может возникнуть ошибка>
Если (условие1) создай исключение1
Если (условие2) создай исключение2
…
Если (условиеn) создай исключениеn
}
В-случае-Ошибки1 выполни <обработка исключения1>
В-случае-Ошибки2 выполни <обработка исключения2>
…
В-случае-Ошибкиn выполни <обработка исключенияn>
Конец-блока-с-исключениями
```

Существует два принципиально разных механизма функционирования обработчиков исключений. *Обработка с возвратом* подразумевает, что обработчик исключения ликвидирует возникшую проблему и приводит программу в состояние, когда она может работать дальше по основному алгоритму. В этом случае после того, как будет выполнена обработка, управление передается обратно в программу, где возникла исключительная ситуация. Обработка с возвратом типична для обработчиков асинхронных

исключений (которые обычно возникают по причинам, не связанным прямо с выполняемым кодом), для обработки синхронных исключений она малопригодна. *Обработка без возврата* заключается в том, что после выполнения обработки исключения управление передается в некоторое заранее заданное место программы и с него продолжается исполнение. То есть при возникновении исключения команда, во время работы которой оно возникло, заменяется безусловным переходом к заданному оператору.

Пример на языке C#:

```
try
{
      result = SafeDivision(a, b);
      Console.WriteLine("{0} деленный на {1} = {2}", a, b, result);
}
catch (DivideByZeroException e)
{
      Console.WriteLine("Попытка деления на ноль!");
}
```

ЗАМЕЧАНИЕ

> По многим свойствам исключения в языках программирования походят на прерывания, относящиеся к механизмам реализации управления вычислительным процессом, в частности, в операционных системах ЭВМ.

Процедурное программирование

Базовыми действиями в императивном программировании остаются присваивание и вызов подпрограммы. В свое время использование подпрограмм стало одной из основополагающих идей и даже получило самостоятельное название — программирование с использованием процедур, или *процедурное программирование*.

Процедурное программирование дает возможность *повторного использования* кода и в некотором смысле реализует принцип матрешки. Процедура — блок, который раскрывается в другом месте, а из данной программы лишь вызывается (указанием ее имени).

ЗАМЕЧАНИЕ

> Процедурное программирование — лишь одна из вариаций древнейшего принципа решения сложных задач «разделяй и властвуй!». Принцип заключается в разбиении сложной задачи на менее сложные подзадачи, каждая из которых либо также разбивается на еще более простые, либо решается доступными средствами.

Структурируются современные программы на части несколькими способами. Программа может делиться на несколько отдельных файлов с

исходными текстами, классы, пакеты, процедуры, модули, функции, пространства имен и т. д. (это зависит от языка, в некоторых из них возможны несколько вариантов разбиения). Пока будем использовать для всех них единый термин — *модуль* (в обобщенном смысле).

В программировании используются несколько принципов разделения программы на модули. Один из них требует, чтобы размер каждой отдельной процедуры (функции, класса) был обозримым, не превышая, скажем, двух стандартных страниц. Другие обоснованные соображения деления программы на модули основываются на функциональной законченности, или разбиении программы на части в соответствии с выполняемыми самостоятельными функциями. При этом руководствуются принципами сильной связи *внутри* модуля и слабой связи *между* модулями (которая тем не менее должна быть достаточной для решения программой поставленных перед нею задач). Слабая взаимозависимость модулей позволяет более удобно заменять версию данного модуля в случае необходимости (нахождение и исправление ошибки или доработка программы) и распределеять задачи программирования в коллективах — а современные программы часто создаются десятками, а то и сотнями разработчиков.

ЗАМЕЧАНИЕ

> Поскольку вызов процедуры требует некоторых затрат времени процессора и памяти, иногда целесообразна замена некоторых процедур малого размера так называемыми встроенными (inline) функциями. Фактически, процедура превращается в дублирующиеся в нескольких местах блоки команд, что позволяет повысить скорость исполнения программы.

Иногда в языках программирования используются так называемые *замыкания* — фактически, вспомогательные функции, определяемые и используемые внутри родительской функции и использующие ее локальные переменные, а не только свои переменные-параметры. В программе замыкание выглядит как функция, находящаяся целиком в теле другой функции. При этом вложенная функция содержит ссылки на переменные внешней. Каждый раз при выполнении родительской функции происходит создание нового экземпляра внутренней функции с новыми ссылками на переменные внешней функции. При этом ссылки на переменные внешней функции действительны до тех пор, пока работает вложенная функция, даже если внешняя функция закончила работу.

Пример на языке PHP:

```
function outer($x) //Определение внешней функции
{
        $y=2; //Локальная переменная внешней функции
```

```
        $inner=function ($a) use ($x, $y) //Определение внутренней
функции
        {
                $b=4; //Локальная переменная внутренней функции
                /* А дальше складываются переменные внутренней и
                 * внешней функций, как будто все они локальные
                 * переменные внутренней функции
                 */
                $res=$x+$y+$a+$b;
                echo $res; //Результат 10 в нашем примере.
        };
        $inner(3); //Вызов внутренней функции
}
outer(1);
```

Механизм замыканий тесно связан с *лямбда-функциями* (безымянными функциями, использующими текущий контекст программы). Первоначально они нашли широкое применение в функциональном программировании (языки Лисп — см. соответствующий раздел настоящей книги, Хаскелл, Scheme), но затем получили распространение и в языках императивной парадигмы (Ruby, PHP, Питон и др.).

Пример на языке Питон:

```
(lambda x: x*2)(3) →  6
```

Контрольные вопросы и упражнения

3. Как выглядит императивная программа?
4. В чем назначение номеров строк в программе? В чем назначение меток?
5. Что такое процедурная и декларативная секции программы?
6. Что такое линейная программа? Приведите примеры.
7. В чем состоят функции и какова структура оператора присваивания?
8. В чем заключается назначение и какова структура условного оператора?
9. В чем заключается назначение, каковы варианты и структура оператора выбора?
10. Каким образом можно организовать повторение одних и тех же действий в программе?
11. Что такое рекурсия? В чем назначение граничного условия в рекурсивных программах?
12. Что такое итерация? В чем преимущества и недостатки итерации по сравнению с рекурсией?

13. Какие виды операторов циклов существуют в языках программирования?
14. В чем заключаются особенности цикла ПОКА? Приведите пример применения.
15. В чем заключаются особенности цикла ДО? Приведите пример применения.
16. В чем заключаются особенности цикла ДЛЯ? Приведите пример применения.
17. В чем заключаются особенности цикла ДЛЯ … ИЗ? Приведите пример применения.
18. В чем заключаются особенности цикла ПОВТОР N РАЗ. Приведите пример применения.
19. В чем заключается теорема Бома —Джакопини?
20. В чем заключается основная идея структурного программирования? Согласны ли вы с ней?
21. Какие существуют дополнительные виды операторов управления в программах? Приведите примеры их использования.
22. Зачем нужно повторное использование кода?
23. В чем заключается принцип встраиваемых функций (inline). Каковы условия их применения? Приведите пример.
24. Что такое исключения в программах? Зачем нужен механизм исключений? Приведите пример использования механизма исключений.
25. Какие виды исключений в программах вы знаете? Какие альтернативные способы обработки нештатных ситуаций можете предложить?
26. Что такое замыкание? Всегда ли создается экземпляр замыкания при выполнении родительской функции? Существует ли связь между замыканиями и лямбда-функциями?

Структуры данных в программировании

> Программы = Алгоритмы + Структуры данных.
>
> *Никлаус Вирт*

В этой главе пойдет речь о таком важном аспекте современных информационных технологий, как структуры данных в языках программирования. Это, если можно так выразиться, оборотная, но необходимая сторона медали. Действительно, хотя алгоритмы определяют,

как и в каком порядке происходит обработка, структуры данных должны четко описывать, что именно обрабатывается, — и на это описание затрачивается значительная часть усилий программиста. И если, как говорит Никлаус Вирт, программы — это алгоритмы плюс структуры данных, в языках программирования должны быть средства описания алгоритмов и средства описания данных.При этом то, насколько удобно и эффективно можно описать данные, зависит от встроенных в язык программирования возможностей.

Некоторые языки программирования требуют, чтобы до их использования все переменные, аргументы и возвращаемые значения функций и процедур были объявлены с указанием типа (Алгол, Ада, Паскаль, C/C++). Другие (Лисп, РАПИРА, PHP) этого не требуют, переменная в них приобретает тип в зависимости от первого присвоенного ей значения. При этом в большинстве языков программирования переменные могут иметь только один тип во все время их существования. Но в некоторых тип может меняться во время исполнения программы. В первом случае говорят о *статической*, во втором — о *динамической типизации*.

Языки программирования могут быть как с явной, так и с неявной системой типов. В первом случае четко фиксируется, к какому типу относится та или иная переменная или константа, во втором об этом отдельно речь не идет, тем не менее данные обычно обрабатываются по-разному. И у первого, и у второго подхода имеются как достоинства, так и недостатки. Динамическая неявная типизация расширяет возможности использования переменных и избавляет программиста от хлопот, связанных с объявлением и контролем соответствия типов. Кроме того, данный подход позволяет создавать более универсальные и проще интегрируемые программные модули. Однако это требует от программиста большей ответственности, поскольку во время выполнения программы возможны неожиданности вроде попыток перемножить две строки или разделить число на логическое значение. При статической типизации такие ошибки обнаружились бы на этапе компиляции. И чем больше размер программы, тем более острой становится эта проблема.

В некоторых языках само имя переменной свидетельствует о типе переменной. Например, в некоторых диалектах Бейсика имя переменной A% свидетельствует, что она целочисленная, а A$ — что строковая. В языке Perl переменная в зависимости от типа получает значок перед именем: для скалярных переменных это $, для массивов — @, для ассоциативных массивов — %.

ЗАМЕЧАНИЕ

Переменная в императивных языках программирования фактически является синонимом ссылки на объект, способный в различные моменты времени

принимать разные значения данного типа. Можно провести параллель с ячейкой памяти ЭВМ.

Контрольные вопросы

1. Почему нужны возможности описания данных в языках программирования?
2. В чем заключается разница между статической и динамической типизацией? Какие существуют способы неявного задания типа данных в языках программирования.
3. Как влияет вариант типизации на надежность программ?

Простые типы данных

Под простыми, или базовыми, типами данных в языках программирования понимаются объекты (переменные или константы), которые не имеют доступной программисту внутренней структуры.

В различных языках имеются разные наборы встроенных базовых типов данных. Так, в Си он минимален — целое число, число с плавающей точкой, одиночный символ. В C++ уже есть специальный логический тип с допустимыми значениями лишь ИСТИНА и ЛОЖЬ (`true` и `false`). Любопытно, что в первом языке программирования высокого уровня Фортран, изначально предназначавшемся для научных расчетов, в базовый набор типов данных входят комплексные числа.

Числа в языках программирования

Начнем рассмотрение с наиболее распространенного и имеющегося практически во всех языках программирования типа данных — целых чисел. При объявлении переменных используются следующие ключевые слова:

- `int` — языки Си, C++, Java и др.;
- `integer` — Паскаль, Модула, Алгол, Ада, Фортран и др.

ЗАМЕЧАНИЕ

Настоящих, в математическом смысле чисел в программах нет. В ЭВМ, имеющих ограниченную память, непредставимы не только действительные числа произвольной точности, но и целые, которые теоретически могут иметь бесконечно большое значение. В языках программирования используют приближения понятия числа с помощью заданного в системе количества битов. Если при выполнении вычислений должно быть получено большее число, чем допускает данный тип, происходит ошибка переполнения.

В некоторых языках существует тип для неотрицательных целых (натуральных) чисел, например `cardinal` в Модуле. В других допустимо

перед наименованием целого типа указывать модификатор, указывающий, например, что число заведомо неотрицательное (`unsigned`), — это позволит за счет использования знакового разряда тем же набором бит представить число большей абсолютной величины. Допустимы и другие модификаторы, позволяющие при необходимости увеличить или уменьшить отводимое под хранение переменной число двоичных разрядов.

Возможная реализация целочисленных типов в Си и диапазоны значений (в большинстве современных реализаций следует умножить приводимые значения на два, поскольку они ориентированы изначально на 32-разрядную архитектуру):

```
int          16 бит      -32768..+32767
unsigned int  16 бит       0..65535
short int      8 бит       -128..+127
unsigned short 8 бит       0..+255
long int      32 бит       -2147483648..+2147483647
unsigned long 32 бит       0..+4294967295
```

Достаточно интересно, что в языке Си допустимо использование префиксов `signed` и `unsigned` без идущего затем указания типа. Например, описание переменной `a` как `int a` будет полностью эквивалентно описанию `signed int a` и даже просто `signed a`;

Размер в битах прямо связан с тем, что для представления двух альтернатив — 0 и 1 — достаточно одного двоичного разряда, четырех — двух разрядов, и вообще n бит дают возможность представления 2^n различных кодов.

В Паскаль-семействе языков, а также Фортране, Алголе и Аде для обозначения целого типа используется ключевое слово `integer` (в Аде наименования типов пишутся с прописной буквы, в Фортране — часто полностью прописными).

Для действительных чисел используются различные наименования их приближений в языках программирования:

- `real` — Паскаль, Модула, Оберон, Алгол, Фортран и др.;
- `float` — Си, C++, Java, C#, Ада и др.

Небезынтересное название типа данных *число с плавающей точкой* (`float`) связано со следующим. При записи чисел в нормализованной экспоненциальной форме — а именно в ней хранится в двоичном виде в памяти число данного типа — происходит перемещение запятой (в англоязычных странах для отделения целой части от дробной используется точка) — она «переплывает» с места на место. Например, 1255,1 представляется как $1{,}2551 \cdot 10^3$.

В памяти при нормализованном представлении типа `float` хранится двоичное число — мантисса с фиксированным числом битов и отдельно порядок — аналогично фиксированное количество битов. Для больших чисел может применяться тип данных с удвоенной длиной мантиссы, например `double` в Си, C++, Java или `DOUBLE PRECISION` в Фортране.

Какие операции в языках программирования могут использоваться для чисел? Как и следовало ожидать, это арифметические действия (сложение, вычитание, умножение и деление) и сравнение. Для чисел допустимы следующие виды сравнения: больше, меньше, равно, не равно, меньше или равно, больше или равно. Результатом сравнения может быть специальная переменная логического типа. В некоторых языках (Си, C++, PL/I, Java, Ruby, C#, Паскаль) к числам применяются также побитовые логические операции — сдвиг, И, НЕ, ИЛИ, исключающее ИЛИ.

ЗАМЕЧАНИЕ

> Не следует проверять на равенство вещественные переменные типов float или double, поскольку операция эта точная, а в памяти хранятся и получаются при вычислениях на компьютере лишь приближенные значения. Для проверки на равенство данных этого типа можно сравнивать модуль разности чисел с некоторым положительным допуском ε.

Следующий важный тип данных — логический. Он может называться:

- `LOGICAL` — в Фортране;
- `bool` — в C++, C#;
- `boolean` — в Паскале, Аде, Java и др.

В большинстве языков допустимые значения для логического типа носят наименования `false` и `true`. Логические переменные могут использоваться в условных операторах. К ним применимы операции сравнения на равенство, а также (не побитовые) логические операции И, НЕ, ИЛИ.

Переменная может использоваться для хранения одиночного символа. Этот тип данных называется:

- `char` — в Си, C++, C#, Java, Паскале и др.;
- `character` — в Фортране, Аде (см. уточнение о регистре символов) и др.

К переменным символьного типа можно применять сравнение на равенство. В языке Си — те же действия, что и к типу `int`, включая арифметику, хотя это довольно опасная практика, используемая в основном для «трюков» программирования.

Во многих языках программирования переменная может связываться с целой строкой символов, среди них есть как языки со строгой статической явной типизацией, так и иные — Бейсик, PHP, Лисп, C++, Python, РАПИРА, Пролог и др. Стоит еще раз отметить, что Фортран — первый из созданных языков программирования высокого уровня — имеет весьма развитую систему типов и по этому критерию превосходит многие более поздние языки. В нем можно использовать то же ключевое слово для описания строк, например объявить переменную `CHARACTER*20` для хранения строки из 20 символов.

К строкам обычно применимы операции сравнения на равенство и неравенство, а также слияние — конкатенация (так называется действие, при котором строка приписывается в конец другой). Во многих языках есть операция взятия символа с указанным порядковым номером. Могут присутствовать операции определения длины строки, вырезания подстроки из строки (РАПИРА, C++) и др.

Отдельным и весьма важным моментом является преобразование типов данных. Допустимо ли в рамках одного выражения использовать переменные, относящиеся к различным типам данных? Во многих ситуациях это представляется полезным, несмотря на пользу строгого контроля типов для надежности программ. Действительно, весьма логично и интуитивно хочется считать допустимой запись, когда в ходе операции деления целых чисел результат сохраняется в переменной с плавающей точкой. Явно ничего страшного не произойдет, если мы к вещественному числу прибавим целое.

В языках программирования выделяют явное и неявное преобразования типов. Неявное преобразование выполняется системой с использованием встроенных правил. Явное преобразование задает программист, используя соответствующий синтаксис языка программирования, например:

```
x = (int)d * 10 + y;   /* преобразование типа в Си */
y = static_cast<short>(65534);    // преобразование в C++
```

Контрольные вопросы

1. Возможно ли полноценное представление математического понятия числа в ЭВМ? Почему?
2. Как представляются целые числа в памяти ЭВМ?
3. Как представляются дробные числа в памяти компьютера? Откуда произошло название типа данных с плавающей запятой?
4. Какие операции над числами допустимы в языках программирования?
5. Понятие и назначение логического типа данных. Примеры.

6. Что такое символьный тип данных, допустимые операции, строковый тип?
7. Зачем нужно преобразование типов данных в программах? Приведите примеры.

Указатели

В ряде языков программирования, например Си, Паскаль, C#, допустимо использование так называемых *указателей*. Они относятся к еще одной разновидности встроенных типов данных. Указатель — константа или переменная, задающая адрес в памяти, где располагаются некоторые другие объекты (данные, функция).

Допустимо, в частности, объявить некую переменную, которая будет указывать на находящееся в памяти целое.

Пример на языке C++:

```
int* pA; // переменная pA указывает на целое число
```

Теперь можно присваивать переменной-указателю значение ссылки (адрес) целого и после этого обращаться к объекту в памяти через указатель на него.

Пример на языке C++:

```
int a=10;
pA=&a; // pA теперь указывает на a
*pA=100; // эта запись присваивает переменной a значение 100
```

Смешивать типы при использовании указателей в общем случае недопустимо. Ссылки могут использоваться и для составных структур данных, в частности, они активно применяются при обработке массивов и списков.

В языках программирования встречается предопределенное наименование для пустой ссылки — указателя, еще не связанного с каким-либо объектом в памяти, часто это `NULL` или `NIL`.

Контрольные вопросы

1. Что такое указатели? Приведите пример их использования в языках программирования.
2. Что такое пустой указатель?

Перечисления

В некоторых языках программирования допускается задание так называемых *перечислений*. Бывает удобно называть объекты в программе привычными именами, а не раз и навсегда заданными наименованиями типов. Перечисления отвечают указанной потребности, например:

```
Type DAY=(Monday,Tuesday, Wednesday,Thursday, Friday, Saturday,
Sunday);  (* Паскаль*)
type Cardsuit is (clubs, diamonds, hearts, spades); -- Ада
enum MyColors {Red = 100, Green = 110, Blue = 120} // C#
```

В них явным образом указывается набор значений, допустимых для данной переменной, — фактически, создается новый тип данных. После его введения допускается объявление переменной данного типа наряду с базовыми, например:

```
MyColors cwet=Green; // C#
```

Обычно при компиляции значения перечислений представляются целыми числами. В зависимости от языка программирования представление может быть либо полностью скрыто от программиста, либо доступно ему, например, путем принудительного преобразования значения типа «перечисление» к значению типа «целое число», либо даже управляемо явно, как показано в примере.

При попытке присвоить переменной перечислимого типа значение, не указанное при его определении, будет выдаваться ошибка. Это позволяет еще более повысить надежность программ по сравнению с простой статической типизацией. Перечислимый тип используется довольно широко и воспринимается как сама собой разумеющаяся особенность современных языков программирования. Тем не менее не обходится без критики со стороны теоретиков и практиков программирования. При разработке языка программирования Оберон перечислимые типы попали в список особенностей, которые были удалены из него. Никлаус Вирт, разработчик языка, назвал следующие причины: «Во все возрастающем числе программ непродуманное использование перечислений… приводит к демографическому взрыву среди типов, что, в свою очередь, ведет не к ясности программ, а к многословию».

ЗАМЕЧАНИЕ

> Добавление типов данных к базовому набору является частью концепции абстрактного типа данных, более подробно рассмотренной в разделе, посвященном объектно-ориентированному программированию.

Составные типы данных

Составные типы данных подразумевают некую внутреннюю структуру,

обычно представляя собой соединение элементов базовых типов, выполненное с использованием предопределенных правил. Элементами составных структур данных, в свою очередь, часто могут быть сложные структуры.

Начнем рассмотрение с наиболее популярной структуры, имеющейся почти во всех языках программирования, — *массивов*.

Массивы

Массив — это проиндексированный (в простейшем случае пронумерованный) набор однородных элементов. Объявление массива в различных языках программирования может выглядеть так:

```
10 DIM A(10) 'Бейсик — одномерный массив из 10 элементов
DIMENSION A(20), B(20,100), C(0:10)  ! Фортран — три массива
Var Vector = array [1..10] of integer;(*Паскаль,с заданием границ*)
int a[12][2]; /* Си — двумерный массив */
int[,] mass = new int[4,5]; /* C# — двумерный массив */
```

Как видно из примера, массивы могут быть одномерными и многомерными — в зависимости от количества индексов.

Массивы могут быть статическими и динамическими (последние разрешены далеко не во всех языках программирования, но имеются, например, в Фортране). При объявлении статического массива программист должен заранее знать его размер. Если это невозможно (Си, Паскаль и пр.), приходится выделять память с запасом — столько, чтобы ее заведомо хватило для целей программы. Динамический массив — это массив, размер которого определяется при выполнении программы, тогда же происходит и выделение памяти под него.

В некоторых языках программирования индекс может изменяться лишь в пределах от единицы до N (Бейсик) или от нуля до $N-1$ (Си. Это обусловлено смещением элемента от адреса начала массива в памяти, причины подробнее рассмотрены в посвященном этому языку разделе.). В других, например Паскале и Фортране, допускается устанавливать произвольные границы изменения индекса для каждого измерения.

ЗАМЕЧАНИЕ

> В некоторых языках программирования, например Си, строки реализованы как одномерные массивы данных символьного типа.

Базовая операция для массива — взятие элемента по индексу. Поэтому массив представляет собой структуру данных с *произвольным доступом* — в любой момент за одинаковое количество времени возможен доступ к любому месту в нем. Как правило, сравнение массивов одной операцией

недопустимо. Некоторые языки программирования, такие как APL и его потомки K и J, изначально ориентированы на обработку массивов, в них операции применимы к целому массиву (ко всем элементам). Например, можно сложить два массива или найти сумму элементов с помощью операции, записываемой одним знаком.

В некоторых языках программирования помимо обычных массивов применяются так называемые *ассоциативные массивы*, в которых в качестве индекса может использоваться не только целое число, но и значение произвольного типа данных. Точнее говоря, ассоциативный массив представляет собой набор пар вида (ключ, значение) и поддерживает операции добавления пары, а также поиска и удаления по ключу (индексу). В некоторых языках поддержка ассоциативных массивов встроена в язык, в других — нужно использовать стандартную библиотеку. Примеры описаний ассоциативных массивов:

```
map<string, string> book; //объявление язык C++, библиотека STL
$names["Петров"]="Петр"; // обращение в языке PHP, фамилия – ключ
print $hash{'cat'}; # язык Perl
```

Ассоциативный массив может быть и многомерным. Многомерные ассоциативные массивы могут содержать несколько ключей, соответствующих конкретному индексу ассоциативного массива.

Пример многомерного ассоциативного массива на языке PHP:

```
<?php
// Многомерный массив
$A["Ivanov"] = array("name"=>"Иванов И.И.", "age"=>"25",
"email"=>"ivanov@mail.ru");
$A["Petrov"] = array("name"=>"Петров П.П.", "age"=>"34",
"email"=>"petrov@mail.ru");
$A["Sidorov"] = array("name"=>"Сидоров С.С.", "age"=>"47",
"email"=>"sidorov@mail.ru");
?>
```

Контрольные вопросы

1. Что такое массив? Приведите примеры объявления в языках программирования.
2. Что такое одномерный и многомерный массивы? Приведите примеры использования.
3. Что такое статический и динамический массивы? Какие преимущества и недостатки имеют динамические структуры данных?
4. В чем заключаются особенности структуры данных с произвольным доступом?

5. Что такое ассоциативный массив? Приведите пример использования.

Списки

Следующей важнейшей структурой данных являются списки. Такого рода структуры в языках программирования весьма разнообразны. Простейшей является *линейный односвязный список*, реализованный на базовом уровне в таких языках программирования, как Питон, Лисп и Пролог.

Линейный список представляет собой набор идущих друг за другом элементов, при записи разделяемых пробелом, запятой или иным знаком, например:

```
(Чайник Кастрюля Самовар)
[a,b,c,d,e]
```

В отличие от массивов, список, как правило, не предполагает возможности взятия элемента по его номеру. Для взятия элемента из середины можно начать последовательно перебирать их с первого, пока не будет найден искомый. В этом смысле список представляет собой структуру *с последовательным доступом*. Для полноты вводится также понятие пустого списка, или списка, не содержащего элементов, для которого используется специальное обозначение:

- `NIL` — в языке Лисп;
- `[]` — одно из обозначений в языке Пролог.

Основными операциями над списками служат:

- взятие первого элемента — так называемой *головы* списка;
- взятие остатка списка, также являющегося списком, или его *хвоста*;
- добавление элемента в список;
- удаление заданного элемента из списка;
- проверка на вхождение элемента в список;
- приписывание списка к списку;
- подсчет количества элементов списка.

В некоторых языках списки можно сравнивать одной операцией.

Список является *динамической* структурой данных — его размер может меняться в ходе исполнения программы. Список является также так называемой *рекурсивной* структурой — его понятие может быть определено рекурсивно следующим образом: пустой список является списком, кроме этого список — это элемент-голова, за которым идет, возможно пустой хвост-список. В силу этого для обработки списков хорошо подходят рекурсивные программы.

Во многих языках программирования списки не являются встроенной структурой (хотя могут быть частью библиотеки, например, для C++ существует мощная библиотека STL), и их поддержка требует конструирования на основе *записей* и *ссылок* (указателей). Это упражнение весьма популярно при изучении Паскаля или Си. Такую реализацию списка иллюстрирует рис. 9.

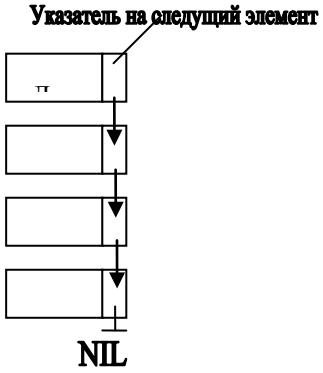


Рис. 9

Элементы подобного списка иногда называют вершинами. Каждая вершина здесь представляет собой элемент однотипных данных, хотя это могут быть и данные составного типа, и указатель на следующий элемент списка. Последний элемент списка содержит пустую ссылку.

В языках программирования, поддерживающих списки на базовом уровне, часто допускается наличие в списке в качестве элементов данных различающихся типов. Элементом списка может быть и список — в этом случае говорят о вложенных списках.

Другие виды списков

Нетрудно заметить, что в линейном односвязном списке возможно лишь движение от головы к хвосту. Иногда удобно иметь возможность обратного продвижения по списку. Для этого достаточно налачия в каждой вершине двух указателей — на предшествующий и последующий элементы. Графически реализацию двусвязного списка можно представить следующим образом (рис. 10).

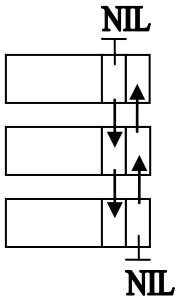


Рис. 10

При реализации двусвязного списка у первого элемента делается пустая ссылка на предшествующий.

Использование нескольких ссылок в каждой вершине позволяет строить и более сложные списочные структуры, например *деревья*.

Дерево — частный случай многосвязного списка. Оно имеет иерархическую организацию. Вершина дерева, не являющаяся ничьим потомком и называемая *корнем дерева* (рис. 11), имеет ссылки на вершины, называемые *потомками*, причем они не могут ссылаться на эту вершину. В нижней части иерархии — вершины-*листья*, не имеющие потомков (ссылки `NIL`).

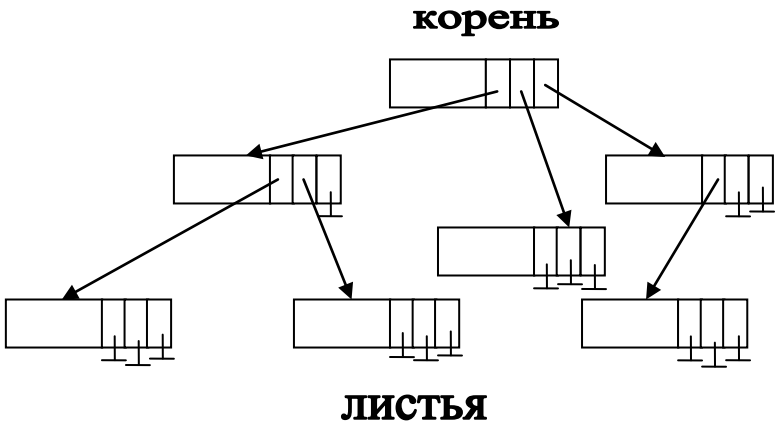


Рис. 11

Чаще всего в практике программирования используются бинарные деревья (у каждой вершины не более двух потомков).

Обработка деревьев обычно производится путем *обхода* вершин дерева в том или ином порядке. Известны алгоритмы обхода деревьев снизу вверх, сверху вниз и слева направо. Популярными операциями над деревьями служат также поиск поддерева в дереве, удаление поддерева и добавление вершин.

Иногда на основе списков строятся такие структуры данных, как стеки и очереди.

Весьма важным при ручной реализации динамических структур данных является следующее обстоятельство. При добавлении элемента в динамическую структуру данных необходимо выделить необходимую память, а при удалении — освободить ее. При этом в процессе работы с динамическими типами данных в языках программирования память выделяется из специального пула, называемого *кучей*. Если в языке программирования отсутствует механизм автоматического управления памятью при удалении — так называемая *сборка мусора*, ответственность за очистку памяти возлагается на программиста, который должен проделать это явным образом.

Сборщик мусора периодически освобождает память, удаляя объекты, которые уже не будут востребованы приложением. Изначально он имеется в таких языках, как Лисп и Java. Однако при выполнении программ на этих языках наличие автоматической сборки мусора может приводить к непредсказуемым задержкам в заранее не известные моменты времени. В связи с этим на Си, например, управление памятью осуществляется вручную с использованием библиотечных функций. Ниже приводится возможный вариант программирования динамического массива на языке Си:

```
#include <alloc.h>
#define N 10
int main()
{
  double* ptd; // объявляем указатель на массив

  /* запрос выделения памяти из «кучи» */
  ptd = (double*)malloc(N * sizeof(double));
  if(ptd != NULL)  /* если памяти нет, malloc возвращает NULL*/
  {
    for(int i = 0;i <N;i++)  ptd[i] = i;
  } else printf("Не удалось выделить память.");

  free(ptd); /* Освобождение памяти */

  return 0;
}
```

Контрольные вопросы

1. Что такое список в языках программирования?

2. Каковы основные операции над списками?
3. Является ли список рекурсивной структурой данных? Почему?
4. В чем заключается принцип низкоуровневой реализации списков в программах на языках, не содержащих встроенной поддержки списков?
5. В чем заключается разница между линейным односвязным и многосвязными списками?
6. Что такое дерево в языках программирования? Перечислите основные операции над деревьями.
7. Каковы особенности ручного управления динамической памятью? Что такое сборщик мусора в языках программирования? В чем заключаются недостатки и достоинства механизма автоматического управления памятью?

Очереди

Очередь представляет собой линейный упорядоченный набор, как правило, однотипных элементов, доступ к которому возможен лишь с конца. Очередь, как и линейный список, — динамическая структура данных, напоминающая очередь за продуктами в магазине. Работа с ней интуитивно понятна из ее названия. Имеется лишь два базовых действия:

- добавление элемента в конец очереди;
- взятие элемента с начала (головы) очереди.

В очереди реализуется принцип «первым пришел — первый на обработку» (FIFO — First Input — First Output) (рис. 12).

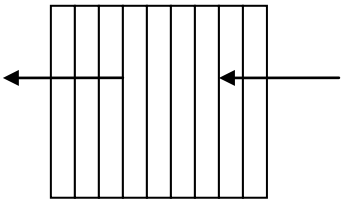


Рис. 12

Очередь может быть пустой. Кроме базовых, популярны следующие операции над очередями:

- проверка на пустоту;
- подсчет количества элементов в очереди;
- копирование значения в голове очереди с оставлением элемента.

Стек

Стек — динамическая структура данных, противоположная очереди. В

стеке реализуется принцип «последним пришел — первым на обработку» (LIFO, Last Input — First Output). Базовые действия над стеком:

- добавление элемента на *вершину* стека;
- взятие элемента с вершины.

Это может быть проиллюстрировано рис. 13.

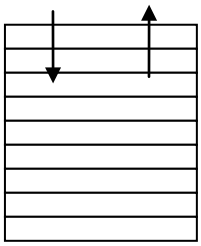


Рис. 13

Стек, как и очередь, может быть пуст. Популярное называние действия по помещению элемента на вершину стека — PUSH, а извлечения из стека — POP. Со стеком, как и очередью, можно выполнить популярные операции:

- проверку на пустоту;
- подсчет количества элементов;
- копирование значения на вершине с оставлением элемента в стеке.

Стек работает по принципу магазина пистолета или стопки книг. Книга, находящаяся на вершине стопки, снимается с нее первой. А до книги, лежащей в самом низу, читатель добирается позже.

Стеки и очереди поддерживаются во многих языках программирования специальными библиотеками, например, в C++ — библиотекой STL.

Контрольные вопросы

8. Что такое стек в языках программирования? Каковы области использования стека? Каковы основные операции со стеком?
9. В чем заключаются особенности структуры данных «очередь»? Каковы основные операции над очередями?

Кортежи

В некоторых языках программирования, например Питон и русскоязычная РАПИРА, имеется такая встроенная структура данных, как *кортеж*. Кортеж — динамическая структура данных, представляющая собой упорядоченную совокупность разнотипных элементов:

```
("a", "b", "mpilgrim", "z", "example") # язык Питон
<"Я",7,"Вася",<1,"Осень">,-25> (* язык программирования РАПИРА *)
```

Как видно из примера, кортеж может входить в состав другого кортежа — допустима вложенность кортежей.

К основным операциям над кортежами относятся:

- взятие элемента по его порядковому номеру;
- проверка на вхождение элемента;
- вырезка подкортежа;
- конкатенация;
- подсчет количества элементов.

Кортеж — структура с произвольным доступом за счет наличия операции взятия элемента по номеру. Кортеж похож на список, массив и строку одновременно и представляет собой весьма мощную структуру данных.

В кортеж, в отличие от множества, один элемент может входить несколько раз.

Множества

В Паскале, Модуле и некоторых других языках программирования возможно использование так называемых *множеств*. Понятие множества, вероятно, известно читателю из математики, где так называется совокупность элементов произвольной природы. Понятие множества в программировании значительно у́же математического понятия. Множество — динамическая структура данных, представляющая собой неупорядоченную совокупность элементов, как правило, однотипных.

Базовые операции над множествами:

- проверка вхождения элемента в множество;
- теоретико-множественные операции — объединение, пересечение, вычитание;
- сравнение множеств на эквивалентность;
- проверка вхождения подмножества в множество;
- удаление и/или добавление элемента.

Далее приводится ряд примеров, иллюстрирующих использование множеств:

```
set<int> s; // объявление множества целых в C++, библиотека STL

(*язык программирования Паскаль*)
Type symbol= set of char; (* объявление типа symbol -множества*)
Var small, capital, latin: symbol; (* три множества *)
…
```

```
small:= ['a' .. 'z']; (* строчные латинские буквы *)
capital:= ['A' .. 'Z']; (* прописные латинские буквы *)
latin:= small + capital; (*объединение множеств small и capital *)

small:= small + ['a'] +['b']; (* добавление поэлементно*)

z:= x*y; (* если x,y,z — множества — пересечение *)

letter:= ['a' .. 'z']; (*множество букв латинского алфавита*)
glasn:= ['a', 'e', 'o', 'u', 'i', 'y']; (*множества гласных букв*)
soglasn:= letter-glasn; (* вычитание множеств*)
```

После приведенного задания множеств `glasn` и `soglasn` операция проверки вхождения элемента `'a' in glasn` возвращает значение `true`, а `'o' in soglasn` — `false`.

Для сравнения множеств на равенство или неравенство в Паскале используются символы = и <>:

```
A:= [2,1,3];
D:= [1,3,2];
```

В этом случае A=D возвращает значение `true`, а A<>D — значение `false`. Операции проверки включения обозначаются <= и >=, после приведенного будет соблюдаться `letter >= glasn` и `soglasn <= letter`.

В множество, в отличие от кортежа, элемент может входить лишь единожды.

Контрольные вопросы

10. Что такое кортеж в языках программирования? Каковы основные операции с кортежами? Сравните списки и кортежи.
11. Что такое множества в языках программирования? Каковы операции над множествами? Сравните массивы, кортежи и множества.

Записи

Весьма важной и популярной структурой данных в языках программирования являются *записи* (иногда запись называется *структурой*). Запись представляет собой поименованную совокупность разнотипных данных. Отдельные элементы записи принято называть *полями*. Доступ к элементу возможен по указанию имени всей структуры и имени поля. Объявление структур часто происходит с определением нового типа данных.

Пример на языке Паскаль:

```
(* определение типа анкета на основе записи *)
Type anketa=record
   fio: string[45]; (* ФИО *)
   pol: char; (* пол *)
   dat_r: string[8];
   adres: string[50];
   curs: 1..5;
   grupp: string[15];
end;
var stud: anketa;
```

Пример на языке C++:

```
// определение структуры student с полями фамилия, сравнить балл и
год //рождения
struct student
{
   char familia[58];
   float srBall
   int godRojdenia;
} s; // объявление переменной s с заданной структурой
```

Записи можно объединять в массивы, включать в списки и т. д. Весьма популярно обращение к полю записи с использованием символа `.` (точка): `stud.grupp` или `s.godRojdenia`.

Файлы

Еще одной структурой данных, фундаментальной по значению в информационных технологиях вообще и используемой в языках программирования, является *файл*. Файл представляет собой поименованную совокупность данных, хранящуюся на диске или передаваемую по сети — локальной или глобальной.

Основные операции над файлами:

- открытие в различных режимах — на чтение, запись, добавление и пр.
- закрытие после использования (обычно для фиксации изменений);
- чтение из файла — или порции байт, или структурированное;
- запись в файл — или порции байт, или структурированное;
- копирование файлов;
- переименование файла на диске;
- удаление файла.

Файлы в зависимости от их содержимого обычно делят на текстовые,

графические, звуковые и т. д. В языках программирования принято использовать деление файлов на так называемые *бинарные* и *текстовые*. На самом деле в зависимости от этого файл не меняется, меняется лишь способ работы с ним.

Например, в Паскале файловый тип данных определяет упорядоченную совокупность произвольного числа однотипных компонентов. Далее приводится возможное объявление файлов в языке программирования Паскаль:

```
var
   f1: file of char;
   f2: file of integer;
   f3: file;
   t: text;
```

Здесь к файлам `f1`, `f2` и `f3` предполагается структурированный доступ, а к файлу `t` — неструктурированный, с чтением и записью символов. Объявленную переменную файлового типа необходимо связать с физическим файлом на диске, что на Паскале записывается как `Assign(f, FileName)`. Имя задается строкой `FileName`.

После задания связи файловой переменной с дисковым именем файла в программе нужно указать направление передачи данных (открыть файл). В зависимости от этого направления говорят о чтении из файла или записи в файл. Например, `Reset(f)` в Паскале открывает для чтения файл, с которым связана файловая переменная `f`. После успешного выполнения процедуры `Reset` файл готов к чтению из него первого элемента. Процедура завершается с сообщением об ошибке, если указанный файл не найден. Если `f` — типизированный файл, то процедурой `reset` он открывается для чтения и записи одновременно. `Rewrite(f)` открывает для записи файл, с которым связана файловая переменная `f`. После успешного выполнения этой процедуры файл готов к записи в него первого элемента. Если указанный файл уже существовал, то все данные из него уничтожаются.

Структурированный доступ к компонентам (называемым в этой ситуации *записями*) файла осуществляется в Паскале с учетом порядкового номера, начиная с нуля. После открытия файла *указатель* стоит в его начале. После каждого чтения или записи указатель сдвигается к следующему компоненту с изменением номера записи. Чтение из файла на Паскале выглядит следующим образом:

```
Read(f, <список переменных>);
```

Типы записей файла и переменных должны совпадать. Необходимо либо точно рассчитывать количество компонентов, либо перед каждым чтением

данных выполнять проверку их существования, для чего использовать специальную функцию `EOF(f)`, которая возвращает `true`, если при чтении очередной записи достигнут конец файла — прочитан последний элемент в файле или файл после открытия оказался пуст.

Запись в файл выглядит так:

```
Write(f, <список переменных>);
```

Возможно смещение указателя файла без фактического чтения. Процедура `Seek(f,n)` смещает указатель файла `f` на *n*-ную позицию.

Весьма важная операция — определение длины файла (количества записей при структурированном доступе и количества байтов — при неструктурированном). Функция `FileSize(f)` возвращает количество компонентов в файле `f`. `FilePos(f)` возвращает позицию указателя.

Текстовый файл воспринимается как совокупность строк, разделенных символом перевода строки. Доступ к каждой строке возможен последовательно, начиная с первой. Одновременные запись и чтение запрещены.

Бинарные файлы воспринимаются как последовательность байтов — подобным образом можно работать с файлами произвольных типов.

Открытие нетипизированного файла на чтение и запись соответственно в Паскале выглядит так:

```
Reset(f, BufSize)
Rewrite(f, BufSize)
```

Параметр `BufSize` задает число байтов, считываемых из файла или записываемых в него за одно обращение. Минимальное значение `BufSize` — 1 байт, если `BufSize` не указан, то по умолчанию он принимается равным 128. Чтение данных из нетипизированного файла выглядит следующим образом:

```
BlockRead(f, X, n, QuantBlock);
```

Эта процедура осуществляет за одно обращение чтение в переменную `X` количества блоков, заданного параметром `n`, при этом длина блока равна длине буфера. Необязательный параметр `QuantBlock` возвращает число блоков, прочитанных текущей операцией. Запись данных в нетипизированный файл производится аналогично:

```
BlockWrite(f, X, Count, QuantBlock);
```

Для нетипизированных файлов можно также использовать `Seek`, `FIlePos` и `EOF`.

Контрольные вопросы

1. Что такое записи в языках программирования? В чем заключается назначение записей? В чем состоит связь между определением записи и объявлением типа данных?
2. Что такое файл? Опишите основные операции над файлами в языках программирования.
3. В чем заключается разница в работе с текстовыми и бинарными файлами?
4. Какие существуют режимы доступа к файлам?
5. Зачем используется указатель на файл?
6. В чем разница между структурированным и байтовым (блочным) доступом к файлу?
7. В чем назначение функции `eof`? Приведите пример ее использования.

Структурирование программ, принцип модульности

Современные программные комплексы зачастую содержат тысячи, десятки тысяч и даже миллионы строк исходного текста на языке высокого уровня. Их сложность современных потрясает. Единственным способом справиться со сложностью, известным человечеству с незапамятных времен, является принцип «разделяй и властвуй», или решение сложной задачи путем разбиения ее на менее сложные подзадачи.

Программирование не является исключением. Принятым термином для обозначения самостоятельной части программы, обособленной в соответствии с некоторыми принципами, приводимыми ниже, является *модуль*. В различных языках программирования могут использоваться и другие похожие термины, например «пакет» или «компонент». В объектно-ориентированном программировании в качестве аналога модулей часто рассматриваются классы. Иногда модули хранятся по принципу «один модуль» - «один файл исходного текста программы». Часто при этом модули компилируются по отдельности, и затем из них собирается полностью программа.

Если программа разбита на модули, мы получаем целый ряд преимуществ. Так, можно поручить создание разных модулей разным программистам, распараллелив работу, тем самым избавив каждого программиста от необходимости перетруждаться и ускорив процесс. Сложность разработки и тестирования отдельно взятого модуля при этом вполне может оказаться

«по силам» обыкновенному программисту, а не гению, способному удержать в голове сотни тысяч операторов и переменных единой большой программы. Более того, можно реализовывать программу «по частям», сначала создав главный модуль, обращающийся к остальным, а вместо них поначалу использовать так называемые «заглушки», постепенно наполняемые содержимым.

Важнейшим преимуществом модульности является возможность так наызываемого *повторного использования кода*. В случае, если у нас имеется написанный ранее и отлаженный модуль, можно при необходиморсти его использовать и в последующих программах без затрат труда, времени и ресурсов. Вообще, в программировании при создании нового продукта процессы выглядят весьма отсталыми по сравнению с такими инженерными дисциплинами, как строительство или машиностроение, где инженер-конструктор, не задумываясь, использует стандартные типовые шурупы, болты и даже значительно более сложные узлы и сборки при создании нового изделия. Использование как можно большего числа ранее созданных модулей – правильный подход к проектированию сложной программной системы. В свою очередь, инженер-программист должен при написании каждого модуля задаться вопросом, а не возможно ли его последующее повторное использование и как нужно доработать/изменить модуль для этого в случае необходимости. В идеале при производстве программного обеспечения было бы неплохо иметь возможность собирать новые приложения, взяв все необходимые компоненты из соответствующих библиотек.

Естественно, модули в программе должны быть согласованы между собой по *интерфейсу*, под которым понимается порядок вызова модуля, передачи ему входных данных (и их стуруктура) и получения результатов его работы.

Весьма важна также так называемая *инкапсуляция* – или, выражаясь проще, сокрытие «внутренностей» модуля от доступа извне. Так, данные внутри модуля обычно недоступны из других модулей, за исключением специально указываемых случаев. Это предохраняет их от случайной порчи информации и повышает надежность программного комплекса в целом.

Еще одним важным преимуществом модульной программы с разделением доступа к данным внутри модулей является упрощение модификации. Часто достаточно для модификации большой программы лишь заменить один модуль другим – исправленным, более эффективным, и пр. При этом подобная замена никак не влияет на остальные модули. Когда необходимо устранить ошибку или улучшить некую функцию, модульность помогает ограничить поиск неисправных или подлежащих модернизации частей

конкретными компонентами.

Чтобы максимизировать эффект от модульности, программист должен стараться создавать модули, имеющие высокую связность внутри модуля и низкую связность между модулями. Это – целое искусство, и в больших программных проектах является одной из задач гуру программирования, выполняющего роль *архитектора системы*. Межмодульная связность характеризует зависимость между модулями, когда, например, модуль А использует некую подпрограмму, располагающуюся в модуле Б, или функция в модуле Б обращается к данным модуля В (существует множество возможных видов связи между модулями).

Несмотря на то, что способность правильно проектировать модульную программную систему обычно приходит с опытом, приведем несколько простых эмпирических правил, помогающих с пользой применять модульность при написании программ. Итак:

1) Хорошо, если программный модуль реализует некую самостоятельную независимую функциональность;
2) Желательно, чтобы каждый модуль был как можно более простым и обозримым (до 100 строк);
3) Данные внутри модуля должны максимально использоваться подпрограммами и функциями, находящимися в этом модуле, и не использоваться из других модулей.

Язык программирования Си

> Эффективность, эффективность и еще раз эффективность!
>
> *Возможный девиз языка программирования Си*

Одним из наиболее известных, сильно повлиявших на дальнейшее развитие информационных технологий и широко используемых до сих пор языков программирования является Си. На его основе были созданы такие наиболее популярные в настоящее время в индустрии программирования языки, как C++, C#, Java, PHP, Objective C и др. При этом можно сказать, что, в отличие от многих других языков, созданных в рамках серьезных инвестиционных программ правительств или крупных корпораций (PL/I, Кобол, Java) или родившихся в стенах исследовательских лабораторий в качестве проверки научной концепции (Smalltalk, Клу, Q и др.), Си появился на свет в результате «хулиганства».

В начале 1970-х годов молодые системные программисты (программисты,

создающие операционные системы и другие служебные программы) Деннис Ритчи и Кен Томпсон, работавшие в американской корпорации AT&T, были в числе других сотен разработчиков задействованы в большом проекте создания многозадачной многопользовательской операционной системы, что нашло свое отражение в названии — MULTICS. Согласно одной из легенд, Ритчи и Томпсон в свободное от работы время были не прочь поиграть. Особенно им полюбилась игра Space Travel, которую они сами написали и запускали на главном сервере компании. Позднее они захотели перенести ее на вспомогательный компьютер PDP-7, собиравший пыль в их комнате. Но он, к сожалению, не имел операционной системы.

Что делают системные программисты, если им хочется запустить на компьютере игру, но на нем отсутствует операционная система? Правильно, пишут операционную систему. Название Unix как отсылка к MULTICS (противопоставление на английском слова «уникальный», «единственный» слову «мульти», «множественный») появилось в 1970 году как шутка Брайана Кернигана — намек на то, что новая система поддерживала лишь пользователя Томпсона и поэтому должна носить название Un-multiplexed Information and Computing Service. Все очевиднее становилось, что изначально устаревшая PDP-7, накладывающая множество раздражающих ограничений на работу операционной системы, не сможет удовлетворять нуждам дальнейшего развития, как и вся серия машин PDP-7. При этом надеяться на разрешение закупки нового оборудования со стороны руководства не приходилось.

Один из сотрудников, увлекавшийся обработкой текстов на компьютере, предложил схитрить и попросить у начальства новейший микрокомпьютер PDP-11 компании DEC для создания инструментов редактирования и форматирования текста. То, что для создания этих инструментов потребуется написать операционную систему, указывалось как сноска. Запрашиваемая сумма была куда меньше, чем раньше — всего 65 тыс. долларов. Не сразу, но начальство согласилось, и в 1970 году был сделан заказ на PDP-11. Все усложнявшаяся операционная система была переписана на новую машину. Перенос оказался непростой задачей, поскольку система была написана полностью на ассемблере. В связи с этим возникла идея переписать операционную систему на язык высокого уровня.

В распоряжении авторов имелся язык программирования BCPL. Его разработал Мартин Ричардс в 1966–1967 годах, когда посещал Массачусетский технологический институт. Это была несколько упрощенная версия языка CPL (Cambridge Programming Language). Первоначально компилятор BCPL был реализован для операционных систем GECOS и MULTICS. Томпсон и Ритчи несколько его

усовершенствовали, назвав сначала B (читается «би»), и затем NB. По сути, B был синтаксически видоизмененным BCPL, который Томпсону удалось втиснуть в 8 Кбайт памяти. Почему язык получил такое имя? Существует две гипотезы его происхождения: от первой буквы либо BCPL, либо другого языка Томпсона — Bon, названного в честь его жены Бонни. Язык B был, фактически, языком промежуточного уровня. Как и BCPL, он бестиповый, работающий со словом (ячейкой машинной памяти), содержащим фиксированное число разрядов, как в ассемблере. Память при этом рассматривается как линейный массив слов, а значение ячейки можно интерпретировать как индекс в этом массиве. Для подобного доступа BCPL использует оператор `!`, а язык B — оператор `*`.

В связи с тем, что B не давал использовать все возможности PDP-11, на который программисты хотели перенести систему, было решено создать еще один улучшенный язык программирования. Переход от B к Си происходил одновременно с созданием компилятора, способного порождать довольно быстрые и компактные программы, не слишком уступавшие в скорости по сравнению с ассемблером. Деннис Ритчи вспоминает: «Я назвал несколько расширенный язык NB — «новый B» (new B)». В язык NB Ритчи ввел первые типы — `int` и `char`. Вместе с массивами и указателями они составили его систему типов.

Откуда появилось название Си? Ритчи разъясняет это так: «Создав систему типов, соответствующий синтаксис и компилятор для нового языка, я почувствовал, что он заслуживает нового имени: NB показалось мне недостаточно четким. Я решил следовать однобуквенному стилю и назвал его C (Си), оставляя открытым вопрос, являлось ли после B это следующей буквой в алфавите или в названии BCPL».

Надо сказать, что при создании Си активно использовалась «самораскрутка», когда язык используется для создания собственного транслятора. Сначала на ассемблере пишется поддержка минимального набора конструкций и возможностей реализуемого языка, затем полученный минимальный язык применяется разработчиком для реализации еще какой-либо возможности и т. д.

Успех Си был неразрывно связан с тем, что в одном месте в одно и то же время появились сразу три шедевра, ставших культовыми: язык программирования Си, операционная система UNIX и мини-компьютер PDP-11 (в Советском Союзе аналогом PDP-11 были ЭВМ семейств СМ-4 и СМ-1420). Кен Томпсон считает: «Наше сотрудничество было образцом совершенства. За те десять лет, что мы проработали вместе, можно вспомнить только один случай нескоординированной работы. Я тогда обнаружил, что мы написали одинаковую ассемблерную программу из 20 строк. Я сравнил наши тексты и поразился, обнаружив, что они совпадают

посимвольно. Результат нашей совместной работы получился намного более значительным, чем вклад нас обоих по отдельности». К 1973 году язык Си стал довольно силен, большая часть ядра UNIX, первоначально написанная на ассемблере PDP-11, была переписана на Си. Это одно из самых первых ядер операционных систем, написанное на языке, отличном от ассемблера. Более ранними были лишь MULTICS, запрограммированная на ПЛ/1, и TRIPOS, использовавшая BCPL.

Таким образом, Си — язык, созданный системными программистами для своих нужд. Это серьезнейшим образом сказалось на некоторых его особенностях:

- Язык имеет низкоуровневые возможности, позволяющие непосредственно работать с машинными ячейками, в том числе битовые операции и так называемую адресную арифметику. В связи с этим некоторые ученые относят Си к языкам промежуточного, а не высокого уровня.
- Си позволяет создавать высокоэффективные — быстрые и не требующие больших объемов памяти программы.
- Язык Си отличается минимализмом и лаконичностью программ.

В отличие от языков, специально создававшихся для первоначального обучения программированию, таких как BASIC (Beginners All-purpose Symbolic Instruction Code — многоцелевой язык символических инструкций для начинающих) или Паскаль, Си дает программисту большие возможности, развязывет ему руки. Но за большую свободу, как известно, приходится платить большей ответственностью. При программировании на Си можно допустить такие ошибки, которые в Паскале невозможны.

Брайан Керниган говорит: «Си — инструмент, острый, как бритва: с его помощью можно создать и элегантную программу, и кровавое месиво». В связи со сравнительно низким уровнем языка многие случаи неправильного использования опасных элементов не обнаруживаются ни при компиляции, ни во время исполнения. Это часто вызывает непредсказуемое поведение программы. Иногда в результате неграмотного использования элементов языка появляются уязвимости. Примером ошибки является обращение к несуществующему элементу массива. Си не имеет проверки индексов массивов (проверки границ). Например, возможна запись в шестой элемент массива из пяти элементов, что, естественно, приведет к непредсказуемым результатам. Похожей является ошибка *переполнения буфера*.

Для того чтобы помочь программирующим на Си решить эти и многие другие проблемы, было создано большое число отдельных от

компиляторов инструментов. Такими инструментами являются программы дополнительной проверки исходного текста программы и поиска распространенных ошибок (статические анализаторы), а также библиотеки, предоставляющие дополнительные функции, не входящие в стандарт языка.

Во многих учебниках по языкам программирования традиционным началом изучения языка является написание на нем программы приветствия, выводящей на экран сообщение «Здравствуй, мир!». Пойдем по этому пути, проторенному Керниганом и Ритчи [11], и мы. Далее приводится вариант подобной программы на Си.

Пример программы «Здравствуй, мир!» на языке Си:

```
/* Программа приветствия */
#include <stdio.h>

int main()
{
   printf ("Здравствуй, мир!\n");
   return 0;
}
```

Разберем построчно текст программы. Первой строкой идет так называемый комментарий, игнорируемый компилятором и служащий для пояснения программы читающему ее человеку. Комментарием в Си считается любая последовательность символов (не только располагающаяся на одной строке, но и многострочная), начинающаяся с комбинации `/*` и заканчивающаяся `*/`.

ЗАМЕЧАНИЕ

> Автор настоятельно советует читателям при написании программы на любом языке начинать ее со строки комментария, в котором кратко описывается назначение программы, могут указываться автор, версия и дата выпуска программы.

Следует обратить внимание на стиль написания программы, в том числе использование вертикальных и горизонтальных отступов для отделения логически значимых частей текста друг от друга (при этом отступы образуют лесенку). Правильный стиль необходим также для того, чтобы программа была более понятной и ясной для человека, включая и самого автора — по прошествии некоторого времени! С точки зрения языка программирования Си пробелы несущественны (игнорируются). Удобство восприятия и «прозрачность» программы для человека — один из главных факторов успеха крупных современных профессиональных программных проектов, в которых задействуется множество участников и создаются

тысячи и десятки тысяч строк программ.

Обратите также внимание на то, что при написании операторов и стандартных функций языка Си используются строчные, а не прописные буквы.

Что же содержится в следующей строке? В ней пока не встречаются инструкции, непосредственно относящиеся к выводу приветствия на экран. Нам придется сделать довольно длительное отступление и погрузиться в особенности построения программ на языке программирования Си.

Как уже говорилось, почти в любом современном языке программирования помимо собственно перечня выполняемых действий программа в обязательном порядке содержит еще несколько секций. Например, программа на языке Си может содержать так называемые *директивы препроцессора*, которые обычно идут в начале текста программы и, чтобы отличаться от собственно операторов программы, начинаются с символа # (диез). Препроцессор — это специальная программа, обрабатывающая текст программы до того, как он передается транслятору, и осуществляющая некоторые предварительные действия над ним в соответствии с заданными директивами.

Препроцессор имеет возможность обрабатывать ряд других директив, включая обработку макроподстановок, то есть замену в тексте программы специально введенных *макроимен* так называемыми *макрорасширениями* (блоками текста), при необходимости — параметрическую замену с указанием или вычислением изменяющихся параметров. Простейший случай макроподстановки — эмулирование констант, например:

```
#define PI 3.1415926
```

После этого все вхождения последовательности символов `PI` в тексте программы будут заменены препроцессором на цепочку символов `3.1415926`. Необходимо обратить внимание на то, что замена осуществляется чисто механически — это не настоящая константа в смысле, принятом в языках программирования, в частности, не проводится контроль типов данных.

Все же в Си одна из главных задач директив — указание, какие из библиотек (предварительно кем-то разработанных наборов программ, выполняющих полезные функции) будут использованы в программе. Директива `#include` указывает на необходимость включения (вставки) в программу содержимого из заголовочного файла `stdio.h` библиотеки стандартных функций ввода-вывода (STandarD Input/Output). Хотя в принципе нет разницы, содержимое текстового файла с каким расширением включается в тело текущего файла, в большинстве случаев это так называемый *заголовочный* файл с расширением `.h`. Заголовочный

файл обычно содержит определения некоторых констант, глобальных переменных и структур данных, а также декларирующие описания — *заголовки* функций, используемых в том или ином программном модуле. В соответствующем файле *исходного текста* программы на языке Си, обычно имеющем одинаковое имя с заголовочным файлом, но с расширением `.c`, должна содержаться *реализация* данной функции.

ЗАМЕЧАНИЕ

> Данное условие не относится к функциям стандартной библиотеки, которые поставляются не в виде исходных текстов, а внутри двоичных файлов с машинными кодами, обрабатываемых специальной программой — компоновщиком при сборке исполняемой программы пользователя. О том, что это заголовок системной библиотеки, говорит указание имени включаемого файла в угловых скобках. Это показывает, что искать данный файл нужно в предопределенных местах среди системных каталогов. Для обычных заголовочных файлов в директиве include используется взятие имени в кавычки.

В данном случае вставка текста из `stdio.h` необходима для дальнейшего использования библиотечной функции `printf`, предназначенной для вывода информации на экран компьютера. Язык Си не содержит встроенных операторов ввода-вывода, в отличие от большинства других языков программирования, что делает его ядро более компактным, при этом поддержка ввода-вывода выносится в стандартные библиотечные функции.

Следующая строка содержит заголовок основной функции программы — `main`.

Любая программа на языке Си является набором функций (в простейшем случае набор превращается в одну — *главную* функцию). Термин «функция» здесь имеет не математический, а особый, специфический смысл. Функция определяется как модуль (часть программы), имеющий в общем случае некоторые аргументы, записываемые в круглых скобках, и способный возвращать некоторое значение. При этом одни функции могут вызывать на выполнение другие. Функции могут располагаться в тексте программы в произвольном порядке, но система определяет точку входа (начала исполнения) программы с начала функции `main`.

Поэтому каждая программа на языке Си обязательно должна включать одну (и только одну!) функцию с именем `main`. Остальные функции могут иметь произвольные имена в соответствии с правилами написания идентификаторов (имен) в языке Си (содержать латинские буквы и цифры и знак _, не содержать в имени пробелов, начинаться с буквы и пр.). Описание функции выглядит следующим образом:

<*заголовок функции*>
<*тело функции*>

В заголовке перед собственно названием функции пишется тип возвращаемого ей значения. В нашем примере перед `main` написано ключевое слово `int` — это означает, что функция возвращает целое число. Поясним, что это за число. Поскольку функция `main` — главная функция программы, то ее выход подразумевает возвращаемое программой в операционную систему значение. При этом существует соглашение, по которому в случае, когда программа завершается нормально — без возникновения нештатных ситуаций, она возвращает ноль, в противном случае — ненулевое значение.

После названия функции `main` идут круглые скобки без каких-либо символов внутри них, что означает, что наша программа не обрабатывает никаких входных значений (в случае необходимости имеется возможность передать в программу набор некоторых значений с использованием командной строки операционной системы).

В следующих после заголовка строках находится *тело* функции (так на программистском жаргоне называется ее основной текст), в котором содержатся операторы языка Си и вызовы других функций, то есть действия, которые она выполняет. Тело функции ограничивается фигурными скобками `{` и `}`. Такая пара скобок называется в языке программирования Си *операторными скобками* — ее используют для группировки любого количества операторов, обрабатываемых потом как один оператор в конструкциях языка.

Наша функция вызывает одну стандартную библиотечную функцию *форматного* (с возможностью форматирования — выравнивания и т. д.) вывода информации на дисплей — функцию `printf`. Аргументом данной функции является текстовая строка (текстовые строки в языке Си принято заключать в двойные кавычки — `"` и м). В конце строки `"«Здравствуй, мир!"` присутствуют символы `\n` — это специальная последовательность символов языка, означающая, что после вывода строки на экран надо перейти к новой строке. Вообще специальные последовательности символов в языке программирования Си, называемые иногда *управляющими*, начинаются с символа `\`.

Обратите также внимание на то, что стандартным разделителем в тексте программ на языке Си, разграничивающим отдельные действия внутри функции, является точка с запятой.

Завершается функция оператором `return 0`. Оператор `return` прекращает выполнение любой функции и возвращает в вызвавшую ее (в нашем случае, поскольку функция главная, она вызывается из

операционной системы) некоторое значение в качестве результата. В данном примере мы передаем в операционную систему ноль в качестве свидетельства того, что функция завершилась нормальным образом. Поскольку мы объявили функцию `main` как возвращающую целое значение, необходимо вставить этот оператор в программу, хотя завершение выполнения функции произошло бы и просто при встрече закрывающей фигурной скобки. В противном случае при трансляции программы мы получили бы предупреждение. Под *предупреждением* понимается сообщение о неточности или наличии в программе потенциально опасной конструкции, которая не является полностью недопустимой, как *ошибка,* в случае которой трансляция завершается неудачей, но должна привлечь внимание программиста.

Завершающая строка программы — закрывающая фигурная скобка, указывающая на окончание функции `main`.

Контрольные вопросы

1. Какова история появления языка программирования Си?
2. В чем основные особенности языка программирования Си? Являются ли они достоинствами или недостатками? Или достоинства - продолжения недостатков?
3. Почему Си называют «инструментом, острым, как бритва»? Удобно ли использовать Си для первоначального обучения программированию?
4. Нужно ли использовать комментарии в программах?
5. В чем заключается назначение препроцессора языка Си? Для чего применяется директива `#define`?
6. Какова структура исходных текстов программ на языке Си. В чем назначение директивы `#include`? Зачем нужны заголовочные файлы?
7. Что такое главная функция программы? Как записывается главная функция в Си?
8. Что такое заголовок и тело функции?
9. Что такое операторные скобки?
10. Что такое компоновщик?
11. В чем заключается функция оператора `return 0`? Почему передается ноль? Зачем передается значение в операционную систему?

Основные понятия языка программирования Си

Важнейшим понятием языка программирования являются *идентификаторы*. Идентификаторы используются для обозначения

переменных и функций. В языке Си идентификаторы представляют собой последовательность букв латинского алфавита, символов подчеркивания и десятичных цифр. Первым символом в идентификаторе должна быть буква.

ЗАМЕЧАНИЕ

Прописные и строчные буквы в языке Си различаются, то есть NAME, Name и name — три разных идентификатора.

Имена объектов в программе не должны совпадать с операторами языка и названиями стандартных функций.

Все переменные в программе должны относиться к тому или иному типу данных. Базовыми типами данных в языке Си являются следующие:

- `int` — целое число;
- `float` — число с дробной частью (с плавающей точкой);
- `char` — одиночный символ;
- `double` — вещественное число двойной точности.

ЗАМЕЧАНИЕ

В языке программирования Си нет выделенного логического типа данных, ЛОЖЬЮ считается нулевое значение переменной, например, целого типа, а ИСТИНОЙ — любое ненулевое значение.

Перед названием типа могут встречаться так называемые *модификаторы*. Например, `unsigned` для чисел означает, что данное число не может хранить отрицательное значение: если при 16-разрядной архитектуре компьютера тип `int` может хранить числа от –32 767 до 32 768, то `unsigned int` — от 0 до 65 535. При 32-разрядной архитектуре указанные значения удваиваются. Вообще говоря, размер выделяемой памяти для переменных того или иного типа в языке Си зависит от конкретного используемого компилятора.

Имеются также модификаторы `short` и `long`, указывающие, соответственно, на короткое (занимающее меньше памяти) и длинное (для хранения больших значений) целое.

При этом можно просто писать `short` и `long` вместо `short int` и `long int`, `unsigned` и `signed` вместо `unsigned int` и `signed int`.

Каждая используемая в программе переменная должна быть предварительно объявлена с указанием типа, и в дальнейшем она может использоваться для хранения значений только указанного типа. Иными

словами, язык программирования Си является языком со *статической типизацией*. Допустимо объявлять переменные одного типа как через запятую в одной строке, так и в разных строках, например:

```
int a,b,c;
int x;
char sym, A21;
```

Важным понятием языков программирования является *преобразование типов*. Так, если в выражении на Си появляются операнды различных типов, они преобразуются к некоторому общему типу, при этом к каждому операнду применяется следующая последовательность правил:

1) Если один из операторов имеет тип `long double`, остальные также преобразуются к данному типу;
2) В противном случае, если один из операндов в выражении имеет тип `double`, остальные преобразуются к данному типу;
3) В противном случае, если один из операндов имеет тип `float`, остальные преобразуются к данному типу;
4) В противном случае, если один из операндов имеет тип `unsigned long`, остальные преобразуются к данному типу;
5) В противном случае, если один из операндов имеет тип `long`, остальные преобразуются к данному типу;
6) В противном случае, если один из операндов имеет тип `unsigned`, остальные преобразуются к данному типу;
7) В противном случае все операнды преобразуются к типу `int`, при этом char преобразуется в `int` со знаком, `unsigned char` - в `int`, у которого старший байт всегда нулевой, `signed char` – в `int`, у которого знаковый бит передается из `char`, тип `short` – в `int` (знаковый или беззнаковый).

Кроме того, возможно явное (задаваемое программистом) преобразование типа, выполняемое, например, следующим образом:

```
X = y + (float)z;
```

Здесь перед вычислениями переменная `z` будет приведена к типу «число с плавающей точкой».

Другим важнейшим понятием языка являются *выражения*, широко распространенные в Си. Они состоят из *операндов* (переменные, константы, числа), соединенных знаками *операций* (сложения, умножения и др.). Порядок выполнения операций определяется их приоритетом и круглыми скобками, используемыми при записи выражений. Выражения широко используются в операторах присваивания, являющихся краеугольным камнем императивного программирования. В языке Си для записи присваивания используется одиночный символ равенства,

например:

```
x=a+b;
```

Удобной особенностью Си является то, что существует удобная сокращенная запись для присваивания одного и того же значения группе переменных, например:

```
f=b=c=d=100;
```

Арифметические операции в языке Си записываются следующим образом:

- `+` — сложение;
- `-` — вычитание и унарный минус;
- `*` — умножение;
- `/` — деление;
- `%` — остаток от деления.

Логические операции в языке Си, используемые при записи условий:

- `&&` — логическое И;
- `||` — логическое ИЛИ;
- `!` — логическое НЕ;

Знаки сравнения:

- `<` — меньше;
- `>` — больше;
- `==` — равно;
- `<=` — меньше или равно;
- `>=` — больше или равно;
- `!=` — не равно.

ЗАМЕЧАНИЕ

> Обратите особое внимание на запись сравнения на равенство — чтобы отличить ее от присваивания, в Си и унаследованных от него языках — C++, C#, Java и др. используются два стоящих рядом знака равенства! Начинающие программисты из-за этого могут допускать весьма трудноуловимые в ошибки, поскольку в Си допускается присваивание внутри условия.

В Паскаль-семействе языков программирования проверка на равенство записывается с помощью одного знака равенства, а присваивание — как `:=`. Однако вспомним, что язык программирования Си создан системными программистами и ему присуща лаконичность почти во всем. В программах присваивания встречаются гораздо чаще, нежели проверки на

равенство, и поэтому для экономии символов и времени программиста присваивание записывается с помощью одного знака равенства, а проверка — двумя идущими подряд знаками. Надо заметить, что и запись отношения неравенства отличается от принятой в Паскале <>, вместо этого используется !=, что по духу соответствует обозначению логического отрицания в Си.

Знаки для записи поразрядных логических операций:

- & — поразрядное логическое И;
- | — поразрядное логическое ИЛИ;
- ~ — поразрядное отрицание;
- ^ — поразрядное логическое исключающее ИЛИ.

В языке есть также и другие операции:

- ++ — инкремент (увеличение на единицу);
- -- — декремент (уменьшение на единицу);
- <усл> ? <в1> : <в2> — условная операция;
- Z= — набор операций, где Z — любая бинарная операция;
- >> — побитовый сдвиг вправо;
- << — побитовый сдвиг влево.

Очень удобными являются операции инкремента и декремента, которые можно рассматривать как сокращенные формы записи выражений x=x+1 и x=x-1. Аналогичным образом используется сокращенная запись вида z+=2 вместо z=z+2, y*=5 вместо y=y*5 и т. д. Здесь ярко проявляется происхождение Си — системные программисты стремились к краткости записи.

Для *условной операции*, или условного выражения, довольно экзотического и редко встречающегося в других языках программирования, результатом становятся значение <в1>, использованное в ее записи в случае, если первое выражение <усл> истинно, и значение <в2> — если значение первого ложно. Например,

```
y = (a>=b) ? a : b; /* если a >b – y=a, иначе y=b*/
```

приводит к присвоению y значения максимума среди a и b.

Контрольные вопросы и упражнения

1. Каковы принципы записи идентификаторов в языке программирования

Си?

2. Какие базовые типы данных языка программирования Си вы знаете?
3. Каков принцип использования логических значений в языке программирования Си?
4. Является ли Си языком со статической или динамической типизацией? Как записывается объявление переменных в языке программирования Си?
5. Как записывается оператор присваивания в языке программирования Си?
6. Какие существуют и как записываются арифметические операции в языке программирования Си?
7. Какие существуют и как записываются логические операции в языке программирования Си?
8. Какие операции, кроме арифметических и логических, возможны в Си? Приведите примеры.
9. Каково условное выражение языка программирования Си? Приведите пример использования.
10. Что такое явное преобразование типов в Си? Приведите пример.

Принципы ввода-вывода в языке Си

Говоря о принципах организации ввода-вывода информации в языке программирования Си, напомним, что сам язык не включает в себя операторы ввода-вывода, а все поддерживающие эти процессы операции осуществляются с помощью стандартных функций. Этот подход был применен разработчиками с целью достижения легкости переносимости между различными платформами.

В языке Си, поскольку он генетически связан с операционной системой UNIX, поддерживается концепция стандартных потоков ввода (`input`), вывода (`output`) и ошибок (`errors`). По умолчанию стандартный ввод связан с клавиатурой компьютера, стандартный вывод — с экраном дисплея, поток ошибок — также с экраном. Стандартные операции ввода-вывода поддержаны библиотечными функциями, описание которых содержится в заголовочном файле `stdio.h`. Это прежде всего `printf` для форматного вывода и `scanf` для ввода информации. Функция `printf` имеет следующий синтаксис:

```
printf("<форматная строка>"[,<список переменных>]);
```

Список переменных может отсутствовать, в этом случае `printf` используется просто для вывода на экран текстовой строки. Форматная

строка представляет собой строку символов, которые выводятся на экран, наряду с форматными символами, не выводимыми, но описывающими формат печати переменных (если они есть). Исторически форматный вывод ведет свою родословную еще от языка Фортран.

ЗАМЕЧАНИЕ

> Количество форматных символов должно строго соответствовать числу выводимых переменных, фигурирующих в списке после запятой!

В списке переменные, значения которых подлежат выводу на экран, разделяются запятыми, например:

```
printf("Результаты вычислений: a= %7.2f b=%7d\n",a,b);
```

В приведенном примере выводятся значения двух переменных — действительной `a` и целой `b`, после этого происходит переход на новую строку.

Форматные символы в общем случае имеют вид

```
%[<признаки>][<ширина>][.<точность>]<тип>
```

Все параметры, приводимые в квадратных скобках, являются необязательными. Обязательно правильное указание типа выводимого значения из перечисленных далее:

- `d` — целое число, допустимо использование `i`;
- `u` — целое число без знака;
- `c` — одиночный символ;
- `s` — строка символов;
- `f` — число с плавающей точкой (действительное);
- `e` — выводится число в экспоненциальной форме, например 1.23e+2.

Признаки могут быть следующими:

- `-` — указывает на выравнивание по левому краю;
- `+` — указывает на необходимость вывода знака числа.

Параметр `<ширина>` задает, сколько позиций символов на экране отводится для вывода значения данной переменной. Этот параметр очень удобен для выравнивания при печати данных, например, в таблицах.

В случае вывода чисел, имеющих дробную часть после десятичной точки, возможно указание параметра `<точность>`, который определяет, сколько знаков внутри определенного параметром ширины поля будет отведено на дробную часть. Например, если значение переменной равно –12.4567, то при применении к ней форматной строки `%7.2f` будет выведено `–12.45`.

Внутри форматной строки возможно появление *специальных символов*, запись которых начинается с символа \. Допустимы следующие комбинации:

- \7 — для вывода звукового сигнала;
- \n — для перехода на новую строку;
- \\ — для печати символа \;
- \r — для вывода символа «возврат каретки»;
- \t — для горизонтальной табуляции;
- %% — для печати символа %;
- \" — для печати символа кавычек.

Функция scanf используется для ввода данных с клавиатуры (со стандартного устройства ввода). Она имеет следующий вид:

```
scanf("<форматная строка>",<список указателей>);
```

В отличие от функции printf, здесь <список указателей> является обязательным. При этом он представляет собой список *адресов*, то есть указателей на переменные, а не самих переменных.

В языке Си для взятия адреса переменной применяется операция & (используется во всех случаях печати, кроме случая символьных строк, поскольку переменные, описывающие строки, фактически являются в нем указателями на массивы символов), которая подробнее рассматривается в последующих разделах. <форматная строка> содержит форматные символы для выводимых переменных, формируемые по тем же правилам, что и при использовании функции printf. Их количество и тип должны строго соответствовать числу и типам вводимых переменных. Пример использования функции scanf:

```
scanf("%d %f",&n,&x);
```

ЗАМЕЧАНИЕ

Внутри форматной строки в scanf запрещено использовать любые другие символы, кроме форматных. Разделять их должен строго один пробел (можно обходиться без разделителя, но это затрудняет чтение программы).

Контрольные вопросы

1. Включает ли язык программирования Си операторы ввода-вывода? Почему?
2. Что такое форматный вывод? Перечислите элементы форматной строки в Си.

3. В чем заключается принцип использования указателей при вводе значений с помощью функции `scanf()`?

Структурирование программ на языке Си

Как уже отмечалось, современные программы могут содержать тысячи, десятки тысяч и даже миллионы строк исходного текста на языке высокого уровня. Сложность современных программных комплексов поражает воображение. Это относится и к программам на языке Си. Единственным способом справиться со сложностью, известным человечеству с незапамятных времен, является принцип «разделяй и властвуй», или решение сложной задачи путем разбиения ее на менее сложные подзадачи.

Итак, сложная программа на языке программирования Си должна некоторым образом разбиваться на части — структурироваться. Ка

Исходный текст программы может содержаться в нескольких файлах. При компиляции указывается список файлов, подлежащих обработке. Часть файлов при этом может иметь расширение `.h` и являться *заголовочными* — в них обычно содержатся определения структур данных, глобальных переменных и заголовки функций. Основная часть файлов исходных текстов на языке Си имеет расширение `.c` (у программ на C++ — `.cpp`).

Но главным механизмом структурирования программ на языке Си служит аппарат функций. В языке не предусматриваются как таковые процедуры, все подпрограммы называются функциями независимо от факта возврата значения.

Как известно, функция содержит заголовок и тело. Каждая функция на Си до ее вызова (выше по тексту программы) должна быть либо определена (заголовок + тело функции), либо объявлена (заголовок, завершающийся символом `;`) с указанием:

- типа возвращаемого функцией значения;
- имени функции;
- перечня аргументов — *формальных параметров*.

ЗАМЕЧАНИЕ

> Тип возвращаемого значения, количество и типы аргументов иногда называют сигнатурой функции.

Пример определения функции:

```
/* функция суммирует целые аргументы a и b */
int sum(int a,int b)
{
   int k,l; /* локальные переменные функции */
```

```
   k=a;l=b;
   return k+l;
}
```

Обратите внимание на то, что внутри функции `sum` объявляются переменные `k` и `l`. Это так называемые *локальные переменные*. Такое название они получили потому, что недоступны извне функции (свойство переменной быть доступной в одной части программы и недоступной в другой называется еще *областью видимости*). Также отметьте то, что в списке формальных параметров имя типа в программах на Си необходимо указывать перед каждой переменной.

ЗАМЕЧАНИЕ

> В разных функциях может фигурировать произвольное число одноименных переменных, хотя они будут совершенно разными, в том числе ссылаться на различные ячейки памяти.

В теле функции доступны:
а) локальные переменные;
б) переменные, перечисленные как формальные параметры;
в) *глобальные переменные*.

Глобальные переменные— переменные, объявления которых в программе на Си находятся вне функций, они доступны из любых функций данного файла исходного текста программы. Если необходимо получить доступ из функции, расположенной в другом файле исходного текста, используется ключевое слово `extern`.

Еще одним важным понятием является *время жизни* переменной. Локальные переменные функции «живут», пока выполняется данная функция. Как правило, место в памяти для них резервируется в стеке – динамической структуре данных, из которого затем по выходу из функции информация удаляется. В отличие от этого, глобальные переменные размещаются в статической части памяти программного модуля и «живут» все время, пока работает программа. Могут использоваться также переменные, хранимые в динамически распределяемой памяти («куче», heap) – они «живут» пока не освобождается занимаемая ими память.

Как было сказано в соответствующей главе, правила написания надежных и легко модифицируемых программ требуют минимальных связей между модулями и минимального использования глобальных переменных — лишь в случаях, когда это действительно обоснованно.

После определения функция может быть вызвана в произвольном месте программы, ее имя может входить в выражения

```
s=sum(2,2); /* вызов функции в операторе присваивания */
```

```
m=n=-2;
printf("сумма равна %d\n",sum(m,n)); /* так тоже можно */
```

Обратите внимание на то, что во втором случае в качестве значений аргументов передаются текущие значения переменных с именами m и n. Эти переменные при вызове называются *фактическими параметрами* функции. Что происходит при обращении к функции? Система берет *текущие значения* переменных m и n, например, это будут числа –2 и –2, при входе в функцию sum использует эти значения для формальных параметров a и b, которые затем используются в теле функции по их именам. При этом модифицировать переменные m и n в вызывающей функции из функции sum напрямую невозможно. Подобный подход при вызове подпрограмм в языках программирования называется передачей параметров *по значению*, что противопоставляется *передаче по ссылке*, используемой, например, в Паскале и позволяющей вызванной процедуре изменять значения переменных вызывающей функции.

ЗАМЕЧАНИЕ

> Если изменение значений вызывающей функции необходимо, для этого используют специальный трюк с операцией &, описанный ниже при рассмотрении функции scanf() и указателей языка Си.

Контрольные вопросы

1. Каковы базовые принципы структурирования программ на языке Си?
2. В чем разница между глобальными и локальными переменными? Стоит ли повсеместно использовать глобальные переменные? Надо ли их полностью исключить? Почему?
3. Что такое «область видимости» переменной?
4. В чем разница между передачей параметров в языках программирования по ссылке и по значению? Как в программе изменить значение переменной вызывающей функции на языке Си?
5. Что означает термин «время жизни» переменной?

Структуры данных и управления языка программирования Си

Наряду с простыми одинарными переменными одного из базовых типов язык Си позволяет объявлять массивы. Под массивом понимается набор (упорядоченное множество) элементов одного типа. При этом возможен доступ к элементам массива путем указания индекса (то есть номера элемента в массиве). При этом, в отличие от многих языков программирования, в языке Си принято соглашение о том, что элементы

массива нумеруются, начиная с нуля, то есть первый элемент — элемент с номером 0. Смысл подобной нумерации, несколько странной для читателя, не являющегося специалистом в программировании, заключается в том, что индекс можно рассматривать как величину смещения от начальной ячейки памяти, от которой в памяти помещается массив. При этом Си оказал огромное влияние на другие языки — даже современные языки Паскаль-семейства.

Объявляются массивы там же, где и обычные переменные, то есть в начале функции, но в квадратных скобках после имени указывается размерность (количество элементов), например:

```
int a,b,c;
float x[20];
char c,str[40];
```

Здесь описаны массив вещественных чисел из 20 элементов, а также массив символов (строка) размерностью 40. Обратите внимание на то, что в языке Си *строки* представляют собой массивы символов, при этом на конец строки указывает символ с кодом (значением), равным нулю.

При работе с массивами часто используют цикл по его элементам, при этом значение переменной, используемой в качестве индекса, либо увеличивается, либо уменьшается на единицу. Поскольку нумерация элементов начинается с нуля, переменная-индекс имеет значение от 0 до $N-1$, где N — количество элементов массива. Связано это с тем, что фактически индекс элемента в массиве на Си представляет собой смещение относительно значения указателя в памяти на первый элемент массива.

Массивы могут фигурировать в качестве формальных и фактических параметров функций. Пример:

```
/* обработка массива в Си */
#include <stdio.h>
#include <math.h>

int Pecat_kornei(float args[],int kol)
{
int i;
float res;
for (i=0;i<kol;i++)
{
    if (args[i]>0)
    {
      res=sqrt(args[i]);
      printf("Корень из %d – го элемента равен %f\n",i,res);
```

```
    }
    else
      printf("Элемент массива номер %d меньше нуля !",i);
}
}
```

Используется стандартная библиотека математических функций, описанных в заголовочном файле `math.h`. Функция `sqrt()` возвращает значение квадратного корня числа.

В приведенной программе используется цикл `for` языка программирования Си, который является весьма гибким инструментом. После ключевого слова `for` в круглых скобках, разделенные точками с запятой, идут три выражения. Первое — группа операторов языка, разделяемых запятыми, которые должны выполниться до первого выполнения цикла. Второе — логическое условие, проверяемое перед каждым выполнением цикла и определяющее, продолжать выполнение или нет. Наконец, третье выражение — действия (через запятую), которые выполняются по окончании каждой итерации. Само тело цикла, то есть действия, выполняемые циклически, заключается в фигурные скобки, если это — не один оператор, в противном случае это просто следующий за круглыми скобками оператор. Выглядит все следующим образом:

```
for ([<ид1>, <ид2>,…]; [<условие>]; [<ит1>, <ит2>, <ит3>…])
  <тело цикла>
```

Здесь <*ид*$_n$> — инициализирующее действие с номером *n*, <*ит*$_n$> — действия-итераторы. Цикл выполняется следующим образом. Сначала производятся действия, приписываемые <*ид*$_n$>. Затем проверяется условие, и если оно истинно, выполняется тело цикла. После выполнения тела цикла по очереди выполняются действия-итераторы, снова проверяется условие и т. д. Процедура может быть проиллюстрирована рис. 14.

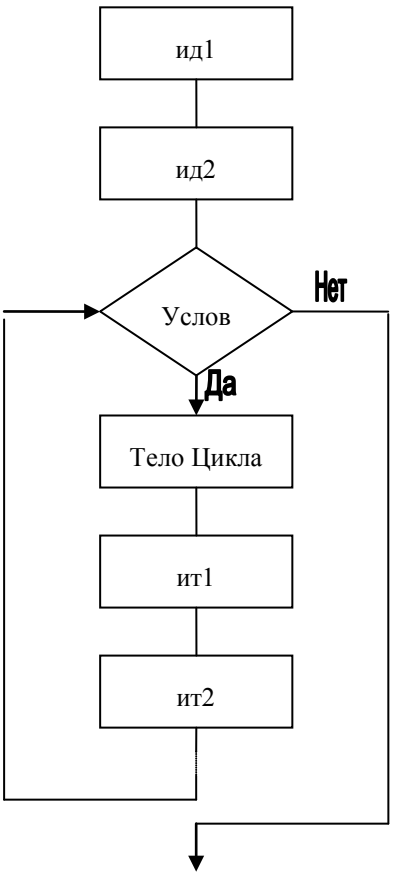


Рис. 14

ЗАМЕЧАНИЕ

Конструкция for может использоваться для задания бесконечного цикла в случае отсутствия инициализирующих действий, условия и итераторов: for (;;);. В этом случае нужно предусмотреть другой способ прерывания исполнения — например, с помощью break, goto или return.

ЗАМЕЧАНИЕ

Если условие изначально ложно, тело цикла может не выполниться ни разу, в отличие от инициализирующих действий.

Цикл `for` часто используется для эмуляции стандартного цикла с параметром – цикла ДЛЯ других языков программирования, хотя им и не является (в строгом смысле слова). Чтобы превратить его в цикл с параметром, необходимо, чтобы инициализирующие действия включали задание начального значения переменной цикла, условие относилось к границе изменения параметра, а действия-итераторы включали изменение значения переменной цикла. Пример:

```
/* умножение всех элементов массива на 2*/
for (i=0; i<N; i++) a[i]*=2;
```

Несмотря на частое использование в качестве цикла с параметром, возможности `for` языка программирования Си гораздо шире. Это иллюстрируют следующие примеры:

```
/* пробег сразу двух массивов во встречных направлениях */
for (i=0,j=M; i<N && j>=0; i++,j--) a[i]=b[j];


/* тело цикла пустое — все делают итераторы*/
for (i=0;i<N;a[i++]=0) ; /* сброс массива */
```

ЗАМЕЧАНИЕ

Программистам не рекомендуется злоупотреблять подобными трюками, поскольку они затрудняют понимание программы человеком и чреваты внесением трудноуловимых ошибок.

С помощью цикла `for` могут быть эмулированы и стандартные циклы вида ПОКА и ДО (читателю предлагается самостоятельно разобраться, каким образом). Несмотря на это отдельные формы этих циклов в языке также присутствуют, примеры можно найти в главе о базовых конструкциях императивных языков.

В примере используется также условный оператор языка Си. Его синтаксис выглядит следующим образом:

```
if (<условие>)
    <действие 1>
else
    <действие 2>;
```

В случае истинности условия (которое может быть сложным, где фигурируют несколько простых условий, связанных логическими связками И — &&, ИЛИ — ||) выполняются <*действия 1*>, в качестве которых может выступать один оператор или группа операторов, заключенных в

операторные скобки (фигурные скобки). Если условие не выполняется, то исполняется <*действие 2*>. Ветвь ИНАЧЕ (`else`) может отсутствовать. Этот случай соответствует сокращенному условному выражению, когда в случае ложности условия просто ничего не происходит. Пример:

```
if (summa>0 && ostatok>0 && (!ograblenie())) vydat_dengi();
```

ЗАМЕЧАНИЕ

> Благодаря наличию в языке Си условного выражения иногда можно обойтись без использования оператора if.

Кроме оператора `if` язык Си поддерживает оператор выбора, выглядящий следующим образом:

```
switch(<пер>)
{
  case <зн1>:<действия1>;break;
  case <зн2>:<действия2>;break;
  case <зн3>:<действия3>;break;
  …
  [default:<действия_если_ни_одно_не_подошло>]
}
```

В операторе выбора для выбора альтернативной ветви используется значение переменной в скобках после `switch` (переменная должна быть перечислимого типа, например целочисленная). Если оно совпадет с <$зн_1$>, выполняются <$действия_1$>, если со <$зн_2$> — <$действия_2$> и т. д. Если действия включают несколько операторов, последние не заключаются в операторные скобки, а лишь записываются один за другим и разделяются точкой с запятой. Строк `case` столько, сколько важных вариантов должна обработать программа. Иногда в оператор выбора добавляется строка `default`, содержащая действия, выполняемые, если значение переменной не совпало ни с одним из <$зн_i$>. Обработка иллюстрируется рис. 15.

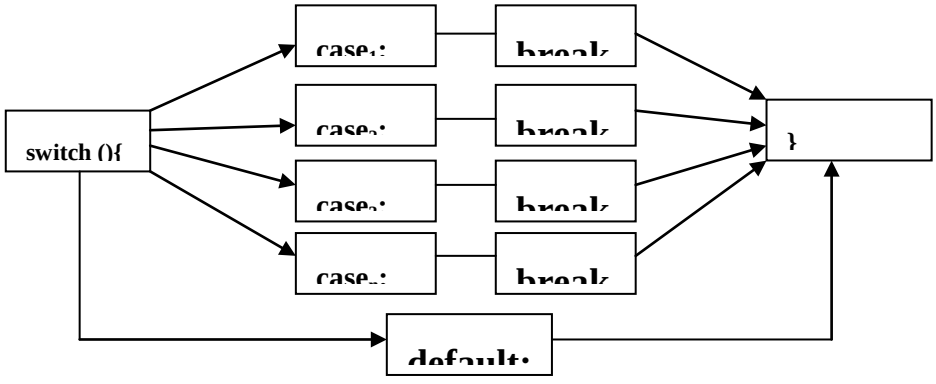


Рис. 15

ЗАМЕЧАНИЕ

Поскольку ветви case реализованы в языке программирования Си фактически как метки, для того чтобы при входе в каждую из веток и выполнения соответствующих действий автоматически не происходили переход к следующей строке и выполнение строки со следующим case, необходимо каждую ветвь заканчивать оператором break.

Пример использования `switch` приводится в лабораторном практикуме (приложение А).

Помимо простых, или одномерных, массивов в языке Си возможна работа с так называемыми многомерными массивами, то есть массивами, элементы которых также являются массивами. В простейшем случае двумерный массив (аналогом которого в математике является двумерная матрица) может быть описан следующим образом:

```
float matr[10][20];
```

Здесь объявляется массив из 10 строк по 20 элементов в каждой. Доступ к элементам массива осуществляется очевидным образом:

```
y+=matr[i][j];
```

ЗАМЕЧАНИЕ

Обратите внимание на то, что каждое измерение записывается в своих квадратных скобках.

В примере к переменной `y` добавляется значение элемента массива `matr`, находящееся на пересечении *i*-й строки и *j*-го столбца матрицы.

Аналогично выполняется работа с трехмерными и имеющими большее число измерений массивами.

Важным элементом программирования является возможность задания значений переменной при объявлении. В некоторых языках, например, принадлежащих к Паскаль-семейству, по умолчанию все переменные инициализируются нулем. В Си по умолчанию переменным не присваивается никакого значения, что служит источником трудноуловимых ошибок, ведь значение неинициализированной явным образом переменной соответствует случайному числу, находившемуся в данной ячейке памяти до запуска программы. Причем может оказаться, что 50 раз при запуске там случайно оказывается, например, ноль и программа работает штатно, а затем происходит неожиданный сбой на 51-м запуске, когда в данной ячейке будет иное значение. В связи с этим хорошим тоном в программах на Си является запись вида

```
int k=0,b=2; /* инициализация при объявлении */
float z=3.21;
```

Подобным образом можно инициализировать и массивы:

```
int a[7]={-7,0,1,2,15,21,-1}; /* одномерный массив*/
char ima[20]="дядя Ваня"; /* строка — массив символов */

int mat[2][3]={ {1, 2, 3},
                {-1,0, 1}
              };
```

Как и в других языках программирования, в Си возможно создание на основе базовых типов данных более сложных конструкций. В некоторых языках подобного рода конгломераты называются записями, в других — структурами и т. д.

Идея структуры состоит в том, что под одним именем группируются сразу несколько параметров (элементов данных). Можно попытаться описать, например, гоночный автомобиль следующим набором параметров: мощность двигателя (целое число лошадиных сил), изготовитель (строка символов), тип мотора (строка символов), количество передач (целое число), максимальная скорость (число с плавающей точкой). В языке программирования Си возможно объявление соответствующей данному описанию структуры:

```
struct
{
  int Power;
  char Manufacturer[40];
  char Motor[40];
  int transmissons;
  float MaxSpeed;
} Auto;
```

После этого объявления можно обращаться к отдельным элементам данных структуры `Auto`. Делается это следующим образом, например:

```
Auto.Power=340;
Auto.MaxSpeed=380.2;
scanf("%s",Auto.Manufacturer);
```

Помимо простого объявления структуры в языке Си возможно объявление структуры с определением нового типа данных с именем, совпадающим с именем структуры. Для этого используется ключевое слово `typedef`. В приведенном примере, например, заголовок мог бы выглядеть следующим образом:

```
typedef struct {…} Auto;
```

После этого можно объявлять переменные нового типа `Auto`, например:

```
Auto car,car2;
```

Дальнейшее обращение к членам данных переменных `car` и `car2` производится стандартным образом, например: `car.Power`, `car.Motor` и т. д.

ЗАМЕЧАНИЕ

Понятие структуры является предком более сложного понятия, используемого в объектно-ориентированных языках программирования (например, C++), — понятия класса.

Контрольные вопросы

1. В чем заключается использование массива в программе на Си? Приведите пример.
2. В чем заключается обработка массива с помощью оператора `for` языка программирования Си? Приведите пример.
3. Как инициировать массивы при объявлении на языке Си? Приведите примеры для различных типов данных.
4. Опишите условный оператор языка программирования Си. В чем разница между полной и сокращенной формами условного оператора?
5. Опишите оператор выбора в языке Си. Зачем он нужен, если есть условный оператор?
6. Приведите пример использования структур (записей) в Си. Как создать новый тип данных в программе на Си с помощью записей?
7. Опишите, как с помощью оператора `for` языка программирования Си представлять бесконечный цикл, цикл ПОКА и цикл ДО. Как прервать бесконечный цикл в языке программирования Си?

Обработка текстовых строк

Текстовая информация представляет собой наиболее часто встречающуюся разновидность информации, обрабатываемой на современных компьютерах. Ушли в прошлое времена, когда компьютер был в основном вычислительным устройством. Сегодня зачастую компьютер — просто пишущая машинка в конторе!

В языке программирования Си текстовые строки рассматриваются как одномерные массивы символов, при этом ограничителем строки при выводе, например, на экран или в файл считается нулевой символ (символ с кодом, равным нулю). В связи с этим для объявления строки с максимальным числом хранимых символов 19 возможно следующее описание:

```
char x[20];
```

Стандартная библиотека функций для работы с текстовыми строками, описанная в заголовочном файле `string.h`, имеет следующие наиболее употребительные функции для работы со строками:

- `strcpy(<`*строка1*`>,<`*строка2*`>)` — для копирования строк;
- `strcmp(<`*строка1*`>,<`*строка2*`>)` — для лексикографического сравнения строк (возвращает ноль, если строки одинаковые, положительное значение, если первая строка «больше», и отрицательное — если «меньше»);
- `strcat(<`*строка1*`>,<`*строка2*`>)` — для приписывания второй строки в конец первой (конкатенации строк);
- `strstr(<`*строка1*`>,<`*строка2*`>)` — для проверки вхождения подстроки в строку (если не найдет, возвращает `NULL`, в противном случае возвращает указатель на начало первого вхождения искомой подстроки в исходную строку);
- `strlen(<`*строка*`>)` — для нахождения длины строки (не считает завершающий строку символ `\0`).

Рассмотрим семантику некоторых из перечисленных функций библиотеки обработки строк. Интересно передать ее по аналогии с обработкой числовых данных с помощью стандартных операций языка программирования Си:

- `strcpy(s1,s2); /* аналог s1=s2 */`;
- `strcat(s1,s2); /* аналог s1+=s2 */`;
- `if (!strcmp(s1,s2)) /* аналог if (s1==s2) */`.

Примеры возможного использования строковых функций:

```
char s1,s2="дядя ",s3="Ваня";
strcpy(s1,s2); /* s1= "дядя " */
if (!strcmp(s1,s2))
{
  strcpy(s2,s3); /* s2= "Ваня" */
  strcat(s1,s2); /* s1="дядя Ваня" */
}
char *str1 = "коля", *str2="оля";
*ptr = strstr(str1, str2);
if (ptr!=NULL) printf("Подстрока: %s\n", ptr);
```

Контрольные вопросы

1. Как представляются строки в «чистом» Си? Что означает символ с кодом ноль (`\0`)?

2. Какой заголовочный файл подключается для использования библиотеки работы со строками в программах на Си? Какие основные функции содержит библиотека?

Использование параметров функции main()

Как уже упоминалось, в Си имеется возможность получения программой некоторых параметров при ее запуске операционной системой ЭВМ. Поскольку исполнение программы на Си начинается с функции `main`, это организовано через ее аргументы. Чтобы указать, что программа использует параметры, вместо имени `main()` с пустыми круглыми скобками необходимо указать набор предопределенных параметров, как показано в примере далее. Фактически, параметры могут задаваться либо в командной строке оболочки операционной системы, либо другими средствами.

```
/* Передача параметров в функцию main*/
#include <stdio.h>
int main (int argc, char *argv[])
{
  int i=0;
  printf ("\n Имя программы %s", argv[0]);
  for (i=1; i>=argc; i++)
     printf ("\n аргумент %d равен %s", argv[i]);

  return 0;
}
```

Итак, первым предопределенным параметром является целочисленная переменная `argc`. Она задает количество аргументов, фактически переданных при запуске программы. Данное обстоятельство позволяет гибко обрабатывать, например, последовательности чисел произвольной длины — подсчет количеств берет на себя операционная система (используется при написании одного из вариантов программы подсчета суммы нечетных в лабораторном практикуме).

Далее идет массив строк `char *argv[]`. Аргументы передаются в программу в виде строк текста, таким образом на программу возлагается обязанность в случае необходимости превратить их в числа (можно использовать, например, функцию ввода из текстовой строки `sscanf()` или `atoi()`). Важным является то обстоятельство, что в качестве первого элемента массива строк по умолчанию передается имя самой программы (путь к исполняемому файлу) `argv[0]`. Соответственно, `argc` всегда как минимум будет равен единице, а фактически набранные пользователем параметры доступны через `argv[1]`, `argv[2]`, `argv[3]`...

ЗАМЕЧАНИЕ

В программе на Си можно получить из операционной системы значение ряда системных параметров — так называемых переменных окружения, для чего в main добавляется третий предопределенный аргумент. Однако эта возможность на практике используется редко: int main (int argc, char *argv[], char *argp[]).

Контрольные вопросы

1. Что такое параметры операционной системы, передаваемые в программу? Каким образом они могут использоваться?
2. Какой параметр идет первым (с индексом ноль) в массиве `argv[]`?
3. Эквивалентны ли следующие записи:
   - `char **argv;`
   - `char *argv[];`
   - `char argv[][]`?

Работа с файлами

Помимо средств ввода-вывода, работающих со стандартными потоками ввода, вывода и ошибок (фактически, это клавиатура и экран дисплея), в стандартных библиотеках языка программирования Си присутствуют средства для работы с информацией, находящейся на дисках и организованной в виде файлов. Существенно облегчает понимание принципов работы тот факт, что эти функции используют те же принципы, что и функции для ввода-вывода со стандартных устройств.

При работе с файлами в языке Си используется понятие *файловый дескриптор*. Для обозначения указателя на файл зарезервировано специальное ключевое слово `FILE` — одно из немногих ключевых слов в языке программирования Си, записываемых прописными буквами. Например,

```
FILE *fp;
```

описывает дескриптор файла с именем `fp`. После объявления файлового дескриптора его необходимо связать с физическим файлом на диске (файла может еще не существовать, если он создается путем открытия его на запись или добавление). Для открытия файла может использоваться стандартная функция `fopen`:

```
fp=fopen("<имя файла>","<режим>");
```

Здесь `<имя файла>` — строка с именем файла, соответствующая его наименованию в операционной системе. `<режим>` может быть следующим:

- `"w"` — запись;
- `"r"` — чтение;
- `"a"` — добавление.

После символа, задающего режим доступа, может идти дополнительный символ, определяющий доступ в текстовом или бинарном режиме. По умолчанию подразумевается текстовый режим доступа.

Если файл открывается на чтение и его нет на диске, `fopen` возвращает `NULL` — специальное предопределенное значение, в противном случае — указатель на файл (дескриптор).

Открытие на запись уже существующего файла предполагает перезапись его содержимого.

Открытие на добавление существующего файла предполагает приписывание информации к его концу.

Для файлового ввода-вывода существует ряд функций, соответствующих функциям работы со стандартным вводом-выводом, в частности:

- `fprintf` — для вывода в файл;
- `fscanf` — для чтения из файла.

Отличие этих функций от `scanf` и `printf` заключается лишь в том, что первым аргументом, идущим после открывающей круглой скобки, должен быть дескриптор файла, например:

```
fprintf(fp,"2 + 2 = %f\n", a);
```

После использования файла его необходимо *закрыть*. Это можно сделать с использованием функции `fclose`, имеющей такой синтаксис:

```
fclose(<дескриптор файла>);
```

Следующая программа осуществляет ввод строки с клавиатуры, сохранение ее в текстовом файле, чтение строки из файла и вывод ее на экран:

```
/* Пример работы с файлом в языке программирования Си */
#include <stdio.h>

int main()
{
  FILE *f;
  char str[40];
  printf("Введите строку:");scanf("%s",str);
  f=fopen("file.txt","w");
  if (f==NULL) return -1;
```

```
  else
  {
fprintf(f,"%s\n",str);
fclose(f);
f=fopen("file.txt","r");
fscanf(f,"%s",str);
fprintf("\nстрока из файла: %s",str);
fclose(f);
return 0;
  }
}
```

Файл одновременно можно открыть только на чтение или только на запись. Поэтому два раза используется вызов функции `fopen` с одной и той же переменной — дескриптором файла, но с разными режимами доступа:

Текст программы:

```
#include <stdio.h>
#include <string.h>

int main()
{
  FILE *fp,*fp1;
  char Str[80];
  int sc=0;

printf ("Программа подсчета числа вхождений подстроки \"пере\" в
файле test.txt\n");
  printf ("и замены слова \"абитуриент\" на слово \"студент\"\n");

  // Открываем исходный файл
  if ((fp=fopen("test.txt","r"))==NULL)
  {
    printf("Ошибка открытия файла test.txt");
    return 1;
  }

  // Создаем промежуточный файл для копирования
  if ((fp1=fopen("test.bak","w"))==NULL)
  {
    printf("Ошибка создания промежуточного файла");
    return 2;
```

```
  }

  while (!feof(fp)) // Пока не конец исходного файла
  {
    fscanf(fp,"%s",Str); // Читаем строку из исходного файла
    if (strstr(Str,"пере")!=NULL) sc++; // Считаем подстроки "пере"
    if (!strcmp("абитуриент",Str))
      strcpy(Str,"студент"); // Замена слов
    if (!feof(fp)) fprintf(fp1,"%s ",Str);
  }

  fclose(fp);
  fclose(fp1);

  if(remove("test.txt")) // Удаляем исходный файл
  {
    printf("Ошибка удаления файла");
    return 3;
  }

  if(rename("test.bak","test.txt")) // Переименовываем промежуточный
файл
  {
    printf("Ошибка переименования промежуточного файла");
    return -1;
  }
  else
    printf("Было найдено %d \"пере\"\n",sc);

  return 0;
}
```

Файл test.txt (пример): "абитуриент сможет перейти через барьер, и наконец стать настоящим человеком, который поступит в вуз и будет переходить с курса на курс, так что абитуриент в конце концов станет настоящим специалистом — инженером!"

Обратите внимание на использование в этой программе функций `rename` и `remove` из стандартной библиотеки, описываемой заголовочным файлом `stdio.h`, для переименования и удаления файлов. В программе сначала создается промежуточный файл `test.bak`, в который из исходного файла копируется вся информация, но слова «абитуриент»

заменяются словами «студент», потом исходный файл удаляется, а промежуточный получает имя исходного файла — `test.txt`.

Контрольные вопросы

1. Что такое файловый дескриптор в программе на языке Си? В чем отличие дескриптора от имени файла?
2. Как открыть файл в программе на языке Си? Как задать режим доступа к файлу?
3. Как используется `NULL` при открытии файла в программах на Си?
4. В чем заключается применение функций `remove()` и `rename()`?
5. Нужно ли закрывать файл после использования? С помощью какой функции это можно сделать?

Сумма нечетных на языке Си

Итак, мы уже располагаем некоторыми знаниями о языке программирования Си. Вернемся к примеру, рассмотренному во введении книги. Напишем программу суммирования нечетных членов числовой последовательности.

Предположим, что числа вводятся пользователем с клавиатуры — столько, сколько необходимо, пока не будет введен ноль, служащий ограничителем последовательности.

```
/* Сумма нечетных */
#include <stdio.h>

int main()
{
  int a,s=0; /*s – сумма, обнуляем*/

  printf("Программа нахождения суммы нечетных членов числового
ряда\n");
  printf("Вводите целые числа или ноль для завершения\n");
  do
  {
    printf(">");scanf("%d",&a);
    if (a%2) s+=a; /* экивалентно if (a%2!=0) s=s+a;*/
  } while (a!=0); /* можно просто while(a); но это менее понятно*/
  printf("сумма нечетных среди введенных =%d\n",s);
  return 0;
}
```

Обратите внимание на некоторые особенности программы. В ней

соблюдены правила хорошего тона: после запуска до пользователя доводится краткая информация о ее предназначении и делается приглашение к вводу чисел. При этом для каждого очередного числа выдается >. Вывод приглашения с помощью `printf()` и ввод с помощью `scanf()` сгруппированы в одной строке, что вполне логично.

Используется цикл `do` (ДО) для ввода чисел и подсчета суммы нечетных. Для определения четности/нечетности введенного числа используется встроенная в язык операция % взятия остатка от деления. Если остаток от деления на 2 ненулевой — число нечетное, в противном случае — четное. Поскольку в языке программирования Си нет выделенного логического типа и любое ненулевое число воспринимается как логическая ИСТИНА, можно записать `if (a%2)` вместо `if (a%2!=0)`. Этим, как и сокращенной записью `s+=a` вместо `s=s+a`, соблюден дух Си. Однако это несколько ухудшает прозрачность программы, и при проверке ограничителя цикла подобный трюк не применяется.

Напишем теперь второй вариант программы. В нем сначала будем запрашивать пользователя о количестве подлежащих обработке чисел. Кроме того, вместо условного оператора `if` применим условное выражение ?:

```
/* Сумма нечетных вторым способом */
#include <stdio.h>
#define MAXN 1000 /* максимальная длина последовательности чисел */

int main()
{
  int a,i,n,s=0; /*s – сумма, обнуляем*/

  printf("Программа нахождения суммы нечетных членов числового
ряда\n");
  printf("Введите количество чисел (<%d):",MAXN);scanf("%d",&n);
  for (i=0;i<n;i++)
  {
    printf("введите очередное число>");scanf("%d",&a);
    s+=((a%2)?a:0); /* прибавляем или само число a, или ноль */
  }
  printf("сумма нечетных среди введенных =%d\n",s);
  return 0;
}
```

В этом варианте следует обратить внимание на ввод ограничения сверху на длину обрабатываемой числовой последовательности с помощью директивы `#define`. Затем у пользователя запрашивается действительное

количество чисел, подлежащих обработке, `n`. Для ввода и обработки используется единый цикл `for`, в котором переменная цикла `i` меняется от 0 до $n - 1$ путем записи `for (i=0;i<n;i++)` — это соответствует стилю Си, поскольку в нем с помощью `for` часто обрабатываются массивы, а в них нумерация начинается с нуля. Всего получится n повторений, что нам и надо. После ввода числа к `s` добавляется результат вычисления условного выражения. Сначала вычисляется остаток от деления `a` на 2. Этот результат по правилам языка программирования Си воспринимается как логическая ИСТИНА, если остаток ненулевой, иными словами, число нечетное, и как логическая ЛОЖЬ, если нулевой, то есть число четное. В первом случае все условное выражение принимает значение, указанное до двоеточия, а именно само число `a`, во втором случае — ноль, указанный после двоеточия. Полученное число прибавляется к `s`, давая возможность получить нужный результат.

Контрольные вопросы

1. Как работает оператор `if (a%2)`? Какая особенность работы с логическими значениями языка программирования Си используется в данной программе?
2. С какими целями используется директива препроцессора `#define`?
3. Поясните, как работает оператор `s+=((a%2)?a:0)`. Могли бы вы предложить альтернативные варианты подсчета суммы нечетных на языке программирования Си?

Сортировка массивов

Весьма важную роль в программировании имеют задачи поиска и сортировки. Неудивительно, что один из томов свого знаменитого труда «Искусство программирования» Дональд Кнут так и назвал — «Сортировка и поиск» [10].

Мы не будем сейчас углубляться в теорию, а интересующихся вопросом отсылаем к упомянутой книге. Заметим лишь, что решать эти задачи можно различными способами, причем существенно различающимися между собой по эффективности — времени выполнения алгоритма и требуемому объему памяти.

Приведем здесь лишь один из примеров реализации сортировки на языке программирования Си — не самого эффективный, но прозрачный алгоритм (сортировка выбором), в котором сначала на первое место в массиве ставится наименьший элемент путем перебора всех элементов, сравнения их с первым и перестановки в случае необходимости, а затем процедура повторяется для второго и оставшихся элементов.

```
/* сортировка перестановками на Си */
#define N 10 /* размер обрабатываемого массива */

int main()
{
  int a[N];
  int i,j,buf;

  printf("Программа сортировки массива\n");
  printf("Введите %d чисел:",N);
  for (i=0;i<N;i++) scanf("%d",&a[i]);

  for (i=0;i<N-1;i++)
    for (j=i+1;j<N;j++)
      if (a[i]>a[j])
      {
        buf=a[i];
        a[i]=a[j];
        a[j]=buf;
      }

  printf("После сортировки:\n");
  for (i=0;i<N;i++) printf("%d ",a[i]);
  printf("\n");

  return 0;
}
```

В программе используется `#define` для задания размера массива. Поскольку обработка с помощью директив препроцессора происходит до компиляции, запись `int a[N]` допустима, несмотря на поддержку лишь статических массивов в Си — к моменту выделения памяти `N` будет заменено в программе конкретным числом.

Для сортировки используются два цикла, один из которых вложен в другой. Внешний цикл использует переменную `i` для выбора сначала первой позиции, куда помещается наименьший в массиве, затем — второй и так до позиции $N-2$ (именно это значит запись `i<N-1`). Во втором цикле, использующем переменную `j`, ее значение изменяется от $i+1$ до $N-1$ (`j<N`), производятся перебор всех оставшихся элементов массива и их сравнение с находящимся на выбранном месте, в случае необходимости выполняется перестановка. Поскольку в языке программирования Си нет операции обмена значений переменных, для перестановки используется

буфер `buf` и «трехходовка» через него (в отличие от ассемблера x8086).

Контрольные вопросы

1. Как работает оператор `if (a%2)`? Какая особенность работы с логическими значениями языка программирования Си используется в данной программе?
2. Какие методы сортировки вам известны? Какие свои идеи вы бы использовали для этого?

Система управления базой данных о студентах

Рассмотрим более сложную, нежели приведенные ранее, программу на языке Си, а именно — систему управления базой данных (СУБД) о студентах. Как правило, в современной практике для создания программ, основными функциями которых являются хранение, поиск и выдача по запросу пользователя информации, — информационных систем используются специализированные средства.

Однако с учебными целями мы проиллюстрируем, как реализовать простую базу данных непосредственно на универсальном языке программирования, каковым является Си. Хранить информацию будем в формируемом программой на диске компьютера файле, например `Students.dat`.

Какую информацию нужно хранить о студенте? Очевидно, что нужно запоминать фамилию, имя и отчество, вероятно, год рождения (это может представлять интерес для военкомата). Логично для студента сохранять группу и успеваемость, которую можно выразить средним баллом, полученным в ходе сдачи предыдущей сессии. Добавим — чтобы внести положительных эмоций — еще размер стипендии и этим в нашем учебном примере ограничимся (читателю не возбраняется создать свою версию данной программы с расширенным перечнем характеристик студента, например, указанием пола, длины волос, любимой рок-группы и т. д.).

Соответствующая структура данных может выглядеть так:

```
/* Описание структуры данных "Студент" */
struct
{
  char Family[50]; // Фамилия
  char Imy[50];  // Имя
  char Otcestvo[50];  // Отчество
  char NGr[7]; // Номер группы
  int GodR; // Год рождения
  float SrBall; // Средний балл
```

```
  float Stip; // Размер стипендии
} Stud;
```

Для удобства дальнейшей обработки фамилия, имя и отчество хранятся в виде отдельных строк. Обозначение группы во многих вузах может включать как цифры, так и буквы, поэтому номер группы тоже представлен строкой `NGr`. Год рождения вполне может представляться целым числом. Средний балл и размер стипендии —числа с плавающей точкой.

Предположим, что после запуска программа будет выдавать на экран «меню» — перечень возможных действий с базой данных.

Реализуем следующие основные операции с базой:

1. Добавление записи о студенте в базу.
2. Поиск студента по фамилии.
3. Поиск по группе — вывод всех учащихся заданной группы.
4. Поиск по возрасту — с указанием диапазона годов рождения «от» и «до».
5. Поиск по успеваемости с заданием диапазона среднего балла.
6. Удаление данных.
7. Выход из СУБД.

Логично соответствующим образом структурировать программу — для каждого основного режима работы реализовать отдельную функцию. Главная программа при этом содержит цикл, в котором производятся запрос выбираемого режима, обработка и т. д., пока не будет выбран пункт «Выход».

Уместным в данной ситуации представляется использование так называемого *нисходящего проектирования программ* при разработке системы. Оно заключается в том, что сначала пишется функция `main` с меню, из которой вызываются функции `Dobavl()`, `Udal()` и т. д. Затем создаются «заглушки» этих функций — заголовки с пустыми телами, имеющими лишь `return`. Далее разрабатывается и отлаживается одна из подлежащих реализации функций, затем — другая (остальные остаются заглушками) и т. д. При этом на каждом этапе мы имеем работоспособную программу, которую можно запустить на выполнение, хотя и с ограниченной функциональностью!

ЗАМЕЧАНИЕ

Несмотря на широкое использование слова «функционал» для обозначения набора функций программы, в русском языке оно является некорректным.

> Функционал — понятие из математики, а в упомянутом контексте правильно говорить «функциональность».

Особое внимание следует уделить удалению записей. Для уникальной идентификации удаляемой записи (нельзя перепутать и отчислить из вуза другого студента!) нужно выбрать комбинацию параметров, не совпадающую ни у одной из возможных групп студентов. Вполне обоснованным представляется использовать фамилию, имя, отчество и год рождения — предполагаем, что даже если в нашем вузе встретятся полные тезки, они не будут рожденными в один год. Реализовать удаление записи можно по-разному. Например, воспользоваться следующей процедурой:

1. Создать копию базы данных с именем `Students.bak`, перенести в нее все записи, кроме удаляемой, например, с помощью цикла и проверки.
2. Удалить исходный файл базы данных.
3. Переименовать промежуточный файл, дав ему исходного — `Students.dat`.

Для удаления и переименования можно воспользоваться библиотечными функциями `remove` и `rename` из все того же `stdio.h`. Их аргументами являются имена файлов.

Альтернативным вариантом удаления является способ, при котором удаляемые записи фактически не удаляются, а лишь особым образом помечаются. Физическое удаление происходит при выполнении специальной операции — сжатия базы данных.

После всего сказанного можно приступать к реализации программы. Возможный вариант содержится в приложении А.

Контрольные вопросы

4. Что такое нисходящее проектирование программ?
5. Применимо ли к программам понятие «функционал»?
6. Какую информацию о студенте вы сочли бы уместным сохранять дополнительно к рассмотренной? Почему?

Особые возможности Си

В этом разделе мы рассмотрим отличительные особенности языка Си, в других языках почти не встречающиеся, что делает их некой экзотикой. Тем не менее их существование вызвано вполне серьезными причинами — потребностями системного программирования.

Одна из важнейших особенностей Си, приближающая его по уровню к языку ассемблера, — возможность использования так называемой

адресной арифметики. Напомним, что указатель — переменная, используемая для хранения значения адреса в памяти ЭВМ, где располагается некоторый другой объект (одиночная переменная, массив, функция и др.). В случае их использования открываются интересные возможности. Переменные, которые являются указателями, нужно объявлять специальным образом:

```
int *a,*b; /* a и b – указатели на целые числа в памяти ЭВМ */
int* a; /* аналогично, где ставить звездочку – дело вкуса */
char *d; /* d – указатель на символ или строку */
```

Переменную-указатель можно сравнить с предопределенным значением `NULL` (пустая ссылка). При выводе с помощью функций форматного вывода `printf` можно использовать для указателей специальный форматный символ `%p` (значение указателя — адрес — обычно представляется в виде шестнадцатеричного числа).

В Си операция & (амперсанд) возвращает адрес объекта:

```
y=&x;  /* y присваивается адрес x */
```

Обратная операция `*` (звездочка) используется для выборки содержимого по указанному адресу в памяти:

```
z=*y; /* z присвоено значение по адресу y */
*y=7; /* занесение семерки по адресу y */
(*z)++;  /* содержимое по адресу z увеличивается на единицу */
```

Указатели можно использовать в довольно сложных выражениях.

Но это не все. Поскольку адрес в памяти ЭВМ, фактически, представляет собой целое число, то к нему применимы арифметические операции — сложение, вычитание и др.:

```
*z++; /* адрес увеличен на размер типа данных в байтах, на который
ссылается   z */
a=*(p+11); /*взятие значения из ячейки, на 11 элементов отстоящей от
p */
```

Можно складывать указатели друг с другом, а также указатели и целые числа.

Как уже говорилось, параметры в языке программирования Си передаются в функции по значению, что не дает прямой возможности изменить переменные, объявленные в вызывающей функции, из вызываемой. В этом случае допустимо использование указателей при объявлении функции, например:

```
int utr3(int* a, int* b,int* d) /* утраиваем три параметра */
{
    *a=(*a)+(*a)+(*a);
```

```
    *b=(*b)+(*b)+(*b);
    *d=(*d)+(*d)+(*d);
    return 0;
}
```

При этом вызов данной функции должен осуществляться, как показано ниже, с использованием операции & для взятия адреса переменной:

```
int a=1,b=2,d=3;
utr3(&a,&b,&d);
```

Указатели в качестве аргументов обычно используются в функциях, которые должны возвращать более одного значения.

В языке Си массив передается с помощью ссылки на первый элемент массива в памяти. В связи с этим следующие записи эквивалентны:

```
int a[] и int *a (при записи аргументов функции);
char a[][], char* a[] и char **a;
a[5] и *(a+5);
```

ЗАМЕЧАНИЕ

> Следует помнить, что объявление вида int *a; фактически не приводит к выделению памяти под массив, в отличие от int a[5]. В связи с этим необходимо быть весьма осторожным, чтобы избежать трудноуловимых ошибок.

Весьма интересной особенностью Си является возможность создавать *переменные-указатели на функции* и строить затем гибкие программы. На самом деле это вполне естественно вытекает из фон-неймановской архитектуры ЭВМ: функция, являющаяся частью программы, находится в памяти, как и данные, соответственно, ее начало имеет адрес. Удивительно, что другие имеративные языки не часто поддерживают данную возможность. Следующий пример содержит иллюстрацию:

```
/* Использование функциональной переменной */
#include <stdio.h>

/* первая функция */
int fplus(int a,int b)
{
  return a+b;
}

/* вторая функция */
int fminus(int a,int b)
{
```

```
  return a-b;
}


int main()
{
  int a=10,b=10,t;
  int (*f)(int,int); /* объявление функциональной переменной */

  clrscr();
  printf("fplus адрес %p\n",fplus);
  printf("fminus адрес %p\n",fminus);

  f=fplus; /* сопоставляем f первую функцию */
  t=f(10,10);
  printf("Вызываем %p., получаем %d\n",f,t);

  f=fminus; /* сопоставляем f первую функцию */
  t=f(10,10);
  printf("Вызываем %p, получаем %d\n",f,t);

  return 0;
}
```

Говоря о функциях в Си, нельзя не отметить такую особенность языка, как функции с переменным числом аргументов. Мы уже пользовались ими — нетрудно заметить, что `printf` и `scanf` принимают произвольное число выводимых или вводимых значений. А может ли программист самостоятельно создать подобные функции? Следующий пример это иллюстрирует:

```
/* среднее арифметическое переменного числа аргументов */
double f(double n, ...)    /* заголовок с переменным числом
параметров */
{
    double *p = &n;        /* --установили на начало списка */
    double sum = 0, count = 0;

    while (*p)          /* пока аргумент не равен нулю */
    {
      sum+=(*p);          /* суммируем */
      p++;              /* берем следующий аргумент */
      count++;          /* подсчет количества аргументов */
```

```
    }

    return ((sum)?sum/count:0);    /* вычисляем среднее */
}
```

Есть одно обстоятельство, которое ограничивает применение таких функций. Передача аргумента не того типа, который задумывался, или не тем способом, который подразумевался при разработке, приведет к катастрофическим последствиям — компилятор Си ничего не проверяет.

Для доступа к списку параметров в примере использован указатель, значением которого будет адрес последнего явного параметра в списке. Чтобы перейти к адресу следующего параметра, надо изменить значение этого указателя. Это означает, что программист при разработке функции с переменным числом параметров должен отчетливо представлять себе типы аргументов, которые будет обрабатывать функция. Кроме того, способ передачи параметров должен быть одинаковым для всех параметров: либо все по значению, либо все по указателю.

Как в программе указать, каково фактическое число переданных параметров? Это можно сделать одним из двух способов:

- явно передать среди обязательных параметров количество аргументов;
- добавить в конец списка аргумент с уникальным значением, по которому будет определяться конец списка параметров;

И тот, и другой способ имеют право на жизнь — все определяется потребностями задачи и вкусами программиста. В примере использован второй способ: последним значением списка параметров считается ноль.

Еще одной интересной особенностью Си являются *смеси* и *битовые поля*, представляющие собой особые разновидности структур. Под полем понимается последовательность соседних битов внутри одного целого значения, что позволяет обращаться к выделенным частям одного машинного слова.

ЗАМЕЧАНИЕ

> Потребность взять определенные участки битов в слове может возникать при взаимодействии с внешними устройствами ЭВМ, а также при решении задач информационной безопасности.

Значение может иметь тип либо `int`, либо `unsigned int` и занимать от 1 до 16 бит. Поля размещаются в машинном слое в направлении от младших к старшим разрядам. В полях типа `int` крайний левый бит — знаковый. Если поле состоит ровно из одного бита, то это либо 0, либо –1. Поля не могут быть массивами и не имеют адресов, поэтому к ним нельзя применять операцию &. Пример:

```
/* битовое поле на Си */
struct prim
{
  int a:2; /* два бита с назанием a, значения от -3 до 1*/
  unsigned b:3; /* три бита с названием b, значения 0-8 */
  int c:1;
} i; /* (справа налево) 0011000=24 */
```

Поля — не единственный способ выделения наборов битов в слове. Могут использоваться также маскирование с помощью операции И — & (биты, которые нам нужны, — значение разряда маски 1, не нужны — 0) и побитовый сдвиг:

```
(c&24)>>3; /* наложили маску 0011000=24 и сдвинули на 3 вправо */
```

Смесь — переменная, которая в разное время может хранить объекты различного типа и размера. Появляется возможность работы в одной и той же области памяти с данными различного вида. Синтаксис следующий:

```
/* использование смеси в Си */
union k
{
  int ik;
  float fk;
  char ck;
} z;
```

Переменная `z` должна быть достаточно велика для того, чтобы хранить любой из трех типов. В один и тот же момент времени `z` может хранить значение только одной из переменных-компонентов. Следующий пример также иллюстрирует работу с битовыми полями и смесями.

```
/* Проба битовых полей и смесей */
#include <stdio.h>

void printbin(int* ch)
{
  int i;
  for (i=15;i>=0;i--)
    printf("%d", ((*ch)>>i) & 1);
  printf("\n");
}

struct bits
{
  unsigned a:3;
  unsigned b:3;
```

```
  unsigned c:4;
} sb;

union uni1
{
  float f;
  char c;
  int k;
} u;

int main()
{
  struct bits* uk=&sb;

  u.f=12.5;
  u.c='c';
  u.k=5;

  printf("Значение смеси %f\n",u.f);

  uk->a=5;
  uk->b=5;
  uk->c=0xB;

  printf("Битовое поле = %d\n",uk->a);
  printbin((int*) uk);

  return 0;
}
```

Как уже отмечалось, язык Си — язык системного программирования. Неудивительна поэтому легкость интеграции Си с ассемблером. Например, во многих версиях Си-систем вполне допустимы прямые ассемблерные вставки в программу по следующему принципу (значение команд ассемблера приводится в соответствующем разделе данной книги):

```
/* ассемблерные вставки в программу на Си */
#include <stdio.h>
#pragma inline
int main ()
{
    int a=10,b=20,c;
    print ("a= %d, b=%d\n",a,b);
```

```
    asm mov ax,10; /* строка на ассемблере — занесение 10 в регистр
AX */
    asm mul a; /* строка на ассемблере — команда умножения AX на a
*/
    c=_AX; /* прямой доступ к регистру AX процессора, с = 100 */
    print ("c= %d \n",c);
    /* целый фрагмент на ассемблере */
    asm {  mov ax,a
       mov bx,b
       xchg ax,bx
       mov a,ax
    mov b,bx
    }
    print ("a= %d, b=%d\n",a,b);
    return 0;
}
```

Как видно из примера (варианта реализации Си-системы), в программе на Си возможно прямое обращение к регистрам процессора, при этом их имена предваряются символом подчеркивания _ (для процессоров архитектуры Intel x86 это _AX, _BX, _DX и пр.). Для вставки одной строки на языке ассемблер достаточно написать в ее начале ключевое слово `asm` — и все дальнейшее до знака перевода строки будет восприниматься как строка ассемблера. Несколько строк можно вставить с помощью конструкции `asm {}`. Чтобы подобные вставки стали допустимыми, используется директива препроцессора `#pragma inline`. Изнутри ассемблерных вставок переменные, объявленные в программе на Си, остаются доступными просто по их именам (контроль типов — применимости ассемблерных команд к данным — ложится на программиста).

Контрольные вопросы и упражнения

1. Что означает запись `(*z)++`? Как изменится значение переменной `z`? Значение по адресу `z`?
2. В чем разница между записями `a[5]` и `*(a+5)` при обращении к элементу массива?
3. Каким образом вы применили бы в программах указатели на функции? Приведите пример.
4. В какой ситуации нужны функции с переменным числом параметов?
5. В каких ситуациях уместно использование битовых полей? Придумайте пример.

6. Возможен ли доступ из ассемблерной вставки к переменной Си?
7. Возможен ли доступ в программе на Си к регистру процессора?

Достоинства и недостатки языка Си

Язык программирования Си был создан довольно давно — в 1971 году. После появления он очень быстро завоевал популярность и потеснил такие языки, как Фортран, Алгол и PL/I. Впоследствии язык Си сыграл значительную роль в развитии языков программирования — стал непосредственным прародителем или оказал влияние на такие языки, как Objective C, C++, Java, C#, PHP, D и др.

Согласно рейтингу языков программирования TIOBE, на октябрь 2013 года чистый Си по популярности находится на первом (!) месте среди языков программирования, опередив Java (второе место), C++ (четвертое) и C# (шестое). Более того, шесть первых мест рейтинга заняты языками, родственными Си!

В чем причина такого успеха?

Несомненными достоинствами языка программирования Си являются:

- высокая скорость и компактность получаемых машинных программ — эффективность, из-за которой язык широко используется при написании встроенных приложений;
- низкоуровневые возможности — также востребованы при создании встроенных приложений и системных программ;
- широкая известность и наличие компиляторов для очень большого числа платформ.

В то же время следует еще раз отметить ряд особенностей языка программирования Си, которые, на взгляд автора (несмотря на то что обычно достоинства являются продолжениями недостатков и наоборот), вполне можно считать его недостатками и которые ограничивают сферу его успешного применения.

Конструкции Си изначально рассчитывались на профессионалов — системных программистов, и поэтому текст программы как минимум не вполне ясен начинающему программисту, а кроме этого он предоставляет весьма широкие возможности — можно сказать: язык «остер, как бритва» — можно порезаться! На Си допустим, к примеру, следующий фрагмент программы (сочинен студентами МАИ):

```
int X;
X = 1;
X+=X++ + ++X; /* Догадайся, чему будет равен X? */
```

Си (и, увы, во многом произошедшие от него языки, хотя в них пытались

эту проблему решать — например, в Java запрещено в явном виде использование указателей и отсутствует адресная арифметика) изобилует потенциально небезопасными конструкциями. Так, в нем можно применить присваивание вместо проверки на равенство внутри оператора `if`, например:

```
if (current->uid=0) retval=1;
```

и это не вызовет ошибки. Кстати, для борьбы именно с этой уязвимостью был предложен метод записи условий в программах на Си, названный нотацией Йоды. Этот персонаж саги «Звездные войны» необычным образом строил предложения, меняя привычный порядок слов. При записи `if` в программе на Си в виде условия Йоды сначала пишется константа, с которой производится сравнение, и лишь после знака — переменная:

```
if (0==current->uid) retval=1;
```

Естественно, присвоить числовой константе значение нельзя, и это вызовет ошибку. Некоторые программисты считают хорошим тоном использовать в программах на Си нотацию Йоды. Однако это не спасает ситуацию в целом. Не вызывает сомнений, что Си – изобилующий потенциальными опасностями и не вполне прозрачный для восприятия человеком язык. В приложении А содержится позаимствованный из книги Павловской [12] пример совершенно нечитаемой и тем не менее корректной для языка Си программы. Поэтому Си нужно с осторожностью использовать для начального обучения программированию. Весьма популярной альтернативой для этого у нас в стране является Паскаль.

Из-за слабого контроля действий программиста (таких особенностей, как ручное управление динамической памятью — отсутствия сборки мусора, не существующей автоматической инициализации переменных, адресная арифметика, отсутствия контроля выхода индекса за пределы массива, и пр.) при программировании на Си потенциально совершается больше ошибок, чем при программировании на Модуле-2, Обероне или Аде. Но несмотря на это он, как кажется автору, неоправданно популярен при создании критических приложений, например, в аэрокосмической промышленности или автоматизированных системах управления технологическими процессами.

Своеобразный промежуточный уровень языка Си, как считают многие исследователи, не позволяет эффективно применять его при написании приложений, требующих высокого уровня абстракции, например систем искусственного интеллекта. Для этих областей гораздо лучше подойдут Пролог или Лисп.

Язык ассемблера (автокод)

Как мы уже знаем, *ассемблеры* возникли в качестве первых отличных от машинных кодов средств программирования и относятся к второму поколению языков программирования. Какой же смысл сегодня, в XXI веке, изучать эту древность? Или язык ассемблера — современное средство? Разберем этот вопрос подробнее. Среди языков программирования ассемблер ближе всего к архитектуре ЭВМ, следовательно, он требует от программиста досконального знания деталей реализации данной ЭВМ, особенностей архитектуры. Это позволяет писать эффективные программы — короткие, быстрые, не требующие большого объема памяти. Где они востребованы и востребованы ли? Ответ на этот вопрос — востребованы, и эта ситуация будет наблюдаться, по всей видимости, еще много лет.

Как бы ни показалось удивительным некоторым читателям, но существуют применения ЭВМ, где доступные ресурсы до сих пор весьма ограниченны. Это, например, *встроенные* микропроцессоры и микроконтроллеры. А ведь именно они представляют собой наиболее многочисленный класс современных ЭВМ, если подсчитать все используемые суперкомпьютеры, мэйнфреймы, мини-ЭВМ (рабочие станции), персональные компьютеры и т. д. Микроконтроллеры повсюду — в телевизорах, телефонах, микроволновых печах, автомобилях, лифтах… Другой пример — системные программы (ядро операционной системы, драйверы). Можно привести в качестве примера также большие программные комплексы, работающие в системах массового обслуживания, в которых имеется узкое «бутылочное горлышко» — фрагмент программы, наиболее часто выполняющийся и критический с точки зрения производительности системы в целом. Такой фрагмент целесообразно писать на ассемблере, даже если остальная программная система использует Java или C#.

Весьма важный аспект — информационная безопасность. Вирусы и вредоносные программы, программы-шпионы используют бреши в уязвимости компьютерных систем со знанием тончайших деталей и особенностей архитектуры. Для того чтобы противостоять им, нужно обладать не меньшими познаниями и искусством программирования на уровне машины. Неудивительно, что умение программировать на ассемблере — своеобразный знак качества программиста, а разработчики подобных программ окружены ореолом таинственности как носители тайного знания. Но это не каждому по плечу!

Рассматривать ассемблер мы будем на примере программирования микропроцессоров Intel семейства x86. Данная архитектура не является ни наиболее простой для изучения, ни наиболее удобной и оптимальной с точки зрения разработки программ. Настоящие ценители

программирования на ассемблере помнят изящество архитектуры ассемблера ЭВМ разработки компании DEC семейства PDP-11 [13]. Причина выбора данного языка заключается в невероятном количестве выпущенных по всему миру компьютеров, основанных на архитектуре x86 — впрочем, далеко не единственной выпущенной компанией Intel, в послужном списке которой такие замечательные архитектуры, как, например, iAPX 432 и Itanium. Из чего следует, что, во-первых, не должно быть проблем с его нахождением и проверкой примеров, а во-вторых, читатель, не исключено, сможет извлечь из материала некоторую практическую пользу для себя. Ведь, как показывает многолетний опыт преподавания, учащиеся не любят изучать новые языки и склонны в практической работе на протяжении долгих лет использовать именно тот язык программирования, который изучили на первом курсе... Ниже представлены этапы исторического развития процессоров Intel.

1971 г. — первый в мире микропроцессор 4004, создан Intel

(внутри 16 разрядов, снаружи 8 разрядов) **1978**

|

80186 8088

|

80286 (IBM PC AT) **1982**

|

80386 (32-разрядный, защищенный режим) **1985**

|

80486 **1989**

|

Pentium — PentiumPro (увеличенный размер кэш-памяти) **1993**

|

Pentium MMX (multi medium extension) **1997**

|

Pentium II **1997**

|

Pentium III **1999**

|

Pentium IV **2000**

|

Core 2 Duo (многоядерный) **2006**

|

Pentium G **2011**

Другим важным обстоятельством является то, что огромный массив существующих программ был разработан именно для x86, так что, весьма вероятно, читателю придется разбираться именно с подобными программами.

ЗАМЕЧАНИЕ

> Данная глава ни в коем случае не является учебником по программированию на языке ассемблера — весьма нетривиальной технической дисциплине, имеющей массу нюансов. Она дает лишь весьма поверхностное общее впечатление, позволяющее в некоторой степени понять особенности программирования на ассемблере и способное стать отправной точкой для его последующего изучения по специализированной литературе, например [13].

Для дальнейшего изложения нам потребуются краткие дополнительные сведения об архитектуре ЭВМ — в частности, о представлении команд. Для представления команды в памяти ЭВМ необходимо закодировать двоичным кодом само действие и операнды — то, над чем производится действие.

Поле команды состоит из двух частей — операционной и адресной. В *операционной* части записывается *код операции* (КОП). Этот код определяет действие, которое команда должна выполнить над данными (обычно арифметическое или логическое). *Адресная* часть команды содержит адреса ячеек памяти — операндов, обрабатываемых в операции. Количество приводимых в команде адресов может быть различным. Основными вариантами архитектур ЭВМ являются: трехадресная; двухадресная; одноадресная.

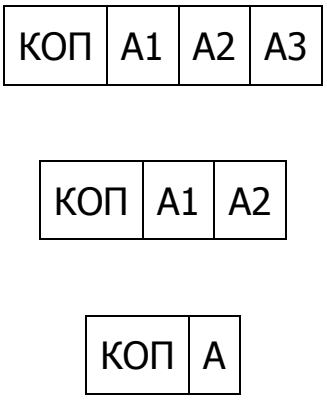


Рис. 17

Большинство операций, реализуемых арифметико-логическими устройствами современных процессоров, — двухместные (бинарные), это означает, что в результате обработки двух операндов получается один результат. Рассмотрим пример вычисления выражения `z=x+y`.

В трехадресной архитектуре — самой полной — часть двоичного кода команды отводится под представление операции. Например, `000110` — сложение (в архитектуре PDP-11). Кроме этого, в ней присутствуют три адреса: адрес первого операнда `x` — A1, адрес второго `y` — A2, и адрес, в который нужно поместить результат, `z` — A3. Трехадресная архитектура является наиболее гибкой, но и наиболее затратной с точки зрения объема памяти, необходимого для представления каждой команды. Если адресное пространство соответствует микропроцессорам 8086, под каждый из адресов потребуется 20 бит. Соответственно, каждая команда должна быть длиннее 60 бит, что не очень хорошо с точки зрения упаковки больших программ.

Большинство современных ЭВМ, в том числе микропроцессоры семейств x86, — двухадресные. В них в команде присутствуют два адреса, один из которых используется как для взятия операнда, так и для помещения по этому же адресу полученного результата. Примером может служить `x=x+y`. Данный подход позволяет без существенной потери гибкости сэкономить заметное количество памяти.

На одноадресной архитектуре были построены первые микропроцессоры. В ней присутствует *аккумулятор* — специальный выделенный регистр в процессоре, в который по умолчанию попадают результаты всех операций. Из него же по умолчанию берется один из операндов. Соответственно, в команде достаточно указать адрес всего одного операнда.

Самый экстремальный случай — безадресная архитектура. В ней команда содержит лишь КОП. Операнды по умолчанию берутся из заранее известного места — чаще всего из стека (два числа на вершине стека) (см. раздел о структурах данных).

ЗАМЕЧАНИЕ

> Не все команды используют адресные поля полностью. Есть действия, по своей природе имеющие один адрес (например, смена знака числа). Есть команды и вовсе не требующие указания адресов операндов (останов, пустая операция, системный сброс и т. д.). В ЭВМ трехадресной и двухадресной архитектур некоторые команды будут одноадресными и безадресными.

При обращении к ячейке памяти в программе на ассемблере можно указать конкретный адрес в виде числа (в большинстве случаев шестнадцатеричного или восьмеричного) — так называемый *абсолютный адрес* (рис. 18).

Рис. 18

Существуют и другие способы адресации (обращения к ячейке). При косвенной адресации в команде указывается регистр или ячейка памяти, где располагается *адрес* операнда, а не сам операнд (рис. 19).

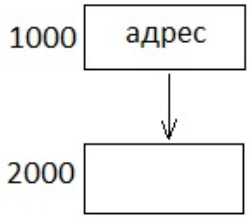


Рис. 19

ЗАМЕЧАНИЕ

> Именно эта низкоуровневая особенность архитектуры ЭВМ повлияла на концепцию указателей в языках Си-семейства.

В некоторых архитектурах реализованы еще более сложные способы адресации. При *автоинкрементной* и *автодекрементной* адресации после обработки данных при косвенной адресации автоматически происходит увеличение или уменьшение адреса, что позволяет, например, эффективно обрабатывать массивы (переходить к следующей ячейке). Эта низкоуровневая особенность повлияла на такую конструкцию языков программирования Си-семейства, как `--` и `++`.

Индексная, или *базовая*, адресация подразумевает, что, как в случае косвенной адресации, в команде указываются ячейка, в которой находится некоторый адрес в памяти, называемый базой, и *смещение* — величина, которую надо предварительно прибавить к базе, чтобы получить адрес, где, собственно, и находятся данные (рис. 20). Этот способ адресации также применяется для обработки массивов.

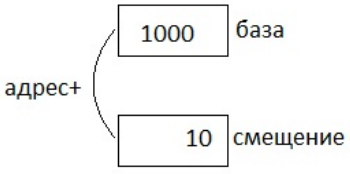


Рис. 20

При *непосредственной* адресации в команде вместо адреса указывается само число, например слагаемое.

ЗАМЕЧАНИЕ

В некоторых архитектурах могут реализовываться и другие более экзотические способы адресации операндов.

Обычно процессор помимо УУ, АЛУ и СК имеет встроенные так называемые *регистры*. Они представляют собой *сверхоперативную память* — команды могут обрабатывать данные без необходимости обращения к ОЗУ ЭВМ, что значительно ускоряет вычисления. Регистры могут указываться в командах вместо адресов данных в памяти.

В архитектуре x86 имеется несколько регистров, которые можно использовать произвольным образом (почти), или *регистров общего назначения*. Они имеют обозначения `AX`, `BX`, `CX`, `DX`, `SI` и `DI`. Несмотря на универсальность, в программах регистры зачастую используются в определенных целях и поэтому имеют дополнительные имена: `AX` — аккумулятор, `BX` — база, `CX` — счетчик, `DX` — данные, `SI` — индекс источника, `DI` — индекс результата.

Определив подходы к представлению команд в памяти ЭВМ, поговорим немного о представлении данных. Компьютер имеет базовый параметр, называемый *разрядностью*, определяющий размер *машинного слова* — порции данных, обрабатываемой процессором при выполнении одной команды. Сегодня разрядность у массовых компьютеров — 16, 32, 64 бит — два, четыре или восемь байт соответственно. Тем не менее обычно минимально адресуемой единицей памяти является байт, и у 16-разрядной ЭВМ, например, адреса соседних ячеек (слов) различаются на 2. При представлении дробных чисел в современных ЭВМ обычно производится нормализация и хранятся мантисса и порядок числа.

Важным является представление знака для чисел. Не вдаваясь в подробности, отметим, что обычно допустима обработка как беззнаковых чисел, так и чисел со знаком. При этом, если число рассматривается как число со знаком, обычно крайний старший разряд слова принимается носителем знака по следующей схеме: ноль — число положительное, единица — отрицательное.

Символы и строки хранятся в памяти ЭВМ с использованием той или иной системы кодирования. Наиболее распространены вариации кода ASCII и Unicod. В кодировке ASCII (American Standard Code for Information Interchange — увы, кодировка американская, и с русскими буквами мы имеем массу проблем...) для кодирования одного символа используется 1 байт (фактически, изначально 7 бит, именно это позволило при создании основанных на ней русскоязычных кодировок добавить кириллицу, но для простоты можно считать, что 8 бит).

Программа на ассемблере может включать несколько секций (некоторые

не являются обязательными), или разделов (например, в ассемблерах MASM и Flat Assembler):

- заголовок;
- описание интерфейса и подключаемых библиотек;
- описание данных (констант и переменных);
- собственно команды.

Правила записи заголовка и интерфейса программы могут различаться в разных версиях программы-переводчика (также называемой ассемблером). Автор рекомендует для программ на ассемблере x86 использовать свободно распространяемый Flat Assembler, снабженный довольно солидным набором библиотек, включающих, в частности, неплохую поддержку компьютерной графики.

Основная часть программы на языке ассемблера состоит из выражений. Все выражения могут быть записаны в следующем условном виде:

[*Метка:*] [*Оператор*] [*Операнды*] [*; Комментарий*]

Выражения могут быть директивами ассемблера и мнемоническими обозначениями команд. Различие между ними следующее. Мнемоника команды ассемблером превращается в машинный код, кототрый затем записывается в память программы и в ходе ее исполнения обрабатывается процессором. Директивы называются также псевдокомандами — это некие предписания ассемблеру, предназначенные для программы-переводчика, а не процессора. Набор команд специфичен для каждого процессора (архитектуры). Директивы определяются реализацией программы ассемблирования.

Сначала рассмотрим запись программы на языке ассемблера с помощью команд:

```
A1:   mov      ax,2 ; занесение в регистр AX процессора числа 2
   mov      bx,2 ; занесение в регистр BX процессора числа 2
     add bx,ax; сложение содержимого BX и AX, результат → в BX
```

Надеемся, приведенный фрагмент не вызвал у читателя затруднений . В данном примере данные заносились и обрабатывались в регистрах общего назначения процессора. Если данные хранятся в памяти, ячейке обычно дается мнемоническое имя с помощью соответствующих *директив* ассемблера. Рассмотрим следующий пример (ассемблер MASM):

```
; в разделе описания данных
S1 dw 2 ; занесение в ячейку числа 2 и ее обозначение как S1
S2 dw 3 ; занесение в ячейку числа 3 и ее обозначение как S2
S3 dw ? ; резервирование ячейки памяти с именем S3
P  db ? ; резервирование одного байта
```

```
MAS   dw 1,2,-4,11,25; начиная с адреса MAS занесены пять чисел
STR   db 'Привет'; начиная с адреса STR идут байты строки символов

; в разделе описания команд они обычно отделены друг от друга
sub   S2,S1; вычитание содержимого ячейки S1 из S2
```

ЗАМЕЧАНИЕ

> В процессорах x86 при записи двухадресных команд адрес, куда помещается результат операции, указывается до запятой. В других ассемблерах может быть иначе.

Команды можно разделить:

- на команды пересылки;
- команды преобразования данных — арифметические и логические операции, сдвиги;
- команды передачи управления — переходов безусловных и условных, вызова и возврата из подпрограммы;
- служебные и специальные команды.

Рассмотрим основные команды пересылки данных. Наиболее часто используемая — mov. Эта двухадресная команда создает копию данных, находящихся по указанному адресу, в другом месте. Примеры:

```
mov   ax,bx; копирование содержимого регистра BX в AX
mov   cx,99; занесение 99 (непосредственная адресация) в CX
mov   S1,S2; копирование значения ячейки S2 в ячейку S1
```

При программировании на ассемблере весьма активно используется системный стек (см. описание стека как структуры данных в соответствующем разделе). Для занесения информации на вершину стека используется команда push, для снятия — pop:

```
push  99; занесение числа 99 в стек
push dx; сохранение в стеке содержимого регистра DX
pop   dx; снятие с вершины стека значения в регистр DX
```

В системе команд процессоров x86 есть еще одна весьма полезная пара команд — pusha и popa. Они не имеют аргументов и предназначены для занесения текущих значений всех регистров процессора в стек и восстановления оттуда соответственно. Используются, например, при вызове подпрограмм. Можно провести параллель между использованием регистров процессора в этой ситуации и локальных переменных — в языках высокого уровня.

ЗАМЕЧАНИЕ

> В настоящей книге не рассматриваются все команды ассемблера x86 — и тем более все возможные команды языков ассемблера различных ЭВМ.

В архитектуру x86 введена еще одна весьма полезная команда, позволяющая двум ячейкам памяти произвести обмен содержимым, — `xchg`. В других системах команд для реализации подобного действия требуется несколько команд `MOV` и промежуточный буфер:

```
xchg  ax,bx; обмен содержимым AX ↔ BX
```

Перейдем к командам преобразования данных. Это в первую очередь арифметические операции, а также логические операции и команды сдвигов. Сложение и вычитание весьма просты:

```
add   ax,2; прибавление числа 2 к регистру AX
sub   N,-1; вычитание минус единицы из ячейки с именем N
```

Существуют две специальные команды для уменьшения (декремента) и увеличения на единицу (инкремента):

```
inc   cx; cx=cx+1
dec   N; уменьшение на единицу содержимого ячейки с именем N
```

Данные команды занимают меньше места в памяти и выполняются быстрее, нежели `sub` и `add` с аргументом `1`. В языке программирования Си очевидна параллель — `--` и `++`.

Несколько более интересны команды умножения и деления. Во-первых, система команд x86 предусматривает по две команды для умножения и деления чисел — со знаком и беззнаковых. Умножение беззнаковых чисел производится командой `mul`, чисел со знаком — `imul`. Во-вторых, команда одноадресная — по умолчанию один из аргументов берется из регистра `AX` и туда же заносится результат. Если результат не умещается в `AX` полностью, старшие биты произведения помещаются в `DX`. Рассмотрим пример:

```
mov   bx,5; первый сомножитель 5 заносим в BX
mov   ax,25 ; второй сомножитель 25 → в AX
mul   bx ; умножаем 5 * 25, результат → в AX
```

Команда деления беззнаковых чисел — `div`, чисел со знаком — `idiv`. Поскольку аргументы воспринимаются как целые числа, а для них полноценное математическое деление невозможно, операция производится как целочисленное деление с занесением неполного частного в младшие биты регистра `AX`, которые обозначаются `AH`, а остатка от деления — в его старшие биты (`AL`). При делении чисел размером 1 байт частное заносится в `AL`, а остаток — в `DL`.

ЗАМЕЧАНИЕ

Это обстоятельство может быть использовано при написании варианта программы подсчета суммы нечетных элементов числовой последовательности.

Логические операции производятся побитово (аналоги в Си — `!`, `|`, `&`):

```
and   ax,64; логическое И с маской 64 (1000000) над AX
or dx,ax; логическое ИЛИ
xor   cx,cx; XOR исключающее ИЛИ – кстати, дает обнуление
```

Имеется набор команд сдвига. Первый аргумент при этом рассматривается как набор битов, которые будут сдвигаться влево или вправо. Второй же аргумент — целое без знака, показывающее, насколько надо сдвинуть первый операнд.

Команда `shl` — *логический* сдвиг влево, ее действие иллюстрирует рис. 21.

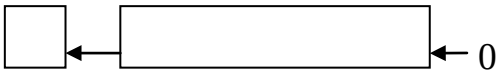


Рис. 21

Обратите вниманиена то, что крайний левый бит не теряется, а попадает в специальный разряд — флаг — процессора, называемый `CF`. В младший разряд заносится ноль. `shr` — логический сдвиг вправо, работает аналогично.

ЗАМЕЧАНИЕ

Логический сдвиг влево и вправо позволяет выполнить быстрое умножение и деление беззнакового целого числа на делители, представляющие собой степени числа 2, не требующее накладных расходов команды div.

Для подобного умножения и деления чисел со знаком можно применять специально предусмотренные в системе команд ассемблера x86 *арифметические* сдвиги: `sal` — арифметический сдвиг влево, `sar` — арифметический сдвиг вправо (рис. 22).

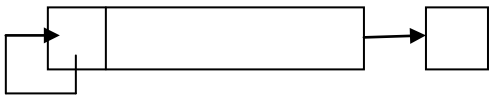


Рис. 22

Особенность *циклических* сдвигов заключается в том, что разряды двигаются по кольцу. Причем существует две разновидности сдвигов в зависимости от использования флага процессора `CF`: `rol` — циклический сдвиг влево, `ror` — циклический сдвиг вправо (рис. 23).

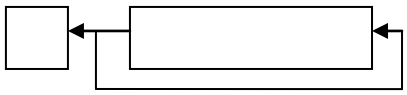


Рис. 23

В команде `rol`, как показано на рисунке, бывший самым левым бит заносится в качестве правого (младшего) и одновременно копируется в `CF`. Команды сдвига через перенос `rcl` — циклический сдвиг влево и `rcr` — циклический сдвиг вправо работают следующим образом (рис. 24).

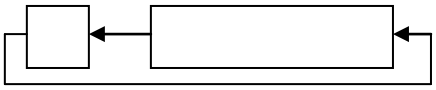


Рис. 24

Поскольку речь зашла о *флагах процессора*, стоит рассмотреть их подробнее. Обычно они используются при выполнении команд условного перехода. В процессорах x86 используются следующие основные флаги:

- `ZF` — флаг нуля, устанавливается в единицу, если результат равен 0;
- `SF` — флаг положительного/отрицательного значения, устанавливается, если результат меньше 0;
- `CF` — бит переноса из старшего разряда результатов;
- `OF` — флаг переполнения, устанавливается, если результат не поместился в разрядную сетку ЭВМ;
- `PF` — флаг четности, устанавливается, если в 8 младших байтах результата содержится четное количество двоичных единиц.

Значения флагов определяются результатом предыдущей выполненной команды преобразования данных. Кроме этого, сформировать флаги можно с помощью специальной команды сравнения операндов `cmp`:

```
cmp   ax,bx; сравнение AX и BX
```

В зависимости от флагов можно выполнить *условный переход к метке*. В языке ассемблера x86 существует богатый набор команд переходов, часть из которых приведены далее, причем допустимы две формы мнемонической записи (указаны через дробь), что, вообще говоря, — не лучший подход с точки зрения соблюдения стиля программирования:

- `JE/JZ` — переход по нулю (`JE` — переход по равенству после `cmp`);
- `JNE/JNZ` — если не ноль (не равны аргументы предыдущей `cmp`);
- `JG/JNLE` — переход, если больше для знаковых;
- `JGE/JNL` — переход, если больше или равно для знаковых;
- `JAE/JNBE` — переход, если больше или равно для беззнаковых;

- JC/JNAE — переход, если бит CF установлен в единицу.

Переходы необходимы для изменения линейного порядка выполнения действий в программе. Для этого место в программе, куда передается управление (с которого продолжается выполнение программы), должно быть помечено. Рассмотрим пример нахождения максимума среди чисел без знака:

```
; максимум среди чисел X и Y → в ячейку Z
   mov   ax,X; берем в AX значение числа из ячейки с именем X
   cmp   ax,Y; сравнение – формирование флагов
   jae   M; переход к метке M если >=
   mov   ax,Y;
M: mov   z,ax
```

Помимо команд условного перехода в программах на языке ассемблера используются безусловный переход и вызов подпрограммы с возвратом. Команда безусловного перехода записывается так:

```
jmp   M1; продолжить с метки M1
```

Если требуется, например, в целях информационной безопасности максимально затруднить понимание логики программы, например, при ее дизассемблировании злоумышленниками, можно насытить ее беспорядочными переходами с помощью jmp.

При исполнении команды вызова подпрограммы call происходит следующее. В системный стек заносится текущее значение регистра — счетчика команд процессора (IP — Instruction Pointer) и управление передается по указанному в команде адресу (метке). По окончании подпрограммы в ней должна встретиться команда ret, при выполнении которой производится обратное действие — извлекается значение с вершины стека и управление передается в вызвавшую программу:

```
; вызов подпрограммы pecat
call  pecat
; сюда возвращается управление после исполнения подпрограммы
; …

; где-то в другом месте исходного текста
; подпрограмма печати – комментарий необходим
pecat:; … действия
   ; действия
   ret ; возврат из подпрограммы
```

Отметим, что требование хорошего стиля программирования подразумевает, как и при написании программ на языках высокого уровня, что следует снабжать подпрограмму комментарием, раскрывающим

выполняемые ею действия.

Система команд процессоров x86 довольно богата — в ней имеются специальные команды для организации циклов. Рассмотрим пример с использованием команды `loop`. Пусть некий фрагмент программы нужно повторить *N* раз. Реализовать это можно, например, следующим образом:

```
   mov   cx,N; регистр CX — счетчик, аналог переменной цикла
L: ;начало тела цикла
   ;… действия цикла
   ;…
   ;…
   dec   cx; CX=CX-1
   cmp   cx,0; CX=0?
   jne   L; CX еще не ноль — возврат к метке L
```

Как видите, в конце цикла используется стандартная тройка команд. Ее можно заменить одной — `loop`! Переписать программу в этом случае можно следующим образом:

```
   mov   cx,N; регистр CX — счетчик, аналог переменной цикла
L: ;начало тела цикла
   ;… действия цикла
   ;…
   ;…
   loop  L; цикл, использует регистр CX по умолчанию
```

Видно, что современные ассемблеры и архитектуры ЭВМ предоставляют довольно гибкие возможности — главное, умело ими пользоваться.

Контрольные вопросы

1. К какому поколению языков программирования относится ассемблер?
2. В чем разница между ассемблером и автокодом? Откуда произошло название «автокод»?
3. Как вы оцениваете перспективы ассемблера в XXI веке?
4. Каковы области применения ассемблера в современных сложных программных проектах?
5. Как работает команда `xchg`? Удобно ли использовать ее на ассемблере и как она могла бы повлиять на языки программирования высокого уровня?
6. В чем заключается разница между арифметическим и логическим сдвигами?
7. Что такое машинное слово?

8. В чем заключается механизм работы индексной адресации?
9. Какие разновидности машинных команд существуют?
10. Как работают команды условного перехода? Зачем они нужны?
11. Для чего применяется команда `loop`?

Сумма нечетных на ассемблере

Приведенной ранее информации достаточно для того, чтобы приступить к написанию программы суммирования нечетных элементов числовой последовательности, проходящей красной нитью через настоящую книгу, на языке ассемблера x86. Будем пользоваться бесплатно распространяемой версией Flat Assembler. Рассмотрим возможный вариант решения:

```
; программа подсчета суммы нечетных в массиве chisla
include 'win32ax.inc'

.data
chisla  dw 1,2,3,4,5,0,0,0,0,0
N     dw 10

.code
start:  ; начальная метка программы, в конце должнj быть .end start
        mov esi,chisla
        mov cx,[N] ; количество элементов в таблице
        mov dx,0 ; в dx будем накапливать сумму нечетных, сначала
обнуляем
l1:     mov ax,[esi]; очередное число заносим в AX
        test ax,1 ; проверяем на нечетность - младший бит на 1
        jz  l2 ; если младший бит двоичного числа равен нулю -
четное!
        add dx,ax ; прибавляем очередной нечетный элемент
l2:     add esi,2
        loop l1 ; цикл по счетчику cx, пока не дойдет до нуля
        ; вызываем outint для вывода целого значения — результата
        mov ax,dx
        call outint
        invoke  ExitProcess,0
```

Данная программа предполагает, что у нас есть возможность вывода на экран результата — мы формируем его в регистре процессора AX. Первая строка — комментарий, который, как и положено, описывает суть нашей программы. Строка с `include` нужна, чтобы мы смогли использовать в программе системный вызов `invoke ExitProcess,0` для корректного

завершения. Директива `.data` FlatAssembler указывает на начало сегмента описания данных в программе, директива `.code` — начало исполняемого блока команд.

В секции описания данных у находится директива `chisla dw 1,2,3,4,5,0,0,0,0,0`, с помощью которой мы заносим в память набор обрабатываемых чисел. В следующей строке — описание ячейки `N`, содержащей размер массива, в нашем случае 10.

Правила FlatAssembler подразумевают, что начало исполняемой программы снабжается меткой `start` (последней строчкой программы должна в этом случае быть директива `.end start`). Итак, вначале в регистр `ESI` заносится адрес начала обрабатываемого массива в памяти. Регистр `ESI` — так называемая расширенная (32-разрядная) версия регистра процессора `SI`, ее использование в данном случае обусловлено использованием современного компьютера. В следующей строке заносим в регистр `CX`, используемый как счетчик, количество чисел. Квадратные скобки необходимы, чтобы указать на необходимость взять в качестве аргумента число по адресу, а не сам адрес ячейки, — косвенную адресацию. Затем мы обнуляем регистр `DX` — в нем будем накапливать сумму нечетных чисел.

Со следующей строки начинается тело цикла, поэтому она имеет метку `l1`. В цикле мы первым делом извлекаем из памяти по адресу, на который указывает `ESI`, очередное число (косвенную адресацию дают квадратные скобки) и заносим его в регистр `AX`. Проверка нечетности числа осуществляется путем вполне уместного при программировании на ассемблере использования низкоуровневых возможностей — команды битового тестирования по маске `test ax,1`. Вспомним, что данные хранятся в памяти ЭВМ в двоичном виде в позиционном коде. Младший разряд числа отвечает при этом за степень 2^0 и равен нулю в случае, если число делится без остатка на 2, и единице — в противном случае. Соответственно, для проверки на нечетность нет необходимости выполнять операцию деления на 2. Достаточно проверить содержимое младшего бита (маска 1). Следующая команда условного перехода `jz` обеспечивает переход к метке `l2` в случае, если результат нулевой (число четное), с помощью чего производится обход команд программы, производящих суммирование.

Следующая операция, которая выполняется либо после суммирования естественным путем, либо после перехода по команде `jz l2`, — переход к следующему числу в памяти. Для этого мы увеличиваем на 2 значение регистра `ESI`. Увеличение на 2 связано с тем, что мы храним обрабатываемые числа в машинных словах, а минимальная единица

адресации процессора x86 — 1 байт. В случае если счетчик не дошел до нуля, с помощью специальной команды цикла `loop` выполняется переход к повтору цикла с метки `l1`.

После выполнения программы сумма нечетных чисел будет накоплена в регистре `DX`. Нам было бы интересно получить это значение на экране. К сожалению, в ассемблере нет удобных средств, подобных `writeln` в Паскале или `cout <<` в C++, позволяющих печатать значения производного типа. Flat Assembler предоставляет системный вызов `MessageBox`, отображающий на экране окно сообщения, но в качестве аргументов он требует строку. Поэтому нам необходимо преобразовать число — сумму нечетных — в строку, иными словами, в его *запись с помощью цифр*.

Здесь, в отличие от языков программирования высокого уровня, видна особенность программирования на низком уровне — многие вещи приходится реализовывать самостоятельно. Итак, напишем подпрограмму (назовем ее `outint`), которая позволит выводить на экран целое число, например, находящееся в регистре `AX`, используемом для передачи аргумента.

Будем реализовывать подпрограмму сразу как универсальную, что является признаком хорошего стиля в программировании. Наша подпрограмма сможет выводить число как в десятичном, так и в двоичном, и в другом виде — с использованием позиционной системы счисления с произвольным основанием. Вспомним правило записи чисел в позиционной системе:

$$N = \sum q^i g_i,\ i = 0 \ldots M,$$

где M — количество разрядов числа; q^i — основание системы счисления в степени i, g_i — значение цифры в позиции i, а i — номер позиции (разряда), начиная с нуля и справа налево. Будем делить наше число на основание системы счисления и накапливать остатки от деления, которые и будут значениями цифр. Далее приводится возможный вариант программы. В секции описания данных появится

```
schisl  dw 10 ; основание системы счисления
Chislo  db '      ' ; строка где будет записываться цифрами число
```

Здесь мы резервируем место (пустую строку из нескольких пробелов) для формирования записи числа, начиная с адреса `Chislo` в памяти, а кроме этого заносим по умолчанию в качестве основания системы счисления в ячейку `schisl 10`.

Сама подпрограмма может выглядеть следующим образом:

```
outint: ; вывод содержимого ax в окне сообщений
```

```
        mov edi,Chislo ; заносим в регистр edi адрес строки для
записи числа цифрами
        add edi,5 ; записывать цифры будем  начиная с правой,
поэтому переходим к концу строки
l:      xor dx,dx ; обнуление dx
        div [schisl] ; после выполнения div в dl автоматически
заносится остаток от деления
        add dl,'0' ; для получения кода цифры прибавляем к числу код
нуля
        mov [edi],dl ; пересылаем в память, где хранится запись
числа цифрами, очередную
        dec edi ;передвигаемся к следующему байту — идем от конца
        or ax,ax ; проверяем ax, где остается частное, на равенство
нулю, если нет -
        jnz l ; продолжаем делить


        ; вызов WinAPI MessageBox для вывода строки
        invoke  MessageBox,HWND_DESKTOP,Chislo,"Сумма
нечетных",MB_OK
        ret
```

Итак, сначала мы заносим в регистр EDI (расширенная версия DI, используется из-за 32-разрядной архитектуры) адрес начала будущей записи числа цифрами в памяти с помощью команды `mov edi,Chislo`. Однако вспомним, что при формировании позиционной записи мы пишем остатки справа налево — следовательно, нужно перейти к концу строки. Поскольку мы отвели пять позиций под запись числа, добавляем 5 к адресу командой `add edi,5` (адресная арифметика в ассемблере). Следующая строка — начало цикла, поэтому она помечена `l`. В ней первая команда иллюстрирует программистский трюк — для обнуления регистра DX используется `xor` значения самого с собой — команда выполняется быстрее, нежели `mov dx,0`. Далее с помощью `div [schisl]` производится деление солержимого AX на основание системы счисления, извлекаемое из соответствующей ячейки памяти с помощью косвенной адресации. Особенностью команды `div` является то, что в данном случае остаток от деления автоматически попадает в младшие биты регистра DX, называемые DL. А нам именно этот остаток и нужен! Пересылаем его по адресу, на который указывает EDI (косвенная адресация), однако предварительно превращаем остаток, который сам по себе является числом от 0 до 9 при основании системы счисления 10, в соответствующий код символа. Сделать это несложно , поскольку в используемой кодировке коды цифр идут подряд. Достаточно с помощью команды `add dl,'0'` прибавить код нуля, чтобы получить необходимый символ — цифру. С помощью команды `dec edi` переходим к предыдущей позиции в памяти.

Как только число будет исчерпано, в качестве частного от деления, попадающего по умолчанию в AX, мы получим ноль, что должно стать сигналом о завершении цикла. Проверку значения AX на ноль можно осуществить с помощью часто используемой в программах на ассемблере логической операции ИЛИ аргумента самого с собой `or ax,ax` (заметим, что в ряде ассемблеров существует специальная команда тестирования содержимого ячейки и установки флагов, например TST). Продолжаем цикл, если это необходимо, с помощью команды условного перехода `jnz l`.

После завершения цикла запись числа будет сформирована в виде строки по адресу `Chislo`. С помощью системного вызова `invoke MessageBox,HWND_DESKTOP,Chislo,"Сумма нечетных",MB_OK` на экране демонстрируется окно сообщений с соответствующим содержимым.

Последняя строчка подпрограммы — команда `ret` возврата в вызывающую программу.

Контрольные вопросы

1. Какой прием используется в рассмотренной программе для преобразования выводимого на экран числа в строку?
2. Почему команда `shr` применима для проверки нечетности числа?
3. Где в приведенной программе используется косвенная адресация? Поясните механизм косвенной адресации.
4. Можно ли использовать команду `xor` для обнуления ячейки памяти? Почему?
5. Зачем в программе используется команда ИЛИ регистра самого с собой?

Макросы в ассемблере

Поскольку, как было продемонстрировано, при программировании на языке ассемблера многие вещи приходится реализовывать вручную, весьма активно применяется «расширение» языка на основе аппарата *макросов*. Говоря коротко, макросы позволяют заменять одни фрагменты текста в программе другими (называются *макрорасширениями*) — этот процесс называется *макроподстановкой* (использование макросов аналогично применению директивы `#define` в языке программирования Си). При этом возможно использование макросов с параметрами, как показано далее:

`<`*имя*`> MACRO <`*формальные параметры*`>`

```
<тело>
ENDM
```

При выполнении макроподстановки в тексте программы <*имя*> будет заменено на <*тело*> с подстановкой в нем указываемых фактических параметров вместо формальных. Параметры макроса записываются через запятую или пробел. Размещать макроопределения в тексте программыследует до первой макрокоманды, использующей макрос. Макросредства при программировании на языках ассемблера настолько популярны, что сам язык иногда называют *макроассемблером*.

ЗАМЕЧАНИЕ

> Определяя набор макросов, вы, по сути, создаете некий новый язык. Если макросов много и они достаточно мощные, иногда можно не узнать исходный язык.

Рассмотрим некоторые примеры макросов (всю мощь данного механизма настоящая глава не передает, она представляет собой, как и другие разделы книги, лишь введение в предметную область. Заинтересовавшегося читателя отсылаем к литературе, например [13].).

```
; определение макроса с параметрами X и Y для суммирования
SUM   MACRO X,Y; X=X+Y
   MOV   AX,Y
   ADD   X,AX
   ENDM


; пример использования макроподстановки в программе
   SUM   A,B ; A и B — фактические параметры
```

Макросы могут содержать не только команды, но и директивы ассемблера, например:

```
MAS   MACRO X,N ; описание массива X из N байт
X  DB N DUP (?)
   ENDM
```

На основе вышеприведенного макроса мы можем определить макрос MAS2 для описания сразу двух массивов этого же размера:

```
MAS2  MACRO X1,X2,K
   MAS    X1,<K>
   MAS    X2,<K>
ENDM


;  обращение к макросу
   MAS2   A,B,20
```

Во втором макросе угловые скобки для параметра используются для однократного использования при макрорасширении. Интересно, что в современных макроассемблерах разрешены определение макросов через уже существующие макросы и даже рекурсивные макроопределения.

В завершение данного раздела отметим, что помимо ассемблера, выполняющего функции трансляции исходного текста программы в машинный код, при создании и эксплуатации низкоуровневых программ широко применяются и другие системные программы. Среди них *дизассемблер* — программа, позволяющая по машинному коду строить текст на ассемблере, что становится возможным из-за наличия прямого соответствия между мнемоникой ассемблера и машинной командой (изоморфизма). Важнейшим инструментом является *отладчик* — программа, позволяющая «на лету» переводить строку ассемблера в код и наоборот, иными словами — просматривать и менять содержимое программной памяти, используя не двоичные коды, а мнемоники ассемблера. Вдобавок к этому она выполняет множество полезных функций по запуску, останову, дампу памяти и пр. *Кросс-ассемблер* — ассемблер, применяемый на инструментальной ЭВМ, — обычно универсального назначения, но строящий машинный код не для нее, а для другого процессора, часто встраиваемого. *Эмуляторы* — программы, позволяющие запускать программы низкого уровня на языках ассемблера и в машинных кодах одного процессора на другой ЭВМ. Часто применяются вместе с кроссассемблерами.

Контрольные вопросы

1. Каковы возможные варианты применения макросов в ваших программах?
2. Что такое кросс-система? Эмулятор? Зачем они используются?
3. В чем сходства и различия дизассемблера и отладчика?
4. Допускается ли вложение одних макроопределений в другие?
5. Можно ли использовать директивы ассемблера в макроопределениях?
6. Зачем нужны параметры в макросах? В каких случаях параметры макросов записываются в угловых скобках?
7. В чем разница между макрорасширением и макроопределением?

Введение в объектно-ориентированное программирование на примере C++

В настоящее время объектно-ориентированный подход занимает в программировании особое место. Наиболее современные и активно используемые в индустрии информационных технологий языки — C++, C#, Java — являются объектно-ориентированными. В мире ежегодно проводятся сотни специализированных конференций, посвященных объектно-ориентированному программированию (ООП). Средства ООП добавляются не только в императивные языки (так из Си получились C++ и Objective C), но и, например, в функциональные (Лисп — CLOS, Common Lisp Object System). Провозглашается, что объектно-ориентированный подход позволяет создавать более легко сопровождаемые и модифицируемые сложные программные системы. Некоторые аналитики даже относят объектно-ориентированные языки к выделенному поколению языков программирования — поколению «3,5».

Данная глава посвящена введению в объектно-ориентированное программирование на примере языка C++.

Прежде всего заметим, что объектно-ориентированный подход может применяться не только в программировании — да и пришел он в программирование, так сказать, со стороны. Если смотреть на мир сквозь призму объектно-ориентированного подхода, все в нем мы воспринимаем как объекты. Объектами будут стул, стол, кран и т. п. К различным объектам применимы различные действия.

ЗАМЕЧАНИЕ

> Преподавательская практика автора позволила сделать небезынтересное наблюдение. В ходе лекции, посвященной ООП, задав аудитории вопрос, какое действие можно произвести со стулом, почти непременно слышишь в ответ радостное: «Сломать!».

Набор действий, или операций, применимых к объекту, определяется его классом, или типом. Кроме этого, объект каждого класса определяется характерным для него набором свойств, или атрибутов. Набором важнейших свойств стула, видимо, можно считать, с одной стороны, габаритные размеры и материал, а с другой — стоимость.

ЗАМЕЧАНИЕ

> Существуют и бесклассовые объектно-ориентированные языки, например Self, Lua.

ООП является довольно молодой областью исследований. В связи с этим в

ней до сих пор не существует общепринятой терминологии. Существует целый набор синонимов, означающих одно и то же и характерных, например, для программистов на C++ или Java. Читателю предлагается запомнить, что *класс* и *тип*; *родительский класс*, *базовый класс* и *суперкласс*; *функция-член*, *метод класса и сообщение*; *переменная-член*, *свойство* и *атрибут*; *подкласс*, *унаследованный класс*, *наследуемый класс*, *дочерний (сыновний) класс* — суть одно и то же.

Весьма важно выделить именно значимые и отличающие данный класс от других наборы свойств объекта. Этот процесс называется *абстракцией*. Абстракция, или абстрагирование, лежит в основе ООП.

Из книги в книгу кочует утверждение, что базовыми принципами, тремя китами, на которых покоится ООП, являются *инкапсуляция*, *наследование* и *полиморфизм*, которые будут рассмотрены далее. Однако с этим не вполне можно согласиться. Нам представляется, что наряду с *абстракцией* важнейшим принципом ООП является *интеграция данных и функций обработки этих данных*, или придание им «активности».

Если традиционно программист должен был описать, с одной стороны, набор пассивных структур данных, подлежащих обработке, а с другой — набор функций для обработки этих данных (причем эти определения были в некотором смысле ортогональны, или независимы — вплоть до хранения в разных модулях программы), то в ООП все иначе. Определяя класс, программист должен сразу задать методы обработки объектов этого класса — *внутри* него. На самом деле данный подход был разработан в рамках концепции *абстрактного типа данных*, предшествовавшей ООП.

В языках программирования существует набор встроенных базовых типов данных — целое число, логический тип, символ и т. д. Что произойдет при попытке применить, например, умножение к строкам? Или операцию взятия остатка от деления — к числам с плавающей точкой? Как правило, ошибка, причем в зависимости от того, относится язык программирования к языкам со строгой статической или с нестрогой динамической типизацией, или на этапе компиляции, или, что хуже, *ошибка времени выполнения* с выдачей в процессе исполнения программы системного сообщения и аварийным остановом.

Концепция абстрактных типов данных позволяет избежать подобных коллизий. Предписывается четко задавать набор операций, применимых к каждому типу данных. Возможность конструирования (добавления в язык) новых типов появилась еще в Паскале и Си. Довольно широкие возможности подобного вида были введены в языки Ада 83, Модула-2 и Модула-3. Однако до своего логического завершения она была доведена в Клу (Clu, название происходит от слова «кластер» — cluster, которым в нем обозначался абстрактный тип данных), созданном в 1974 году

Барбарой Лисков в МИТ.

ЗАМЕЧАНИЕ

> В середине 1980-х годов компилятор Клу реализован для советских суперкомпьютеров «Эльбрус», язык был отобран среди прочих кандидатов (Ада, Модула-2, Симула) как наиболее целостно и полно реализующий концепцию абстрактных типов данных и при этом довольно простой в реализации. Барбара Лисков стала лауреатом премии Тьюринга 2008 года. В ее представлении проектирование языка Клу и создание серии эффективных компиляторов для него отмечены как фундаментальный вклад в информатику, доказавший практическую осуществимость идей абстракции данных и превративший теоретическую концепцию в общепризнанный подход.

В языке Клу уже были реализованы такие особенности ООП, как *инкапсуляция* и *полиморфизм*. Впрочем, инкапсуляция — не уникальное в программировании именно для ООП явление, что не позволяет ее считать отличительной его чертой. Что понимается под этим словом? На самом деле, можно использовать простой русский эквивалент — *сокрытие*, и известно оно еще с начала использования локальных переменных процедур и внедрения модульности в программировании. Ясно, что лучше, чтобы как можно больше данных были скрыты от внешнего наблюдателя внутри некоторого программного модуля, были локальными для него. Во-первых, это позволяет повысить безопасность, предохраняя информацию от случайной или преднамеренной порчи, а во-вторых, чем меньше связей между модулями, тем проще и быстрее можно модуль заменить при модернизации или сопровождении программы.

А что же имеется в виду под полиморфизмом, также поддержанным в рамках концепции абстрактного типа данных?

Рассмотрим еще раз пример с операциями, вроде бы не характерными для определенных типов данных. Умножать строки, видимо, бессмысленно в любом случае. А складывать? Во многих языках программирования определена операция *конкатенации*, или слияния строк, когда в конец одной строки приписывается другая. При этом часто она обозначается знаком «плюс», который используется и для обозначения сложения! Пример на Бейсике:

```
10 A$="дядя " + "Ваня"
20 ? A$
«дядя Ваня»
```

В то же время мы понимаем, что, например, при делении целых чисел необходимо выполнить несколько иные операции, нежели при делении чисел с плавающей точкой. В некоторых языках программирования, например Фортране, встроен также тип комплексных чисел. При их делении нужно выполнить целый ряд операций, отличных от простого

деления. При этом обозначение всех этих операций деления совпадает.

Полиморфизмом называется использование одноименных операций, по-разному применяющихся к данным различных классов.

Все же языки с поддержкой абстрактных типов данных не относят безоговорочно к объектно-ориентированным. Принято считать, что для этого необходима поддержка в языке такой принципиальной возможности, как *наследование*. Что под этим понимается?

Определив однажды некий полезный тип данных, мы можем в дальнейшем захотеть использовать его в качестве основы для несколько модифицированного типа, обладающего дополненными или измененными свойствами и методами, не отказываясь тем не менее и от базовой версии. Наследование — механизм, дающий возможность подобного *повторного использования* ранее созданных классов для построения на их основе унаследованных классов, сохраняющих атрибуты и методы базового класса. Естественно, имеется возможность расширения функциональности базового класса и изменения некоторых функций в случае необходимости.

ЗАМЕЧАНИЕ

> Повторное использование — базовый принцип, без которого было бы невозможно современное программирование. Принцип положен в основу библиотек программных модулей, переноса программ на различные платформы и т. д.

Когда же появились первые действительно объектно-ориентированные языки программирования?

Первым объектно-ориентированным языком программирования в истории является Симула. Он был разработан в конце 1960-х годов сотрудниками норвежского вычислительного центра (Осло) Кристеном Нюгордом и Оле-Йоханом Далем с целью моделирования сложных систем, что отразилось на его названии (simulation — моделирование). Наиболее известна версия Симула-67. Язык является расширением Алгола-60. Симула в значительной степени опередила свое время, программисты 1960-х годов оказались не готовы воспринять ее ценности, она не выдержала конкуренции прежде всего с Фортраном. Тем не менее этот язык активно использовался в образовательном процессе в высших учебных заведениях, особенно в Скандинавии. О влиянии достоинств и недостатков Симулы на разработку им языка C++ пишет Бьерн Страуструп в своей книге «Дизайн и эволюция C++». Без сомнения, идеи, заложенные в этот язык, оказали влияние и на другие реализации ООП, такие как Smalltalk.

Smalltalk (произносится «смóлток», в переводе — светская беседа) — объектно-ориентированный язык программирования с динамической типизацией, разработанный в лаборатории Xerox PARC Аланом Кэйем,

Дэном Ингаллсом, Тедом Кэглером и другими в 1970-х годах. Язык был представлен как стандартная версия Smalltalk-80. Smalltalk представляет собой воплощение идей ООП в чистом виде, в нем всё — объекты, включая даже условную конструкцию! Решение принимается с помощью посылки сообщения `ifTrue:` логическому объекту, управление переходит к указанному фрагменту текста, если значение истинно. В языке всего три базовых конструкции: посылка сообщения объекту, присваивание объекта переменной, возвращение объекта из метода. Smalltalk оказал большое влияние на развитие многих современных языков программирования. Помимо прочего, в нем, в частности, была разработана модель построения пользовательского интерфейса по принципу MVC (Model — View — Controller).

Заметим, что в языке Smalltalk, в отличие от многих других объектно-ориентированных языков, в которых обычно говорится о действии над объектами, имеющими пассивный характер, объекты рассматриваются как активные сущности, обрабатывающие посылаемые им сообщения. Примером из сказки может служить знаменитое «Сезам, откройся!» Аладдина, которое может быть представлено на Смолтоке как `Sezam open`.

ЗАМЕЧАНИЕ

> С точки зрения практического программирования, то, как рассматриваются объекты — в активном или страдательном залоге, — не очень важно. Отличие имеет скорее философское и аналитически-языковое значение.

Несмотря на существование и довольно активное применение Smalltalk, все же сейчас господствующее положение занимают объектно-ориентированные языки программирования, построенные как расширение уже существовавших ранее и широко использовавшихся языков. При подобном подходе в хорошо знакомый и привычный программистским массам язык добавляются некие полезные возможности, при этом часто возникает ситуация, когда любая программа на базовом языке является допустимой программой на расширенном языке. Это позволяет преодолеть косность мышления программистов путем плавного перехода к использованию новых возможностей — одной, затем другой, третьей и последующих. Ярким примером является C++, который будет рассмотрен далее.

Язык возник в начале 1980-х годов, когда его автор Бьерн Страуструп придумал ряд усовершенствований языка Си. В конце 1970-х Страуструп начал работать в Bell Labs над задачами теории очередей в приложении к моделированию телефонных вызовов. Он обнаружил, что попытки применения существующих в то время языков моделирования

неэффективны, а использование эффективных машинных языков, дающих быстрый и компактный код, например Си, слишком сложно из-за их ограниченной выразительной силы. Вспомнив свою диссертацию, Страуструп решил дополнить язык Си возможностями, имеющимися в языке Симула. Язык Си был базовым языком системы UNIX, на которой работали компьютеры Bell. Страуструп добавил к нему возможность работы с классами и объектами. В результате практические задачи моделирования оказались доступными для решения как с точки зрения времени разработки (благодаря использованию Симула-подобных классов), так и с точки зрения времени вычислений (благодаря быстродействию Си). Ранние версии языка, первоначально именовавшегося «Си с классами» («C with classes»), стали доступны в 1980 году. В 1985 году вышло первое издание книги «Язык программирования C++», содержащей описание языка, что было чрезвычайно важно из-за отсутствия официального стандарта. В 1989 году состоялся выход C++ версии 2.0. В 1998 году был опубликован стандарт языка ISO/IEC 14882:1998 (известный как C++98), разработанный комитетом по стандартизации C++ (ISO/IEC JTC1/SC22/WG21 working group). Наиболее свежей версией языка, закрепленной в соответствующем стандарте, является C++11. Наименование языка, происходит от оператора инкремента Си ++ — увеличение значения переменной на единицу. Название «C+» не было использовано, потому что является синтаксической ошибкой в Си, кроме того, этим именем уже был назван другой языко. Также язык не был назван D, поскольку, по словам Страуструпа, «является расширением Си и не пытается устранять проблемы путем удаления элементов Си».

ЗАМЕЧАНИЕ

Язык D позже был создан, но это другой язык программирования.

Рассмотрение C++ начнем с некоторых полезных добавлений, которые не относятся непосредственно к объектно-ориентированному программированию, но делают написание программ более удобным. Итак, в языке были официально введены однострочные комментарии, начинающиеся с двух косых черт `//` и заканчивающиеся символом перевода строки. Примеры использования:

```
// функция sum осуществляет суммирование комплексных чисел
complex sum(complex a,complex b);


int s; // в s накапливаем сумму нечетных
```

Другой весьма удобной возможностью является так называемый *потоковый ввод-вывод*. Вместо использования библиотечных функций

scanf и printf с не вполне прозрачной обязательной системой обозначений форматных символов стало возможным применение конструкций вида

```
cout << "введите значения переменных a,b и d:" << endl;
cin >> a >> и >> d;
```

Здесь cin — наименование стандартного потока ввода, обычно связанного с клавиатурой, cout — название стандартного потока вывода, назначаемого обычно на экран дисплея; endl — обозначение символа перевода строки. Для применения потокового ввода-вывода необходимо подключить стандартную библиотеку с заголовочным файлом iostream.h.

Еще одно весьма удобное изменение — разрешение объявлять переменные в произвольном месте программы, а не только в начале тела функции или вне функций для глобальных переменных. Популярной конструкцией при программировании на C++ становится использование глубоко локальных переменных, время жизни которых ограничивается, например, исполнением конкретного оператора цикла, как в следующем примере:

```
for (int i=0;i<N;i++)
{
 …
}
```

После выхода из цикла при переходе к следующему после символа } действию программа «забывает» о существовании и использовании переменной i. Это позволяет программиступовторно использовать это имя с произвольными целями и снизить количество трудноуловимых ошибок в программах.

Однако давайте перейдем к воплощению идей объектно-ориентированного программирования в C++. Объявление класса выглядит следующим образом:

```
class <имя класса>
{
    <переменные-члены класса>
    <функции-члены класса>
};
```

На самом деле функции-члены класса и переменные-члены класса могут описываться внутри определения класса в произвольном порядке.

ЗАМЕЧАНИЕ

Членами класса могут быть и константы.

Приведем пример определения класса человек (chelovek — жаль, но в большинстве реализаций C++ использование кириллических символов не допускается):

```
class chelovek
{
  public:
    int godRogd; // год рождения
    char Fam[50]; // фамилия
    char Ima[20]; // имя
    char Otc[50]; // отчество

    // Вывод информации о человеке на экран
    void pecat_kto_takoi()
    {
        cout << Fam << Ima << Otc << godRogd << "года рождения";
    }

};
```

Разберем пример более подробно. У класса объявляется набор общедоступных членов — об этом говорит *модификатор доступа* public. Это переменные-члены класса, описывающие фамилию, имя и отчество с помощью строк и год рождения — с использованием целого числа, а также одна функция-член pecat_kto_takoi(), служащая для вывода информации о человеке на экран (void в заголовке функции означает, что она не возвращает никакого значения, return со значением в таких функциях не используется).

В приведенном примере функция-член определена прямо в теле класса. Это не подходит для длинных сложных функций — для них предусматривается возможность задания при определении класса лишь ее объявления и реализации в другом месте текста программы. При этом определение функции должно предваряться именем класса, после которого идет двойное двоеточие, как в примере:

```
// определение функции-члена вне класса
class primer
{
   int delai_nechto(); // функция-член, лишь объявление — заголовок
   float x; // переменная-член класса
};
…
// реализация функции-члена класса primer
int primer:: delai_nechto()
```

```
{
  …
}
```

После объявления класса мы получаем право создавать в программе на C++ объекты данного класса, причем полностью равноправные с объектами встроенных типов:

```
chelovek Liza, Andrei, Nasta; // три объекта класса человек
```

и обращаться к их членам, отделяя имя объекта и имя члена с помощью точки:

```
Liza.pecat_kto_takoi();
Andrei.godRogd=1971;
```

Среди функций-членов класса имеются две особые разновидности — *конструктор* и *деструктор*. Функция-конструктор вызывается каждый раз, когда нужно создать объект данного класса, а деструктор — при его уничтожении. Имя конструктора в языке программирования C++ совпадает с именем класса, а имя деструктора образуется добавлением в начало наименования класса символа ~.

Пример конструктора для класса `человек` (обратите внимание на еще одну особенность — у функции-конструктора не указывается тип возвращаемого значения):

```
// конструктор класса человек
chelovek::chelovek(char I[],char F[],char O[])
{
  strcpy(Ima,I);
  strcpy(Fam,F);
  strcpy(Otc,O);
}
```

После задания данного конструктора допустимым становится создание нового объекта класса `человек` следующим образом:

```
chelovek Dima("Дмитрий","Геннадьевич","Романов",1971);
```

Класс может иметь несколько конструкторов, различающихся сигнатурой (количеством и типом аргументов). Так, можно задать конструктор, который будет инициализировать фамилию, имя и отчество пустыми строками:

```
// другой конструктор для класса человек
chelovek::chelovek()
{
  strcpy(Ima,"");
  strcpy(Fam,"");
```

```
  strcpy(Otc,"");
}
```

Наличие нескольких конструкторов — частный случай так называемой *перегрузки* функций в объектно-ориентированных языках, тесно связанной с полиморфизмом. Все задаваемые программистом конструкторы должны быть объявлены при определении класса. В нашем случае это выглядит так:

```
chelovek::chelovek(char I[],char F[],char O[],int G);
chelovek::chelovek();
```

Если программист при описании класса не создает ни одного конструктора явным образом, система создает один, не имеющий аргументов и называемый *конструктором по умолчанию*.

Помимо создания объекта путем объявления переменной данного класса возможно объявление переменной-указателя на объект данного класса. В этом случае используется звездочка:

```
chelovek *Natasha;
```

Следует помнить, что при объявлении указателя объект на самом деле не создается и впоследствии нужно выполнить специальное действие:

```
Natasha=new chelovek;
```

После этого можно обращаться к членам класса, используя не точку, а стрелку `->`:

```
Natasha->godRogd=2003;
Natasha->pecat_kto_takoi();
```

ЗАМЕЧАНИЕ

> Нужно четко различать объекты и классы. Каждый из создаваемых объектов класса имеет уникальный набор значений атрибутов, хотя все объекты и имеют одинаковый набор (имен) атрибутов. Образно выражаясь, класс — это выкройка штанов, а объект — конкретные штаны, сшитые по выкройке. По одной выкройке можно сшить много штанов.

Поговорим о доступе к членам класса. В C++ используются три возможных модификатора доступа — `public`, `private` и `protected`. Модификатор доступа заканчивается двоеточием и влияет на идущие после него в классе переменные-члены и функции-члены, расположенные до следующего модификатора доступа.

Если модификаторы доступа отсутствуют, по умолчанию все члены класса считаются закрытыми (`private`). Закрытые члены класса доступны лишь изнутри объектов самого класса и из так называемых дружественных функций (объявляемых с использованием специального ключевого слова `friend`). Защищенные, или `protected`, члены класса доступны в

объектах данного класса и унаследованных из него классов (потомках). Публичные члены public доступны отовсюду.

Рассмотрим наследование в C++ на следующем примере. Определим класс студент (student). Никто не будет, вероятно, спорить с тем, что студент — человек. Соответственно, было бы удобно унаследовать класс student от класса chelovek. Записать это на языке C++ можно следующим образом:

```
// класс студент, унаследованный от человека
class student:public chelovek
{
  char Gruppa[25]; // группа
  float srball; // средний балл
  float stipendia; // размер стипендии
};
```

Все атрибуты человека — фамилия, имя, отчество, год рождения — сохраняются и у студента. В то же время для студента указываются еще группа (Gruppa) и успеваемость, выражаемая средним баллом, и может указываться стипендия.

Во многих языках программирования возможно лишь одиночное наследование, когда подкласс имеет ровно один суперкласс. В языке программирования C++ поддерживается множественное наследование. Поясним этот механизм на примере.

Рассмотрим класс автомобиль. С одной стороны, автомобиль можно рассматривать как транспортное средство. Важнейшими свойствами для транспортного средства являются тип, вместимость, мощность, скорость, расход топлива и т. д. С другой стороны, автомобиль — недешевый товар, для которого весьма важны цена, марка (производитель), цвет (особенно для покупательниц женского пола, чтобы соблюсти гармонию с цветом туфель). Определим два класса — транспортное средство и товар:

```
// транспортное средство
class trans
{
  float skorost; // скорость
  float moshnost; // мощность
  int vmestimost; // вместимость
};

// товар
class tovar
{
```

```
  char cwet[25]; // цвет
  char proizvoditel[50]; // производитель
  float cena; // цена
};
```

Класс автомобиль, выступающий одновременно в двух ипостасях — как транспортное средство и товар, может быть определен путем наследования от этих двух классов (рис. 25):

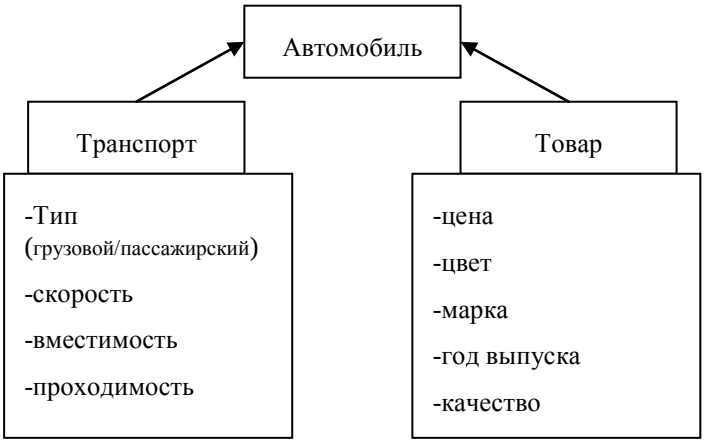


Рис. 25

```
// автомобиль — пример множественного наследования
class automobile: trans,tovar
{
};
```

При этом объекты класса automobile будут иметь все атрибуты как транспортного средства, так и товара.

Как уже говорилось, в унаследованных классах методы базового класса могут быть уточнены — *переопределены*. При этом функция, которая изменяется в наследуемом классе, называется виртуальной, и перед ее именем в объявлении базового класса необходимо поставить ключевое слово virtual.

Рассмотрим переопределение функций на примере базового класса *геометрическая фигура* и унаследованных классов линия и прямоугольник. Определение базового класса геометрических фигур может выглядеть следующим образом:

```
/* геометрическая фигура,атрибуты:координаты на плоскости и цвет */
class Fig
{
```

```
   public:
     void SetXY(int nx, int ny) // задание координат
       {
       x=nx;
       y=ny;
       }
       void Pokras(int cvet)
       {
       color=cvet;
       }

       void Skryt(); // скрыть – нарисовать черным
     {
        Pokras(BLACK);
        Risui();
     }
     virtual int Risui()=0;

   private:
     int x, y;
     int color;
};
```

Обратите внимание на следующее обстоятельство. Непосредственный доступ к координатам `x`, `y` и цвету фигуры закрыт (модификатор `private`). Публичными объявлены функция `SetXY`, при использовании которой возможна модификация координат, и функция `Pokras` для изменения цвета фигуры. Данный подход широко применяется в объектно-ориентированных программах. Еще один весьма важный момент связан с объявлением функции `Risui`, предназначенной для отображения фигуры на экране. Она не только объявлена как виртуальная — фигурирует запись `=0`, довольно оригинальная. Подобный необычный синтаксис в языке программирования C++ предназначен для объявления так называемых чистых виртуальных функций — функций, не подлежащих определению в базовом классе. Действительно, мы не можем изобразить фигуру вообще — не конкретно прямоугольник, треугольник или окружность, а геометрическую фигуру. Конкретный вариант функции `Risui` определяется в наследуемых классах:

```
// класс линия
class Linia: public Fig
{
   public:
```

```
      void Risui();
      {
         setcolor(color);
         line(x,y,xk,yk);
      }

// конструктор
        Linia (int ix,int iy, int ixk,int iyk)
        {
        x=ix;y=iy;xk=ixk;yk=iyk;
          }
     private:
        int: xk,yk;  // координаты конца линии
};

// класс прямоугольник
class Priam: public Fig
{
   public:
      void Risui();
      {
         setcolor(color);
         rectungle(x,y,xk,yk);
      }

          // конструктор
          Priam (int ix,int iy, int ish,int iv)
        {
           x=ix;y=iy;shir=ish;vys=iv;
          }
    private:
      int: shir,vys;  // ширина, высота
};
```

Класс, содержащий хотя бы одну чистую виртуальную функцию, называется *абстрактным классом*. Создать объект абстрактного класса нельзя. Такой класс используется лишь как часть иерархии наследования классов.

Контрольные вопросы

1. Применим ли объектно-ориентированный подход за рамками программирования?

2. Можете ли вы назвать функциональные языки программирования, поддерживающие ООП?
3. Метод и функция-член в ООП — одно и то же? Атрибут и переменная-член? Подкласс и наследуемый класс? Чем определяется применяемая терминология?
4. В чем заключается концепция абстрактного типа данных?
5. Что такое полиморфизм? Приведите примеры.
6. Опишите концепцию наследования в ООП. Какова связь между наследованием и повторным использованием?
7. В чем, по вашему мнению, заключаются недостатки и преимущества множественного наследования?
8. Что такое дружественная функция?
9. Зачем используется переопределение функций?
10. Какой из языков ООП появился первым и когда? С какими целями он создавался и родственен ли Алголу?
11. Является ли Smalltalk языком чистого объектно-ориентированного программирования с наиболее полным воплощением концепции ООП? Кем и с какими целями был создан язык Smalltalk?
12. Что такое модификатор доступа? Какие модификаторы вам известны? Если в программе на C++ не указать модификатор доступа, какими будут члены класса по умолчанию — закрытыми или открытыми?
13. Является ли концепция инкапсуляции уникальной для ООП?
14. Что такое деструктор? Когда он вызывается?
15. Что такое конструктор? Может ли у класса быть несколько конструкторов?
16. У каждого ли класса существует конструктор? Что такое конструктор по умолчанию?
17. Что такое виртуальная функция? Может ли она быть вызвана на выполнение?
18. Что такое чистая виртуальная функция и абстрактный класс?
19. Поясните принцип действия модификатора доступа `protected`. Что такое защищенные члены класса?
20. Возможен ли доступ к членам класса через указатели? Приведите примеры.

Достоинства и недостатки ООП

Как уже говорилось, в настоящее время объектно-ориентированное программирование является магистральным путем развития в индустрии информационных технологий. Апологеты ООП говорят прежде всего о следующих достоинствах объектно-ориентированных систем:

- возможности повторного использования;
- упрощении создания сложных систем за счет использования абстракции;
- удобстве сопровождения и модификации объектно-ориентированных программ.

В то же время у ООП есть и ряд довольно авторитетных критиков. К ним, в частности, относится такой известный в мире программирования человек, как Никлаус Вирт, создатель языков Паскаль, Модула и Оберон. Он утверждает, в частности, что ООП — не более чем тривиальная надстройка над структурным программированием и преувеличение его значимости, выражающееся в том числе во включении в языки программирования все новых модных «объектно-ориентированных» средств, безусловно, вредит качеству разрабатываемого программного обеспечения. Н. Вирт очень удивлен тем вниманием, которое уделяется ныне ООП. Другой критик ООП — известный специалист по программированию Александр Степанов, который участвовал в создании C++ вместе с Бьерном Страуструпом, а впоследствии написал стандартную библиотеку шаблонов языка программирования C++ (STL). А. Степанов полностью разочаровался в парадигме ООП. Он пишет: «Я уверен, что ООП методологически неверна. Она начинает с построения классов. Это как если бы математики начинали бы с аксиом. Но реально никто не начинает с аксиом, все начинают с доказательств. Только когда найден набор подходящих доказательств, лишь тогда на этой основе выводится аксиома. То есть в математике вы заканчиваете аксиомой. То же самое и с программированием: сначала вы должны начинать развивать алгоритмы, и только в конце этой работы приходите к тому, что вы в состоянии сформулировать четкие и непротиворечивые интерфейсы. Именно из-за этой неразберихи в ООП так популярен рефакторинг — из-за ущербности парадигмы вы просто обречены на переписывание программы уже в тот самый момент, когда только задумали ее спроектировать в ООП-стиле».

Патриарх свободно распространяемого ПО Ричард Столлмэн также известен своим критическим отношением к ООП. Особенно любит он шутить насчет мифа о том, что ООП «ускоряет разработку программ»: «Как только ты сказал слово “объект”, можешь сразу забыть о

модульности».

Критики, в частности, обращают внимание на следующие обстоятельства:

- Внедрение ООП требует дополнительных затрат на обучение программистов и накладных расходов при создании программ — от 12 до 75 % времени.
- Наследование — самая большая провокация в индустрии, ни в каком моделировании наследования не существует.
- Инкапсуляция и повторное использование возможны и без ООП при правильной модульной организации программной системы.

В то же время ряд наиболее сложных и успешно используемых программных систем созданы именно в рамках объектно-ориентированного подхода, его применяют такие гиганты индустрии, как Microsoft, IBM, Oracle.

Оставляем за читателем право самостоятельно сделать выводы о пользе и недостатках объектно-ориентированного подхода к программированию.

Контрольные вопросы

1. В чем заключаются плюсы объектно-ориентированных программных систем?
2. Можно ли сказать, что объектно-ориентированный подход — пустое усложнение, не несущее выгод? Обоснуйте свое мнение.

Список литературы

1. Современный компьютер: Сб. науч.-попул. Статей; Пер. с англ./Под ред. В.М.Курочкина; предисл. Л.Н. Королева — М: Мир, 1986.
2. Хофштадтер Д. Гедель, Эшер, Бах — эта бесконечная гирлянда. — Самара: Бахрах-М, 2000.
3. Тьюринг А. Может ли машина мыслить. — М.: Физматгиз, 1950.
4. Лорьер Ж.-Л. Системы искусственного интеллекта. — М.: Мир, 1991.
5. Ермаков И. Е. Лекции с обзором языков программирования.
6. Непейвода Н. Стили и методы программирования. Курс лекций. Учебное пособие. — М.: Интернет-университет информационных технологий, 2005.
7. Пентковский В. М. Язык программирования Эль-76. Принципы построения языка и руководство к пользованию. — 2-е изд., испр. и доп. — М.: Физматлит, 1989.
8. Хамби Э. Программирование таблиц решений. — М.: Мир, 1976.
9. Вирт Н. Систематическое программирование. Введение. — М.: Мир, 1977.
10. Кнут Д. Искусство программирования. В 3 т. — М.: Вильямс, 2012.
11. Керниган Б., Ритчи Д. Язык программирования Си. 2-е изд.— М.: Издательский дом «Вильямс», 2013.
12. C/C++. Программирование на языке высокого уровня / Т.А. Павловская — СПб.: Питер, 2011 .
13. Пустоваров В. И. Язык ассемблера в программировании информационных и управляющих систем. — М.: ДЕСС, 1998.
14. Стерлинг Л., Шапиро Э. Искусство программирования на языке Пролог. — М.: Мир, 1990.
15. Тюгашев А.А. Графические языки программирования и их применение в системах управления реального времени. — Самара: Изд-во СНЦ РАН , 2009.
16. ГОСТ 19.701–90. Единая система программной документации. Схемы алгоритмов, программ, данных и систем. Условные обозначения и правила выполнения. — М.: Госстандарт, 1991.
17. Коварцев А. Н. Автоматизация разработки и тестирования программных средств / Изд-во Самар. гос. аэрокосм. ун-та — Самара, 1999.

18. Харьковский З. Путеводитель автостопщика по потаенным знаниям // Компьютерра.- 2005. № 12. — С. 42–52.

19. Страуструп Б. Язык программирования C++. 3-е изд. — М.: Бином, 2011.

20. Кулямин В. В. Перспективы интеграции методов верификации программного обеспечения // Труды ИСП РАН. — 2009. — Т.16:-С.73-88,

21. Дейкстра Э. Дисциплина программирования. — М.: Мир, 1978.

22. Пароннджанов В. Д. Как улучшить работу ума. — М.: Дело, 2001.

23. Кауфман В. Языки программирования. Концепции и принципы. — М.: ДМК Пресс, 2011.

24. Пратт Т., Зелковиц М. Языки программирования: разработка и реализация / Под общей ред. А. Матросова. — СПб.: Питер, 2002.

25. Абельсон Х., Сассман Д. Структура и интерпретация компьютерных программ. — М.: Добросвет, 2010.

УНИВЕРСИТЕТ ИТМО

Миссия университета – генерация передовых знаний, внедрение инновационных разработок и подготовка элитных кадров, способных действовать в условиях быстро меняющегося мира и обеспечивать опережающее развитие науки, технологий и других областей для содействия решению актуальных задач.

КАФЕДРА КОМПЬЮТЕРНЫХ ОБРАЗОВАТЕЛЬНЫХ ТЕХНОЛОГИЙ

Кафедра «Компьютерные образовательные технологии» (КОТ) создана в 2001 году на факультете информационных технологий и программирования (ИТП) Санкт-Петербургского государственного университета информационных технологий, механики и оптики (НИУ ИТМО) для реализации учебного процесса по ряду профильных дисциплин направления подготовки специалистов 230200 - «Информационные системы». С 2003 года кафедра КОТ стала выпускающей по специальности 230202 - «Информационные технологии в образовании», а с 2009 года по направлению подготовки магистров 230200 «Информационные системы». В апреле 2011 г. кафедра КОТ перешла в состав факультета компьютерных технологий и управления (КТиУ). В настоящее время в рамках реализации ФГОС кафедра КОТ перешла на двухуровневую подготовку выпускников (бакалавр, магистр) направления 09.03.02 – «Информационные системы и технологии». Мы предлагаем для этого образовательную программу «Автоматизация и управление в образовательных системах». Ведется подготовка аспирантов по специальности 05.13.06 - «Автоматизация и управление технологическими процессами и производствами (образование)» по техническим наукам. С 2015 года открыто новое направление подготовки магистров – «Программная инженерия систем реального времени»

Тюгашев Андрей Александрович

Основы программирования. Часть I.

Учебное пособие

В авторской редакции
Редакционно-издательский отдел Университета ИТМО
Зав. РИО Н.Ф. Гусарова
Подписано к печати
Заказ №
Тираж
Отпечатано на ризографе

Редакционно-издательский отдел
Университета ИТМО
197101, Санкт-Петербург, Кронверкский пр., 49