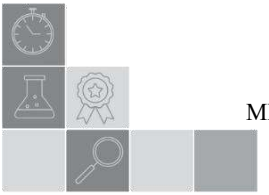
Министерство науки и высшего образования Российской Федерации
Министерство внутренних дел Российской Федерации

Московский университет Министерства внутренних дел
Российской Федерации имени В.Я. Кикотя

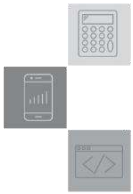
ПРОГРАММИРОВАНИЕ: ЯЗЫКИ, МЕТОДЫ И ТЕХНОЛОГИИ

Учебное пособие



Москва
Московский университет
МВД России имени В.Я. Кикотя

2021



УДК 004.42
ББК 32.973
П78

Рецензенты:
заместитель начальника центра – начальник отдела
(специальный отдел) ЦИТСиЗИ УМВД России
по Пензенской области **Я. Ю. Багрий**; врио начальника отдела
Управления защиты информации ДИТСиЗИ
МВД России **И. В. Кирсанов**

Коллектив авторов:
О. В. Пименова, Е. Н. Клочкова, К. Р. Аветисян,
Г. Г. Плотников, И. С. Мельцева, А. О. Тычкова

П78 **Программирование: языки, методы и технологии** : учебное пособие / [О. В. Пименова и др.]. – М. : Московский университет МВД России имени В.Я. Кикотя, 2021 – 153 с.
ISBN 978-5-9694-1005-3

В учебном пособии рассматриваются основные понятия и принципы алгоритмизации, системы вычисления и основы языка блок-схем, дается классификация языков программирования и их краткая характеристика.

Предназначено для курсантов и слушателей, обучающихся по специальностям «Безопасность информационных технологий в правоохранительной сфере» и «Информационная безопасность автоматизированных систем».

УДК 004.42
ББК 16.23

ISBN 978-5-9694-1005-3

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Одним из базовых понятий информатики, вычислительной техники и программирования является алгоритм, представляющий собой некоторые правила преобразования информации. Умение составлять алгоритмы и программы для решения различных практических задач является элементом алгоритмической культуры современного специалиста в области информационных технологий.

Цель данного учебного пособия – изучение основ алгоритмизации, уяснение понятий «технология программирования», «методология программирования». Кроме того, пособие содержит большое число задач, которые могут быть использованы для организации практических занятий.

Основы программирования излагаются на языке C++, являющимся основным и изучаемым в ходе подготовки специалистов.

Учебное пособие обобщает опыт преподавания дисциплины «Программирование: языки, методы и технологии» и способствует формированию навыков алгоритмического мышления и программирования вычислительных процессов.

ГЛАВА 1
Программа как формализованное описание процесса обработки данных

1.1. Интуитивное и строгое понятие алгоритма

Понятие алгоритма – одно из основных понятий математики – возникло задолго до появления вычислительных машин. На протяжении многих веков люди пользовались интуитивным понятием алгоритма, которое можно выразить следующим образом:

Алгоритм – это строгая и четкая конечная система правил, которая определяет последовательность действий над некоторыми объектами и после конечного числа шагов приводит к достижению поставленной цели.

В частности, система правил является алгоритмом, если ее можно вручить в качестве инструкции разным людям, не знакомым с сутью дела, и они, следуя этой системе правил, будут действовать одинаково.

Существует несколько версий происхождения термина «алгоритм».

Арабская версия. Арабский математик Абу Абдулла Мухаммед ибн Муса аль-Хорезми сформулировал правила счета с использованием индийских цифр и назвал их «Китаб аль-джебр валь-мукабала» («Книга о сложении и вычитании»). При переводе на европейские языки и латынь слова подверглись искажению: аль-джебр – алгебр – алгебра, аль-Хорезми – алгорезми – алгоритм.

Греческая версия (миф Аргонавтов). Арго́ – в древнегреческой мифологии – легендарный корабль, на котором в XIV в. до н. э. аргонавты отправились в переход через Эгейское море, пролив Босфор и Черное море к побережью Колхиды. Дело в том, что строевой лес в Греции – редкость и строительство корабля Арго

считали чудом. Порядок, в котором строился корабль, назвали алгоритмом (Арго – Арго-ритм – алгоритм).

За основную принимают арабскую версию, хотя и существует множество других.

Основными *свойствами алгоритма* являются следующие:

1) массовость алгоритма – свойство, означающее, что каждый алгоритм, разработанный для решения некоторой задачи, должен быть применим для решения задач этого типа при всех допустимых значениях исходных данных;

2) однозначность (детерминированность) – заключается в том, что при задании одних и тех же исходных данных несколько раз алгоритм будет выполняться абсолютно одинаково и всегда будет получен один и тот же результат. На каждом шаге выполнения алгоритма всегда точно известно, что делать дальше, а каждое действие однозначно понятно исполнителю и не может быть истолковано неопределенно. Благодаря этому свойству выполнение алгоритма носит механический характер;

3) результативность (направленность) – означает, что выполнение алгоритма обязательно должно привести к решению поставленной задачи либо к сообщению о том, что при заданных исходных величинах задачу решить невозможно. Алгоритмический процесс не может обрываться безрезультатно.

Алгоритм подразумевает следующие *основные элементы*:

1) данные – формализованный (при помощи языка и алфавита) образ объекта, над которым выполняется алгоритм. Данные поступают в начале алгоритма и в преобразованном виде приходят как результат его выполнения. Они должны формироваться из конечного числа символов алфавита, состоять из групп символов-слов и быть построенными в соответствии с правилами грамматики и построения формул. Не все слова являются данными для конкретного алгоритма;

2) память – возможность хранения результатов промежуточных вычислений;

3) элементарный шаг – возможность представления алгоритма в виде конечного числа элементарных шагов.

Пример. Алгоритм подсчета людей в зрительном зале: пройти по всем рядам и для каждого присутствующего человека прибавлять единицу к общему счетчику, в котором сначала был ноль.

Объекты этого алгоритма – люди в зале и числа. Над людьми выполняется действие «найти следующего», а над числами – действие «прибавить единицу».

При составлении алгоритма его можно представить следующими основными способами:

– словесное описание;

– псевдокоды (полуформализованные описания алгоритмов на условном алгоритмическом языке, включающие в себя как элементы языка программирования, так и фразы естественного языка, общепринятые математические обозначения и др.);

– структурограммы;

– графическое представление;

– программа.

Словесное описание (запись на естественном языке) – представление алгоритма в виде перечня этапов (действий, шагов) для обработки данных в произвольном изложении на естественном языке.

Псевдокод занимает промежуточное место между естественным и формальным языками. С одной стороны, он близок к обычному естественному языку, поэтому алгоритмы могут на нем записываться и читаться как обычный текст. С другой стороны, в псевдокоде используются некоторые формальные конструкции и математическая символика, что приближает запись алгоритма к общепринятой математической записи. В псевдокоде не приняты

строгие синтаксические правила для записи команд, присущие формальным языкам, что облегчает запись алгоритма на стадии его проектирования и дает возможность использовать более широкий набор команд, рассчитанный на абстрактного исполнителя. Однако в псевдокоде обычно имеются некоторые конструкции, присущие формальным языкам, что облегчает переход от записи на псевдокоде к записи алгоритма на формальном языке.

Структурограммы изображают последовательность действий не с помощью линий перехода от блока к блоку, а в виде вложенных друг в друга фигур. Каждый блок структурограммы имеет прямоугольную форму и может быть вложен в любой внутренний прямоугольник другого блока (рис. 1).

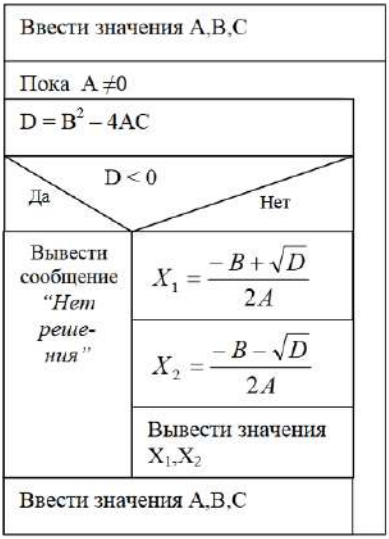


Рис. 1. Структурограммы

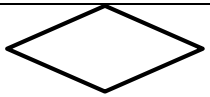
Графическое представление – изображение алгоритма в виде схемы, состоящей из графических символов.

В схеме алгоритма каждому типу действий (вводу исходных данных, вычислению значений выражений, проверке условий,

управлению повторением действий, окончанию обработки и т. д.) соответствует геометрическая фигура (табл. 1), называемая символом действия. Символы действия соединяются линиями передачи управления, определяющими очередность выполнения действий.

Таблица 1

Вид и описание блоков блок-схемы

Вид блока	Назначение блока
	Начало или конец алгоритма
	Ввод или вывод данных
	Вычисление или действие
	Логический блок проверки условия
	Начало циклического процесса (описание параметра итерационного повторения операций)

Правила применения символов:

1. Символ предназначен для графической идентификации функции, которую он отображает, независимо от текста внутри этого символа.

2. Символы в схеме должны быть расположены равномерно. Следует придерживаться минимального числа длинных линий;

3. Большинство символов задумано для включения текста внутри символа. Не должны изменяться углы и другие параметры, влияющие на соответствующую форму символов. Символы должны быть, по возможности, одного размера. Предпочтительна горизонтальная ориентация символов.

4. Минимальное количество текста, необходимого для понимания функции данного символа, надо помещать внутри данного символа. Текст для чтения должен записываться слева направо и сверху вниз независимо от направления потока. Если объем текста, помещаемого внутри символа, превышает его размеры, следует использовать символ комментария.

Правила выполнения соединений:

1. Потоки данных или потоки управления в схемах показывают линиями. Направление потока слева направо и сверху вниз считается стандартным.

2. В схемах нужно избегать пересечения линий. Пересекающиеся линии не имеют логической связи между собой, поэтому изменения направления в точках пересечения не допускаются.

3. Две или более входящие линии могут объединяться в одну исходящую линию. Если две или более линии объединяются в одну линию, место объединения должно быть смещено.

4. Линии в схемах должны подходить к символу либо слева, либо сверху, а исходить либо справа, либо снизу. Линии должны быть направлены к центру символа.

5. При необходимости линии в схемах следует разрывать, избегая излишних пересечений или слишком длинных линий, а также если схема состоит из нескольких страниц. Соединитель в начале разрыва называется внешним соединителем, а соединитель в конце разрыва – внутренним соединителем.

Если необходимо внести бóльшую ясность в схему (например, при соединениях), на линиях используются стрелки. Если поток имеет направление, отличное от стандартного, стрелки должны указывать это направление.

Программная реализация алгоритма – это *компьютерная программа*, написанная на каком-либо алгоритмическом языке про-

граммирования, например C++, Pascal, Basic и т. д. Программа состоит из команд определенного языка программирования. Отметим, что одна и та же блок-схема может быть реализована на разных языках программирования. Ответ при этом получает ЭВМ, а не человек.

В зависимости от состава и последовательности выполняемых действий алгоритмы принято разделять на:

1) линейные – действия выполняют последовательно по одному разу, при этом заданная последовательность действий не изменяется в зависимости от исходных и промежуточных данных;

2) разветвляющиеся – содержат альтернативные действия процесса обработки данных, состав последующих действий зависит от результата предыдущих действий (от выполнения некоторых условий);

3) циклические – многократно повторяют одну и ту же группу действий для формирования результата решения задачи.

Алгоритм Евклида. Общая идея

Древнегреческий математик Евклид предложил алгоритм вычисления общего наибольшего делителя двух натуральных чисел A и B. Суть этого алгоритма в том, чтобы вычитать из большего числа меньшее, занося результат на место большего, и действовать так до тех пор, пока числа не станут равны между собой. Эти равные числа и будут общим наибольшим делителем исходных двух чисел. В алгоритме Евклида используется тот факт, что общий наибольший делитель чисел A и B является общим наибольшим делителем их разности и любого из чисел A, B. Поэтому можно в паре чисел (A, B) заменить большее число на эту разность, а затем искать общий наибольший делитель для новой пары, в которой одно число уменьшилось.

Строгая запись алгоритма Евклида:

1. Рассмотреть A как первое число и B как второе число. Перейти к п. 2.
2. Сравнить первое и второе числа. Если они равны, то перейти к п. 5. Если нет, то перейти к п. 3.
3. Если первое число меньше второго, то переставить их. Перейти к п. 4.
4. Вычесть из первого числа второе и рассмотреть полученную разность как новое первое число. Перейти к п. 2.
5. Рассмотреть первое число как результат. СТОП.

Этот набор правил является алгоритмом, потому что, следуя ему, любой человек, умеющий вычитать, может получить общий наибольший делитель для любой пары чисел.

Недостаток интуитивного понятия алгоритма – нельзя доказывать невозможность алгоритмического решения многих сложных задач, так как не было строго определено само понятие алгоритма. Как следствие, возникла насущная проблема: построить формальное определение алгоритма, аналогичное известному интуитивному понятию.

В вычислительных алгоритмах объектами работы являются числа. В алгоритме шахматной игры объекты – это фигуры и позиции, и нужно выбрать очередной ход. При алгоритмизации производственных процессов объектами служат показания приборов и управляющие клавиши, и нужно найти такое нажатие клавишей, при котором процесс пошел бы лучшим образом. Однако во всех случаях можно считать, что алгоритм имеет дело не с объектами реального мира, а с образами этих объектов.

Уточнение понятия «алгоритм» до уровня математического объекта связано с так называемыми алгоритмическими моделями:

1. Теория вычислимых функций (Черч, Гедель, Клини, 1930–1936 гг.).

2. Абстрактные вычислительные машины (Тьюринг, Пост, 1937 г.).

3. Комбинаторные модели (Марков младший, 1954 г.).

Теория вычислимых функций – алгоритм всегда вычисляет некоторую функцию. Входные данные соответствуют выходным. Алгоритм рассматривается как способ вычисления функций. Из простых функций можно строить более сложные композиционные функции. Если элементарные функции вычислимы, то и их композиция вычислима. Любая вычислимая функция может быть построена из конечного числа вычислимых функций.

На начальных этапах становления компьютерных наук возникла необходимость анализа математических задач на возможность их решения алгоритмическим путем. При этом выяснилось, что само понятие практически не может быть выражено обычным человеческим языком. Тем не менее ряд исследователей создали математически строгий инструмент для проверки исследуемых задач на алгоритмическую разрешимость. Независимо друг от друга Тьюринг, Пост и Марков предложили свои подходы, сравнение которых дало сопоставимые результаты. Суть этих решений – создание абстрактной вычислительной машины с неограниченными ресурсами и бедным набором операций. Это подразумевало неограниченный универсализм – любая алгоритмически решаемая задача может быть представлена такой машиной. Бедный набор операций – максимальное упрощение решения, что дает наиболее строгое доказательство. Такой наиболее известной вычислительной машиной придумал Алан Тьюрин. Является ли та или иная вычислительная схема или способ преобразования данных алгоритмом, проверяется на приведенных алгоритмических моделях. Схема или способ являются алгоритмом, если они могут быть представлены как комбинация элементарных вычислимых функций и реализованы в виде машины Тьюринга

(Поста) или в виде комбинаторной модели Маркова. Все три модели показывают сопоставимые результаты.

Люди же заменяют это определение более или менее точными словесными интуитивными определениями. Для большинства прикладных задач этого достаточно, их алгоритмическая разрешимость очевидна, а существующие современные способы разработки не требуют математической строгости описаний.

Алан Тьюринг высказал предположение, что любой алгоритм в интуитивном смысле этого слова может быть представлен эквивалентной машиной Тьюринга. Это предположение известно как тезис Черча – Тьюринга.

За время своего существования человечество придумало множество алгоритмов для решения разнообразных практических и научных проблем, однако существуют и такие, для которых невозможно придумать алгоритмы их решения. Первой фундаментальной теоретической работой, связанной с доказательством алгоритмической неразрешимости, была работа Курта Геделя – его известная теорема о неполноте символических логик. Это была строго формулированная математическая проблема, для которой не существует решающего ее алгоритма. Усилиями различных исследователей список алгоритмически неразрешимых проблем был значительно расширен. Некоторые будут описаны ниже в качестве примера.

Проблема эквивалентности алгоритмов. По двум произвольным заданным алгоритмам (например, по двум машинам Тьюринга) определить, будут ли они выдавать одинаковые выходные результаты на любых исходных данных.

Проблема тотальности. По произвольно заданному алгоритму определить, будет ли он останавливаться на всех возможных наборах исходных данных. Другая формулировка этой задачи – является ли частичный алгоритм P всюду определенным.

Вычисление совершенных чисел. Совершенные числа – это числа, которые равны сумме своих делителей, например: 28 = 1 + 2 + 4 + 7 + 14.

Сегодня принято при доказательстве алгоритмической неразрешимости некоторой задачи сводить ее к ставшей классической задаче – «задаче останова».

1.2. Формализация. Алгоритмизация. Понятие программы

Формализация – отображение результатов мышления в точных понятиях или утверждениях, раскрывающих сущность изучаемых процессов действительности.

Абстрагирование – выделение существенных свойств, имеющих отношение к решаемой проблеме.

Классификация – сортировка объектов по выделенным свойствам.

Могут быть различные средства и способы отображения одинаковых (в смысле существенных свойств) объектов в виде условных обозначений (изображения, картинки, пиктограммы, сочетания звуков, наборы символов, букв, алфавиты, языки и т. д.).

Формализм (формализованный язык) создается для точного выражения мыслей с целью исключения возможности для неоднозначного понимания.

Алгоритм, записанный на понятном вычислительной машине алгоритмическом языке, называется программой. Соответственно, программирование заключается в создании и исследовании программ для вычислительных машин. Для удобства программирования предлагаются алгоритмические языки. Это ставит задачу разработки алгоритмов перевода программ на этих языках в такие программы, которые понятны вычислительной машине в силу ее конструкции.

Правила записи алгоритма для выполнения его вычислительной машиной оказываются очень жесткими – автомат не может ничего додумывать за человека.

Совокупность средств и правил записи алгоритмов называется алгоритмическим языком.

Составляющие процесса создания программы:

1) алгоритмизация – формализация данных об объекте и необходимых преобразований. Запись их в виде алгоритма на определенном формальном языке;

2) верификация – проверка или доказательство правильности разработанного алгоритма;

3) реализация или кодирование – преобразование формализованного алгоритма в последовательность команд конкретного вычислительного устройства;

4) тестирование – сопоставление результата, выдаваемого программой, с результатом реализации алгоритма. Тестирование не выявляет ошибок составления алгоритма.

Программа, написанная на алгоритмическом языке, должна быть интерпретирована, транслирована, скомпилирована, т. е. преобразована в системы команд конкретных технических устройств, выполняющих требуемые действия.

Этот процесс носит многоэтапный и сложный характер. Он представляется в виде модели, состоящей из виртуальных машин разного уровня – от электрических импульсов отдельных чипов и микросхем до языков высокого уровня – алгоритмических языков и интегральных объектно-ориентированных сред разработчика.

1.3. Абстрактные вычислительные машины

В абстрактных вычислительных машинах алгоритм должен давать однозначные результаты вне зависимости от того, кто его исполняет, и это может делать машина. Если для задачи можно

сформулировать модель в виде абстрактной машины, то она алгоритмически разрешима.

При комбинаторной модели данные могут быть любыми, но операции должны быть комбинаторными, т. е. перестановка, вставка, сдвиг, удаление со сдвигом и т. д. Идея близка к идее абстрактных машин. Иногда это называют машиной Маркова.

Особенности математического подхода:

1. Прикладная эффективность не рассматривается.

2. Инструмент вычислений должен быть наиболее примитивным, при этом позволяющий решать все разрешимые алгоритмические задачи. Проверить результаты, полученные наиболее простым, элементарным, примитивным способом, много проще.

3. Универсализм ставится выше вычислительной эффективности.

4. Физическая реализация вычислительной машины накладывает дополнительные сложности, связанные с возможностью ее удобного использования. Они сводят на нет ценность такой машины для абстрактного анализа.

5. Принципиальное отличие машины Тьюринга от вычислительных машин в том, что ее запоминающее устройство представляет собой бесконечную ленту, а у вычислительной машины может быть очень большое, но уж во всяком случае не бесконечное запоминающее устройство.

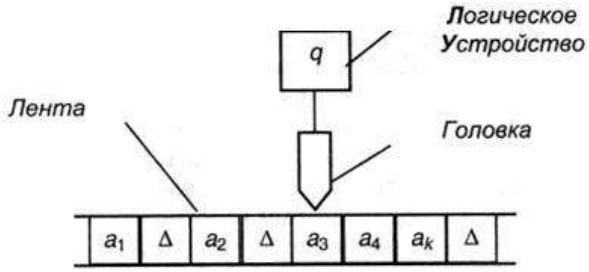


Рис. 2. Машина Тьюринга

В каждой машине Тьюринга есть три части (рис. 2):

– неограниченная в обе стороны лента, разделенная на ячейки;

– логическое устройство;

– головка.

С каждой машиной Тьюринга связан алфавит символов A и набор внутренних состояний Q (всего N состояний, обозначаемых $q_0, q_1, \ldots q_{n-1}$). В каждой ячейке ленты записан один символ из A (считается, что A содержит «пустой» символ Λ («лямбда»), а отсутствие записи в ячейке интерпретируется как запись символа Λ). Логическое устройство может находиться в одном из состояний $q_0, \ldots q_{n-1}$. В каждый момент времени головка обозревает одну из ячеек ленты.

Логическое устройство содержит команды, совокупность которых называется программой машины Тьюринга. Для каждого символа алфавита A и каждого состояния $q_0, \ldots q_{n-1}$ программа содержит в точности одну команду, выполнение любой из которых заключается в следующем:

– заменяется записанный в обозреваемой ячейке символ на новый символ;

– головка сдвигается на одну ячейку вправо или влево (может и не сдвигаться);

– переход в новое состояние.

Команда может быть обычной или заключительной.

Работа машины Тьюринга состоит из однотипных тактов. Каждый такт состоит в выполнении одной команды. Предполагается, что первоначально машина Тьюринга находится в состоянии 0. Таким образом, если задать информацию на ленте и положение головки, работа машины Тьюринга определяется однозначно. Работа считается завершенной, если выполнилась заключительная команда.

Полученное последнее содержимое ленты является результатом работы машины.

Машина Тьюринга была первым устройством, которое могло использовать алгоритмы для решения арифметических задач. Для многих экспертов это была первая теоретическая концепция современного компьютера. Интересно, что основная концепция машины Тьюринга до сих пор изучается в области информатики по всему миру. Основываясь на изобретенной машине, Тьюринг работал над ACE (the Automatic Computing Engine) в период с 1945 по 1947 г. Он также представил доклад о том, что компьютер может хранить программы в памяти (что и делают современные компьютеры). Алан Тьюринг разрабатывал и другие теории, концепции, например шифратор речи, Тьюринг-Велшман Bombe, «Колосс», Hut 8, the Naval Enigma и т. д.

Вскоре пришло и понимание того, что эти новые свойства вычислительных машин по сути описывают архитектуру некоторого абстрактного универсального вычислителя, который сейчас принято называть машиной фон Неймана. Она является абстрактной моделью ЭВМ, однако эта абстракция отличается от машины Тьюринга, которая может обрабатывать входные данные любого объема, поэтому этот исполнитель алгоритма принципиально нельзя реализовать.

Машина фон Неймана не поддается реализации по другой причине: многие детали в архитектуре этого вычислителя, как он описывается в учебниках, не конкретизированы. Это сделано специально, чтобы не сковывать творческого подхода к делу у инженеров-разработчиков новых ЭВМ. Можно сказать, что машина фон Неймана рассматривается не на внутреннем, а только на концептуальном уровне видения архитектуры. В некотором смысле машина фон Неймана подобна абстрактным структурам данных. У абстрактных структур данных, например двоичных деревьев,

для их использования необходимо предварительно выполнить конкретную реализацию: произвести отображение на структуры данных в некотором языке программирования, а также реализовать соответствующие операции над этими данными.

В машине фон Неймана зафиксированы прогрессивные особенности архитектуры, которые в той или иной степени должны были быть присущи всем компьютерам того времени. Разумеется, практически все современные ЭВМ по своей архитектуре в значительно отличаются от машины фон Неймана. Эти отличия удобно изучать, проводя сравнения и сопоставления с машиной фон Неймана.

Основополагающие свойства архитектуры машины фон Неймана были сформулированы в виде одноименных принципов, которые многие годы определяли основные черты архитектуры нескольких первых поколений ЭВМ.

Рассмотрим основные *принципы архитектуры фон Неймана*:

1. *Принцип линейности и однородности памяти*. Память машины фон Неймана – это линейная (упорядоченная) и однородная последовательность некоторых элементов, называемых ячейками. В любую ячейку памяти другие устройства машины могут записать и считать информацию, причем время чтения из любой ячейки одинаково для всех ячеек памяти. Время записи в любую ячейку тоже одинаково (это и есть принцип однородности памяти). Разумеется, время чтения из ячейки памяти может не совпадать со временем записи в нее. Такая память в современных компьютерах называется памятью с произвольным доступом (Random Access Memory – RAM). На практике современные ЭВМ могут иметь участки памяти разных видов. Например, некоторые области памяти поддерживают только чтение информации (по-английски эта память называется Read Only Memory – ROM), данные в такую память записывают один раз при изготовлении этой

памяти, они сохраняются при отключении электрического напряжения. Другие области памяти могут допускать запись, но за большее время, чем в остальную память (это так называемая полупостоянная память), и др. Ячейки памяти в машине фон Неймана нумеруются от нуля до некоторого положительного числа N (это и означает, что память линейная), причем число N в «настоящих» ЭВМ часто является степенью двойки минус единица. Адресом ячейки называется ее номер. Каждая ячейка состоит из более мелких частей, именуемых разрядами и нумеруемых также от нуля и до определенного числа. Количество разрядов в ячейке обозначает разрядность памяти. Каждый разряд может хранить одну цифру в некоторой системе счисления.

В большинстве ЭВМ используется двоичная система счисления, так как это более выгодно с точки зрения инженерной реализации. В этом случае каждый разряд хранит одну двоичную цифру или один бит информации. Восемь последовательных бит составляют один байт. Сам фон Нейман тоже был сторонником использования двоичной системы счисления, что позволяло хорошо описывать архитектуру узлов ЭВМ с помощью логических (булевских) выражений. Содержимое ячейки называется машинным словом. С точки зрения архитектуры машинное слово – это минимальный объем данных, которым могут обмениваться между собой различные узлы машины. Из каждой ячейки памяти можно считать копию машинного слова и передать ее в другое устройство компьютера, при этом оригинал не меняется. При записи в память старое содержимое ячейки пропадает и заменяется новым машинным словом.

2. *Принцип неразличимости команд и данных*. С точки зрения программиста машинное слово представляет собой либо команду, либо подлежащее обработке данное (это число, символьная информация, элемент изображения и т. д.). Для краткости в

дальнейшем будем называть такую информацию числами. Данный принцип фон Неймана заключается в том, что числа и команды неотличимы друг от друга, – в памяти и те и другие представляются некоторым набором разрядов, причем по внешнему виду машинного слова нельзя определить, что оно собой представляет – команду или число. Из неразличимости команд и данных вытекает следствие – принцип хранимой программы. Этот принцип очень важен, его суть состоит в том, что программа хранится в памяти вместе с числами. Чтобы понять важность этого принципа, рассмотрим, как программы вообще появляются в памяти машины. Во-первых, команды программы могут наравне с числами вводиться в память «из внешнего мира» с помощью устройства ввода (этот способ был основным в первых компьютерах). Теперь надо помнить, что на вход алгоритма можно в качестве входных данных подавать запись некоторого другого алгоритма (в частности, свою собственную запись). Остается сделать последний шаг в этих рассуждениях и понять, что выходными данными алгоритма тоже может быть запись некоторого другого алгоритма.

Таким образом, одна программа может в качестве результата своей работы поместить в память компьютера другую программу. Именно так и работают компиляторы, переводящие (транслирующие) программы с одного языка на другой. Следствие принципа хранимой программы то, что она может изменяться во время счета самой этой программы. В частности, такая программа может самомодифицироваться, т. е. изменять себя во время своего счета. В настоящее время самомодифицирующиеся программы применяются крайне редко, в то время как на первых ЭВМ использование самомодифицирующихся программ часто было единственным способом реализации некоторых алгоритмов на языке машины.

Заметим также, что когда фон Нейман (с соавторами) писал техническое задание на свою машину, многие из ЭВМ того времени хранили программу в памяти одного вида, а числа – в памяти другого, поэтому этот принцип был в то время революционным. В современных ЭВМ и программы, и данные, как правило, хранятся в одной и той же памяти.

3. *Принцип автоматической работы*. Этот принцип для цифровых вычислительных машин еще называют *принципом программного управления*. Машина, выполняя записанную в ее памяти программу, функционирует автоматически, без участия человека, если только такое участие не предусмотрено в самой программе, например, при вводе данных. Пример устройства, которое может выполнять команды, как и ЭВМ, но не в автоматическом режиме – обычный (непрограммируемый) калькулятор. Последовательность команд для калькулятора задает сам человек. Программа – непустое множество записанных в памяти (не обязательно последовательно) машинных команд, задающих действия, описывающих шаги работы алгоритма. Команды программы обрабатывают хранимые в памяти компьютера данные.

Таким образом, программа – это запись алгоритма на языке машины. Язык машины – набор всех возможных операций, выполняемых командами, и допустимые форматы этих команд. Каждая команда однозначно определяется своим кодом операции (в простейшем случае это просто номер команды). Например, в машинном языке младшей модели 8086 фирмы Intel 135 команд, а машинный язык современных наиболее распространенных компьютеров этой фирмы содержит более трехсот различных кодов операций.

4. *Принцип последовательного выполнения (последовательной работы)*. Устройство управления выполняет некоторую ко-

манду от начала до конца, а затем по определенному правилу выбирает следующую команду для выполнения, затем другую и т. д. При этом каждая либо сама явно указывает на ту, которая будет выполняться за ней (такие команды называются командами перехода), либо следующей будет выполняться команда из ячейки, расположенной в памяти непосредственно вслед за той ячейкой, в которой хранится только что выполненная команда. Этот процесс продолжается, пока не будет выполнена специальная команда останова либо при выполнении очередной команды не возникнет аварийная ситуация (например, деление на ноль). Аварийная ситуация – это аналог безрезультативного останова алгоритма. Например, для машины Тьюринга – это чтение из клетки ленты символа, которого нет в заголовке ни одной колонки таблицы.

Арифметико-логическое устройство

В архитектуре машины фон Неймана арифметико-логическое устройство (АЛУ) может выполнять следующие действия:

1. Считать содержимое некоторой ячейки памяти (машинное слово), т. е. может поместить копию этого машинного слова в некоторую другую ячейку, расположенную в самом АЛУ. Если ячейки памяти расположены не в основной памяти, а в других устройствах ЭВМ, то они называются регистровой памятью или просто регистрами. Таким образом, АЛУ может считать машинное слово из памяти на один из своих регистров.

2. Записать машинное слово в некоторую ячейку памяти – поместить копию содержимого одного из своих регистров в эту ячейку памяти. Если не имеет значения, какая операция (чтение или запись) производится, говорят, что происходит обмен машинным словом между регистром и основной памятью ЭВМ. Таким образом, как уже говорилось, машинное слово – это минимальная

порция информации для обмена между регистрами и основной памятью.

3. АЛУ может также выполнять различные операции над данными в своих регистрах, например сложить содержимое двух регистров, обычно называемых регистрами первого R1 и второго R2 операндов, и поместить результат этой операции на третий регистр, называемый в русскоязычной литературе, как правило, сумматором S. В англоязычной литературе этот регистр называется Accumulator и сокращается до буквы A.

ГЛАВА 2
Структурное и объектное программирование

2.1. Принципы структурного программирования. Библиотеки стандартных процедур и функций

В 60-е гг. прошлого столетия сложилась методика структурного программирования, которая позволила надежно создавать достаточно сложные программные продукты. Эта методика характеризовалась тем, что использовались осмысленные имена переменных, которые локализовались внутри модуля, обмен данными между модулями проходил через глобальные переменные или через переменные, описанные во внешнем модуле, который вызывал текущий. Функции, процедуры и данные организованы по иерархическому принципу. Применялось проектирование по принципу «сверху вниз». Программы стали понятнее, упростился процесс поиска ошибок и отладки.

Используется структурное программирование при создании средних по размеру приложений (несколько тысяч строк исходного кода). Структура программы должна отражать структуру решаемой задачи, чтобы алгоритм решения был ясно виден из исходного текста. Для этого надо иметь средства для создания программы не только с помощью трех простых операторов, но и с помощью средств, более точно отражающих конкретную структуру алгоритма. С этой целью в программирование введено понятие подпрограммы – набора операторов, выполняющих нужное действие и не зависящих от других частей исходного кода. Программа разбивается на множество мелких подпрограмм, каждая из которых выполняет одно из действий, предусмотренных исходным заданием. Комбинируя эти подпрограммы, удается формировать итоговый алгоритм уже не из

простых операторов, а из законченных блоков кода, имеющих определенную смысловую нагрузку, причем обращаться к таким блокам можно по названиям.

Структурное программирование основано на модульной структуре программного продукта и типовых управляющих структурах алгоритмов обработки данных различных программных модулей.

Типы управляющих структур:

– последовательность;
– альтернатива (условие выбора);
– цикл.

Распространены две методики (стратегии) разработки программ, относящиеся к структурному программированию:

– программирование «сверху вниз»;
– программирование «снизу вверх».

Программирование «сверху вниз», или нисходящее программирование, – это методика разработки программ, начинающаяся с определения целей решения проблемы, после чего идет последовательная детализация, заканчивающаяся детальной программой.

Сначала выделяется несколько подпрограмм, решающих самые глобальные задачи (например, инициализация данных, главная часть и завершение), потом каждый из этих модулей детализируется на более низком уровне, разбиваясь на небольшое число других подпрограмм, и так происходит до тех пор, пока вся задача не окажется реализованной.

В данном случае программа конструируется иерархически – сверху вниз: от главной программы к подпрограммам самого нижнего уровня, причем на каждом из них используются только простые последовательности инструкций, циклы и условные разветвления.

Такой подход удобен тем, что позволяет человеку постоянно мыслить на предметном уровне, не опускаясь до конкретных операторов и переменных. Кроме того, есть возможность некоторые подпрограммы не реализовывать сразу, а временно откладывать, пока не будут закончены другие части. Так, если имеется необходимость вычисления сложной математической функции, то выделяется отдельная подпрограмма такого вычисления, но реализуется она временно одним оператором, который просто присваивает заранее выбранное значение. Когда все приложение написано и отлажено, можно приступить к реализации этой функции.

Программирование «снизу вверх», или восходящее программирование, – эта методика начинается с разработки подпрограмм (процедур, функций), пока проработка общей схемы не закончилась.

Такая методика менее предпочтительна по сравнению с нисходящим программированием, так как часто приводит к нежелательным результатам, переделкам и увеличению времени разработки.

Очень важная характеристика подпрограмм – это возможность их повторного использования. С интегрированными системами программирования поставляются большие библиотеки стандартных подпрограмм, которые позволяют значительно повысить производительность труда за счет использования чужой работы по созданию часто применяемых подпрограмм.

Подпрограммы бывают двух видов: процедуры и функции. Отличаются они тем, что процедура просто выполняет группу операторов, а функция вдобавок вычисляет некоторое значение и передает его обратно в главную программу (возвращает значение). Это значение имеет определенный тип.

В языках C, C++, C# понятия процедуры нет. Его заменяет функция, которая всегда формально возвращает значение. Последнее может быть использовано для сообщения программе, вызывающей функцию, о корректности выполняемых вычислений.

Иногда тип возвращаемого значения может быть заменен словом void. Но такая функция просто является той, где поле для возвращаемого значения пустое.

Чтобы работа функции имела смысл, ей надо получить данные из внешней программы, которая эту подпрограмму вызывает. Данные передаются подпрограмме в виде параметров или аргументов, которые обычно описываются в ее заголовке так же, как переменные функции вызываются, как правило, путем простой записи их названия с нужными параметрами и активизируются только в момент их вызова. Операторы, которые находятся внутри функции, выполняются, только если эта функция явно вызвана. Функции могут быть вложенными, – допускается вызов функции не только из главной программы, но и из любых других программ. В некоторых языках программирования допускается вызов подпрограммы из себя самой. Такой прием называется рекурсией и опасен тем, что может привести к зацикливанию – бесконечному самовызову.

Достоинства структурного программирования:

– повышается надежность программ (благодаря хорошему структурированию при проектировании программа легко поддается тестированию и не создает проблем при отладке);

– повышается эффективность программ (структурирование программы позволяет легко находить и корректировать ошибки, а отдельные подпрограммы можно переделывать (модифицировать) независимо от других);

– уменьшаются время и стоимость программной разработки;

– улучшается читаемость программ.

Таким образом, технология структурного программирования при разработке серьезных программных комплексов основана на следующих принципах:

1) программирование должно осуществляться сверху вниз;

2) весь проект должен быть разбит на модули (подпрограммы) с одним входом и одним выходом;

3) подпрограмма должна допускать только три основные структуры – последовательное выполнение, ветвление (if, case) и повторение (for, while, repeat);

4) недопустим оператор передачи управления в любую точку программы (goto);

5) документация должна создаваться одновременно с программированием в виде комментариев к программе.

Структурное программирование эффективно используется для решения различных математических задач, имеющих алгоритмический характер.

2.2. Проблема многократного использования разработанного кода. Объектно-ориентированное программирование

Концепция ООП возникла в середине 80-х гг. XX в. Главная ее идея в том, что программное приложение, как и окружающий нас мир, должно состоять из объектов, обладающих собственными свойствами и поведением.

Объектно-ориентированный подход (ООП) – это целый набор концепций и идей, позволяющих осмыслить задачу, стоящую при разработке компьютерной программы, а затем найти путь к ее решению более понятным, значит, более эффективным способом. До появления объектно-ориентированного подхода во многих языках программирования давно уже возникла потребность в несколько более расширенных типах данных, чем простые целые и вещественные числа, логические «истина-ложь» и текстовые строки. Иначе говоря, при большом количестве атрибутов какой-

либо сущности работать с несколькими массивами весьма затруднительно. В основе объектно-ориентированного подхода в C++ и C# лежит понятие класса.

Классы в C++ – это абстракция, описывающая методы, свойства еще не существующих объектов. Объекты – конкретное представление абстракции, имеющее свои свойства и методы. Созданные объекты на основе одного класса называются экземплярами этого класса. Эти объекты могут иметь различное поведение, свойства, но все равно будут объектами одного класса.

ООП позволяет резко сократить объем и трудоемкость разработки программ, имеющих дело со множеством связанных друг с другом объектов.

В отличие от процедурного программирования, где порядок выполнения операторов программы определяется порядком их следования и командами управления, в ООП порядок выполнения процедур и функций определяется прежде всего событиями.

В объектно-ориентированном программировании существует три основных принципа построения классов:

1. *Инкапсуляция* – свойство, позволяющее объединить в классе и данные, и методы, работающие с ними, и скрыть детали реализации от пользователя.

2. *Наследование* – свойство, позволяющее создать новый класс-потомок на основе уже существующего, при этом все характеристики класса родителя присваиваются классу-потомку.

3. *Полиморфизм* – свойство классов, позволяющее использовать объекты классов с одинаковым интерфейсом без информации о типе и внутренней структуре объекта.

ГЛАВА 3
Алгоритмические языки

3.1. Общая характеристика алгоритмических языков

Алгоритмические языки (называемые также языками программирования) – подобно машинам Тьюринга, Поста и нормальным алгоритмам Маркова – предназначены для представления различных алгоритмов. Однако в рамках этого общего назначения они преследуют существенно иные цели.

Основная же цель, которая ставится при разработке алгоритмического языка, состоит как раз в том, чтобы предложить некое формализованное средство общения между человеком и вычислительной машиной, предназначенное для изложения алгоритмов.

Алгоритмический язык имеет богатые выразительные возможности. Четкое, недвусмысленное определение этого языка представляет собой сложную проблему. Развитым аппаратом для таких целей служат формальные грамматики. Грамматическое описание любого языка (в том числе и языка, на котором мы говорим) включает в себя его алфавит, синтаксис и семантику.

Алфавит языка представляет собой набор основных символов (букв языка), которые могут быть использованы при записи алгоритма.

Синтаксис – это правила построения фраз, позволяющие определить, правильно или неправильно написана та или иная фраза. Другими словами, синтаксис языка представляет собой набор правил, устанавливающих, какие комбинации символов являются текстами на этом языке.

Семантика определяет смысловое значение фраз языка.

Программа как формализованное описание вычислительного процесса

Описать процесс – значит определить последовательность состояний заданной информационной среды. Чтобы по заданному описанию требуемый процесс порождался автоматически на каком-либо компьютере, надо, чтобы это описание было формализованным. Такой подход позволяет дать другое определение компьютерной программы – формализованное описание процесса обработки информации.

Для обозначения объектов алгоритмического языка используется широко распространенное в математике понятие переменной величины. Переменные алгоритмических языков отличаются от переменных, которыми мы оперируем в алгебре. В алгебре переменная – это величина с неизвестным значением. В программировании – ячейка памяти.

Фундаментальным действием в любом алгоритмическом языке является присваивание, которое изменяет значение некоторой переменной.

В алгоритмических языках предусматриваются определенные правила построения композиций операторов, т. е. объединения их в более крупные операторы. Каждое правило построения композиции предписывает свой порядок выполнения операторов, входящих в композицию.

В основе управления алгоритмических языков лежат три типовых операторных конструкции:

– следование: начало S_1, S_2. ... S_k конец;
– выбор: если B, то S_1 иначе S_2;
– повторение (цикл): пока B выполнять S.

Помимо рассмотренных основных конструкций управления в алгоритмических языках обычно предусматривается также возможность прямой передачи управления на любой оператор. С этой целью операторам даются имена (метки).

Перевод текстов, написанных на алгоритмическом языке, на машинные команды

Для выполнения программа должна быть загружена в среду исполнения. Загрузке программы может предшествовать ряд преобразований, цель которых – приведение программы к необходимому виду.

Программа после каждого преобразования размещается на внешнем запоминающем устройстве в виде файлов. Часть программы, которая хранится в одном файле, называется модулем. В основном вся программа хранится в одном файле. Имена файлов, как правило, назначает разработчик, а их расширения назначаются автоматически по правилам, принятым в среде исполнения.

Модуль, содержащий программу в виде, готовом для загрузки в среду исполнения, называется исполняемым модулем.

Различают две основные схемы преобразования исходного модуля в исполняемый модуль: трансляцию и интерпретацию.

Интерпретация. Во время выполнения программы на языке L1 (преобразование языка L1 в L0) каждая из ее команд должна оперативно декодироваться и выполняться программой, написанной на языке L0. Таким образом, при запуске программа на языке L1 начинает выполняться сразу, но каждая ее команда перед выполнением должна быть декодирована.

Трансляция. Перед выполнением программа, написанная на языке L1, должна быть преобразована в программу на языке L0, другой специально созданной для этой цели программой, напи-

санной на языке L0. После этого полученная на языке L0 программа может быть непосредственно выполнена центральным процессором компьютера.

Виртуальные машины

Чтобы не заниматься анализом программ, написанных для конкретного типа процессоров, Таненбаум в 1999 г. предложил создать модель гипотетического компьютера, или виртуальную машину, для каждого из уровней. Следовательно, виртуальная машина VM1 (назовем ее так) может выполнять команды, написанные на языке L1.

Каждая виртуальная машина может быть реализована либо программно, либо аппаратно. Однако независимо от этого программист может писать программы для виртуальной машины VM1 так же, как если бы он это делал для реального компьютера. Кроме того, если реализовать виртуальную машину VM1 на аппаратном уровне, то написанные для VM1 программы можно непосредственно запускать на выполнение, минуя промежуточные этапы трансляции или интерпретации. Кроме того, программы, написанные для виртуальной машины VM1, могут интерпретироваться или транслироваться, а затем выполняться виртуальной машиной VM0.

3.2. Пятиуровневая модель виртуальных машин для современного компьютера

Рассмотрим дополнительные уровни модели виртуальных машин для современного компьютера (рис. 3):

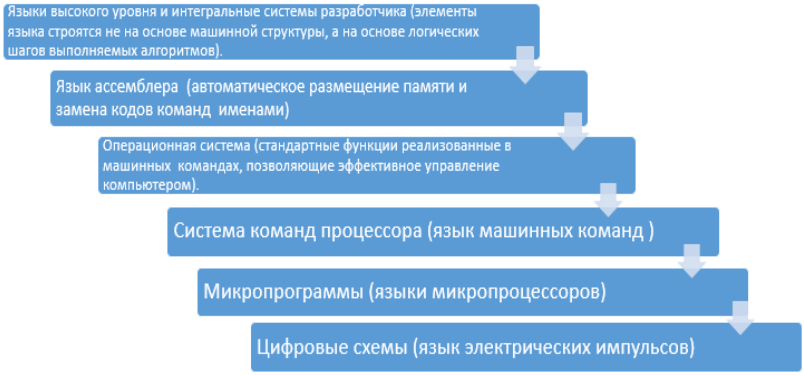


Рис. 3. Иерархия языков программирования (система виртуальных машин по Таненбауму)

Операционная система (уровень 3). Дополнительные уровни виртуальных машин созданы, чтобы более продуктивно использовать время работы программиста. На уровне 3 машина уже может вести работу с пользователем в диалоговом режиме и выполнять его команды, такие как загрузка в память программ и их выполнение, отображение содержимого каталога и т. д. Программа, или виртуальная машина, с помощью которой реализованы эти возможности, называется операционной системой компьютера. Программы, составляющие операционную систему, заранее оттранслированы в машинный код, который выполняет виртуальная машина уровня 2.

Язык ассемблера (уровень 4). Выше уровня операционной системы находятся уровни трансляторов с языков программирования, которые делают возможным разработку крупномасштабных программных проектов. В языке ассемблера, который в иерархической структуре относится к уровню 4, используются короткие мнемоники команд, такие как ADD, SUB и MOV. Благодаря им облегчается процесс трансляции программы в машинный код, соответствующий уровню 2 или системе команд процессора.

Языки высокого уровня (уровень 5). Этому уровню соответствуют языки программирования: C++, C#, Visual, Basic и Java. В программах, написанных на этих языках, обычно используются сложные операторы, которые транслируются сразу в несколько команд языка ассемблера, соответствующих уровню 4. Например, практически во всех отладчиках кода C++ предусмотрено специальное окно, в котором отображаются операторы исходного кода и команды на языке ассемблера, которые получились в результате трансляции. Программы, которые относятся к уровню 5, обычно транслируются компиляторами, соответствующими программам уровня 4. Чаще всего компиляторы содержат встроенный ассемблер, с помощью которого исходный код уровня 4 транслируется непосредственно в машинный код.

3.3. Машинный язык и ассемблер

Электронные микросхемы управляются так называемыми микропрограммами, которые состоят из микрокоманд. Этот набор команд не предоставляется пользователям микросхем для программирования и часто является секретом фирмы. Как правило, микрокоманды вызывают некоторые последовательности электрических импульсов, выполняемых процессами, для которых эта микросхема создана. Например, вывод символа на экран, вывод текста на принтер, подача системного сигнала и многое др.

Центральный процессор собирается из таких микросхем. Однако он имеет свою систему команд, каждая из которых выполняет определенное действие и, в свою очередь, вызывает необходимые последовательности микрокоманд для каждой микросхемы.

Эта система команд документируется и предоставляется программистам. Ее обычно называют системой машинных команд.

Фактически набор машинных команд и представляет собой полный функционал процессора – весь набор действий, функций и возможностей. Из этих команд и состоит исполняемый код программы. Программы, написанные на этом языке, наиболее оптимальны и экономичны при исполнении их процессором.

Машинный код в компьютерной технологии – система команд, которые центральное процессорное устройство компьютера может выполнять сразу, без перевода.

Машинная команда разделяется на группы бит или поля. Поле кода операции (КОП), что должен делать компьютер, а остальные поля, называемые операндами, идентифицируют требуемую команде информацию. Операнд может содержать данное, часть адреса данного, косвенный указатель или другую информацию, относящуюся к обрабатываемым данным. Общий формат машинной команды представлен следующим образом:

Код операции | Операнд | … | Операнд

Наиболее распространенной системой команд является система команд процессора 8086/8088. Хотя сами эти процессоры являются устаревшими и широко не применяются, принципы, заложенные в их основу, наследуются последующими версиями и современными процессорами. Иногда применяется термин архитектура x86-x64. Это расширение архитектуры процессора 8086 на 64 бита с почти полной обратной совместимостью. Практически сами коды команд процессора остаются в прежнем виде, и все изменения касаются распространения этой системы команд на больший объем памяти и более широкие возможности современных процессоров.

3.4. Ассемблеры, компиляторы и дезассемблеры

Компилятор – это программа, которая читает текст на исходном языке и переводит его на другой целевой язык. Поскольку текст на целевом языке пишет человек, то написанный им текст не идеален и содержит ошибки. По мере того как программа натыкается на эти ошибки исходного текста, она выдает пользователю сообщения.

Компиляция состоит из двух частей: анализа и синтеза. Анализ предполагает разбиение программы на смысловые части и формирование некоторого промежуточного логического представления в иерархическом древовидном виде. Синтез – это преобразование содержания промежуточного логического представления в новую программу на целевом языке.

Часто в качестве промежуточного логического представления используется дерево синтаксического разбора. В нем каждый узел – это операция, а дочерние узлы – аргументы (рис. 4).

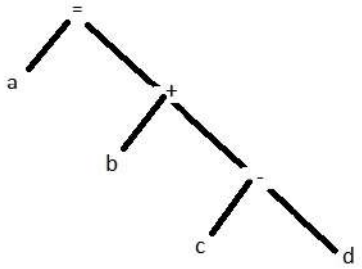


Рис. 4. Дерево синтаксического разбора выражения $a = b + c – d$

Интерпретатор вместо создания целевой программы сразу выполняет выделенные элементы выражения.

При компиляции анализ состоит из трех фаз:

1) линейный (лексический анализ) – поток символов исходной программы считывается и группируется в последовательности символов с определенным совокупным значением – токены (token);

2) иерархический анализ – символы и токены иерархически группируются во вложенные конструкции;

3) семантический анализ – проверка корректности совместного размещения компонентов программы.

Лексический анализ

При лексическом анализе выражение a = b + c – d будет разбито на следующие токены:

Идентификатор –	a
Знак присвоения –	=
Идентификатор –	b
Знак сложения –	+
Идентификатор –	c
Знак вычитания –	–
Идентификатор –	d
Символ разделитель операторов –	;

При этом пробелы, разделяющие символы этих токенов, игнорируются.

Иерархический или синтаксический анализ

Цель этого анализа – объединение токенов исходной программы в синтаксические фразы, которые используются для синтеза выходной. Грамматические фразы исходной программы также представляются в виде дерева. Иногда такое дерево называется деревом синтаксического анализа (рис. 5).

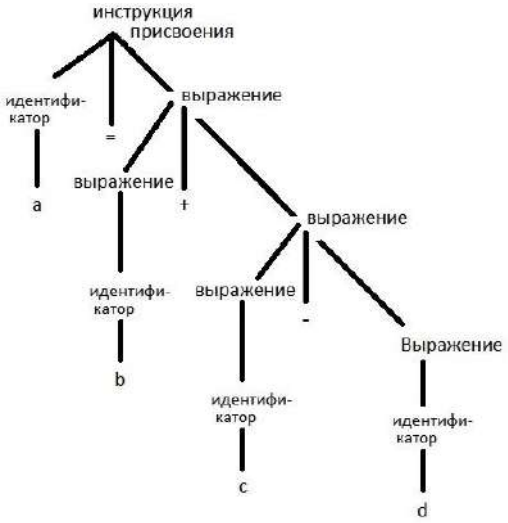


Рис. 5. Дерево синтаксического анализа

При *семантическом анализе* используются иерархические структуры, полученные во время синтаксического анализа для идентификации операторов, операндов, выражений и инструкций. Проверка типов производится, когда компилятор проверяет, имеет ли каждый оператор операнды типа допустимого спецификациями данного языка.

Таблица символов представляет собой структуру данных, содержащую записи о каждом идентификаторе с полями его атрибутов (рис. 6). Атрибуты идентификаторов – это отведенная идентификатору память, его тип, область видимости.

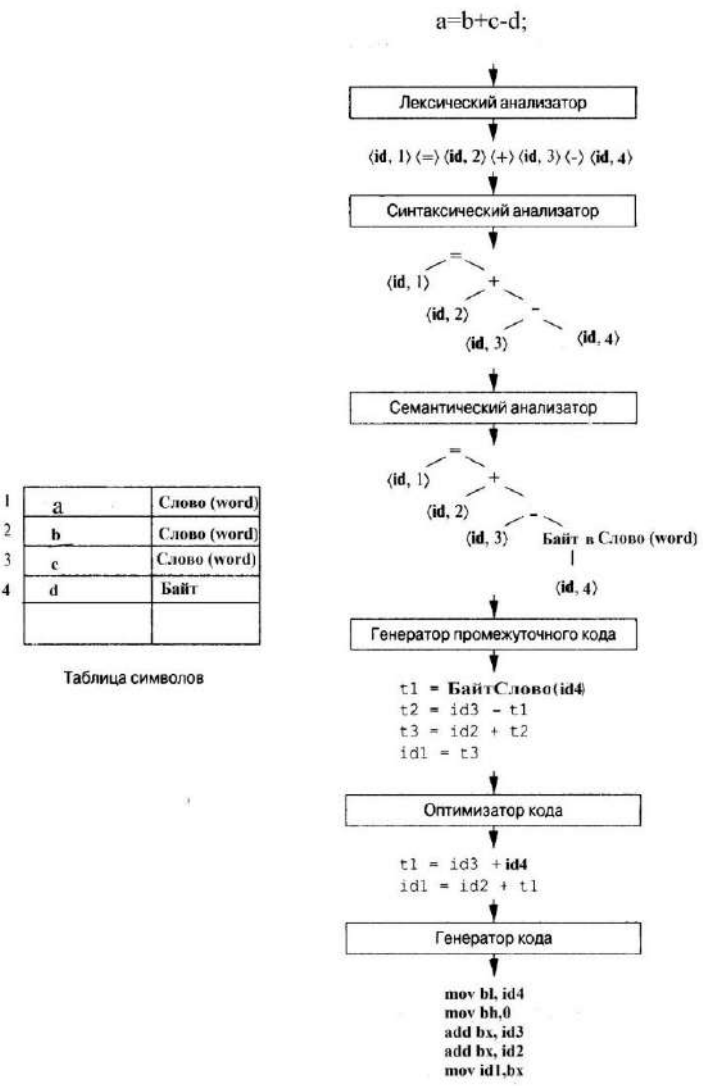


Рис. 6. Схема действия простейшего компилятора

ГЛАВА 4
Коды, команды и форматы, применяемые в программировании

4.1. Системы счисления и коды, применяемые в ЭВМ

Под системой счисления понимают способ представления любого числа с помощью некоторого алфавита символов, называемых цифрами. Существуют различные системы счисления. От их особенностей зависят наглядность представления числа при помощи цифр и сложность выполнения арифметических операций.

В ЭВМ используются только позиционные системы счисления с различными основаниями. Позиционные системы счисления характеризуются тем, что одна и та же цифра имеет различное значение, определяющееся позицией цифры в последовательности цифр, изображающих число.

Любое неотрицательное число в позиционной системе счисления (СС) может быть представлено в следующем виде:

$$Д = R_{n-1} * Осн^{n-1} + R_{n-2} * Осн^{n-2} + \ldots + R_1 * Осн^1 + \\ + R_0 * Осн^0 + R_1 * Осн^1 + R_2 * Осн^2 + \ldots$$

где *Д* – десятичный эквивалент числа;

R_i – значение *i*-го разряда;

Осн – основание системы счисления;

Осн в степени *i* – вес *i*-го разряда;

n – число разрядов целой части числа.

В вычислительном устройстве аппаратно реализуется лишь одна форма представления и хранения числа в памяти, позволяющая записывать в них лишь последовательность нулей и единиц –

целые двоичные числа без знака определенной длины. Для размещения в памяти всего многообразия числовых данных используется кодировка.

В программировании современных компьютерных систем применяются следующие коды системы счисления: двоичная (BIN), десятичная (DEC), шестнадцатеричная (HEX) и непозиционная двоично-десятичная (BCD) системы счисления.

Двоичная арифметика

Правила выполнения арифметических действий над двоичными числами определяются арифметическими действиями над одноразрядными двоичными числами.

Сложение	Вычитание	Умножение
0 + 0 = 0	0 – 0 = 0	0 * 0 = 0
0 + 1 = 1	1 – 0 = 1	0 * 1 = 0
1 + 0 = 1	1 – 1 = 0	1 * 0 = 0
1 + 1 = 1	10 – 1 = 1	1 * 1 = 1

Правила выполнения арифметических действий во всех позиционных системах счисления аналогичны.

Как и в десятичной системе счисления, сложение двоичных чисел начинается с правых (младших) разрядов. Если результат сложения цифр младшего значащего разряда обоих слагаемых не помещается в этом же разряде результата, то происходит перенос. Цифра, переносимая в соседний разряд слева, добавляется к его содержимому. Такая операция выполняется над всеми разрядами слагаемых от младшего значащего разряда до старшего значащего разряда.

Операция вычитания двоичных чисел аналогична операции в десятичной системе счисления. Операция вычитания начинается, как и сложение, с младшего значащего разряда. Если содержимое

разряда уменьшаемого меньше содержимого одноименного разряда вычитаемого, то происходит заем 1 из соседнего старшего разряда. Операция повторяется над всеми разрядами операндов от младшего значащего разряда до старшего значащего разряда.

Пример. Вычесть два числа в десятичном и двоичном представлении (формат – 1 байт).

Заем (единица)	1	01100000
Уменьшаемое	$109_{(10)}$	$01101101_{(2)}$
Вычитаемое	$049_{(10)}$	$00110001_{(2)}$
Разность	$060_{(10)}$	$00111100_{(2)}$

Как и в десятичной системе счисления, операция перемножения двоичных многоразрядных чисел производится путем образования частичных произведений и последующего их суммирования. Частичные произведения формируются в результате умножения множимого на каждый разряд множителя, начиная с младшего значащего разряда. Каждое частичное произведение смещено относительно предыдущего на один разряд. Поскольку умножение идет в двоичной системе счисления, каждое частичное произведение равно либо 0 (если в соответствующем разряде множителя стоит 0), либо является копией множимого, смещенного на соответствующее число разрядов влево (если в разряде множителя стоит 1). Поэтому умножение двоичных чисел идет путем сдвига и сложения. Таким образом, количество частичных произведений определяется количеством единиц в множителе, а их сдвиг – положением единиц (младший значащий разряд частичного произведения совпадает с положением соответствующей единицы в множителе). Положение точки в дробном числе определяется так же, как и при умножении десятичных чисел.

Теперь рассмотрим машинный вариант операции перемножения. Общий алгоритм перемножения имеет вид:

$$Z = X * Y = \text{sign}(Z) * |X| * |Y|,$$

$$\text{sign}(Z) = \begin{cases} +, \text{sign}(X) = \text{sign}(Y) \\ -, \text{sign}(X) \neq \text{sign}(Y) \end{cases},$$

где sign(X) – функция определения знака числа.

Как отмечалось выше, операция перемножения состоит в формировании суммы частичных произведений, которые суммируются с соответствующими сдвигами относительно друг друга. Этот процесс суммирования можно начинать либо с младшего, либо со старшего частичного произведения. В ЭВМ процессу суммирования придают последовательный характер, т. е. формируют одно частичное произведение, к нему с соответствующим сдвигом прибавляют следующее и т. д. (т. е. не формируют все частичные произведения, а потом их складывают). В зависимости от того, с какого частичного произведения начинается суммирование (старшего или младшего), сдвиг текущей суммы осуществляется влево или вправо. При умножении целых чисел для фиксации результата в разрядной сетке число разрядов должно равняться сумме числа разрядов в *x* и *y*.

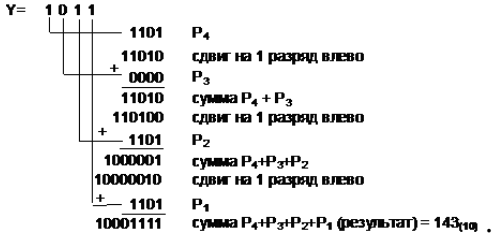
Рассмотрим на примере два машинных варианта выполнения умножения целых чисел: начиная со старшего частичного произведения («старшими разрядами вперед») и начиная с младшего частичного произведения («младшими разрядами вперед»).

Пример. Найти произведение двух чисел:

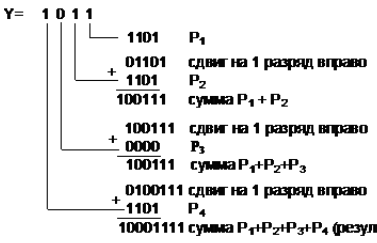
$X * Y = 1101_{(2)} * 1011_{(2)} = 13_{(10)} * 11_{(10)} = 143_{(10)}$.

Обозначим P_i – *i*-е частичное произведение.

1. Умножение старшими разрядами вперед:

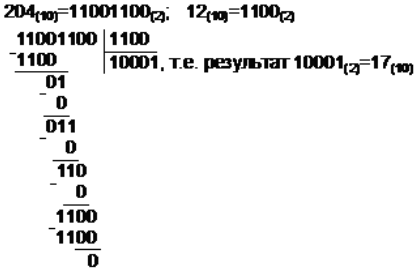


2. Умножение младшими разрядами вперед:



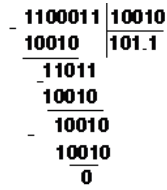
Деление – операция, обратная умножению, поэтому при делении двоичных чисел, как и в десятичной системе счисления, операция вычитания повторяется до тех пор, пока уменьшаемое не станет меньше вычитаемого. Число этих повторений показывает, сколько раз вычитаемое укладывается в уменьшаемом.

Пример. Вычислить $204_{(10)}$ /$12_{(10)}$ в двоичном коде:



.

Таким образом, процедура деления не так проста для машинной реализации, поскольку постоянно приходится выяснять, сколько раз делитель укладывается в определенном числе. В общем случае частное от деления получается дробным, причем выбор положения точки совершенно аналогичен тому, как это делается при операциях с десятичными числами:



.

Машинное представление информации

Центральный процессор и входящие в него микропроцессоры обрабатывают данные в виде двоичных наборов (рис. 7). Минимальный числовой элемент составляет 1 бит, тетрада – 4 бита, байт (byte) – 8 бит, слово (word) – 16 бит или длинное (double word) – 32 бит.

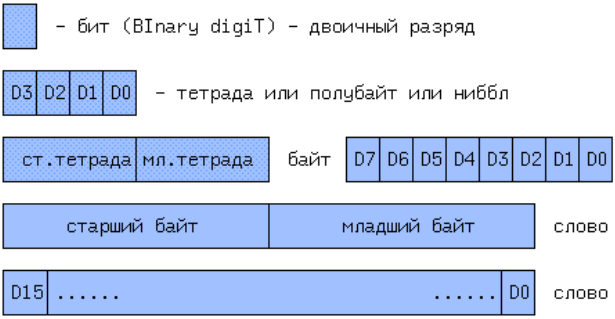
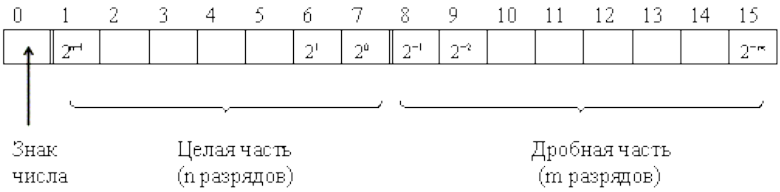


Рис. 7. Машинное представление информации

Числа с фиксированной точкой

Представление числа с фиксированной точкой активно использовалось на ранних этапах развития вычислительной техники, однако применяется и в настоящее время в отдельных частных случаях (форматы представления денежных сумм). Запись числа с фиксированной точкой обычно имеет знаковый и цифровой разряды. Запятая в разрядной сетке может быть зафиксирована в принципе после любого разряда.

Пример. Ячейка с целой и дробной частью:



Как частный случай числа с фиксированной точкой может быть рассмотрена запись целого числа (в этом случае все разряды, кроме знакового, используются для записи целой части).

Пример. Ячейка с записью целого числа:

К достоинствам использования чисел с фиксированной точкой относятся простота выполнения арифметических операций и высокая точность изображения чисел. К недостаткам – небольшой диапазон представления чисел.

Представление целых чисел со знаком

Вариантов представления целых чисел со знаком много. Выбор того или иного способа определяется скоростью выполнения основных арифметических операций. В предыдущем примере рассматривается представление чисел в прямом коде. Самый левый, самый старший бит отводится под знак (0 – положительное число, 1 – отрицательное), остальные биты под число. Например, для корректного сложения чисел надо выделить знак, принять решение, какое число из какого вычитать или к какому прибавить, записать эти числа уже без знаков, провести операцию сложения (или вычитания) в соответствии с правилами арифметических операций для обычных двоичных чисел, определить результат, определить знак результата, записать результат и знак в ячейку для результата. Наличие такого набора операций заметно замедляет вычисления. При этом если оба числа положительные, то бóльшую часть этих операций можно опустить, оставив практи-

чески только двоичное вычитание. Кроме того, в данном представлении существуют два нуля – положительный и отрицательный. Это усложняет логические операции сравнения.

Необходим код, позволяющий корректно складывать и вычитать целые положительные и отрицательные числа без дополнительных операций и проблем. Можно различать положительные и отрицательные числа путем инверсии бит.

Пусть 00001001 = 9, а 11110110 = –9.

Тогда (00001001 = 9) + (11110110 = –9) = 11111111 (рис. 8).

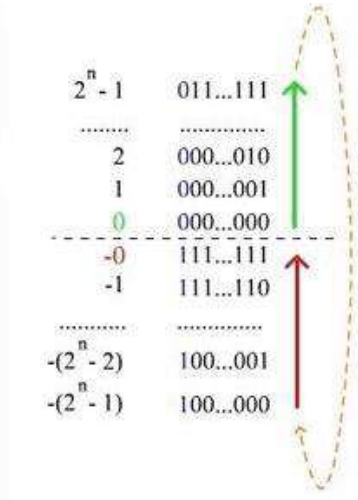


Рис. 8. Представление чисел в инверсном коде

Но +0 = 00000000, –0 = 11111111, таким образом, сумма равна 0. Сложение дает правильный результат, но остается проблема двух нулей. Для ее исключения достаточно сместить все отрицательные значения вверх (влево) по числовой оси на 1. Для этого нужно к отрицательному значению прибавить 1. Если число соответствует отрицательному нулю, то после этого сложения младший байт будет содержать положительный ноль, а в старший будет перенесена единица. Если мы не пользуемся этим байтом, то мы не

будем замечать никаких некорректностей. Однако в схемах сложения чисел, состоящих из нескольких байт или слов, это может исказить результат. Как следствие, обычное сложение двоичных чисел заменяется сложением без переноса единицы в старший байт или слово, т. е. допускает переполнение ячейки. Такой код с прибавлением единицы с переполнением назван дополнительным.

В результате:

00001001 = 9.

11110111 = –9.

1 00000000 – самая левая единица просто исчезает в результате переполнения.

Проблема двух нулей при сложении исчезает.

Алгоритм представления целого числа со знаком выглядит так:

1. Если самый левый бит равен 0, то число положительное и просто переносится в ячейку.

2. Если самый левый бит равен 1, то число отрицательное и к нему применяется операция инверсии (кроме старшего бита) и прибавляется 1, но при этом отключается перенос разрядов в старший бит или слово.

Представления вещественных чисел в виде чисел с плавающей точкой. Представление числа в экспоненциальном виде

Экспонента – число, обозначающее степень, которое пишется в виде верхнего индекса справа от цифры или символа. Например, в выражении a^4 = (a * a * a * a) экспонентой является 4, а число *a* – основанием. Наиболее часто экспонентой называют экспоненту по основанию *e* (*e* = 2,718…), а также функцию $f(x) = e^x = exp_e(x) = exp(x)$. Однако в общем случае экспонентой называют функцию и с другим основанием – $f(x) = a^x = exp_a(x)$. В частности, экспонента по основанию, равному основанию системы

счисления, может быть использована для обозначения расположения запятой или точки при представлении вещественного числа.

Например, скорость света:

$$299\ 792\ 458 \text{ м/с} = 2{,}99792458 * 10^8 \text{ м/с}.$$

При этом число 2,99792458 является мантиссой, а $exp_{10}{}^{+8}$ или 10^8 – экспоненциальной частью или характеристикой. Мантисса по абсолютному значению не превышает основания системы счисления – 10. Такое представление получило название нормализованного. Денормализованным представлением вещественного числа считается число, мантисса которого приведена по модулю к интервалу [0; 1]. Тогда скорость света представляется в виде:

$$299\ 792\ 458 \text{ м/с} = 0{,}299792458 * 10^9 \text{ м/с} = 2{,}99792458 * exp_{10}{}^{+9}.$$

Экспоненциальное представление возможно и для основания 2. В этом случае оно интересно, поскольку может быть использовано для представления вещественных чисел в памяти ЭВМ. Вводимые в ЭВМ числа набираются в произвольном и удобном для пользователя буквенно-цифровом коде, а затем переводятся в формат, который хранит двоичное представление.

Рассмотрим пример перевода числа в десятичном экспоненциальном представлении в двоичное нормализованное. Приведем пример, который содержится в описании формата IEEE 754. Пусть задано десятичное число 155,625. Преобразуем его в двоичное число с плавающей точкой в экспоненциальном нормализованном виде:

$$155{,}625 = 1 * 2^7 + 0 * 2^6 + 0 * 2^5 + 1 * 2^4 + 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + \\ + 1 * 2^0 + 1 * 2^{-1} + 0 * 2^{-2} + 1 * 2^{-3}$$

или:

$$155{,}625 = 128 + 0 + 0 + 16 + 8 + 0 + 2 + 1 + 0{,}5 + 0 + 0{,}125.$$

В итоге:

$155{,}625_{10} = 10011011{,}101_2$ – число в десятичной и в двоичной системе с плавающей точкой.

Приведем полученное число к нормализованному виду в десятичной и двоичной системе:

$1{,}55625 \cdot exp_{10}^{+2} = 1{,}0011011101 * exp_2^{+111}$.

В результате мы получили основные составляющие экспоненциального нормализованного двоичного числа:

Мантиссу M = 1,0011011101 Экспоненту $exp_2 = +111$.

Формальное представление чисел в стандарте IEEE 754 для любого формата точности

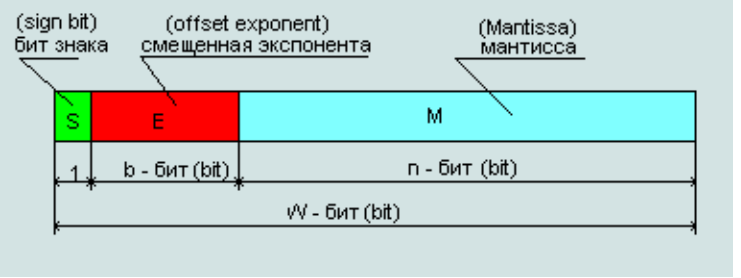


Рис. 9. Представление числа в формате IEEE 754

На рис. 9 показано представление числа в формате IEEE 754, где: S – бит знака, если S = 0 – положительное число;

S = 1 – отрицательное число;

E – смещенная экспонента двоичного числа, такая, что:

$exp_2 = E - (2^{(b-1)} - 1)$, exp_2 – экспонента двоичного нормализованного числа с плавающей точкой;

$(2^{(b-1)} - 1)$ – заданное смещение экспоненты (в 32-битном ieee 754 (b = 8) оно равно +127, а в 64-битном (b = 11) – 1023).

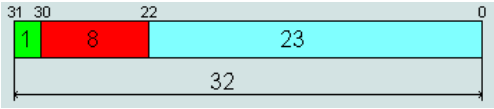
M – остаток мантиссы двоичного нормализованного числа с плавающей точкой. У нормализованной двоичной мантиссы первый бит всегда равен 1, так как число лежит в диапазоне $1 \le M < 2$.

Нет смысла записывать единицу в отведенные для мантиссы биты. Туда записывают остаток от мантиссы – без первого бита.

Формула вычисления десятичных чисел с плавающей точкой, из чисел, представленных в стандарте IEEE 754:

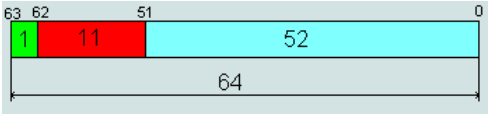
$$F = (-1)^S 2^{\left(E - 2^{(b-1)} + 1\right)} \left(1 + \frac{M}{2^n}\right).$$

Основное применение в технике и программирование получили форматы 32 и 64 бита (рис. 10, 11). Например, в VB используют типы данных single (32 бита) и double (64 бита). В языках C и C++ аналогично используют float (32 бита) и double (64 бит).



$$F = (-1)^S 2^{(E-127)} \left(1 + \frac{M}{2^{23}}\right).$$

Рис. 10. Формат числа одинарной точности 32 бита



$$F = (-1)^S 2^{(E-1023)} \left(1 + \frac{M}{2^{52}}\right).$$

Рис. 11. Формат числа двойной точности 64 бита

Рассмотрим преобразование двоичного числа 10011011,101 в формат single-precision (32 бита) стандарта IEEE 754.

Остальные форматы представления чисел в IEEE 754 являются увеличенной копией single-precision.

Чтобы представить число в формате single-precision IEEE 754, нужно привести его к двоичному нормализованному виду. Мы уже проделали это преобразование ранее над числом 155,625. Теперь рассмотрим, как двоичное нормализованное число преобразуется к 32-битному формату IEEE 754.

Преобразование в 32-битный формат IEEE 754 представлено ниже.

Число может быть положительным или отрицательным. Поэтому отводится 1 бит для обозначения знака числа (0 – положительное или 1 – отрицательное). Это самый старший бит в 32-битной последовательности.

Далее пойдут биты экспоненты, для этого выделяют 1 байт (8 бит).

Экспонента может также быть положительным или отрицательным числом. Для определения знака экспоненты, чтобы не вводить еще один бит знака, добавляют смещение к экспоненте в половину байта +127 (0111 1111). Иными словами, если наша экспонента равна +7 (+111 в двоичной), то смещенная экспонента равна 7 + 127 = 134. А если бы наша экспонента была – 7, то смещенная экспонента была бы равна 127 – 7 = 120. Смещенную экспоненту записывают в отведенные 8 бит. При этом, когда будет нужно получить экспоненту двоичного числа, мы просто отнимем 127 от этого байта.

Оставшиеся 23 бита отводят для мантиссы. Как уже указывалось, значение первого бита мантиссы всегда равно 1 и оно не записывается, а в отведенные 23 бита записывают остаток от мантиссы.

В табл. 2 представлено десятичное число 155,625 в 32-битном формате IEEE 754.

Таблица 2

Десятичное число в 32-битном формате

1 бит знак числа	8 бит смещенный порядок	23 бит остаток от мантиссы	Число, записанное в формате IEEE 754, представленное в шестнадцатеричном коде
0	1000 0110	001 1011 1010 0000 0000 0000	431BA000 (hex)
0(dec)	134(dec)	1810432(dec)	

Преобразования числа формата 32 бит IEEE 754 в десятичное число

Чтобы записать число в стандарте IEEE 754 или восстановить его, необходимо знать три параметра:

1. S – бит знака (31-й бит).
2. E – смещенная экспонента (30–23 биты).
3. M – остаток от мантиссы (22–0 биты).

Это целые числа, записанные в числе IEEE 754 в двоичном виде.

Приведем формулу для получения десятичного числа из числа IEEE754 одинарной точности:

$$F = (-1)^S 2^{(E-127)} \left(1 + \frac{M}{2^{23}}\right),$$

где *F* – десятичное число.

Проверяем наш пример:

$F = (-1)^0 * 2^{(134 - 127)} * (1 + 1810432 / 2^{23}) = 2^7 * (1 + 0{,}2158203125) = 128 * 1{,}2158203125 = 155{,}625.$

Двоично-десятичный код

В цифровых устройствах, где основная часть операций связана не с обработкой и хранением информации, а с самим ее вводом и выводом на какие-либо устройства отображения с десятичным представлением полученных результатов, имеет смысл проводить вычисления в десятичной системе счисления. Но ЭВМ требует информацию только в двоичной форме. Следовательно, десятичные цифры нужно кодировать каким-либо легко реализуемым и быстрым способом. Для этих целей используется двоично-десятичный код, в котором каждая десятичная цифра 0 ... 9 изображается соответствующим 4-разрядным числом (от 0000 до 1001). Такой код называется еще кодом 8421 (цифры, соответствующие весам двоичных разрядов).

Двоично-десятичный код (ДДК), или Binary Coded Decimal (BCD), может быть *упакованным*, когда в одном байте хранятся две десятичные цифры, либо *неупакованным* – по одной цифре в байте. Упакованное число 1996 представляется в виде двух байтов: 0001 1001 и 1001 0110. Для знака числа отводится дополнительный байт. Например в формате (ДД) девять байтов отводится для размещения 18 цифр, а в старшем бите десятого байта находится знак числа.

Суммирование двоично-десятичных чисел можно производить по правилам обычной двоичной арифметики, а затем производить *двоично-десятичную коррекцию*, которая заключается в проверке каждой тетрады на допустимые коды. Если в какой-либо тетраде обнаруживается запрещенная комбинация, то это говорит о переполнении. В этом случае необходимо произвести двоично-десятичную коррекцию. Двоично-десятичная коррекция заключается в дополнительном суммировании числа шесть (число запрещенных комбинаций) с тетрадой, в которой произошло переполнение или перенос в старшую тетраду.

Примеры:

$$\begin{array}{r} 0001\ 1000 \\ +\ 0001\ 0011 \\ \hline 0010\ 1011 \end{array} \Rightarrow \begin{array}{r} 0010\ 1011 \\ +\ 0000\ 0110 \\ \hline 0011\ 0001 \end{array}$$

$$\begin{array}{r} 0001\ 1001 \\ +\ 0001\ 1001 \\ \hline 0011\ 0010 \end{array} \Rightarrow \begin{array}{r} 0011\ 0010 \\ +\ 0000\ 0110 \\ \hline 0011\ 1000 \end{array}.$$

Для записи одного десятичного разряда используется четыре двоичных бита. Эти четыре бита называются тетрадой. Иногда встречается название, пришедшее из англоязычной литературы, – нибл. При помощи четырех бит можно закодировать шестнадцать цифр. Лишние комбинации в двоично-десятичном коде являются запрещенными.

Запишем пример двоично-десятичного кода:

1258 = 0001 0010 0101 1000.

При записи десятичных чисел часто требуется записывать знак числа и десятичную запятую (в англоязычных странах точку). Двоично-десятичный код часто применяется для набора телефонного номера или набора кодов телефонных служб. В этом случае кроме десятичных цифр часто применяются символы «*» или «#». Для записи этих символов в двоично-десятичном коде применяются запрещенные комбинации.

Достаточно часто в памяти процессора для хранения одной десятичной цифры выделяется одна ячейка памяти (восьми-, шестнадцати- или тридцатидвухразрядная). Это делается для повышения скорости работы программы. Для того чтобы отличить такой способ записи двоично-десятичного числа от стандартного, способ записи десятичного числа, как это показано в примере, называется упакованной формой двоично-десятичного числа.

Буквенно-цифровой код

По своей природе компьютеры могут работать лишь с числами. И для того, чтобы они могли хранить в памяти и обрабатывать буквы или другие символы, каждому из них должно быть поставлено в соответствие некоторое число, т. е. использована та или иная система кодировки символов. Буквы, цифры, математические символы, знаки препинания, символы для рисования линий, управляющие символы и некоторые другие кодируются однобайтовыми числами. Существует несколько разновидностей таких кодов, например, ASCII, КОИ-7, КОИ-8, альтернативный код ГОСТ, основной код ГОСТ и др.

Система представления символов в персональных компьютерах базируется на Американском стандартном коде для обмена информацией (American Standard Code for Information Interchange), который был введен в 1963 г. и ставил в соответствие каждому символу семиразрядный двоичный код, обеспечивающий представление 128 символов. ASCII-код включал две группы символов:

1) управляющие символы, используемые в коммуникационных протоколах для передачи команд периферийным устройствам;

2) символы пишущей машинки – цифры, буквы и специальные знаки.

Управляющие символы имеют коды с номерами от 0 до lAh. К управляющим относится также символ с кодом 7Fh. Каждый управляющий символ выполняет строго определенную функцию. Все остальные символы относятся к алфавитно-цифровой группе (группе символов пишущей машинки).

ASCII и 7-битовый код для обмена информацией (КОИ-7) отображают первые 128 символов и входят в состав остальных кодировок. Дополнительные символы и русский алфавит входят в

восьмибитовые расширенные коды (КОИ-8, альтернативный и основной). Общее число символов в этих кодах равно 256. Таблица некоторых кодов приведена ниже. Следует отметить, что нулевой код (NULL) не кодирует цифру ноль и вообще никак не отображается:

0 – 0011 0000	A – 0100 0001 – 41_{16}
1 – 0011 0001	B – 0100 0010 – 42_{16}
2 – 0011 0010	…
…	Z – 0101 1010 – $5A_{16}$
9 – 0011 1001	+ – 0010 1011 – $2B_{16}$
	= – 0011 1101 – $3D_{16}$

Низкоуровневая работа с клавиатурой

Клавиатура является основным средством ввода текстовой информации в компьютер, поэтому программист в первую очередь должен изучить способы кодирования текстовых символов и особенности использования функций операционной системы. Если у программиста возникает потребность в работе на уровне аппаратного обеспечения, он должен также разобраться с особенностями программирования контроллера клавиатуры и контроллера прерываний.

Кроме ASCII-кодов для идентификации клавиш используются также скан-коды. Скан-коды в старых клавиатурах (появившихся до использования микроконтроллеров) являлись порядковыми номерами клавиш: нумерация велась сверху вниз, справа налево. С целью сохранения совместимости со старым программным обеспечением микропроцессоры современных клавиатур преобразуют действительные порядковые номера клавиш в номера, соответствующие латинской раскладке на клавиатуре IBM XT с учетом дополнительных клавиш клавиатуры IBM AT. В резуль-

тате распределение номеров перестало быть строго упорядоченным. Кроме того, функции BIOS выполняют также перекодировку скан-кодов с целью упрощения анализа этих кодов в прикладных программах. Поэтому существует несколько различных видов таблиц скан-кодов:

– собственная внутренняя таблица встроенного микроконтроллера клавиатуры;

– таблица для обмена кодами между контроллером клавиатуры и специализированным клавиатурным микропроцессором системной платы;

– таблица кодов, которые клавиатурный микропроцессор передает подпрограммам BIOS;

– таблица кодов, которые BIOS передает прикладным программам. При работе с функциями BIOS интерес представляет последняя из этих таблиц.

Шестнадцатеричная система счисления

Эта система счисления имеет основание Осн = 16.

Шестнадцатеричная система счисления позволяет еще короче записывать многоразрядные двоичные числа и, кроме того, сокращать запись 4-разрядного двоичного числа, т. е. полубайта, поскольку $16 = 2^4$. Шестнадцатеричная система также применяется в текстах программ для более краткой и удобной записи двоичных чисел.

Для перевода числа из двоичной системы счисления в шестнадцатеричную надо разбить это число влево и вправо от точки на тетрады и представить каждую тетраду цифрой в шестнадцатеричной системе счисления.

Пример. Двоичное число $10101011111101_{(2)}$ записать в шестнадцатеричной системе:

0010	1010	1111	$1101_2 = 2AED_{16}$
2	A	E	D

Пример. Двоичное число $11101,01111_{(2)}$ записать в шестнадцатеричной системе:

0001	1101	0111	$1000_2 = 1D78_{16}$
1	D	7	8

Для перевода числа из шестнадцатеричной системы счисления в двоичную необходимо, наоборот, каждую цифру этого числа заменить тетрадой.

В заключение следует отметить, что перевод из одной системы счисления в другую произвольных чисел можно осуществлять по общим правилам. Однако на практике переводы чисел из десятичной системы в рассмотренные системы счисления и обратно осуществляются через двоичную систему счисления.

4.2. Формат машинных команд. Команды языка ассемблера

Программы, выполняемые процессором, находятся в оперативной памяти компьютера. Процессор забирает из памяти очередную команду, выполняет ее, переходит к следующей команде, снова выполняет ее и так до конца программы. Команды процессора могут не только менять содержимое регистров, но и записывать числа в память компьютера, состоящую из отдельных, идущих друг за другом байтов. Все байты компьютерной памяти пронумерованы. Самому первому присвоен нулевой номер. Номер последнего байта определяется объемом оперативной памяти, которой располагает компьютер. Номер байта обычно называют адресом.

Память компьютера делится на основную, оперативную и вторичную оперативную. Оперативная и вторичная оперативная память является энергозависимой, а основная – энергонезависимой. В энергонезависимой части памяти находится BIOS – базовая система ввода/вывода.

Существует виртуальная память, которая находится на внешних устройствах. При этом интерфейс доступа к ней такой же, как и к оперативной памяти.

Процессор Intel поддерживает доступ к виртуальной памяти на аппаратном уровне.

Схема управления памятью, которая поддерживает представление программиста об организации хранения программ и данных, называется сегментацией. Сегмент – область памяти, внутри которой поддерживается линейная адресация.

Первоначально сегменты появились в связи с необходимостью обобществления процессами фрагментов программного кода, это позволило исключить дублирующую информацию. Память перестала быть линейной и превратилась в двумерную. Адрес состоит из двух компонентов: номера сегмента и смещения внутри сегмента.

Таким образом, логические и физические адресные пространства ни по организации, ни по размеру не соответствуют друг другу. Максимальный размер логического пространства определяется размерностью процессора.

Логическое пространство значительно превышает размер физического пространства.

Процессор и операционная система осуществляют процесс отображения логических адресов в физические. Этот процесс называется связыванием адресов.

В архитектуре x86 в обычном незащищенном режиме процессора общий объем оперативной памяти разбивается на одинаковые сегменты по 64 Кб (65536 байт)

Регистры процессора и их краткое описание

Регистры процессора – внутренние ячейки памяти процессора, которые служат для хранения информации с практически мгновенным доступом. В отличие от оперативной памяти для чтения и записи в регистры не нужно обращаться к внешнему устройству через шину, потому что регистры встроены в процессор и являются одной из его основных частей.

Существуют три типа регистров: регистры общего назначения, сегментные и служебные регистры.

Регистры общего назначения. Регистр EAX – универсальное хранилище. Обычно используется как буферная память для вычислений, передачи параметров и возврата результата выполнения подпрограммы (функции). Часто используется при системных вызовах операционных систем.

Регистр ECX используется для счетчиков. Команды циклов процессора основаны именно на этом регистре. Эти команды автоматически меняют значение этого регистра.

Регистр EBX применяется для указания адреса памяти. Его еще называют регистром базы. Часто используется в командах доступа к оперативной памяти и в паре со смещением.

Регистр EDX похож на регистр процессора EAX. Применяется для передачи данных, часто используется при системных вызовах операционных систем для передачи параметров.

Регистр ESP – указатель стека. Команды работы со стеком автоматически управляют значением этого регистра. Регистр EBP применяется для прямой адресации в стеке, например для доступа к локальным (автоматическим) переменным.

Регистр ESI используется в командах обработки набора байт. Перед использованием этих команд в регистр процессора ESI записывается адрес источника.

Регистр EDI применяется в командах обработки набора байт. Перед использованием этих команд в регистр ESI записывается адрес назначения.

Сегментные регистры. Регистр CS – указатель на сегмент кода. В защищенном режиме в этот регистр записывается селектор сегмента кода.

Регистр DS – указатель на сегмент данных. В защищенном режиме процессора в этот регистр записывается селектор сегмента данных. Именно этот регистр используется для расчета реального адреса в оперативной памяти по умолчанию.

Регистр SS – указатель на сегмент стека. В защищенном режиме в этот регистр записывается селектор сегмента стека.

Регистр ES – дополнительный сегментный регистр данных процессора. Часто используется для доступа к статическим данным программы, доступным только для чтения.

Регистр FS – дополнительный сегментный регистр данных. Впервые появился в процессоре Intel 80386.

Регистр GS – дополнительный сегментный регистр данных. Впервые появился в процессоре Intel 80386.

Служебные регистры. Регистр флагов содержит битовые поля с флагами состояния. Отражает текущее состояние процессора. Большинство команд меняют биты этого регистра.

Регистр EIP – указатель на исполняемую инструкцию. Обычная запись значения в этот регистр невозможна. Он изменяется при каждом выполнении команды. Команды перехода записывают значение в этот регистр, и процессор автоматически переходит на выполнение команд по нужному адресу.

Регистр CR0 служит для чтения и изменения режима работы микропроцессора. С помощью этого регистра можно перевести процессор в защищенный режим, изменив только 1 бит.

Регистр CR1 зарезервирован и недоступен программисту.

В регистр CR2 помещается адрес страницы, которая не найдена в памяти при страничной адресации. Служит для организации виртуальной памяти и файла подкачки.

Регистр CR3 хранит в себе физический адрес каталога страниц при страничной адресации памяти процессора. С помощью этого регистра и каталога страниц процессор определяет, какому логическому линейному адресу соответствует та или иная страница физической памяти компьютера.

Код регистра зашифрован в трех битах, которые могут входить как в состав кода операции машинного кода, так и в специальный байт MRM.

32-разрядные регистры могут хранить числа от 0 до 4294 967 295 (0FFFFFFFFh), 16-разрядные могут хранить числа от 0 до 65 535 (0FFFFh), а в 8-разрядные регистры можно загружать максимальное число 255 (0FFh).

При этом 16-разрядные и 8-разрядные регистры размещаются внутри регистров большего размера согласно приведенной ниже схеме (рис. 12).

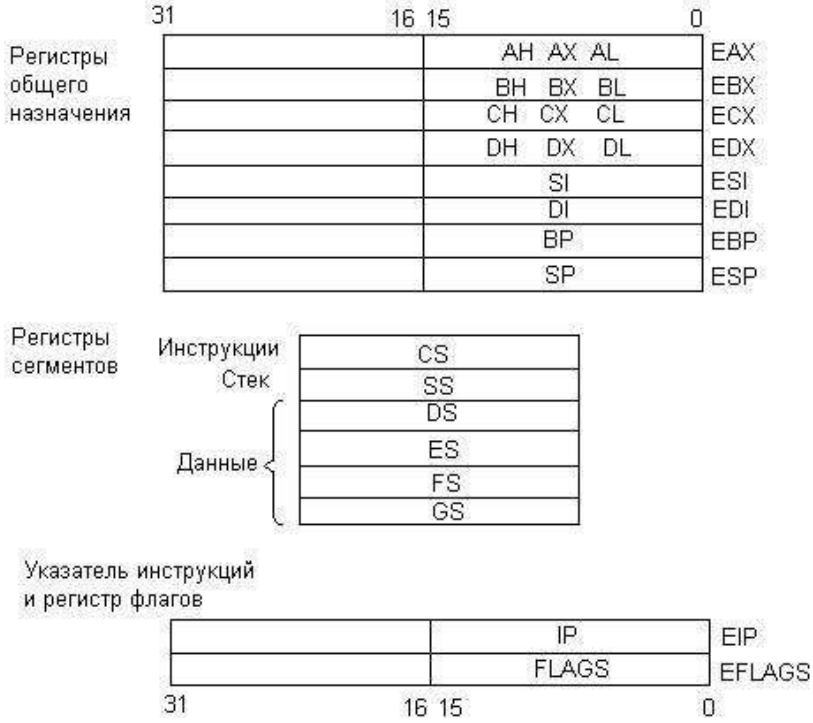
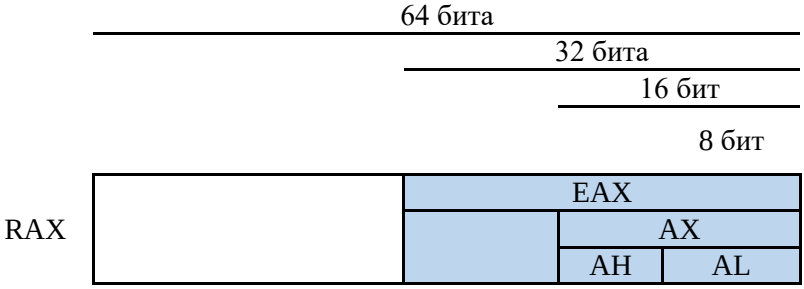


Рис. 12. Регистры архитектуры x86-32

Регистры архитектуры x86-64

Архитектура x86-64 в отличие от архитектуры x86 является полностью 64-битной. Она естественным образом расширяет регистры общего назначения x86 до 64 битов и увеличивает их количество. Также удваивается число регистров XMM. Регистры FPU/MMX остаются без изменений (рис. 13).



Регистры SSE/SSE2 128 бит	Регистры общего назначения	64 бита	
XMM0	RAX		EAX
XMM1	RBX		EBX
XMM2	RCX		ECX
XMM3	RDX		EDX
XMM4	RSI		ESI
XMM5	RDI		EDI
XMM6	RBP		EBP
XMM7	RSP		ESP
XMM8	R8		
XMM9	R9		
XMM10	R10		
XMM11	R11		
XMM12	R12		
XMM13	R13		
XMM14	R14		
XMM15	R15		

Рис. 13. Регистры архитектуры x86-64

В отличие от процессоров x86, у которых все вещественные вычисления производились в сопроцессоре, а блоку XMM отводились только векторные операции, процессоры x86-64 практически все вещественные вычисления выполняют в блоке XMM.

Формат машинных команд для архитектуры x86

Для процессора 32-бита длина машинной команды может быть от одного до 16 байт.

Таблица 3

Структура машинной команды для режима 16 бит

Компонент команды	Число байт
Префикс команды	0–1
Префикс замены сегмента	0–1
Код операции	1–2
Байт MRM – (mod, reg, r/m)	0–1
Поле для задания адреса	0–2
Непосредственный операнд	0–2

Таблица 4

Структура машинной команды для режима 32 бит

Компонент команды	Число байт
Префикс команды	0–1
Префикс изменения размера адреса	0–1
Префикс изменения размера операнда	0–1
Префикс замены сегмента	0–1
Код операции	1–2
Байт MRM – (mod, reg, r/m)	0–1
Байт SIB – (scale, index, base)	0–1
Поле для задания адреса	0–4
Непосредственный операнд	0–4

Префиксы

Команда начинается с кода операции. Но перед командой могут быть префикс и даже сразу несколько префиксов. Однако специальных полей под префиксы не выделяется. Часть кодов первого байта отнесена к кодам операций, а часть оставлена под префикс. Префиксы перед командами используются нечасто. Еще реже применяются сразу два префикса подряд. Префикс действует только в пределах той команды, перед которой он стоит.

Префиксы команды:

Код F0 – префикс блокировки шины, команда LOCK (запирать). Этот префикс употребляется только с теми командами, которые поддерживают такую возможность, – блокировку шины.

Коды F2, F3 – префиксы повторения, команда REP (repeat – повторять) и другие команды этой группы. Такие префиксы употребляются только с цепочечными командами. Префиксы группы REP позволяют организовать циклическое выполнение цепочечной команды.

Код F1 остается свободным.

Префикс изменения размера адреса:

Код 67 – если общий режим выполнения программы равен 32 битам, то для команды, перед которой есть префикс 67, устанавливается атрибут размера адреса 16 бит.

Если общий режим выполнения программы равен 16 битам, то для команды, перед которой есть префикс 67, устанавливается атрибут размера адреса 32 бита.

Действие префикса зависит от конкретной команды.

Префикс изменения размера операнда:

Код 66 – если общий режим выполнения программы равен 32 битам, то для команды, перед которой есть префикс 66, устанавливается атрибут размера операнда 16 бит.

Если общий режим выполнения программы равен 16 битам, то для команды, перед которой есть префикс 66, устанавливается атрибут размера операнда 32 бита.

Действие префикса зависит от конкретной команды.

Префиксы замены сегмента:

Для команд, в которых тем или иным способом задается адрес памяти, этот адрес памяти всегда относится к некоторому сегменту, заданному по умолчанию. С помощью префикса замены сегмента можно изменить сегмент, заданный по умолчанию. Однако такая замена возможна не во всех случаях. Это зависит от конкретной команды:

Код 26 – сегмент по умолчанию заменяется на сегмент ES.

Код 2E – сегмент по умолчанию заменяется на сегмент CS.

Код 36 – сегмент по умолчанию заменяется на сегмент SS.

Код 3E – сегмент по умолчанию заменяется на сегмент DS.

Код 64 – сегмент по умолчанию заменяется на сегмент FS.

Код 65 – сегмент по умолчанию заменяется на сегмент GS.

Код операции

Если у команды нет префиксов, то она начинается с кода операции. Поле с кодом операции всегда присутствует в любой машинной команде. Если команда состоит из одного байта, то это байт кода операции.

Код операции (сокращенно – КОП) может состоять из одного байта или из двух байт. Если первый байт кода операции имеет значение 0F, то в этом коде операции есть еще и второй байт. Если КОП состоит из одного байта, то этот байт является основным байтом кода операции. Если из двух байт, то основным следует считать второй байт кода операции.

Байт MRM (mod, reg, r/m) – это байт режима адресации.

Байт MRM имеется не во всех командах. Это определяется конкретным кодом операции (точнее, основным кодом операции), входит ли байт MRM в состав данной машинной команды или не входит.

Например, команда сложения имеет четыре разных варианта с байтом MRM, еще четыре варианта с «сокращенным» байтом MRM. Итого получается десять разных кодов операции, десять разных вариантов машинной команды.

Первым в этой группе идет байт (mod, reg, r/m), т. е. собственно байт MRM, затем, если требуется, может идти второй байт режима адресации, это байт SIB, а затем, при требовании, может быть еще и поле для задания адреса.

Байт MRM делится на три битовых поля (двухбитовое поле (mod), трехбитовое поле (reg), трехбитовое поле (r/m):

7	6	5	4	3	2	1	0
mod			reg			r/m	

Поле reg определяет первый операнд команды, операнд-приемник (destination). Поле r/m определяет второй операнд команды, операнд-источник (source). Обычно бывает именно такое распределение между reg и r/m. Но бывает и наоборот, это зависит от конкретной команды.

Поле reg задает регистр общего назначения. Три бита – это восемь вариантов, восемь разных регистров.

Поле r/m задает либо регистр, либо память. Это поле действует совместно с полем mod, вместе будет пять битов и получается 32 варианта. Это дает восемь вариантов для задания регистров и 24 варианта для задания формы и режима адресации памяти.

В режиме «32 бита» после байта MRM может стоять еще и второй байт режима адресации – байт SIB, так что количество разных форм задания адреса памяти еще более возрастает.

В системе команд x86 есть случаи, когда байт MRM применяется в сокращенном виде только из-за того, что для данной конкретной команды не требуется, чтобы байт MRM задавал два операнда, так как в этой команде нужен только один операнд. В этом случае поле reg байта MRM просто не используется, а в документации оговаривается, что всегда должно быть (reg = 0).

Байт SIB (scale, index, base) – это второй байт режима адресации. Байт SIB возможен в команде только в режиме 32 бита и только в том случае, когда в команде уже есть байт MRM (в полной форме или в сокращенной форме NNN).

Этот дополнительный байт для задания режима адресации включается в состав команды только при некоторых определенных значениях в полях mod и r/m байта MRM.

Аналогично байту MRM байт SIB делится на три битовых поля:

7	6	5	4	3	2	1	0
scale			index			base	

Байт SIB позволяет применять еще более сложные формы задания адреса в памяти.

Поле для задания адреса

В этом поле из 1 или 2, или 4 байт, расположенном внутри команды, может быть задан либо полный адрес, либо смещение относительно некоторого адреса.

С точки зрения терминологии не всегда можно отличить «смещение» от «полного адреса». Например, как называть полный адрес внутри сегмента, если такой адрес всегда задается смещением от начала сегмента.

Поле задания адреса может по-разному использоваться в разных командах. Однако, рассматривая общую структуру команды, важно различать следующие два случая:

1. Если в формат команды входит байт MRM, то поле для задания адреса (если оно есть) является приложением к байту MRM, оно может потребоваться в разных формах задания адреса, применяемых в байте MRM. Причем от кода байта MRM зависит, будет ли в данной машинной команде поле для задания адреса или не будет.
2. Если в формате команды нет байта MRM, то поле для задания адреса используется в той конкретной форме, которая предписана для данной конкретной машинной команды.

В системе команд x86 невозможен случай, чтобы в некоторой команде оказалось сразу два поля для задания адреса. Поле для задания адреса есть не во всех машинных командах.

В принципе знание машинных команд и правил их написания позволяет писать программы, которые могут быть выполнены центральным процессором компьютера.

Непосредственный операнд

Форматы многих команд предусматривают возможность задавать константу непосредственно внутри команды. Поле для непосредственного операнда есть не во всех машинных командах.

Если в формате команды есть поле для задания непосредственного операнда, то это поле в команде всегда будет последним.

Основные команды языка ассемблер

Команда MUL – умножение без знака

Когда operand имеет размер byte: AX = AL * operand.

Когда operand имеет размер word: (DX AX) = AX * operand.

Запись «DX: AX» означает, что старшее слово результата будет находиться в DX, а младшее – в AX.

Пример:

```
MOV AL, 200; AL = 0C8h
MOV BL, 4
```

MUL BL; AX = 0320h (800)

RET.

CF = OF = 0, когда старший байт результата 0.

Команда IMUL – умножение со знаком

Когда operand имеет размер byte: AX = AL * operand.

Когда operand имеет размер word: (DX AX) = AX * operand.

Запись «DX: AX» означает, что старшее слово результата будет находиться в DX, а младшее – в AX.

Пример:

MOV AL, –2

MOV BL, –4

IMUL BL; AX = 8

RET.

Команда NEG – изменение знака числа

Два шага:

1. Инвертирование всех битов операнда.
2. Прибавление к операнду, биты которого уже инвертированы 1.

Пример:

MOV AL, 5; AL = 05h

NEG AL; AL = 0FBh (–5)

NEG AL; AL = 05h (5)

RET.

Команда NOT

Инвертирование каждого бита операнда.

Если bit = 1, заменить его на 0.

Если bit = 0, заменить его на 1.

Пример:

MOV AL, 00011011b

NOT AL; AL = 11100100b

RET.

Инкремент INC: operand = operand + 1.

Пример:

```
MOV AL, 4
INC AL; AL = 5
RET.
```

Пример. Рассмотрим программу на ассемблере, выводящую на экран букву «А». Ее код имеет вид:

```
MOV AH, 02h
MOV DL, 41h
INT 21h
INT 20h.
```

Воспользуемся эмулятором Emu8086. Данный эмулятор позволяет запустить программы, разработанные для архитектуры этого микропроцессора (рис. 14).

#make_COM# – 1-я строка. Эта директива – специфическая для Emu8086. Она используется для определения типа создаваемого файла. В нашем случае это файл с расширением .COM.

ORG 100h – 2-я строка. Эта команда устанавливает значение программного счетчика в 100h, потому что при загрузке COM-файла в память DOS выделяет под блок данных PSP первые 256 байт (десятичное число 256 равно шестнадцатеричному 100).

Код программы располагается только после этого блока. Все программы, которые компилируются в файлы типа COM, должны начинаться с этой директивы.

MOV AH, 02h – 3-я строка. Инструкция (или команда) MOV помещает значение второго операнда в первый операнд, т. е. значение 02h помещается в регистр AH. 02h – это ДОСовская функция, которая выводит символ на экран. А записываем мы эту функцию (ее номер) именно в регистр AH, который использует прерывание 21h.

MOV DL, 41h – 4-я строка. Код символа «A» заносится в регистр DL. Код символа «A» по стандарту ASCII – это 41h.

INT 21h – 5-я строка. Это и есть то самое прерывание 21h – команда, которая вызывает системную функцию DOS, заданную в регистре AH (в нашем примере это функция 02h).

Команда INT 21h – основное средство взаимодействия программ с ОС.

INT 20h – 6-я строка. Это прерывание, которое сообщает операционной системе о выходе из программы и о передаче управления консольному приложению. Следовательно, при использовании INT 20h в нашем примере управление будет передаваться программе (рис. 15). Непосредственно в Emu8086 это прерывание можно заменить командой ret.

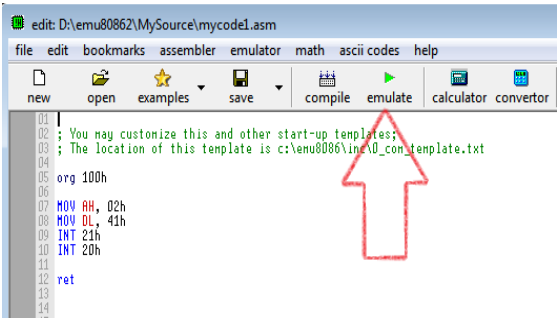


Рис. 14. Запуск программы в режиме эмуляции MS DOS

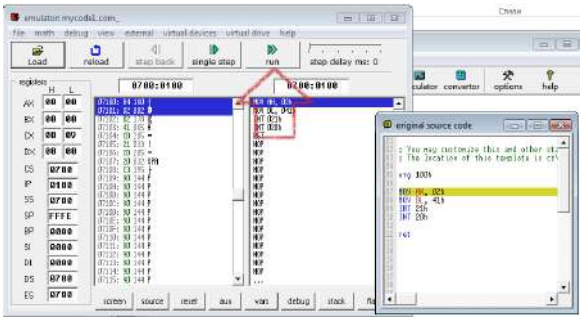


Рис. 15. Экран результатов преобразований в машинный код

В окнах представлены содержимое регистров, последовательность машинных команд, последовательность ассемблерных команд (рис. 16). Для запуска программы необходимо нажать кнопку run.

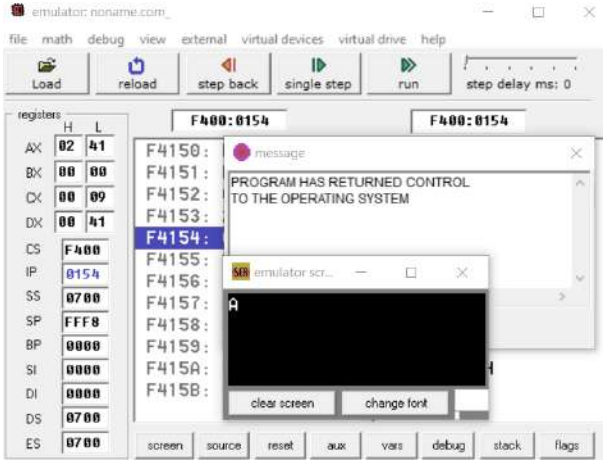


Рис. 16. Экран Emulator screen

Экран Emulator screen показывает результат выполнения программы. Message – экран, где выводится сообщение о том, что управление возвращено операционной системе.

Пример. Печать трехзначного десятичного целого без знака:

org 100h

Сегмент описания данных:

.data

Задание массива цифр:

nd db '0','1','2','3','4','5','6','7','8','9'

Задание массива чисел, соответствующих цифрам:

mmd db 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

Массив десятичных знаков, составляющих выводимого числа:

num3 db 3 dup(?).

Выводимое число длиной 2 байта (слово):

num dw?.

Промежуточная переменная, для сохранения модуля числа (остатка от деления нацело):

md db?.

Сегмент исполняемого кода:

.code

Mov num, 949 – запись выводимого числа в переменную num.

Определение старшего разряда десятичного числа:

MOV AX, num – запись числа в регистр AX (регистр AX имеет длину 2 байта и является составным. Он состоит из двух однобайтовых регистров – AH и AL. В AH содержатся старший байт двухбайтного регистра AH, а младший байт – в AL.

MOV BL, 100 – запись числа 100 в регистр BL.

DIV BL – деление содержимого AX на содержимое BL. Результат деления в AL, а в AH – остаток от деления.

mov md, AH – запоминание остатка от деления – число, на основе которого будет вычисляться средний разряд.

mov num3[1], Al – запоминание старшего разряда.

Определение среднего разряда десятичного числа:

MOV AL, md – занесение в регистр AL остатка от деления.

mov AH, 0 – обнуление регистра AH. Теперь регистр AX содержит число только из AL.

MOV BL, 10 – запись числа 100 в регистр BL.

DIV BL – деление содержимого AX на содержимое BL. Результат деления в AL, а в AH – остаток от деления.

mov md, AH – запоминание остатка от деления – число, на основе которого будет вычисляться младший разряд.

mov num3[2], AL – запоминание среднего разряда.

Определение младшего разряда десятичного числа:

mov num3[3], AH – запоминание младшего разряда.

mov DI, 1 – задание начального значения индекса разряда.

mov cx, 3 – задание счетчика циклов.

Loplab2: – метка, на которую передает управление внешний цикл (перебор разрядов).

mov Al, num3[DI] – запись старшего разряда в регистр AL из массива num3.

inc DI – увеличение индекса на единицу.

push cx – сохранение счетчика внешнего цикла в стеке, чтобы использовать его во внутреннем цикле.

MOV CX, 10 – запись в регистр CX (освободившийся счетчик) количества внутренних циклов – 10 (число десятичных цифр).

mov si, 0 – задание начального значения индекса.

Loplab: – метка, на которую передает управление внутренний цикл (перебор десятичных символов).

cmp AL, mmd[si] – сравнение десятичного разряда числа с его значением из массива.

JE m11 – если они равны, то передать управление на метку m11.

JMP end11 – в противном случае на метку end 11.

m11: mov ah, 2 – занести в регистр AH номер функции для печати.

mov dl, nd[si] – занести в регистр DL десятичный символ, соответствующий разряду числа.

int 21h – вызвать прерывание и выполнить функцию печати символа.

end11: – метка, на которую передается управление в случае пропуска символа.

inc si – увеличение индекса на единицу.

LOOP Loplab – оператор, реализующий внутренний цикл.

pop cx – извлечение из стека значение счетчика внешнего цикла.

LOOP Loplab2 – выполнение внешнего цикла.

mov ah, 0 – занесение в регистр номера функции ожидания нажатия произвольной клавиши.

int 16h – вызов прерывания и выполнение функции.

Ret – выход из программы и передача управления операционной системе (рис. 17).

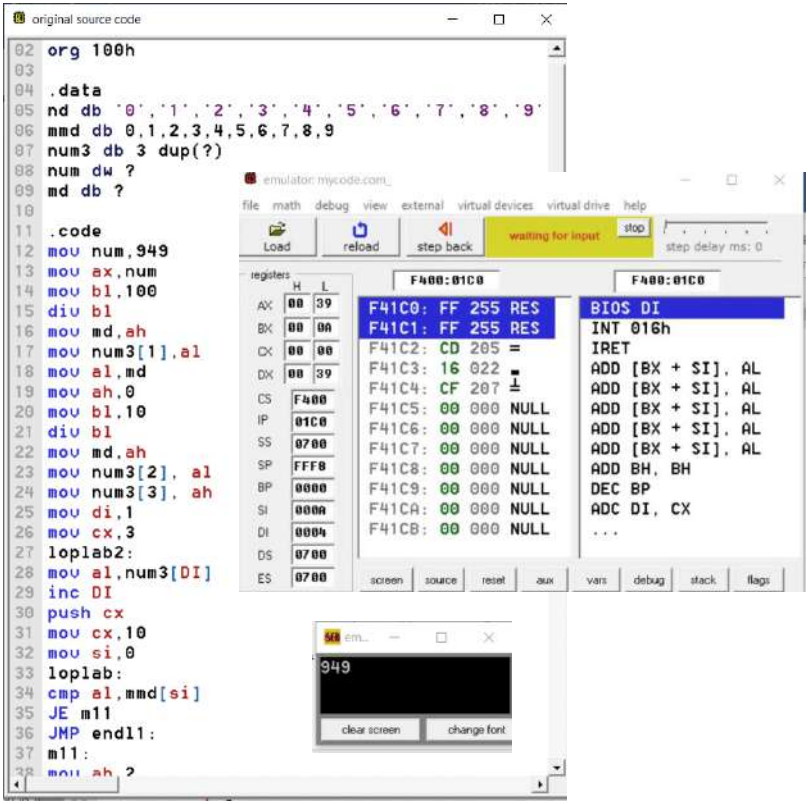


Рис. 17. Результат выполнения программы на экране

ГЛАВА 5
Основные конструкции языков программирования

5.1. Алфавит, идентификаторы, ключевые слова

Алфавит C++ включает:

– прописные и строчные латинские буквы и знак подчеркивания;

– арабские цифры от 0 до 9;

– специальные знаки: ? { } , ¦ [ ] ( ) + – / % * . \ ‘ : ? < = > ! & # ~ ; ^;

– пробельные символы: пробел, символы табуляции, символы перехода на новую строку.

Из символов алфавита языка формируются лексические единицы.

Лексические единицы программы на языке C++ – это идентификаторы, ключевые слова, литералы, операторы и разделители. Идентификатором является ряд букв и цифр, начинающихся с букв. Строчные и прописные буквы рассматриваются отдельно. Первым символом идентификатора может быть буква или знак подчеркивания, но не цифра. Пробелы внутри имен не допускаются.

Для улучшения читаемости программы следует давать объектам осмысленные имена. Существует соглашение о правилах создания имен, называемое *венгерской нотацией.* Это название получено благодаря программисту компании Microsoft венгерского происхождения Чарльзу Симони, предложившему ее еще во времена разработки первых версий MS-DOS. Эта система стала внутренним стандартом Microsoft, по которому каждое слово, составляющее идентификатор, начинается с прописной буквы, а вначале

ставится префикс, соответствующий типу величины, например *iMaxLength, IpfnSetFirstDialog*.

Название *верблюжья нотация (CamelCase)* получено, поскольку заглавные буквы внутри слова напоминают горбы верблюда.

Еще одна традиция – разделять слова, составляющие имя, знаками подчеркивания: «number_of_galosh». Этот стиль называется *змеиный регистр*.

Длина идентификатора по стандарту не ограничена, но некоторые компиляторы и компоновщики налагают на нее ограничения. Идентификатор создается на этапе объявления переменной, функции, типа и т. д., после чего его можно использовать в последующих операторах программы. При выборе идентификатора необходимо иметь в виду следующее:

– идентификатор не должен совпадать с ключевыми словами и именами используемых стандартных объектов языка;

– не рекомендуется начинать идентификаторы с символа подчеркивания, поскольку они могут совпасть с именами системных функций или переменных, кроме того, это снижает мобильность программы;

– на идентификаторы, используемые для определения внешних переменных, налагаются ограничения компоновщика (использование различных компоновщиков или версий компоновщика накладывает разные требования на имена внешних переменных).

Ключевыми словами являются идентификаторы, которые зарезервированы и не могут быть использованы в качестве идентификатора объекта. Список основных ключевых слов C++ представлен в табл. 5.

Таблица 5

Список ключевых слов C++

asm	auto	bool	break	case
catch	char	class	const	const_cast
continue	default	delete	do	double
dynamic_cast	else	enum	explicit	export
extern	false	float	for	friend
goto	if	inline	int	long
mutable	namespace	new	operator	private
protected	public	register	reinter-pret_cast	return
short	signed	sizeof	static	static__cast
struct	switch	template	this	throw
true	try	typedef	typeid	typename
union	unsigned	virtual	void	while

Литералы – это такие символы, которые в рамках принятой интерпретации определяют все свои особенности, в том числе и собственное значение. Литералы подразделяются на числовые, знаковые, с плавающей точкой и строки (строчные литералы).

5.2. Понятие переменной и ее роль в конструировании других элементов языка

Переменная в математике – это символ, который может принимать ряд заранее спланированных значений. При этом механизм и порядок подстановки этих значений не обсуждаются. Язык математических формул позволяет описывать достаточно сложные вычисления, при этом достаточно обычных математических операций. Однако бывают случаи, когда вычисления нужно описать по шагам. И на разных шагах переменная может иметь различные значения. Для этого случая вводится операция присвоения значения. Часто она обозначается так же, как равенство. Однако для строгого и четкого описания операцию присвоения

обозначают не так, как операцию равенства. Возможны различные варианты, например двоеточие и равно «:=», как в алгоритмических языках Алгол и Паскаль, или же логическое равенство обозначают двойным знаком равно, а присвоение для удобства – одним знаком равенства (C, C++, C#, Java и т. д.).

Оператор присваивания. Инкременты

Операндом называется элемент данных, над которым производят машинные операции. Пусть одному операнду присваивается результат обработки другого операнда. Операнд, которому присваивается значение, называется левым, а тот, который подвергается обработке, правым. Тогда действие присвоения значения левому операнду в результате обработки правого называется оператором присваивания:

левый операнд = правый операнд.

Типичная ошибка начинающих программировать на C++: они путают оператор присваивания «=» и логическую операцию равенства «= =».

Примеры оператора присваивания Y = 7, 8 – присвоение действительного числа, J = 8 – присвоение целого числа, c = ’h’ – присвоение символа.

Язык C++ допускает расширение синтаксиса присвоения новыми операциями, которые используются для экономной записи и удобства.

Дополнительные конструкции, расширяющие синтаксис операции присвоения:

1) += операция сложения с присваиванием;
2) –= операция вычитания с присваиванием;
3) *= операция умножения с присваиванием;
4) /= операция деления с присваиванием;
5) %= операция остатка от деления с присваиванием.

Все эти операции выполняют действия между левым и правым операндами и присваивают результат левому. Отличие от присвоения в том, что левый операнд указывается один раз слева, а не слева и справа как при обычном присвоении.

A = A + 5; эквивалентно A + = 5.

A = A – 5; эквивалентно A – = 5.

A = A * 5; эквивалентно A * = 5.

A = A/5; эквивалентно A/ = 5.

A = A%5; эквивалентно A% = 5.

Для целых переменных предусмотрены операции увеличения на единицу и уменьшения на единицу, называемые инкрементами и декрементами соответственно. Префиксный инкремент сначала изменяет переменную на единицу, а потом использует ее в выражении, а постфиксный сначала использует в выражении, а потом изменяет.

Синтаксис операций инкремента и декремента:

++/*имя переменной*/; // префиксный инкремент (преинкремент);

/*имя переменной*/++; // постфиксный инкремент (постинкремент);

– – /*имя переменной*/; // префиксный декремент (предекремент);

/*имя переменной*/– –; // постфиксный декремент (постдекремент).

Побитовые операторы

Побитовые операторы обращаются с переменной как с набором битов. Они используются в тех случаях, когда необходимо получить доступ к отдельным битам данных. Побитовые опера-

торы могут выполнять действия только с целочисленными значениями. В отличие от логических операторов, сравниваются не операнды, а соответствующие биты этих операндов.

Побитовое И. Оператор & записывает в бит результата единицу, только если оба сравниваемых бита равны 1:

```
int a=7, m =7, d; //описание и задание начальных значений
d = a & m; //операция побитового И для a и m
printf("   %d", d); // вывод на экран
while (_kbhit() == 0); // остановка консоли.
```

Побитовое ИЛИ. Оператор | записывает в результирующий бит единицу, если хотя бы один из сравниваемых битов равен 1.

Побитовое исключающее ИЛИ. Оператор ^ записывает в результирующий бит единицу, если сравниваемые биты отличаются друг от друга.

Побитовое отрицание. Оператор ~ инвертирует значение битов числа.

Примеры для этих операторов можно получить путем замены & на требуемую операцию в примере, приведенном выше.

Операторы сдвига. Операторы сдвига на определенное количество бит << и >>.

Пример деления числа на четыре путем сдвига на два разряда:

```
int main()
{   int a=4, d; // описание переменных и
                // задание им начальных значений
    d = a >> 2; // операция сдвига на два разряда
    printf("   %d", d); // вывод на экран
    while (_kbhit() == 0); // остановка консоли.
    return 0;
}
```

Переменная может быть любым специфическим математическим объектом, который подразумевает определенный набор абстрактных операций: способ представления при вводе-выводе,

ограничения на диапазон ее изменений, т. е. существует определенный способ интерпретации ее значений. Таким образом, для целей описания алгоритмов можно определить следующие основные свойства переменной:

1. Имя – набор символов, позволяющих ее идентифицировать-идентификатор.

2. Тип – образ действия со значениями переменной, которые будет выполнять человек или машина, исполняющие алгоритм, или способ интерпретации значений переменной.

Таким образом, чтобы определить переменную согласно правилам алгоритмических языков или языков высокого уровня, необходимо задать ее имя и тип. Ниже приведем примеры описания переменных наиболее часто встречающихся типов:

int g; имя – g, тип целый – int;

float f; имя – f, тип вещественный одинарной точности – float;

double d; имя – d, тип вещественный двойной точности – double;

char s; имя – s, тип символьный – char. Предназначен для хранения кода символа клавиатуры длиной в 1 байт.

Для того чтобы воспользоваться этими описанными именами переменных в качестве хранилищ данных заданного типа, необходимо вызвать их по имени:

g = 238 – для целого значения;

f = 3,1459; d = 86400,0033387660099 – для действительных;

s = ’h’ – для символьных (’h’ – форма задания символьной константы).

Итак, переменную языка высокого уровня, в том числе переменную языка C++, можно определить как имя и тип, который, в свою очередь, определяется длиной и способом ее интерпретации. Интерпретация может охватывать широкую область использования переменной – от разглядывания на экране до использования в

сложных процессах обработки, но она строго привязана к типу и не допускает использования не по назначению.

Машинные языки оперируют понятием ячейки памяти, которая имеет адрес и длину, и в принципе для машины не имеет значения, что в этих ячейках хранится и какие значения заносятся и выводятся. Внедрение идеи автоматического распределения памяти не только привело к тому, что появилась возможность называть ячейку буквами, но и позволило организовать вспомогательный контроль за тем, как содержание этих ячеек используется, ввести ограничения, позволяющие избавиться от путаницы, т. е., когда выполняются операции с данными, которые для этих операций не предназначены, а попали в обработку по недосмотру. Эту проверку возложили на компилятор вместе с обычной проверкой синтаксических ошибок. Типы данных могут позволять пользователю хранить знак переменной, а могут разрешать хранить только абсолютные значения. Это необходимо, чтобы числа со знаками не использовались там, где их не может быть по условиям задачи. Для этого при задании типа данных указываются слова signed или unsigned. Их иногда называют модификаторами. Они указывают компилятору, что первый бит числа содержит знак – signed или он освобождается и используется для хранения значения – unsigned. Эти модификаторы применимы к типам данных char, short, int, long и ряду специальных типов, образованных на их базе (_int32).

Для наиболее часто употребляемых данных в языке существуют стандартные типы. Их дополняет механизм организации специфических типов в соответствии с запросами пользователя. Этот механизм настолько мощно развит, что позволяет задавать типы данных с заданным пользователем перечисляемым множеством значений, сложные имена, которые позволяют выделять значения по индексам, полям, вызывать внутри имени функции-

члены, создавать новые имена путем наследования, управлять доступом к полям и функциям членам и многое др. Возможна отдельная компиляция сложных имен и создание на их основе динамических библиотек. В данном случае будут рассмотрены некоторые элементарные свойства имен переменных, необходимые для написания простых учебных программ и выполнения упражнений.

Ниже приводится табл. 6, описывающая основные стандартные типы данных в языке C++.

Таблица 6

Стандартные типы данных в языке C++

Тип	Размер в байтах	Диапазон значений
char	1	от –128 до 126
unsigned char	1	от 0 до 255
short	2	от –32 768 до 32 767
unsigned short	2	от 0 до 65 536
enum	2	от –2 147 483 648 до 2 147 483 647
long	4	от –2 147 483 648 до 2 147 483 647
unsigned long	4	от 0 до 4 294 967 295
int	4	от –2 147 483 648 до 2 147 483 647
unsigned int	4	от 0 до 4 294 967 295
float	4	от $3,4 * 10^{-38}$ до $3,4 * 10^{38}$
double	8	от $1,7 * 10^{-308}$ до $1,7 * 10^{308}$
long double	10	от $3,4 * 10^{-4932}$ до $3,4 * 10^{4932}$
bool	1	true или false

Константы, используемые в алгоритмах и программах, можно описать как переменные, значение которых не меняется. Но в этом случае при написании сложных текстов произойти так, что значение константы может быть изменено по ошибке. И компиляторы, и отладчики не сообщат вам о ней. Чтобы этого избежать, в

языке вводится инструмент описания констант. Константа – это переменная, которой можно присвоить значение только один раз. Она также вызывается по имени, как и переменная, но после ее объявления операция присвоения для нее невозможна. Фактически мы вводим формальный запрет на помещение нашего имени в левой части оператора присваивания. Сделать это можно с помощью квалификатора const. При этом до появления квалификатора const имя переменной можно использовать как обычную переменную и хранить в ней число.

Например, задание числа $\pi = 3{,}1459$ будет:

double PI = 8,9; // использование переменной до применения к ней квалификатора.

const double PI = 3,1459.

Константу можно задать и путем директивы #define препроцессору в начале файла. В этом случае константа будет объявлена в начале файла, и ее объявление будет действовать во всем файле. Препроцессор просто заменит во всем тексте файла выбранное вами имя на его значение. Использовать это имя как переменную до объявления ее константой не получится. Объявление константы через директиву компилятора:

#define PI = 3,1459.

Различие способов задания состоит в том, что const запрещает использовать в операторе присваивания имя, которое описано как переменная, а #define производит замену имени на заданное значение во всем файле и во всех модулях внутри него (глобальная константа). Однако в большинстве практических случаев оба способа дают одинаковые результаты.

5.3. Функции и процедуры

Скомпилированная и скомпонованная программа представляет собой исполняемый файл – PE (Portable Executable), имеющий расширение .exe. Этот файл может быть запущен на выполнение непосредственно из операционной системы. По сути, точка запуска этого файла является исполняемым образом главной функции программы, которая единственная и имеет имя main. Интегральная среда разработчика Visual Studio позволяет сразу создать главную функцию для консольного приложения:

```
#include"stdafx.h"

int main()
{
......
return 0;
}
```

Файл «stdafx.h» – файл, куда включают заголовочные файлы, которые с целью ускорения компиляции пропускаются через препроцессор. При обычной компиляции этот файл можно пропустить, но для этого нужно отключить опцию Precompiled Headers. Начинающим программистам можно пока игнорировать этот инструмент, он потребуется позднее для более сложных программ.

int main() – заголовок главной функции. В скобках могут быть размещены параметры, которые передаются в главную функцию при вызове файла *.exe в режиме командной строки. int – тип возвращаемого значения. Это значение возвращается в точку вызова через оператор – return. Для главной функции это значение – 0. Получая его, операционная система считает, что вызываемый файл выполнен успешно.

return 0; – оператор возврата значения главной функции.

Написанный программный код производит «холостой запуск». Эта программа не выполняет никаких действий. Таким образом, ее и не заметишь, поэтому, чтобы увидеть какую-нибудь реакцию, надо что-нибудь вывести или напечатать.

Выделять обособленные участки программы, обычно выполняющие какие-то специфические действия, – давняя идея программистов. Язык С++ имеет для этого соответствующий инструмент – это функция. Можно сказать, что язык С++ состоит из функций.

Таким образом, программа на языке С++ состоит из единственной главной функции и функций, которые вызываются из главной. Синтаксис описания функции следующий:

Тип_возвращаемого_значения Имя_функции (тип_аргумента имя_аргумента, …)

{

Тело функции.

return (возвращаемое значение);

}

Пример использования функции на языке С++. Вычислим функцию $u(x) = x^2 + 5$:

```
double u(double x)
{ double res;  res=x*x+5;
   return(res);
}
```

.

double – означает, что переменная имеет вещественный тип с двойной точностью (16 десятичных знаков).

Для того чтобы вычислить эту функцию для заданного значения x, необходимо вызвать эту конструкцию с заданным значением. Например, x = 5. u(5) – результат равен 30.

Пример:

```
int main()
{ double y; double arg=5;
 y=u(arg);
printf( "%-.10f", y);
system("pause");
}
```

system(“pause”); – задержка экрана консоли в С++.

Однако один программист не в состоянии разработать все функции, необходимые для эффективной работы. Поэтому между программистами существует разделение труда. Одни пишут программы для решения практических задач (проблемные программисты), другие создают часто употребляемые функции, обслуживающие компьютер, осуществляющие стандартные функции работы с файлами, ввода-вывода, элементы компьютерной графики, обмена данными по сети (системные программисты). Первые решают поставленную перед ними целевую задачу, а другие создают для них инструменты и накапливают их в виде стандартизированных библиотек.

Частью стандартной библиотеки С++ является заголовочный файл <iostream>, который содержит основные сведения, необходимые для всех операций с потоками ввода-вывода.

Все его возможности размещены в пространстве имен std, поэтому для использования данного заголовочного файла либо приходится приписывать префикс std, либо указывать пространство имен через using namespace:

```
using namespace syd;
...
cin>>a;
cout<<"вы ввели "<<a;
```

или

```
std::cin>>a;
std::cout<<"вы ввели"<<a;
```

Так, данный заголовочный файл включает объекты cin, cout, cerr, clog, которые соответствуют стандартным потокам ввода-вы-

вода и стандартным потокам вывода сообщений об ошибках. Объект стандартного потока ввода cin связан со стандартным устройством ввода, обычно с клавиатурой. Операция взять из потока (cin – the standard input stream – стандартный поток ввода), показанная в приведенном ниже операторе, означает, что величина переменной x должна быть введена из объекта cin в память cin >> x. Объект стандартного потока вывода cout связан со стандартным устройством вывода, обычно с экраном дисплея. Операция поместить в поток (cout – standard output stream – стандартный поток вывода), показанная в приведенном ниже операторе, означает, что величина переменной x должна быть выведена из памяти на стандартное устройство вывода cout << x.

Объекты cerr и clog связаны со стандартным устройством вывода сообщений об ошибках.

Пример практической реализации операции ввода-вывода. В программе показан вывод строки, использующий одну операцию поместить в поток:

```
#include<iostream>
using namespace std;
 int main()
 {
    setlocale (0,"rus");
    cout<<"Вы изучаете язык C++!\n";
    system("pause");
    return 0;
 }
```

Переход на новую строку в этих программах осуществляется с помощью управляющей последовательности \ n. Эту же операцию можно осуществить и с помощью манипулятора потока endl (end line – конец строки).

setlocale (LC_ALL, "Russian"); или setlocale (0, "rus"); позволяет осуществлять вывод в консольном приложении текста на русском языке.

Для решения вычислительных задач часто используется библиотека math.h стандартных математических функций (табл. 7).

Таблица 7

Стандартные математические функции

Матем-я функция	Описание	Функция	Пример
$\|a\|$	Модуль или абсолютное значение от a – abs, если а целое и fabs – вещественное	fabs(a) abs(a)	fabs(–3,0) = 3,0 abs(–5) = 5
$\sqrt{a}$	Корень квадратный из a, причем, a не отрицательно	sqrt(a)	sqrt(9,0) = 3,0
a^b	Возведение a в степень b	pow(a, b)	pow(2, 3) = 8
e^a	Вычисление экспоненты e^a	exp(a)	exp(0) = 1
sin(a)	a задается в радианах	sin(a)	
cos(a)	a задается в радианах	cos(a)	
tg(a)	a задается в радианах	tan(a)	
arcsin(a)	Арксинус a, где $–1,0 < a < 1,0$	asin(a)	asin(1) = 1,5708
arccos(a)	Арккосинус a, где $–1,0 < a < 1,0$	acos(a)	
arctg(a)		atan(a)	
arctg(x/y)		atan2(x, y)	
ln(a)	Натуральный логарифм a (основанием является экспонента)	log(a)	log(1,0) = 0.0
lg(a)	Десятичный логарфм a	log10(a)	log10(10) = 1

Окончание табл. 7

Наименьшее целое $\geq a$	Округление a до наименьшего целого, но не меньше чем a	ceil(a)	ceil(2,3) = 3,0 ceil(–2,3) = –2,0
Наибольшее целое $\leq a$	Округление a до наибольшего целого, но не больше чем a	floor(a)	floor(12,4) = 12 floor(–2,9) = –3
Остаток от деления a на b	Вычисление остатка от a/b	fmod(a, b)	fmod(4,4; 7,5) = 4,4 fmod(7,5; 4,4) = 3,1

Таким образом, любой процесс обработки данных может быть представлен в виде функции либо в виде группы функций, которые вызывают друг друга и обмениваются между собой данными. Элементами данных являются другие конструкции языка C++, такие как имена переменных, массивы структуры, сложные типы данных, организуемые пользователями.

Сфера действия описания имен

Областью видимости переменной могут быть:

1. Программный блок.
2. Функция.
3. Файл.
4. Вся программа.

Переменные, объявленные внутри функции, действуют только внутри этой функции. Переменные, объявленные внутри файла и имеющие спецификатор static, доступны во всем файле, начиная со строки объявления. Переменные, объявленные в одном из файлов программы на внешнем уровне (без спецификатора static), видимы в пределах всей программы. Переменные со спецификатором extern видимы во всей программе.

Указатели в С++

В языке С++ предусмотрены операции, позволяющие работать не только со значениями переменной, но и их адресами. Для этого существуют две операции: «*» и «&». Первая операция позволяет по адресу, содержащемуся в переменной, получить его значение, а вторая обратная операция – определение адреса по имени переменной.

**a* – находит значение по адресу, который содержится в *a*.

&a – получение адреса переменной *a*.

Указанные инструменты позволяют гибко управлять адресным пространством памяти, организовывать массивы, передавать данные в другие области видимости переменных. В частности, это передача данных в функции через параметры и обратно. Функции С++ позволяют передачу данных через параметры только в одном направлении – внутрь функции. Обратное возвращение значений производится через механизм указателей. В описании функции, куда надо передать параметр, он фигурирует как указатель *, а при вызове этой функции соответствующий параметр передается как адрес – &. Адрес попадает в функцию, и через указатель устанавливается связь между параметром вне функции и параметром внутри.

Например, описание функции funprob будет следующим:

```
int funprob (int *x)
{*x=...//указатель ищет ячейку по
      // адресу и посылает туда значение
}
```

Вызов функции во внешней программе с параметром *y* будет:

```
funprob (&y);
```

У переменной *y* находится адрес и передается параметру *x*, затем операция * находит по этому адресу ячейку в памяти компьютера, и, работая с **x*, функция фактически работает с внешней переменной *y*.

Таким образом, имена параметров внутри и снаружи обращаются к одной и той же памяти в адресном пространстве компьютера.

5.4. Операторы изменения порядка выполнения программы

Простейшие вычисления представляют собой линейные алгоритмы и простую последовательность действий. Однако для решения некоторых задач бывает необходимо изменять порядок следования операторов или команд в зависимости от исходных данных и алгоритма их обработки. Самый простой и часто используемый способ – использование условного оператора if.

Синтаксис его приведен ниже:

if (условие) {группа команд или операторов, которые выполняются, если условие истинно}

Группа команд или операторов, которые выполняются после того, как условие проанализировано.

Эта конструкция работает следующим образом: сначала анализируется условие, которое записано в скобках после слова if. Если условие истинно, то выполняется оператор, следующий за условием сразу. Если нужно, чтобы выполнилась сразу группа операторов, то она заключается в фигурные скобки – { }. Если условие не истинно, то восстанавливается обычный порядок вычислений – выполняются следующие по порядку операторы, а оператор или выделенная группа операторов, связанная с условием, не выполняется.

Типичная ошибка начинающих программировать на C++ – это точка с запятой после условия: if ($a < 3$); $a = a + 1$. Эта последовательность операторов не будет увеличивать a на единицу, если a меньше трех, а будет увеличивать a на единицу всегда. Данное выражение означает – если a меньше трех, то выполнить пу-

стой оператор, который расположен между скобкой и точкой с запятой –); , и только затем в обычном порядке выполняется увеличение *a* на единицу.

Может случиться, что условие сформулировано так, что оно не принимает ложного значения. Это соответствует случаю «неистинно», и обычный порядок следования операторов продолжается. Если нужно обязательно передать управление группе операторов, когда условие ложно, то используется конструкция else. Она фактически означает – «в противном случае».

Синтаксис конструкции if else приведен ниже:

if (условие) {группа команд или операторов, которые выполняются если условие истинно}

else {группа команд или операторов, которые выполняются, если условие ложно}.

После того как условие проанализировано и нужная альтернатива выполнена, восстанавливается обычный порядок следования операторов.

Пример использования оператора if else. Изменение возвращаемого значения, если переменная приняла недопустимое значение:

```
int process( double *d)
{
int er;    *d=*d*3.8;
if (fabs(*d) >100.0) er=1;else er=0;
return (er);
}
```

Рассмотри код подробно:

int process(double *d) – заголовок функции;

double *d – ссылочная переменная (параметр функции, который не только передает значение внутрь функции, но и возвращает измененные значения после проводимых над ним вычислений). Тип этой переменной действительный.

{ – скобка, открывающая описание функции;

*d = *d * 3,8; – присвоение переменной нового значения и выполнение вычислений с ее участием;

if (fabs(*d) > 100,0) er = l; else er = 0; – условный оператор анализа значения переменной d на превышение ограничений по модулю (100). Если модуль d больше 100, то переменной er присваивается значение 1. В противном случае присваивается значение 0.

return (er); – оператор возврата значения переменной er;

} – скобка, закрывающая описание функции.

Если условие сформулировано так, что возможно несколько альтернатив, то, добавляя операторы else, можно получить нужное количество альтернативных реакций.

Если альтернатив набирается много, то удобнее использовать конструкции switch\case.

switch (целочисленное выражение) {
case константа1;
выражение 1;
break;
case константа 2;
выражение 2;
break;
case константа 3;
выражение 3;
break;
…
case константа n;
выражение n;
break;
default: оператор (действие в случае, если не выбрана ни одна из альтернатив).

Пример программы на использование оператора switch:

```
using  namespace std;
int main()
{
    setlocale (0,"rus");
    char x;
    cout<<"Введите первую букву имени функции\n";
    cout<<"S-Sin\nC-Cos\nA-Atan\n";
    cin>>x;
    switch(x)
    {
        case's':
        case'S':
            cout<<"Вычисление синуса аргумента в радианах\n";
            break;
        case'c':
        case'C':
            cout<<"Вычисление косинуса аргумента в радианах\n";
            break;
        case'a':
        case'A':
            cout<<"Вычисление арктангинса аргумента в радианах\n";
            break;
        default:
            cout<<"Ошибка\n";
                }
    cout<<"\n----------------------------------------\n";
    system ("pause");
    return(0);}
```

.

Для того чтобы большой объем данных обработать по одному и тому же алгоритму применяются конструкции, позволяющие выполнять одну и ту же группу операторов несколько раз. Такая же ситуация получается при решении задачи итерационным методом. В этом случае применяются *циклы*.

Цикл for применяется, когда число повторений известно заранее.

for (инициализирующее выражение, условное выражение, модифицирующее выражение) – выражение, которое нужно повторить.

Если нужно повторить несколько операторов, то они заключаются в {}.

Инициализирующее выражение – выражение, где устанавливается начальное значение счетчика цикла. Условное выражение – условие прекращения цикла. Пока это условие истинно, цикл выполняется. Как только оно становится ложным цикл прекращается.

Пример программы, которая печатает номера 0, 1, 2, 3:

```
#include<iostream>

int main()
{
for (int i=0;i<4;i++) //Цикл повторяет своё "тело"
            // целое число раз , которое
            // соответствует диапазону перебора
            // значений индекса i; i=0 - конструкция
            // указывает начальный индекс перебора
             // i<4 - конструкция указывает конечный
             // индекс перебор перибираемые значения 0123
    {    printf("  %d", i);

        printf("\n");
    }
system("pause");
return 0;
}
```

```
D:\Работа\проги\Пособие_2.exe
 0
 1
 2
 3
Для продолжения нажмите любую
```

Цикл for можно использовать и для организации итерационных алгоритмов.

Однако при организации итерационных решений счетчик числа повторений делает конструкцию излишне сложной. Для этих алгоритмов достаточно прерывать циклы по условию. При этом вычислительная схема получается более ясной. В этих случаях применяются конструкции while и while do.

Цикл с предусловием while сначала проверяет условие, а затем выполняет тело цикла.

while (условие продолжения)

{

Тело цикла: группа операторов, которые нужно повторять

}

Если условие продолжения истинно, то цикл повторяется. В случае цикла с предусловием тело цикла может быть не выполнено ни разу.

Может случиться, что проверка условия до выполнения тела цикла приводит к некорректному выполнению алгоритма, тогда применяется конструкция с постусловием.

do

{

Тело цикла.

}

while (условие повторения цикла).

Тело цикла повторяется, пока выполняется условие. В случае цикла с постусловием тело цикла выполняется хотя бы один раз.

Пример конструкции с постусловием – вычисление Дзета-функции Римана для m = 8:

$$\zeta(m) = \sum_{n=1}^{\infty} \frac{1}{n^m},$$

```
double s,y,eps=0.000001,v=1;
int i, k;
int n, m=8;
    s = 0.0; i = 0;
    do {
        y = s; i++; v = v*i;
        s = s + 1.0/pow(v,m);
        }
     while (fabs(s-y)>eps);
```

В данной конструкции: pow(v, m) – функция возведения величины v в степень m; fabs(s–y) – вычисление абсолютного значения разности между последующим и предыдущим значением величины (данные функции относятся к библиотеке math.h).

Инструкции перехода

Инструкции перехода призваны по необходимости передать управление другому участку программы или группе операторов. Наиболее известна из них инструкция goto.

goto метка;

…

Метка: оператор, которому передается управление.

В C++ применение этой инструкции считается плохим стилем. В этом случае получаются плохо структурированные и плохо читаемые коды, что противоречит идеологии языка. Тем не менее она может использоваться при отладке и исследовании неизвестного кода, т. е. вспомогательным образом.

Инструкция break позволяет выходить из цикла до его окончания, до того как его условие станет ложным. Фактически это конструкция аналогична goto, только управление передается первому оператору после окончания цикла. Это позволяет вводить дополнительные условия прекращения цикла. Обычно это особые решения, возникающие в алгоритме и требующие действий по нестандартной схеме.

Инструкция continue прекращает выполнение текущего тела цикла и передает управление на новую итерацию. Также позволяет выполнять особые условия, возникающие в процессе выполнения алгоритма.

Инструкция return прекращает выполнение той функции, где она вызвана. Она может иметь параметр – возвращаемое значение функции. Если она применяется в функции main и для этой функции предусмотрено возвращаемое значение, то это значение 0, если функция завершена успешно. Если возвращается другое значение, то операционная система воспринимает ее, как завершенную с ошибкой, и выдает сообщение, что программа завершилась с данным кодом.

5.5. Массивы

Большое количество однородных величин, обрабатываемых типовым одинаковым способом, неудобно, а чаще невозможно хранить в отдельных именах. Необходима конструкция, позволяющая хранить данные под одним именем, но вызывать их по номеру. Такая конструкция существует в алгоритмических языках, она называется массивом. На машинном уровне он определяется начальным адресом, длиной элемента, числом элементов. Для того чтобы обратиться к элементу массива, необходимо длину элемента в байтах умножить на номер элемента –1 и прибавить к начальному адресу массива. Тогда получится адрес элемента массива, который соответствует заданному номеру.

В языках C, C++, C# массив – это элемент языка, предназначенный для хранения однородных данных, доступ к которым осуществляется по номеру. Массив характеризуется: именем, типом хранимых элементов, размерностью (числом хранимых элементов), нумерацией элементов. Синтаксически эта конструкция может быть описана в виде:

Тип Имя массива [константное выражение 1], [константное выражение 2]… [константное выражение n].

Одномерные массивы. Чаще всего используются одномерные массивы. Объявление массива выглядит следующим образом:

```
const int N = 15; //константа
const int M = 15;  //константа
int i, j; //описание индексов массивов
// Описание статического одномерного массива
int A[10]; double Z[10]; // размерность задана числовой константой
int B[N]; double Y[N]; // размерность задана буквенной константой.
```

Доступ к элементам массива осуществляется выражением A[i], где i – индекс элементов массива, который начинается с нуля.

Объявление массива можно совмещать с заданием элементам массива начальных значений. Эти значения перечисляются в списке инициализации после знака равенства, разделяются запятыми и заключаются в фигурные скобки, например:

int A[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}.

Многомерные массивы – это массивы, элементами которых являются одномерные массивы. Например, двумерный массив может быть объявлен следующим образом:

// Описание статического многомерного массива

int A2[10][5]; double Z2[10][5]; // размерность задана числовыми константами

int B2[N][M]; double Y2[N][M]; // размерность задана буквенными константами

int C[N][10]; double X[N][10]; // размерность смешанная.

Доступ к значениям элементов многомерного массива обеспечивается через индексы, каждый из которых заключается в квадратные скобки. Например, A2[3][2] – значение элемента, лежащего на пересечении четвертой строки и третьего столбца (напоминаем, что индексы начинаются с 0).

Инициализация двумерного массива:

int B[3][3] = {{1, 2, 3},{3, 2, 1},{1, 2, 3}}.

Так как массив является структурным типом, то основные операции с ним надо проводить с помощью оператора цикла.

Пример программы, в которой создается массив из 10 элементов. Вычисляется и отображается среднее значение всех элементов, а также минимальное и максимальное значение:

```
#include <iostream>
using namespace std;
int main()
{   setlocale(0,"rus");
    int i, min, max;
    float sred;
    int nums[10]={10,18,75,0,1,-2,4,-3,11,8};

    //Вычисление среднего значения
    sred=0;
    for (i=0;i<10;i++)
    sred+=nums[i];  //Суммируем значения массива
    sred/=10;       //Вычисляем среднее значение
    cout<<"\nСреднее значение равно "<<sred<<"\n";

    //Определение минимального и максимального значения
    min=max=nums[0];
    for(i=1;i<10;i++)
    {
    if (nums[i]<min) min=nums[i]; //минимальный элемент
    if (nums[i]>max) max=nums[i]; //максимальный элемент
    }

    cout<<"\nМинимальное значение:"<<min<<"\n";
    cout<<"\nМаксимальное значение:"<<max<<"\n";
    system ("pause");
    return 0;
}
```

.

Результат выполнения программы:

```
Среднее значение равно 12.2

Минимальное значение:-3

Максимальное значение:75
Для продолжения нажмите любую клавишу . . .
```

.

Пример программы, которая вычисляет количество положительных, отрицательных и нулевых элементов двумерного массива размерностью 2 на 3. Для задания размера массива часто удобно использовать константы. В данном примере число строк двумерного массива задается константой row, а число столбцов – константной col. Использование констант для задания размера массивов делает программу более наглядной и масштабируемой,

так как при любом изменении размеров массива в программе достаточно изменить только значения констант.

```
#include <iostream>
#define row 5
#define col 5
using namespace std;
int main()
{
    setlocale(0,"rus");
    float b[row][col];
    int i=0, j=0, n=0,p=0,zero=0;
    cout<<"\nОпределить кол-во полож.-ых и отр.-ых эл-ов";
    cout<<"\nмассива b["<<row<<","<<col<<"]\n";
    for (i=0; i<row;i++)
    {
        for (j=0; j<col; j++)
        {
            cout<<"\nВведите b["<<i+1<<","<<j+1<<"]=";
            cin>>b[i][j];
            }
            cout<<"\n\n";
            }
        for(i=0; i<row; i++)
        {
            for(j=0; j<col; j++)
            {
                if(b[i][j]>0) p+=1;
                if(b[i][j]<0) n+=1;
                if(b[i][j]==0) zero+=1;
                }
        }
    cout<<"\nЧисло положительных элементов="<<p;
    cout<<"\nЧисло отрицательных элементов="<<n;
    cout<<"\nЧисло нулевых элементов="<<zero<<"\n";

    system ("pause");
    return 0;}
```

Особенность передачи массива в функцию

Имя массива в функцию передается только по ссылке. При этом в явном виде знаки * и & не используются.

Пример:

```
int  func( int A[ ], N)
{ int d;  d=A[N];
 return (d);
}
Res=func(A,4); //получение элемента с номером 4
```

Динамические массивы

Рассмотренные ранее конструкции определяют массив с момента описания и до момента прекращения работы программы. Также за ними закрепляется необходимая память. Такие массивы называются *статическими*. Их размерность задается константами и не может быть изменена в процессе работы программы. Это сильно ограничивает возможности эффективного использования памяти, поэтому существуют инструменты, которые позволяют организовать память по принципу «кучи» (heap). В языке С для динамического распределения памяти используется библиотека <malloc.h>. Она содержит функции malloc, calloc, realloc, free, которые используются для динамического распределения памяти. Кроме того, для создания динамических массивов в С++ рекомендуется использовать функцию создания объекта new и функцию удаления delete.

Пример создания динамического массива функцией malloc. Создается массив размерностью 10, он заполняется значениями квадратов индексов, выводится на экран, а затем память освобождается функцией free:

```
#include<iostream>
#include <malloc.h>
#include <conio.h>
int main()
{
double  *c;
int N=10; int i;
// создание динамического массива размерностью N.
c = (double *)malloc(N*sizeof(double));
for (i = 0; i < N; i++)
{
    c[i] = i*i;
  }
for (i = 0; i < N; i++)
printf("\n c[%d]=%f", i, c[i]);
_getch();
free(c); // освобождение памяти
return 0;
}
```

.

Функция realloc позволяет перераспределить уже ранее выделенную память.

Пример увеличения памяти на пять элементов. Программа создает динамический массив размерностью N, заполняет его квадратами индексов, затем увеличивает массив на пять элементов с выделением дополнительной памяти, а затем добавленные элементы заполняются просто значениями индексов. После печати результатов память освобождается:

```
#include<iostream>
#include <malloc.h>
#include <conio.h>
int main()
{
  double  *c; // указатель на массив
int N=10; int i; //
// создание динамического массива размерностью N.
c = (double *) calloc (N, sizeof(double));
for (i = 0; i < N; i++)
{
c[i] = i*i;  //заполнение вновь созданного раздела
  }
// увеличе-ние размерности на 5 элементов;
c = (double *) realloc (c, (N + 5)*sizeof(double));
 for (i = N; i < N+5; i++)
    {
       // заполнение дополнительных элементов
        c[i] = i;
    }
    for (i = 0; i < N+5; i++)
        printf("\n c[%d]=%4.2f", i, c[i]);
    _getch();
    free(c); // освобождение памяти
    return 0;
}
```

```
D:\Работа\прогр
c[0]=0.00
c[1]=1.00
c[2]=4.00
c[3]=9.00
c[4]=16.00
c[5]=25.00
c[6]=36.00
c[7]=49.00
c[8]=64.00
c[9]=81.00
c[10]=10.00
c[11]=11.00
c[12]=12.00
c[13]=13.00
c[14]=14.00
```

.

Применение функций new и delete позволяет легко выделять память как под сложные объекты и структуры, так и под обычные переменные.

Синтаксис оператора new:

Объект = new Тип Инициализатор.

Пример создания переменной с начальным значением, равным π:

```
double  * Ppi;
 Ppi = new double;
 *Ppi = 3.14159;
printf("\n %f", *Ppi);
delete (Ppi); Ppi = NULL;
```

Пример создания одномерного массива:

```
double  *c;
int N = 10; int i; double  * Ppi;
// создание динамического массива размерностью N.
c = new double[N];
for (i = 0; i < N; i++)
{
    c[i] = i*i;
 }
for (i = 0; i < N; i++)
printf("\n c[%d]=%f", i, c[i]); // вывод значений массива
delete c;
```

Одномерный массив является образом строки матрицы или вектора. Двумерный массив – соответственно строка, каждому элементу которой ставится в соответствие столбец (вектор). Поэтому двумерный динамический массив создается в два этапа. Создается массив указателей на указатели строк, затем они разворачиваются в массив указателей на элементы столбцов.

Пример создания многомерного динамического массива:

```
#include<stdio.h>
#include <malloc.h>
using namespace std;
int main()
{
    int N=3,M=5;

int **A;
A=new int*[N];   //массив указателей на указатели строк
  for (int i=0;i<N;i++)
  {
      A[i]=new int [M]; //массивы указателей на
                        // элементы столбцов
  }
  for(int i=0;i<N;i++)  //Цикл повторяет своё "тело" целое
                        //число раз , которое соответствует
                        // диапазону перебора значений индекса i;
                        // i=0 - конструкция указывает начальный
                        // индекс перебора; i<4 - конструкция
                        //указывает конечный индекс перебора
                        // перибираемые значения 0 1 2
                        // перебор по второму индексу
   {
     for(int j=0; j<M;j++) //Цикл повторяет своё "тело" целое
                        // число раз , которое соответствует
                        // диапазону перебора значений индекса j;
                        // j=0 - конструкция указывает начальный
                        // индекс перебора; j<2 - конструкция
                        // указывает конечный индекс перебора
                        // перибираемые значения 0 1 2 3 4
    {
       A[i][j]= i*2+j*3;   // i*2+j*3 - выражение зависимое
                           // от индекса вставлено для примера.

    printf(" A[%d][%d]=%d",i,j,A[i][j]);
    printf("\t");
    }
        printf("\n");
    }
    // Освобождение памяти после использования
    // двумерных динамических массивов
   for (int i = 0; i < N;  i++)
      delete[] A[i];
  delete[] A;

  system("pause");
    return 0; }
```

Результат программы:

```
A[0][0]=0       A[0][1]=3       A[0][2]=6       A[0][3]=9       A[0][4]=12
A[1][0]=2       A[1][1]=5       A[1][2]=8       A[1][3]=11      A[1][4]=14
A[2][0]=4       A[2][1]=7       A[2][2]=10      A[2][3]=13      A[2][4]=16
Для продолжения нажмите любую клавишу . . .
```

Применение malloc, calloc, realloc

Ниже приводится пример создания массива типа unsigned char (это целая величина от 0 до 255 значений) функцией calloc. Обычно unsigned char используется для хранения изображения с 256 градациями серого. Создается двумерный массив, заполняется суммой индексов по строкам и столбцам и выводится на экран консоли:

```
#include <iostream>
#include<stdio.h>
#include <malloc.h>
using namespace std;
int main()
{
int i, j, n, m;
unsigned char **R; // Указатель на указатели
n = 5; m = 5;
//указатели на строки
R = (unsigned char **)calloc(n, sizeof(unsigned char *));
if (R == NULL) { exit(-1); }
for (i = 0;i<n;i++) {
// указатели на элементы  столбцов
R[i] = (unsigned char  *)calloc(m, sizeof(unsigned char));
    if (R[i] == NULL) { exit(-1); }
}
for (i = 0;i<n; i++)
    for (j = 0;j<m; j++)
    {
        R[i][j] = i+j; // вызов элементов динамического
                       //массива по индексам и задание ему
                      // значения
    }
for (i = 0;i<n; i++)
{
    for (j = 0;j<m; j++)
    {
printf(" R[%d][%d]=%d", i, j, R[i][j]);
printf("\t");
    }
    printf("\n");
}
free(R); // освобождение памяти  занятой под массив R
system("pause");
return 0; }
```

```
D:\Работа\проги\Пособие_1.exe
R[0][0]=0       R[0][1]=1       R[0][2]=2       R[0][3]=3       R[0][4]=4
R[1][0]=1       R[1][1]=2       R[1][2]=3       R[1][3]=4       R[1][4]=5
R[2][0]=2       R[2][1]=3       R[2][2]=4       R[2][3]=5       R[2][4]=6
R[3][0]=3       R[3][1]=4       R[3][2]=5       R[3][3]=6       R[3][4]=7
R[4][0]=4       R[4][1]=5       R[4][2]=6       R[4][3]=7       R[4][4]=8
Для продолжения нажмите любую клавишу . . .
```

Создание динамических массивов функциями malloc и realloc аналогично функции calloc. Для получения этих синтаксических конструкций достаточно воспользоваться примерами, изложенными ранее.

5.6. Перечисления, объединения, битовые поля

Перечисления – конструкция языков C/C++, которая призвана сделать программный код более понятным и читаемым. Они создаются с помощью ключевого слова enum. Для описания перечисления применяется следующая синтаксическая форма:

enum необязательный_тег { константа1 [=значение 1],
константа2 [=значение 2], …
константа n [=значение n]}

Пример перечислений:

```
#include <iostream>
using namespace std;
enum e_acomany {
    Audi,
    BMW,
    Cadillac,
    Ford,
    Jaguar,
    Lexus,
    Maybach,
    RollsRoyce,
    Saab
};


int main()
{
    e_acomany car_for_me;
    for (car_for_me = (e_acomany)0;
    car_for_me < Saab;
    car_for_me = (e_acomany)((int)car_for_me + 1))

    {
        cout << "\n" << car_for_me;
    }

    system("pause");
    return 0;
}
```

.

Объединение – это переменная специального типа, которая в разные моменты времени может содержать данные разных типов.

Ее конструкция похожа на структуру, но при этом под все поля выделяется одна область памяти, размер которой определяется размером самого большого поля. Конструкция объединения существует только в C/C++. В языке C# для выполнения тех же действий используются классы.

Объединение создается с помощью ключевого слова union:

union тег {

тип 1 имя 1;

тип 2 имя 2;

тип 3 имя 3;

…

тип n имя n;}

Пример объединения:

```
//Пример объединения
#include <iostream>
using namespace std;
 union
{
    short int name1;
    int name2;
    long int name3;
} myUnion;
int main()
{   myUnion.name1 = 22;
    cout << myUnion.name1 << endl;
    myUnion.name3 = 222222222;
    cout << myUnion.name3 << endl;
    cout << myUnion.name1 << endl; // снова обращаемся к name1
    system("pause");
    return 0;
}
```

```
D:\Работа\проги\Пособие_4.exe
22
222222222
-10354
Для продолжения нажмите любую клавишу
```

.

Битовые поля

В языках C/C++/C# имеется возможность задать в виде поля отдельные группы бит. Обычно они используются для передачи, хранения, задания кодов, характеризующих состояние некоторого объекта или устройства. Такие коды принято называть флагами. Простейший флаг имеет длину один бит и два состояния – 0 или 1. Обычно ими описываются состояния типа «вкл.», «выкл.».

В этих ситуациях в C/C++ применяется конструкция «битовое поле», которая предусмотрена как элемент структуры. Битовые

поля отличаются от обычных полей структуры тем, что за именем поля идет двоеточие с указанием длины поля в битах. Поле p5 в примере, приведенном ниже, имеет длину 1 бит. При вызове этой конструкции fll.p5 этому полю присваивается значение 1.

Пример структуры с битовыми полями:

```
#include <iostream>
#include "conio.h"

int main()
{
    struct flagibitu {
        unsigned char
    p1 : 1,
    p2 : 1,
    p3 : 1,
    p4 : 1,
    p5: 1,
    p6: 1,
    p7: 1,
    p8: 1;
    };


    struct flagibitu fll;
    fll.p5 = 1;

    printf("   %d", fll.p5);
    while (_kbhit() == 0);
    return 0;
}
```

.

В языке C# имеется класс BitArray, который расположен в пространстве имен System.Collections. Он позволяет программировать те же самые действия.

При использовании битовых полей фактически используется синтаксис структуры. Единственное ограничение – к битовым полям нельзя применять операцию взятия адреса.

5.7. Структуры, функции-члены структур. Классы, функции-члены классов

Сохранение однородных данных под одним именем дает эффект, если к этим данным применяются один и тот же алгоритм, функция, программа. Однако возникает целесообразность под одним именем объединять разнородные данные, например свойства какого-то предмета, сущности, объекта. Разбросанные по программе отдельные переменные создают трудности при программировании – может возникнуть элементарная путаница. При передаче их в функцию в качестве параметров конструкция получается громоздкой. Поэтому в программах с большим количеством переменных их лучше классифицировать по принадлежности к какой-нибудь сущности, предмету или объекту. Впервые такая конструкция появилась в языке PL-1 и получила название структуры. Затем была подхвачена Виртом и появилась в языке Pascal. Язык С продолжил эту традицию. В С++ структура практически мало чем отличается от понятия класса и объекта.

Итак *структура* – это объединенное в единое целое множество поименованных элементов в общем случае разных типов.

Синтаксис структуры

Структура создается с помощью ключевого слова struct, необязательного тега или имени экземпляра, а затем списка членов структуры:

```
struct   Тег {
Тип 1 имя 1;
Тип 2 имя 2;
…
Тип n имя n;
};
```

Вызов в программе.

Тег. Имя 1 … Тег. Имя n.

Иными словами, для обращения к объектам, входящим в качестве элементов в конкретную структуру, чаще всего используются уточненные имена. Общей формой уточненного имени элемента структуры является выражение с двумя операндами и операцией «точка» между ними:

Пример, в котором нужно ввести с клавиатуры информацию о пяти студентах (фамилия, имя, пол, возраст), вывести ее на экран в виде таблицы и выбрать студентов мужского пола моложе 20 лет:

```
#include <iostream>
#include<Windows.h>
using namespace std;

int main()
{
SetConsoleCP(1251);
SetConsoleOutputCP(1251);
struct person
{char fam[15]; char name[10];
int vozrast; char pol;
} student[5];
int i; cout<<"Введите информацию\n";
for (i=0;i<5;i++)
{cout<<"студент"<<i+1<<"\n";
 cout<<"Фамилия->";
 cin>>student[i].fam;
 cout<<"Имя->";
 cin>>student[i].name;
 cout<<"Пол->";
 cin>>student[i].pol;
 cout<<"Возраст->";
 cin>>student[i].vozrast;}

 cout<<"Фамилия\tИмя\tПол\tВозраст\n";
for (i=0;i<5;i++)
cout<<student[i].fam<<"\t"<<student[i].name<<"\t"<<
          student[i].pol<<"\t"<<student[i].vozrast<<"\n";
cout<<"Студенты мужского пола моложе 20 лет:\n";
for (i=0;i<5;i++)
if (student[i].pol=='M'&&student[i].vozrast<20)
cout<<student[i].fam<<"\t"<<student[i].name<<"\t"<<
           student[i].pol<<"\t"<<student[i].vozrast<<"\n";
system("pause");
}
```

После того как введем все данные, программа выдаст следующий результат:

```
Фамилия Имя     Пол     Возраст
Иванов  Иван    М       18
Петров  Петр    М       26
Сидорова        Елена   Ж       19
Кулич   Семен   М       19
Новиков Тимур   М       28
Студенты мужского пола моложе 20 лет:
Иванов  Иван    М       18
Кулич   Семен   М       19
Для продолжения нажмите любую клавишу . . .
```

Передача структуры внутрь функции в качестве параметра осуществляется просто по имени. Возврат значений структуры из функции осуществляется так же, как и возврат обычной переменной через указатели «*» и через получение адреса – «&».

Указатели на члены структуры не содержат символ «*», а точка, отделяющая член структуры, заменяется на «–>».

Для примера разберем передачу структуры examp в функцию make.

```
void make (examp)
{
examp.поле1= 6…; выполняются действия со структурой и ее полями.
}
void main()
{ make(examp); вызов функции.
}
```

Данные, передаваемые в функцию, при этом не возвращаются. Для возвращения необходимо использовать указатели.

```
void make( *examp)
{
 examp –>поле 1 = 6…; выполняются действия со структурой и ее полями.
}
void main()
```

```
{
 make(&examp); вызов функции.
}
```

В C++ понятие структуры практически приближено к понятию класса. Фактически и структура, и простейший класс имеют близкий набор возможностей. Различие заключается в форме реализации и механизме конструирования сложных программ из элементарных элементов. Структура – это часть разрабатываемой программы, она используется только в модуле, где она объявлена. Класс – это отдельный изолированный элемент программы, он может использоваться многократно в разных программах, компилироваться отдельно, и на его базе могут создаваться динамические библиотеки dll.

Классы

Класс – сложный тип данных, определяемый пользователем. Каждый класс содержит данные и набор функций, манипулирующих с этими данными. Компоненты – данные класса – называются данными-элементами. Компоненты – функции класса – называются функциями-элементами. Класс, определяемый пользователем, называется объектом. Классы в C++ являются естественным продолжением структуры (struct) в C.

Определение класса состоит из двух частей: заголовка, включающего ключевое слово class, за которым следует имя класса, и тела, заключенного в фигурные скобки. После такого определения должны стоять точка с запятой:

Внутри тела объявляются данные-члены и функции-члены и указываются уровни доступа к ним. Таким образом, тело класса определяет список его членов.

Синтаксис объявления класса:

Class имя класса: список классов разделителей

{ public:

Данные, методы, свойства, события, доступные всем.

}

{ protected:

Данные, методы, свойства, события, доступные только классам потомков

}

{ private:

Данные, методы, свойства, события, доступные только внутри класса.

}.

Тело класса определяет отдельную область видимости. Объявление членов внутри тела помещает их имена в область видимости класса. Наличие в двух разных классах членов с одинаковыми именами – не ошибка, эти имена относятся к разным объектам.

После того как тип класса определен, на него можно ссылаться двумя способами:

– написать ключевое слово class, а после него – имя класса;

– указать только имя класса.

Оба способа сослаться на тип класса эквивалентны. Первый заимствован из языка C и остается корректным методом задания типа класса; второй способ введен в C++ для упрощения объявлений.

Данные-члены класса объявляются так же, как переменные. Функции-члены класса объявляются в его теле. Это объявление выглядит точно так же, как объявление функции в области видимости пространства имен.

Функции-члены отличаются от обычных функций следующим:

– функция-член объявлена в области видимости своего класса, следовательно, ее имя не видно за пределами этой области.

К функции-члену можно обратиться с помощью одного из операторов доступа к членам – точки (.) или стрелки (–>);

– функции-члены имеют право доступа как к открытым, так и к закрытым членам класса, тогда как обычным функциям доступны лишь открытые. Конечно, функции-члены одного класса, как правило, не имеют доступа к данным членам другого класса.

Функция-член может быть перегруженной. Однако она способна перегружать лишь другую функцию-член своего класса. По отношению к функциям, объявленным в других классах или пространствах имен, функция-член находится в отдельной области видимости и, следовательно, не может перегружать их.

Пример простейшего класса. Класс содержит функции, выполняющие различные операции с символьными, действительными, целыми переменными:

```
#include <iostream>
#include <math.h>
#include <cstdio>
#include <conio.h>

// Описание класса
class procClas
{
    char simbol;  //переменная член класса
public:
    void makeSimb();
    int process(char* s, int* g, double* d);
};
//Функции члены класса
//Функция заполнения символа
void procClas::makeSimb()
{
    simbol = 'f';
}
```

```
//Функция process
int procClas::process(char* s, int* g, double* d)
{   int er; *s = simbol; *g = *g + 5; *d = *d * 3.8;
    if (fabs(*d) > 100.0) er = 1; else er = 0;
    return (er);
}
int main()
{   procClas ProcObj; //Описание экземпляра класса
    int erm; int g; double d; char s;
    FILE* f0, * fl;
    errno_t erin, erout;

        //Ввод переменных из файла
    erin = fopen_s(&f0, "D:\\ref.txt", "r");
    fscanf_s(f0, "%c", &s); fscanf_s(f0, "%d", &g);
    fscanf_s(f0, "%le", &d);
    fclose(f0);
    //Вызов функции члена класса - занесение символа f
                                  // в переменную класса.
    ProcObj.makeSimb();
    //Вызов функции члена класса
    erm = ProcObj.process(&s, &g, &d);

        // Вывод полученных параметров в файл
    erout = fopen_s(&fl, "D:/wrf.txt", "w");
    fprintf(fl, "%c %d %-.10f", s, g, d);
    printf("\n wrf.txt= %c %d %-.10f ", s, g, d);
    printf("\n error %d ", erm);
    fclose(fl);
    while (_kbhit() == 0); // останов консоли
    return 0;
}
```

5.8. Класс как инструмент создания объектов. Закрытый характер переменных класса. Принципы ООП

Модификаторы доступа public и private

Все свойства и методы классов имеют права доступа. По умолчанию все содержимое класса доступно для чтения и записи только для него самого. Для того чтобы разрешить доступ к данным класса извне, используют модификатор доступа public. Все функции и переменные, которые находятся после модификатора public, становятся доступными из всех частей программы.

Закрытые данные класса размещаются после модификатора доступа private. Если отсутствует модификатор public, то все функции и переменные по умолчанию являются закрытыми.

Обычно приватными делают все свойства класса, а публичными – его методы. Все действия с закрытыми свойствами класса реализуются через его методы.

Создание объекта через указатель

При создании объекта лучше не копировать память для него, а выделять ее в «куче» с помощью указателя. И освобождать ее после того, как закончена работа с объектом.

При создании статического объекта для доступа к его методам и свойствам используют операцию прямого обращения – «.» (символ точки). Если же память для объекта выделяется посредством указателя, то для доступа к его методам и свойствам используется оператор косвенного обращения – «–>».

Конструктор и деструктор класса

Конструктор класса – это специальная функция, которая автоматически вызывается сразу после создания объекта этого

класса. Он не имеет типа возвращаемого значения и должен называться так же, как класс, в котором он находится.

Деструктор класса вызывается при уничтожении объекта. Имя деструктора аналогично имени конструктора, только в начале ставится знак тильды ~. Деструктор не имеет входных параметров.

Пример простого класса – паспортные данные сотрудника:

```
#include "iostream"
#include <stdio.h>
#include <conio.h>

//Пример применения класса

//Описание класса
class Pasport
{
    long Number;         //переменная класса - номер паспорта
    char *Kem_Vydan;     //переменная класса - кем выдан
    int data[3];         //переменная класса-дата выдачи
    char *propiska;      //переменная класса -где прописан

public:
    Pasport()          //Конструктор класса
    {
        Number = 333444;                    //номер паспорта
        Kem_Vydan = " OVDCokolMoskva ";     // кем выдан
        // дата выдачи
        data[0] = 1995;  //год
        data[1] = 10;    //месяц
        data[2] = 14;    //число
        propiska = "Moskva ";// прописка
    }
    ~Pasport() {}     //Деструктор класса
    void getNumber();  // методы класса
    void getKem_Vydan();
    void getDaTA();
    void getPropiska();
    void StepUstarDoc(int TecDat);

};   // точка с запятой нужна!
```

```
//Описание методов класса
void Pasport::getNumber()
{
    printf("\n Number 45 00 %d", Number);
}
void Pasport::getKem_Vydan()
{
    printf("\n Kem_Vydan= %s", Kem_Vydan);
}
void Pasport::getDaTA()
{
    printf("\n DATA vydachi god= %d", data[0]);
    printf("  mesac= %d", data[1]);
    printf(" chislo= %d", data[2]);
}
void Pasport::getPropiska()
{
    printf("\n propiska= %s", propiska);
}

void Pasport::StepUstarDoc(int TecDat)
{
    printf("\n vremya ustarevanya god= %d", TecDat - data[0]);
}

int main()
{
    int D;
    D = 2016;   //грубое задание текущей даты
    Pasport PassIvanov;  //создание экземпляра класса и вызов конструктора
    PassIvanov.getNumber();  //Вызов методов для вывода переменных класса
    PassIvanov.getKem_Vydan();
    PassIvanov.getDaTA();
    PassIvanov.getPropiska();
    PassIvanov.StepUstarDoc(D);
    _getch();
    return 0;
}
```

Ключевые черты ООП:

1. Инкапсуляция – это определение классов – пользовательских типов данных, объединяющих свое содержимое в единый тип и реализующих некоторые операции или методы над ним. Классы обычно являются основой модульности, инкапсуляции и абстракции данных в языках ООП.

2. Наследование – способ определения нового типа, когда последний наследует элементы (свойства и методы) существующего, модифицируя или расширяя их.

3. Полиморфизм – позволяет единообразно ссылаться на объекты различных классов (обычно внутри некоторой иерархии). Это делает классы еще удобнее и облегчает расширение и поддержку программ, основанных на них.

Наследование (inheritance) – это процесс, посредством которого один объект может приобретать свойства другого. Точнее, объект может наследовать основные свойства другого объекта и добавлять к ним черты, характерные только для него. Наследование является важным, поскольку оно позволяет поддерживать концепцию иерархии классов (hierarchical classification). Применение иерархии классов делает управляемыми большие потоки информации. Например, описание жилого дома. Дом – это часть общего класса, называемого строением. С другой стороны, строение – это часть более общего класса – конструкции, который является частью еще более общего класса объектов, который можно назвать созданием рук человека. В каждом случае порожденный класс наследует все связанные с родителем качества и добавляет к ним свои собственные определяющие характеристики. Без использования иерархии классов для каждого объекта пришлось бы задать все характеристики, которые бы исчерпывающе его определяли. Однако при использовании наследования можно описать объект путем определения того общего класса (или классов), к которому он относится, с теми специальными чертами, которые делают объект уникальным. Наследование играет очень важную роль в ООП.

C++ позволяет наследоваться по разным зонам видимости.

```
class B : public A{ //public наследование
};
class C : protected A{ //protected наследование
};
class Z : private A{ //private наследование
}.
```

Полиморфизм (от греч. polymorphos) – это свойство, которое позволяет одно и то же имя использовать для решения двух или более схожих, но технически разных задач. Целью полиморфизма применительно к объектно-ориентированному программированию является использование одного имени для задания общих для класса действий. Выполнение каждого конкретного действия будет определяться типом данных. Например, для языка Си, в котором полиморфизм поддерживается недостаточно, нахождение абсолютной величины числа требует трех различных функций: abs(), labs() и fabs(). Эти функции подсчитывают и возвращают абсолютную величину целых, длинных целых и чисел с плавающей точкой соответственно. В C++ каждая из этих функций может быть названа abs(). Тип данных, который используется при вызове функции, определяет, какая конкретная версия функции действительно выполняется. В C++ можно использовать одно имя функции для множества различных действий. Это называется *перегрузкой функций* (function overloading).

В более общем смысле, концепцией полиморфизма является идея «один интерфейс, множество методов». Это означает, что можно создать общий интерфейс для группы близких по смыслу действий. Преимущество полиморфизма в том, что он помогает снижать сложность программ, разрешая использование того же интерфейса для задания единого класса действий. Выбор конкретного действия в зависимости от ситуации возлагается на компилятор.

Полиморфизм может применяться и к операторам. Фактически во всех языках программирования ограниченно применяется полиморфизм, например в арифметических операторах. Так, в Си символ + используется для складывания целых, длинных целых, символьных переменных и чисел с плавающей точкой. В этом случае компилятор автоматически определяет, какой тип арифметики требуется. В C++ вы можете применить эту концепцию и к

другим типам данных, заданным вами. Такой тип полиморфизма называется *перегрузкой операторов* (operator overloading).

Ключевым в понимании полиморфизма является то, что он позволяет вам манипулировать объектами различной степени сложности путем создания общего для них стандартного интерфейса для реализации похожих действий.

Полиморфизм – возможность объектов с одинаковой спецификацией иметь различную реализацию. Полиморфизм реализуется с помощью наследования классов и виртуальных функций. Класс-потомок наследует сигнатуры методов класса-родителя, а реализация в результате переопределения метода этих методов может быть другой, соответствующей специфике класса-потомка.

5.9. Инструменты автоматизации для работы с классами. Интегрированные среды разработчика

Программирование в среде Visual Studio Windows Forms C++, C#.

Windows Forms – самостоятельная интегральная среда разработчика, которая поддерживается библиотеками Microsoft .NET Framework.

Windows Forms не зависит от языка разработки и позволяет создавать ПО как на C#, C++, так и на VB.Net, J# и др. По сути это оболочка, позволяющая более удобно управлять функциями Win-API.

Средства Windows Forms обеспечивают быструю разработку пользовательского интерфейса, поскольку он создается из стандартных элементов, а соответствующий программный код генерируется автоматически. Затем требуется лишь доработать сгенерированный код, чтобы добиться необходимой функциональности.

Применяя Windows Forms, можно построить удобный пользовательский интерфейс, в том числе главное окно приложения,

выбирая подходящие элементы управления, с которыми взаимодействует пользователь.

Windows Forms ориентировано на реакцию на событие. Событие – такое свойство объекта, которое описывает воздействие на программу извне (от кнопки, порта, таймера, клавиатуры и т. д.).

Основные элементы Windows Forms

Форма – это окно, служащее основой окна приложения или диалогового окна, в которое можно добавлять другие компоненты, предназначенные для взаимодействия с пользователем.

На форме размещаются более мелкие элементы управления – «кнопки», «блоки», «редакторы», «прогресс-бары» и другие графические элементы управления, иногда их называют компоненты, элементы и т. д.

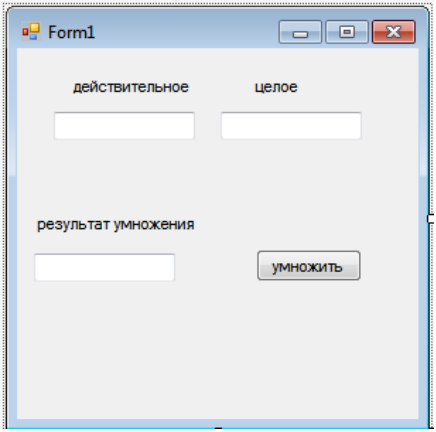
Формы добавляются при помощи специального меню. Элементы или компоненты размещаются на палитре компонент или на панели элементов.

Все формы и компоненты являются объектами и соответствуют классам в библиотеке классов. Каждый такой объект имеет доступные пользователю входные и выходные параметры, при помощи которых он может перенастраивать объекты под требования своей задачи.

Эти параметры делятся на две группы. Первая – свойства, описывают статическое состояние объекта, и события – изменение состояния объекта в результате управляющего воздействия.

Каждое событие порождает обработчик – встроенная в класс пустая функция (метод), которую дописывает пользователь. Пользователь туда вписывает код, который должен выполняться, если данное событие произойдет.

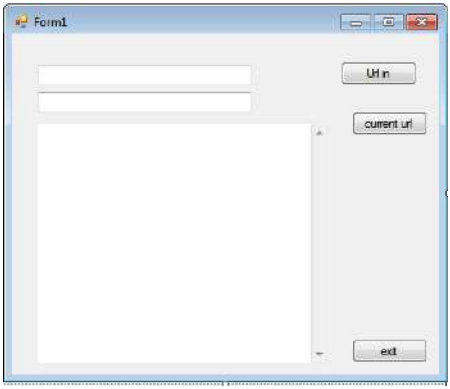
Пример элементов, необходимых для ввода чисел:




```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace WindowsFormsApp3
{
    Ссылок: 2
    public partial class Form1 : Form
    {
        ссылка: 1
        public Form1()
        {
            InitializeComponent();
        }
            ссылка: 1
            private void button1_Click(object sender, EventArgs e)
            {
                double a, c; int b;
                a = double.Parse(textBox1.Text);
                b = int.Parse(textBox2.Text);
                c = a * b;
                textBox3.Text = c.ToString();
            }
    }
}
```

Пример простейшего веб-браузера:




```
namespace WindowsFormsApp3
{
    Ссылок: 2
    public partial class Form1 : Form
    {
        ссылка: 1
        public Form1()
        {
            InitializeComponent();
        }
        ссылка: 1
        private void button1_Click_1(object sender, EventArgs e)
        {
            Close();
        }
        ссылка: 1
        private void textBox1_TextChanged(object sender, EventArgs e)
        {

        }
        ссылка: 1
        private void button2_Click(object sender, EventArgs e)
        {
            webBrowser1.Navigate(new Uri("http://" + textBox1.Text));
        }
        ссылка: 1
        private void button3_Click(object sender, EventArgs e)
        {
            textBox2.Text = webBrowser1.Url.ToString();
        }
        ссылка: 1
        private void webBrowser1_DocumentCompleted(object sender,
            WebBrowserDocumentCompletedEventArgs e)
        {
        }
    }
```

ГЛАВА 6
Стиль программирования

Программы должны составляться так, чтобы их могли прочитать в первую очередь люди, а не машины, т. е. удобочитаемость программ – существенна, поэтому имеет смысл говорить о стандарте стиля программ.

Легко читаемая программа создает впечатление, что ее автор хорошо знал, что делал.

Программа должна передавать логику и структуру алгоритма настолько, насколько это возможно.

Стиль программирования – набор приемов или методов программирования, которые используют опытные программисты, чтобы получить надежные, эффективные, удобные для применения и легко читаемые программы.

Общие рекомендации просты, не требуют для выполнения особых усилий и направлены на более четкую структуризацию программы. Суть их заключается в следующем.

Правило стандартизации стиля: если существует более одного способа сделать что-либо и выбор произвольный, остановитесь на одном способе и всегда его придерживайтесь.

Легче избежать путаницы, придерживаясь стандартных приемов, хороший набор стандартных приемов позволяет сконцентрировать внимание на главной задаче.

Невозможно выработать единого промышленного стандарта для всех программ, но в пределах единого проекта, задачи, разработки вполне возможно.

Комментарии: желательность комментариев очевидна. Программы с пояснительными комментариями значительно легче отсле-

живать. Читая чужую программу, программисты немало времени отслеживают логику чужой программы, т. е. не комментируемая программа – свидетельство дилетантского похода.

Комментарии следует писать в процессе написания программы. Бессмысленно вставлять комментарии после того, как программа завершена, так как цель комментариев – облегчить понимание программы.

Комментарии бывают трех типов:

1. Вводные.
2. Оглавление.
3. Пояснительные.

Вводные комментарии. Каждая программа, подпрограмма или процедура должны начинаться с комментариев, поясняющих, что она делает. Минимальная информация, содержащаяся в вводных комментариях, должна включать следующие пункты:

1. Назначение программы.
2. Указания по вызову программы и ее использованию.
3. Список и назначение основных переменных или массивов.
4. Указание по вводу-выводу. Список всех файлов.
5. Список используемых подпрограмм.
6. Назначение применяемых математических методов.
7. Сведения о времени выполнения программы (иногда).
8. Требуемый объем памяти.
9. Специальные указания оператору.
10. Сведения об авторе.
11. Дату написания программы.

Пояснительный комментарий. Пояснениями нужно сопровождать те части программы, которые трудно понять без комментариев. Следует сопровождать комментариями те действия, которые, с вашей точки зрения, могут быть не совсем понятны другому. Комментарии не должны объяснять синтаксис языка

программирования, а должны раскрывать логику программы или цель конкретного действия. Если логика программы понятна только на основании прочтения комментариев, это качественные комментарии.

Комментарии легче читаются, если они отделяются пустыми строками (до и после комментариев). Дополнительный метод выделения комментариев – заключение их в прямоугольник из специальных символов (звездочки, слеши, скобки и т. д.). Располагаются комментарии таким образом, чтобы это не делало программу менее наглядной. Неправильные комментарии хуже, чем их отсутствие.

Выбор имен переменных, файлов: имена переменных должны наилучшим образом определять те величины, которые они представляют. Если ограничения на длину имени отсутствуют, используйте имена постольку длинные, поскольку это нужно, но не длиннее, чем необходимо.

Имена переменных не должны совпадать со служебными словами.

Избегайте схожих по виду имен и подобных по написанию символов.

Различие имен должно быть всегда явно ощутимым.

Когда имена содержат лишнюю информацию, это тоже плохо, так как программа расширяется, становится громоздкой.

При выборе имен переменных старайтесь установить, что обозначает эта переменная на естественном языке, и выбирайте наиболее подходящее слово.

Имена файлов. Использование одних и тех же имен для одинаковых файлов в различных программах обеспечивает их быструю идентификацию (узнаваемость).

Иногда использование префикса помогает определять, какие поля связаны логически.

При выборе имен записей используйте имена, ориентированные на содержание записи, а не на конкретное задание.

При написании программы целесообразно использовать систему отступов. И хотя отступы не оказывают влияния на логику программы, они существенно улучшают ее читаемость.

ГЛАВА 7
Тестирование программных средств и применение отладчиков

7.1. Понятие надежной программы. Методы ручного тестирования

Продуктом технологии программирования является программное средство (ПС), содержащее программы, выполняющие требуемые функции.

Под программой часто понимают «правильную» программу, т. е. не содержащую ошибок.

Будем считать, что в программе имеется ошибка, если она не выполняет того, что разумно ожидать от нее пользователю. «Разумное ожидание» пользователя формируется на основании документации по применению этой программы. Следовательно, понятие ошибки в программе является существенно неформальным.

Недоказуемость правильности путем тестирования

В связи с тем, что задание на ПС обычно формулируется неформально, а также из-за неформализованности понятия ошибки в ПС нельзя доказать формальными методами (математически) правильность ПС. Нельзя показать правильность ПС и тестированием. Тестирование может лишь продемонстрировать наличие в ПС ошибки. Поэтому понятие правильной ПС неконструктивно в том смысле, что после окончания работы над созданием ПС мы не сможем убедиться, что достигли цели.

Надежность программного средства

Альтернативой правильного ПС является надежное ПС. Надежность ПС – это его способность безотказно выполнять определенные функции при заданных условиях в течение заданного периода времени с достаточно большой вероятностью. При

этом под отказом в ПС понимают проявление в нем ошибки. Таким образом, надежное ПС не исключает наличия в нем ошибок – важно лишь, чтобы эти ошибки при практическом применении этого ПС в заданных условиях проявлялись достаточно редко. Убедиться, что ПС обладает таким свойством, можно при его испытании путем тестирования, а также при практическом применении. Следовательно, фактически мы можем разрабатывать лишь надежные, а не правильные ПС.

Оценка степени надежности программных средств

Разрабатываемое ПС может обладать различной степенью надежности.

Степень надежности можно характеризовать вероятностью работы ПС без отказа в течение определенного периода времени.

При оценке степени надежности ПС следует также учитывать последствия каждого отказа. Некоторые ошибки в ПС могут вызывать лишь неудобства при его применении, тогда как другие ошибки могут иметь катастрофические последствия, например угрожать человеческой жизни. Поэтому для оценки надежности ПС иногда используют дополнительные показатели, учитывающие стоимость (вред) для пользователя каждого отказа.

Ручной контроль обычно используют на ранних этапах разработки. Все проектные решения, принятые на том или ином этапе, должны анализироваться с точки зрения их правильности и целесообразности как можно раньше, пока их можно легко пересмотреть. Поскольку возможность практической проверки подобных решений на ранних этапах разработки отсутствует, большое значение имеет их обсуждение, которое проводят в разных формах.

Различают статический и динамический подходы к ручному контролю. При статическом подходе анализируют структуру, управляющие и информационные связи программы, ее входные и

выходные данные. При динамическом – выполняют ручное тестирование, т. е. вручную моделируют процесс выполнения программы на заданных исходных данных.

Исходными данными для таких проверок являются: техническое задание, спецификации, структурная и функциональная схемы программного продукта, схемы отдельных компонентов и т. д., а для более поздних этапов – алгоритмы и тексты программ, а также тестовые наборы.

Доказано, что ручной контроль способствует существенному увеличению производительности и повышению надежности программ и с его помощью можно находить от 30 до 70 % ошибок логического проектирования и кодирования. Следовательно, один или несколько методов ручного контроля обязательно должны использоваться в каждом программном проекте.

Основные методы ручного контроля:

– инспекции исходного текста;
– сквозные просмотры;
– проверка за столом;
– оценки программ.

Инспекции исходного текста. Инспекции исходного текста представляют собой набор процедур и приемов обнаружения ошибок при изучении текста группой специалистов. В эту группу входят автор программы, проектировщик, специалист по тестированию и координатор – компетентный программист, но не автор программы. Общая процедура инспекции предполагает следующие операции:

– участникам группы заранее выдается листинг программы и спецификация на нее;

– программист рассказывает о логике работы программы и отвечает на вопросы инспекторов;

– программа анализируется по списку вопросов для выявления исторически сложившихся общих ошибок программирования.

Список вопросов для инспекций исходного текста зависит как от используемого языка программирования, так и от специфики разрабатываемого программного средства. В качестве примера ниже приведен список вопросов, который можно использовать при анализе правильности программ:

1. Контроль обращений к данным.

Все ли переменные инициализированы?

Не превышены ли максимальные (или реальные) размеры массивов и строк?

Не перепутаны ли строки со столбцами при работе с матрицами?

Присутствуют ли переменные со сходными именами?

Используются ли файлы? Если да, то при вводе из файла проверяется ли завершение файла?

Соответствуют ли типы записываемых и читаемых значений?

Использованы ли нетипизированные переменные, открытые массивы, динамическая память? Если да, то соответствуют ли типы переменных при «наложении» формата? Не выходят ли индексы за границы массивов?

2. Контроль вычислений.

Правильно ли записаны выражения (порядок следования операторов)?

Корректно ли выполнены вычисления над неарифметическими переменными?

Корректно ли выполнены вычисления с переменными различных типов (в том числе с использованием целочисленной арифметики)?

Возможно ли переполнение разрядной сетки или ситуация машинного нуля?

Соответствуют ли вычисления заданным требованиям точности?

Присутствуют ли сравнения переменных различных типов?

3. Контроль передачи управления.

Будут ли корректно завершены циклы?

Будет ли завершена программа?

Существуют ли циклы, которые не будут выполняться из-за нарушения условия входа? Корректно ли продолжатся вычисления?

Существуют ли поисковые циклы? Корректно ли отрабатываются ситуации «элемент найден» и «элемент не найден»?

4. Контроль межмодульных интерфейсов.

Соответствуют ли списки параметров и аргументов по порядку, типу единицам измерения?

Не изменяет ли подпрограмма аргументов, которые не должны изменяться?

Не происходят ли нарушения области действия глобальных и локальных переменных с одинаковыми именами?

Кроме непосредственного обнаружения ошибок, результаты инспектирования позволяют программисту увидеть другие сделанные им ошибки, получить возможность оценить свой стиль программирования, выбор алгоритмов и методов тестирования. Инспекция является способом раннего выявления частей программы, с большей вероятностью содержащих ошибки, что позволяет при тестировании уделить внимание именно этим частям.

Сквозные просмотры. Сквозной просмотр, как и инспекция, представляет собой набор способов обнаружения ошибок, осуществляемых группой лиц, просматривающих текст программы. Такой просмотр имеет много общего с процессом инспектирования, но отличается процедурой и методами обнаружения ошибок. Группа по выполнению сквозного контроля состоит из трех-пяти

человек: председатель или координатор, секретарь, фиксирующий все ошибки, специалист по тестированию, программист и независимый эксперт.

Сквозной просмотр предполагает выполнение следующих процедур:

– участникам группы заранее выдают листинг программы и спецификацию на нее;

– участникам заседания предлагают несколько тестов;

– участники заседания мысленно выполняют каждый тест в соответствии с логикой программы, при этом состояние программы (значения переменных) отслеживается на бумаге или доске;

– при необходимости программисту задают вопросы о логике проектирования и принятых допущениях.

В большинстве сквозных просмотров при выполнении самих тестов находят меньше ошибок, чем при опросе программиста.

Проверка за столом. Исторически данный метод ручного тестирования появился первым, так как он не требует наличия группы специалистов. Это – проверка исходного текста или сквозные просмотры, выполняемые одним человеком, который читает текст программы, проверяет его на наличие возможных ошибок по специальному списку часто встречающихся ошибок и «пропускает» через программу тестовые данные. Исходя из принципов тестирования проверку за столом должен проводить человек, не являющийся автором программы. Метод наименее результативен, так как проверка представляет собой полностью неупорядоченный процесс, при ней отсутствует обмен мнениями и здоровая конкуренция.

Оценка программ. Этот метод непосредственно не связан с тестированием, но его использование также улучшает качество программирования. Его используют для анонимной оценки программы в терминах ее общего качества, простоты эксплуатации и

ясности. Цель метода – обеспечить сравнительно объективную оценку и самооценку программистов.

Такая оценка выполняется следующим образом. Выбирается программист, который должен выполнять обязанности администратора процесса. Администратор набирает группу от шести до двадцати участников, которые должны заниматься разработкой сходных программ. Каждому участнику предлагается представить для рассмотрения две программы, с их точки зрения – наилучшую и наихудшую. Отобранные программы случайным образом распределяются между участниками. Им дают по четыре программы – две наилучшие и две наихудшие, но не говорят, какие программы плохие, а какие – хорошие. Программист просматривает эти программы и заполняет анкету, в которой оценивает их качество по семибалльной шкале. После этого результаты оценки сверяют, а проверяющий дает общий комментарий и рекомендации по улучшению программ.

7.2. Отладчики, отладчик, встроенный в интегральную систему разработчика

Отладка – это процесс локализации и исправления ошибок, обнаруженных при реализации и тестировании программного обеспечения.

Локализацией называют процесс определения строки программы, выполнение которого вызвало нарушение нормального вычислительного процесса. Для исправления ошибки нужно определить ее причину, т. е. определить фрагмент, содержащий ошибку. Причины ошибок могут быть как очевидные, так и очень глубоко скрытые.

Рис. 18. Классификация ошибок по этапу обработки программы

В соответствии с этапом обработки, на котором проявляются ошибки, различают (рис. 18):

– *синтаксические ошибки* – ошибки, фиксируемые компилятором (транслятором, интерпретатором) при выполнении синтаксического и частично семантического анализа программы;

– *ошибки компоновки* – ошибки, обнаруженные компоновщиком (редактором связей) при объединении модулей программы;

– *ошибки выполнения* – ошибки, обнаруженные операционной системой, аппаратными средствами или пользователем при выполнении программы.

Синтаксические ошибки относят к группе самых простых, так как синтаксис языка, как правило, строго формализован и ошибки сопровождаются развернутым комментарием с указанием их местоположения. Определение причин таких ошибок труда не составляет, и даже при нечетком знании правил языка за несколько прогонов удается удалить все ошибки данного типа.

Ошибки компоновки, как следует из названия, связаны с проблемами, обнаруженными при разрешении внешних ссылок. Например, предусмотрено обращение к подпрограмме другого модуля, а при объединении модулей данная подпрограмма не найдена или не стыкуются списки параметров. В большинстве

случаев ошибки такого рода также удается быстро локализовать и устранить.

Ошибки выполнения относятся к самой непредсказуемой группе. Прежде всего, они могут иметь разную природу и, соответственно, по-разному проявляться. Часть ошибок обнаруживается и документируется операционной системой. Выделяют четыре способа проявления таких ошибок:

1) появление сообщения об ошибке, зафиксированной схемами контроля выполнения машинных команд, например переполнении разрядной сетки, ситуации «деление на ноль», нарушении адресации и т. д.;

2) появление сообщения об ошибке, обнаруженной операционной системой, например нарушении защиты памяти, попытке записи на устройства, защищенные от записи, отсутствии файла с заданным именем и т. д.;

3) «зависание» компьютера, как простое, когда удается завершить программу без перезагрузки операционной системы, так и «тяжелое», когда для продолжения работы необходима перезагрузка;

4) несовпадение полученных результатов с ожидаемыми.

Причины ошибок выполнения очень разнообразны, а потому и локализация может оказаться крайне сложной. Все возможные причины ошибок можно разделить на группы:

– неверное определение исходных данных,
– логические ошибки,
– накопление погрешностей результатов вычислений.

Неверное определение исходных данных происходит, если возникают любые ошибки при выполнении операций ввода-вывода: ошибки передачи, ошибки преобразования, ошибки перезаписи и ошибки данных.

Логические ошибки имеют разную природу. Так, они могут следовать из ошибок, допущенных при проектировании, например при выборе методов, разработке алгоритмов или определении структуры классов, а могут быть непосредственно внесены при кодировании модуля.

К последней группе относят:

– *ошибки некорректного использования переменных*, например неудачный выбор типов данных, использование переменных до их инициализации, использование индексов, выходящих за границы определения массивов, нарушения соответствия типов данных при использовании явного или неявного переопределения типа данных, расположенных в памяти при использовании нетипизированных переменных, открытых массивов, объединений, динамической памяти, адресной арифметики и т. д.;

– *ошибки вычислений*, например некорректные вычисления над неарифметическими переменными, некорректное использование целочисленной арифметики, некорректное преобразование типов данных в процессе вычислений, ошибки, связанные с незнанием приоритетов выполнения операций для арифметических и логических выражений, и т. д.;

– *ошибки межмодульного интерфейса*, например игнорирование системных соглашений, нарушение типов и последовательности при передаче параметров, несоблюдение единства единиц измерения формальных и фактических параметров, нарушение области действия локальных и глобальных переменных;

– *другие ошибки кодирования*, например неправильная реализация логики программы при кодировании, игнорирование особенностей или ограничений конкретного языка программирования.

Накопление погрешностей результатов числовых вычислений возникает, например, при некорректном отбрасывании дроб-

ных цифр чисел, некорректном использовании приближенных методов вычислений, игнорировании ограничения разрядной сетки представления вещественных чисел в ЭВМ и т. д.

Все указанные выше причины возникновения ошибок следует иметь в виду в процессе отладки.

Интегрированные средства отладки. Большинство современных сред программирования (Delphi, Builder C++, Visual Studio и т. д.) включают средства отладки, которые обеспечивают максимальную эффективность. Они позволяют:

– выполнять программу по шагам, причем как с заходом в подпрограммы, так и выполняя их полностью;

– предусматривать точки останова;

– выполнять программу до оператора, указанного курсором;

– отображать содержимое любых переменных при пошаговом выполнении;

– отслеживать поток сообщений и т. д.

Применять интегрированные средства в рамках среды достаточно просто. Используют разные приемы в зависимости от проявлений ошибки. Если получено сообщение об ошибке, то сначала уточняют, при выполнении какого оператора программы оно получено. Для этого устанавливают точку останова в начало фрагмента, в котором проявляется ошибка, и выполняют операторы в пошаговом режиме до проявления ошибки.

Аналогично поступают при «зависании» компьютера.

Если получены неправильные результаты, то локализовать ошибку обычно существенно сложнее. В этом случае сначала определяют фрагмент, при выполнении которого получаются неправильные результаты. Для этого последовательно проверяют интересующие значения в узловых точках. Обнаружив значения,

отличающиеся от ожидаемых, по шагам трассируют соответствующий фрагмент до выявления оператора, выполнение которого дает неверный результат.

Отладка с использованием независимых отладчиков. При отладке программ иногда используют специальные программы – отладчики, которые позволяют выполнить любой фрагмент программы в пошаговом режиме, проверить содержимое интересующих программиста переменных. Как правило, такие отладчики позволяют отлаживать программу только в машинных командах, представленных в шестнадцатеричном коде.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Аблязов, Р. З. Программирование на ассемблере на платформе x86-64 / Р. З. Аблязов. – Саратов : Профобразование, 2019 // URL: http://www.iprbookshop.ru/88005.

2. Агапов, В. П. Основы программирования на языке C# : учебное пособие / В. П. Агапов. – М. : Московский государственный строительный университет, ЭБС АСВ, 2012. – URL: http://www.iprbookshop.ru/16366.

3. Ашарина, И. В. Объектно-ориентированное программирование в C++ : учебное пособие / И. В. Ашарина. – М. : Горячая линия – Телеком, 2012. – URL: http://www.iprbookshop.ru/12008.

4. Белов, А. В. Программирование микроконтроллеров для начинающих и не только / А. В. Белов. – СПб. : Наука и техника, 2016. – URL: http://www.iprbookshop.ru/60657.html.

5. Давыдова, Н. А. Программирование : учебное пособие / Н. А. Давыдова, Е. В. Боровская. – М. : БИНОМ ; Лаборатория знаний, 2012.

6. Ишков, А. Д. Оформление заявок на государственную регистрацию программ для электронных вычислительных машин и баз данных : справочное пособие / А. Д. Ишков, А. В. Степанов. – М. : Московский государственный строительный университет, ЭБС АСВ, 2012. – URL: http://www.iprbookshop.ru/16361.

7. Кауфман, В. Ш. Языки программирования. Концепции и принципы / В. Ш. Кауфман. – Саратов : Профобразование, 2019. – URL: http://www.iprbookshop.ru/88014.

8. Кирнос, В. Н. Информатика 2. Основы алгоритмизации и программирования на языке C++ : учебно-методическое пособие / В. Н. Кирнос. – Томск : Эль Контент, Томский государственный университет систем управления и радиоэлектроники, 2013. – URL: http://www.iprbookshop.ru/14011.

9. Курипта, О. В. Основы программирования и алгоритмизации : практикум / О. В. Курипта, О. В. Минакова, Д. К. Проскурин. – Воронеж : Воронежский государственный архитектурно-строительный университет, ЭБС АСВ, 2015. – URL: http://www.iprbookshop.ru/59123.html.

10. Мейер, Б. Объектно-ориентированное программирование и программная инженерия / Б. Мейер. – М. : Интернет-Университет информационных технологий (ИНТУИТ), Ай Пи Эр Медиа, 2019. – URL: http://www.iprbookshop.ru/79706.

11. Подбельский, В. В. Язык Си#. Базовый курс [Электронный ресурс] : учебное пособие / В. В. Подбельский. Электрон. текстовые данные. – М. : Финансы и статистика, 2011. – Режим доступа: http://www.iprbookshop.ru/18866. ЭБС «IPRbooks».

12. Туральчук, К. А. Параллельное программирование с помощью языка C# / К. А. Туральчук. – М. : Интернет-Университет информационных технологий (ИНТУ-ИТ), 2016. – URL: http://www.iprbookshop.ru/39560.html.

13. Шарп, Дж. Microsoft Visual C# : подробное руководство / Дж. Шарп ; пер. с англ. Н. Вильчинский. – 8-е изд. – СПб. : Питер, 2017.

Учебное издание

Пименова Оксана Владимировна,
кандидат технических наук

Клочкова Екатерина Николаевна

Аветисян Карэн Рафаелович

Плотников Герман Геннадьевич,
кандидат технических наук

Мельцева Ирина Сергеевна

Тычкова Анастасия Олеговна

ПРОГРАММИРОВАНИЕ: ЯЗЫКИ, МЕТОДЫ И ТЕХНОЛОГИИ

Редактор *Чамарова Н. В.*
Корректор *Лосева О. С.*
Компьютерная верстка *Чамарова Н. В.*

Московский университет МВД России имени В.Я. Кикотя
117997, г. Москва, ул. Академика Волгина, д. 12

Подписано в печать 20.09.2021	Формат 60×84 1/16	Тираж	282 экз.
Заказ № 646	Цена договорная	Объем	4,47 уч.-изд. л. 8,89 усл. печ. л.