ГОУВПО
«Воронежский государственный технический университет»

Кафедра «Автоматизированные и вычислительные системы»

50 - 2008

ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНЫХ СИСТЕМ

МЕТОДИЧЕСКИЕ УКАЗАНИЯ
к выполнению контрольной работы
по дисциплине «Технология программирования»
для студентов специальности 230101
«Вычислительные машины, комплексы, системы и сети»
заочной и заочной сокращенной форм обучения

Воронеж 2008

Составители: канд. техн. наук Н.И. Гребенникова,
канд. техн. наук Т.И. Сергеева

УДК 681.3

Технология разработки программных систем: методические указания к выполнению контрольной работы по дисциплине «Технология программирования» для студентов специальности 230101 «Вычислительные машины, комплексы, системы и сети» заочной и заочной сокращенной форм обучения / ГОУВПО «Воронежский государственный технический университет»; сост. Н.И. Гребенникова, Т.И. Сергеева. Воронеж, 2008. 41 с.

Методические указания содержат теоретические и практические сведения о выполнении основных этапов разработки программных средств – кодирования и тестирования.

Предназначены для студентов второго курса.

Табл. 3. Ил. 7. Библиогр.: 4 назв.

Рецензент канд. техн. наук, доц. А.А. Кисурин

Ответственный за выпуск зав. кафедрой д-р техн. наук, проф. С.Л. Подвальный

Печатается по решению редакционно-издательского совета Воронежского государственного технического университета

ВВЕДЕНИЕ

Выполнение контрольной работы является одной из важных форм самостоятельной работы студентов при изучении курса «Технология программирования».

1. МЕТОДИЧЕСКИЕ РЕКОМЕНДАЦИИ

1.1. Цели и задачи контрольной работы

Цели и задачи контрольной работы. В процессе выполнения контрольной работы студенты:

углубленно изучают наиболее важные вопросы, относящиеся к курсу технологии программирования, учатся всесторонне анализировать процесс разработки программных систем и делать на основе этого правильные, научно обоснованные теоретические и практические выводы;

приобретают опыт грамотного анализа научно-технической литературы, использования стандартов, справочников технической документации по математическому и программному обеспечению;

приобретают навыки научно-исследовательской и проектно-конструкторской работы в области исследования и разработки программных систем;

овладевают навыками определения основных этапов и работ, выполняемых при проектировании программных средств.

1.2. Требования к уровню выполнения контрольной работы

При выполнении контрольной работы необходимо:

проблемное изложение теоретического материала;

соблюдение логической последовательности в изложении материала;

четкость структуры работы;

убедительность аргументаций;

краткость и четкость формулировок;

грамотное изложение;

оформление работы в соответствие с принятыми стандартами.

1.3. Структура контрольной работы

По своей структуре контрольная работа должна включать следующие разделы: титульный лист; задания; содержание; введение; разделы и подразделы основной части; заключение; список литературы; приложение.

Оформление работы.

Выполняется на бумаге размером А4, ориентация - книжная, верхнее поле - 25 мм, нижнее поле - 25 мм, левое поле - 30 мм, правое поле 15 мм: в печатном виде (шрифт - Times New Roman Cyr, размер шрифта - 14, межстрочный интервал - одинарный). К работе рекомендуется приложить дискету с текстами разработанных программ.

На титульном листе должны быть указаны ВУЗ, факультет, дисциплина, группа, Ф.И.О. студента, Ф.И.О. преподавателя, год, город.

Объем теоретической части контрольной работы должен составлять 5-10 страниц машинописного текста без учета приложений.

Условия практических заданий приводятся полностью. Ответы излагаются четко и аккуратно.

Работа подписывается автором, указывается дата ее выполнения. Страницы необходимо пронумеровать и обязательно оставить поля для возможных замечаний рецензента.

2. РАЗРАБОТКА ПРОГРАММНЫХ СИСТЕМ

2.1. Стиль программирования

Под **стилем программирования** понимают набор приемов и методов, применяемых для получения правильных, эффективных, удобных и легко читаемых программ.

Один из приемов программирования - **включение комментариев** в программу. Комментарии не должны перефразировать программу, а должны давать дополнительную информацию, поясняющую смысл действий, выраженных операторами. Комментарии располагаются так, чтобы программа выглядела более наглядно. *Комментарии бывают вводными, блоковыми и операторными*.

Вводные комментарии располагаются перед текстом программы, функции, процедуры и включают в себя:

назначение программы;

указания по вызову программы и ее использования;

список и назначение основных переменных и массивов;

указания по обмену с внешними устройствами и файлами;

список используемых процедур и функций;

наименование применяемых математических методов, ссылки на литературные источники, в которых содержится их описание;

сведения о времени выполнения программы;

специальные указания операторам;

требуемый объем памяти;

сведения об авторах, дату написания или последней версии программы.

Блоковые комментарии сопровождают отдельные модули и блоки программы. *Операторный комментарий* описывает назначение отдельного оператора (комментарий с минимальной информацией).

Пропуск строк используют для выделения логически связанных блоков программы. Для удобства и легкости чтения

программы применяют *отступы*. Принято использовать не более 2-3 уровней вложенности, шаг отступа полагать равным 3-5 позициям. Последний уровень не должен выходить за рамки экрана.

В арифметических выражениях перенос выполняется после знака операции, в списках - после разделителя.

Пробелы служат для выделения всех конструкций программы (ключевых слов, идентификаторов, отдельных операторов и т.д.).

Идентификаторы формируются по следующим правилам: осмысленность (имя напоминает соответствующую переменную в исходной программе); включение максимальной информации о данном объекте; исключение в аббревиатуре гласных из слова; использование префиксов или постфиксов.

В одной строке обычно располагается 1 оператор.

Принято упорядочивать списки: последовательность элементов должна иметь определенную логику.

Правильно оформленная программа должна быть составлена с учетом следующих нормативов:

размер модуля 10-15 непустых строк,
размер имен 5-10 символов,
комментарии 15-25 % от размеров модуля,
отступы 24-48 %,
пустые строки 15-30 %,
длина строки 15-25 символов,
пробелы в строке 4-10,
поименованные константы 15-25 %.

Пример комментирования функции:

```
{Вычисление тестовой функции f=x*x*x/(1+x+x*x).}
{ Параметры:                                    }
{       x - аргумент функции.                   }
{ Возвращаемые значения:                        }
{       f - вычисленная функция.                }
{ Используемые процедуры и функции:    нет.     }
```

```
{ Пример расчета функции в точке 0.1:   f(0.1);     }
{ Автор: Иванов А.С., 12.02.2007г.                   }
 function f(x: real) : real;
 begin
 f:= sqr(x)*x/(1+x+sqr(x));
 end;
```

2.2. Модульное программирование

Модульное программирование - это процесс разработки программ, реализующих логические части алгоритма всей задачи. Все модули программируются независимо друг от друга, что позволяет правильно организовать процесс разработки всей системы.

Принципы модульного программирования:

каждый модуль не зависит от его контекстного использования;

модули формируются в большие программы без предварительных знаний о внутренней работе модулей.

Для обеспечения *межмодульной независимости* требуются:

независимость модуля от источника входных данных;

независимость модуля от места назначения выходных данных;

независимость модуля от предыстории;

локализация объектов внутри исполняемого модуля;

определение для каждого модуля алгоритма решения задачи, набора и области допустимых входных и выходных значений.

2.3. Структурное кодирование

Структурное кодирование - это метод написания хорошо структурированной программы, который позволяет получать программный продукт более удобный для тестирования, модификаций и использования.

После декомпозиции и определения множества модулей, формирующих проектируемую систему, начинается кодирование сверху вниз: в первую очередь кодируется модуль самого верхнего уровня (главная программа). Это позволяет проводить тестирование параллельно с кодированием модулей.

Для тестирования сверху вниз в программу включаются "заглушки". "Заглушки" реализуются в виде фиктивных модулей и выполняют простую обработку данных до тех пор, пока не будет закодирован реальный модуль.

2.4. Защитное программирование

Это стиль написания программ, при котором выявленные ошибки легко обнаруживаются и идентифицируются программистом. Предполагается включение в программу средств исключения ошибок. Необходимо руководствоваться следующими принципами.

1). *Общее недоверие*. В каждом модуле входные данные подлежат проверке по типу, по области допустимых значений, по правдоподобности. Вводятся контроль итогов промежуточных вычислений, автоматические проверки переполнения, исчезновения порядка, существования файлов и т.д. Проверяются длины элементов информации (для символьных строк) на критические значения, контролируются обязательные элементы полей или записей данных, контрольные разряды (метод избыточного кодирования).

2). *Немедленное обнаружение ошибок*. Средства обнаружения ошибки по возможности размещаются ближе к источнику ошибки.

3). *Изолирование ошибок*. Исключается влияние волнового эффекта, когда возникновение ошибки в одной части программы распространяется на другие части. В этом помогает модульный метод, при котором данные внутрь модуля передаются только через интерфейс.

По возможности не следует применять принцип умолчания. Например, при наличии n условий в программе необходимо предусмотреть n+1 проверок, из которых n - рабочие, а одна избыточная, направленная на выявление ошибки.

Рассмотрим пример использования в Turbo Pascal функции val для проверки вводимой информации. Ниже представлен фрагмент программы для ввода с клавиатуры числа из отрезка [0; 2].

```
var str: string;    { строка вводимых символов }
      n:  real;       { введенное число}
     code:  integer; { код преобразования строки в число}

begin    { Повторение ввода до тех пор, пока не введено число из интервала [0; 2]}
  repeat
     writeln('Введите число из отрезка [0; 2]');
     readln(str); {чтение символов с клавиатуры }
     val(str,n,code);       {преобразование строки в число}
  until (code=0) and (n<=2) and (n>=0);
end.
```

2.5. Эффективность программ

Участок программы, поглощающий до 90 % времени выполнения, называется *критической областью*. Обычно объем этой области не превышает 5-10 % всего программного кода. Основная задача повышения эффективности программ - поиск критической области и ее оптимизация. Выделяют оптимизацию времени выполнения и ресурсов.

Основным способом поиска критической области является *профилирование* программ, т.е. оценка времени выполнения отдельных фрагментов программы. Выявив те фрагменты исходного текста, которые являются «основными потребителя-

ми» ресурса, необходимо перепрограммировать эти фрагменты, решая задачу оптимизации.

Современные компиляторы, создающие на основе исходного текста программы исполняемый код, как правило, выполняют и оптимизацию этого кода, размещая инструкции процессору таким образом, чтобы увеличить скорость выполнения программы. *Оптимизирующий компилятор* применяется для уменьшения длины программы, но следует иметь в виду, что оптимизация компилятором выполняется достаточно осторожно и возможности такой автоматической оптимизации ограничены.

Оптимизация памяти достигается за счет применения записей с вариантами и частями, динамического распределения памяти, сокращения числа объектов в заголовках функций и процедур.

Для увеличения независимости команд и данных в пределах одного модуля целесообразно воспользоваться *структурным кодированием*. Модули, обрабатывающие исключения, ошибочные или редко возникающие ситуации следует размещать вне основных сегментов.

Оптимизация предполагает изменение существующей и уже отлаженной программы, в процессе этого изменения в программу могут быть внесены ошибки. Для устранения ошибок потребуется время, поэтому к оптимизации следует прибегать, только если в этом есть реальная необходимость. Оптимизировать программу размером в два десятка строк и с временем выполнения в несколько секунд нет смысла. Однако если решение задачи требует несколько десятков часов, оптимизация может оказаться действительно необходимой.

Оптимизация программы по затратам процессорного времени особенно важна для расчетных программ, в которых большой удельный вес имеют математические вычисления. Для создания *эффективной по времени программы* используются следующие приемы.

2.5.1. Инициализация объектов данных

Во многих программах какую-то часть объектов данных необходимо инициализировать, то есть присваивать им начальные значения. Такое присваивание выполняется либо в самом начале программы, либо, например, в начале цикла. Правильная инициализация объектов данных позволяет сэкономить драгоценное процессорное время. Так, например, если речь идет об инициализации массивов, использование цикла, скорее всего, будет менее эффективным, чем объявление этого массива типизированной константой.

Следует обращаться к данным для считывания записи в порядке их расположения в памяти.

2.5.2. Программирование арифметических операций

В том случае, когда значительная часть времени работы программы отводится арифметическим вычислениям, немалые резервы повышения скорости работы программы таятся в правильном программировании арифметических (и логических) выражений. Важно понимать, что различные арифметические операции значительно различаются по быстродействию. Самыми быстрыми являются операции сложения и вычитания. Более медленным является умножение, затем идет деление. Относительно много времени тратится на обращение к подпрограммам.

Быстродействие также зависит и от типа операндов. Операция сложения оказывается наиболее медленной для операндов типа Byte и ShortInt. Объясняется это особенностями реализации целочисленной арифметики. Несмотря на то, что переменные указанных типов занимают один байт, арифметические операции для них — двухбайтовые. При выполнении операций над типами Byte и ShortInt производится обнуление старшего байта регистров, второй операнд копируется из па-

мяти в регистр. Это и приводит к дополнительным затратам времени.

Оптимизация арифметических выражений с операндами различных типов производится за счет группировки однотипных объектов.

Пример.

```
      integer
      /     \
A + I + B + J   следует заменить на (A+B)+(I+J);
 \     /
  real
```

Программируя арифметические выражения, следует выбирать такую форму их записи, чтобы количество «медленных» операций было сведено к минимуму.

Пример. Пусть необходимо вычислить многочлен 4-й степени:

$$ax^4 + bx^3 + cx + dx + e.$$

При условии, что вычисление степени производится перемножением основания определенное число раз, нетрудно найти, что в этом выражении содержится 10 умножений («медленных» операций) и 4 сложения («быстрые» операции). Однако это же выражение можно записать в следующем виде:

$$(((ax + b)x + c)x + d)x + e.$$

Такая форма записи называется схемой Горнера. В этом выражении имеется 4 умножения и 4 сложения. Общее количество операций уменьшилось почти в два раза, соответственно уменьшится и время вычисления многочлена.

Таким образом, в целях минимизации времени вычисления промежуточных результатов выражение

$y=a_0*x_n+a_1*x_{n-1}+ .. +a_n,$

используя схему Горнера, следует заменить более экономичным выражением

$y=((..(a_0*x+a_1)*x+a_1)*x+ .. +a_n.$

Различается и время выполнения циклов. Время выполнения цикла со счетчиком и цикла с постусловием при всех

прочих равных условиях совпадает, цикл с предусловием выполняется несколько дольше (примерно на 20-30 %).

При использовании вложенных циклов следует иметь в виду, что затраты процессорного времени на выполнение такой конструкции могут зависеть от порядка следования вложенных циклов. Так, например, вложенный цикл со счетчиком

```
for   j  := 1   to   100000 do
    for   k  := 1   to   1000 do a := 1;
```

выполняется примерно на 10 % дольше, чем цикл

```
for    j  := 1    to    1000 do
    for   k := 1   to   100000 do a : = 1;
```

На первый взгляд, и в первом и во втором случае 10 000 000 раз выполняется оператор присваивания, и затраты времени на это должны быть одинаковы в обоих случаях. Но это не так. Объясняется тем, что инициализация цикла, то есть обработка процессором его заголовка с целью определения начального и конечного значений счетчика, а также шага приращения счетчика требует времени. В первом случае 1 раз инициализируется внешний цикл и 100 000 раз — внутренний, то есть всего выполняется 100 001 инициализаций.

Во втором случае, как нетрудно подсчитать, таких инициализаций оказывается лишь 1001.

Аналогично ведут себя вложенные циклы с предусловием и с постусловием. При программировании вложенных циклов по возможности следует делать цикл с наибольшим числом повторений самым внутренним, а цикл с наименьшим числом повторений — самым внешним.

При вычислении сумм часто используются циклы, содержащие одинаковые операции, относящиеся к каждому слагаемому. Это может быть, например, общий множитель:

```
summa  := 0;
for  i:= 1 to 1000 do  summa:= summa + a*x[i];
```

Очевидно, что другая форма записи этого цикла оказывается более экономной:

```
summa  := 0;
for    i   := 1   to    1000 do summa := summa + x[i];
summa  := a * summa;
```

так как она содержит всего одно умножение при том же числе сложений, против 1000 умножений в первом случае.

2.5.3. Программирование логических операций

Оптимизация при программировании логических выражений достигается путем написания первой той части логического выражения, которая вероятней всего истинна.

2.5.4. Инвариантные фрагменты кода

Внутри цикла могут встречаться выражения, фрагменты которых никак не зависят от управляющей переменной цикла. Их называют инвариантными фрагментами кода. Современные компиляторы часто определяют наличие таких фрагментов и выполняют автоматическую оптимизацию. Такое возможно не всегда, и иногда производительность программы зависит целиком от того, как запрограммирован цикл.

Пример фрагмента программы:

```
for    I:= 1    to   n    do
begin
…
   for    k  := 1   to   p   do
      for   m := 1   to   q   do
      begin
      a[k, m]:= sqrt(x*k*m–j)+ abs(u*I – x*m + k);
      b[k, m]:= sin(x*k*I) + abs(u*I*m + k);
      end;
…
am  := 0;
```

```
bm  := 0;
for   k  := 1 to p do
        for m := 1   to   q   do
        begin
            am  := am + a[k,m] / c[k];
            bm:= bm + b[k,m] / c[k];
        end;
end;
```

Здесь инвариантными фрагментами кода являются слагаемое sin(x*k*i) в первом цикле по переменной m и операция деления на элемент массива c[k] во втором цикле по m. Значения синуса и элемента массива не изменяются в цикле по переменной m, следовательно, в первом случае можно вычислить значение синуса и присвоить его вспомогательной переменной, которая будет использоваться в выражении, находящемся внутри цикла. Во втором случае можно выполнить деление после завершения цикла по m. Таким образом, можно существенно сократить количество трудоемких арифметических операций.

2.6. Отладка программ

Различают синтаксические и семантические ошибки. Синтаксические выявляются в процессе компиляции программы. Семантические (смысловые) связаны с неверным пониманием постановки задачи, неверной алгоритмизацией и кодированием. Кроме того, различают ошибки времени подготовки программы (синтаксические) и времени выполнения программы (runtime error), которые обнаруживаются при прогонах готовой программы.

Источником ошибок в 90 % случаев является сам программист и постановщик задачи; 10 % относят к ошибкам системы программирования, ОС, аппаратуры.

Отладка складывается из *контроля программы, локализации ошибок, исправления ошибок, предупреждения ошибок и тестирования*.

Контроль программы заключается в обнаружении ошибок при написании текста. Контроль складывается из 3-х этапов: просмотра, проверки, прокрутки.

Локализация ошибок состоит в поиске точки, содержащей ошибку, и в получении тестовых результатов, их анализе, сверке с эталоном, выявлении факта ошибки, предположении о характере и месте расположения ошибки.

Точка обнаружения и точка возникновения ошибки обычно не совпадают. Локализации ошибки проводится по характеру точки обнаружения. Основным способом локализации является обратное отслеживание.

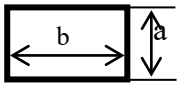
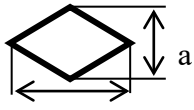
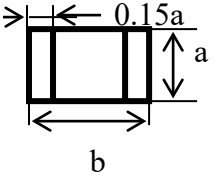
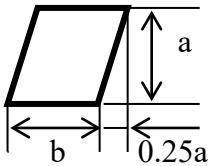
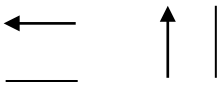
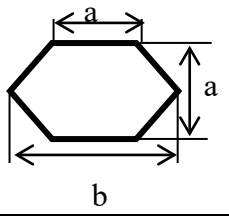
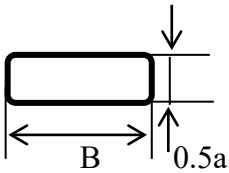
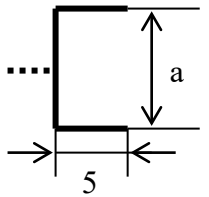
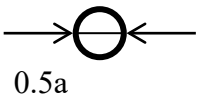
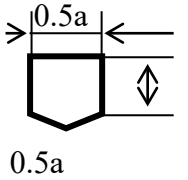
Прогон теста обычно заканчивается одним из трех результатов: аварией, зацикливанием, нормальным завершением с получением результата, правильного или неправильного.

Предупредить ошибки обычно помогают применение поэтапной разработки программной системы, правильный выбор уровня языка программирования, наглядность текста, специальная подготовка тестов, стандарты программирования, применение защитного программирования.

2.7. Методические рекомендации по разработке структурных схем

Условные графические обозначения символов, используемых для составления структурной схемы алгоритма, стандартизированы. Некоторые, часто используемые обозначения приведены в таблице.

Основные блоки структурной схемы

Процесс (вычисления)		Решение (проверка условия)	
Предопределенный процесс (подпрограмма)		Ввод-вывод	
Соединительные линии		Модификация (начало цикла)	
Начало-конец		Комментарии	
Внутристраничный соединитель		Межстраничный соединитель	

Примечание. Значение a принимается из ряда чисел 10; 15; 20… мм; b = 1,5 a.

Отдельные блоки алгоритмов соединяются между собой линиями. Линии, идущие сверху вниз и слева направо, прини-

маются за основные и, если не имеют изломов, стрелками не обозначаются.

Алгоритм любой сложности может быть представлен комбинацией трех базовых структур:

- линейные (следование);
- разветвляющиеся;
- циклические (повторение).

В **линейном** вычислительном процессе операторы выполняются в той последовательности, в которой они записаны (рис. 2.1).

Рис. 2.1. Линейная структура (общий вид)

Разветвляющимися называются такие вычислительные процессы, в которых имеет место разветвление выполняемой последовательности действий в зависимости от результата проверки какого-либо условия. Альтернативные пути алгоритма помечаются соответствующими метками: истина/ложь, да/нет и ведут к общему выходу (рис. 2.2). В частном случае может оказаться, что для одного из выбранных путей действий предпринимать не нужно. Такая структура получила название «**обход**» или структура «**если – то**» (рис. 2.3).

Рис. 2.2. Разветвляющаяся структура **Если** – **То - Иначе**

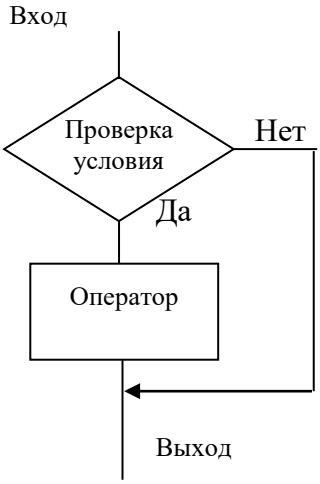


Рис. 2.3. Разветвляющаяся структура **Если** – **То** (обход)

Циклические алгоритмы – это алгоритмы, в которых происходит повторение некоторой последовательности действий. Циклы бывают арифметические (число повторений заранее известно) и итерационные (число повторений неизвестно, выход их цикла осуществляется при выполнении некоторого условия).

Группа операторов, повторяющихся в цикле, называется телом цикла. В арифметическом цикле задается управляющая переменная, которая изменяется от начального значения до конечного с определенным шагом (рис. 2.4).

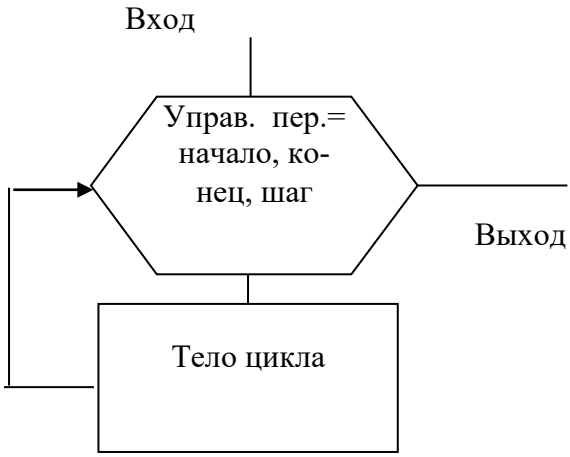


Рис. 2.4. Арифметический цикл (цикл с параметром)

Вложенный цикл представлен на рис. 2.5.

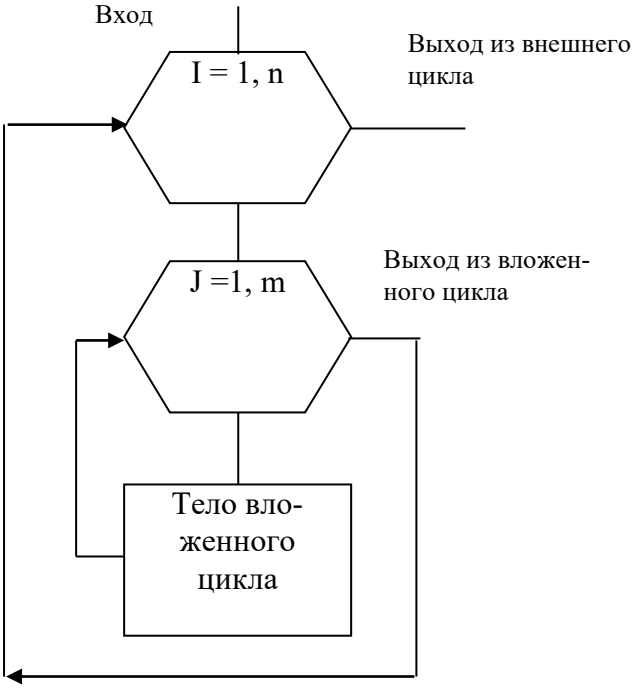


Рис.2.5. Структура вложенного цикла

2.8. Методические рекомендации по разработке программы

Общая схема организации программы следующая:

```
type
  имя = описание данных типа массив;
var
  Описание типов переменных;
(*процедура ввода массива *)
procedure имя1 (формальные параметры);
begin
…
end;
```

```
(*процедуры вывода массива*)
procedure имя2 (формальные параметры);
begin
…
end;
(*процедура вычислений*)
procedure имя3 (формальные параметры);
begin
…
end;
(*основная программа*)
begin
    …
    имя1 (фактические параметры);
    …
    имя3 (фактические параметры);
    …
    имя2 (фактические параметры);
    …
end.
```

Общий вид процедуры следующий:

```
procedure f (var  g1:t1; g2:t2; …);
Раздел описаний
begin
Раздел операторов
end;
```

здесь f – имя процедуры; g_i - формальные параметры (g1 – формальный параметр-переменная, записывается со служебным словом var; g2 – формальный параметр-значение); t_i - типы формальных параметров.

Особенности использования процедуры:

- процедура помещается в главной программе после раздела var и перед разделом begin основной программы;
- вызов процедуры: f (b1, b2, …); здесь b_i - фактические параметры;
- возврат из процедуры происходит на оператор, следующий за вызовом;
- формальные и фактические параметры должны соответствовать по количеству, порядку следования и типу;
- параметры–значения используются для передачи значений в подпрограмму (выполняют роль входных параметров процедуры). Они могут получать значения фактических параметров, но не могут передавать значения фактическим параметрам;
- параметры–переменные записываются со служебным словом var в заголовке процедуры. Параметры-переменные выполняют роль как входных, так и выходных параметров процедуры. Они могут получать значения фактических параметров, изменять их в процедуре и возвращать новые значения фактическим параметрам;
- тип массива, передаваемого в процедуру, должен быть описан в разделе type основной программы.

Пример. Даны две матрицы A(2,2) и B(3,3). Найти в каждой матрице сумму положительных элементов, используя процедуру.

```
{Описание нового типа}
type
mt =array [1..3,1..3] of real;
{Раздел описания типов глобальных переменных}
var
a, b: mt;
suma, sumb:  real;
i, j: integer;
```

```
{Процедура ввода матриц}
procedure vvod (var x: mt, n: integer; xx: char);
begin
writeln  (‘введите матрицу  ’, xx);
write (‘количество элементов= ’, n);
for i :=  1 to n do
for j :=  1 to n do
read (x[i,j]);
readln;
end;

{Процедура вывода матриц}
procedure vivod (x:mt; n:integer; xx: char);
begin
writeln (‘матрица  ’,xx);
for i := 1 to n do
begin
for j  :=  1 to  n
write (x[i,j]:5:1,’  ’);
writeln;
end;
end;

{Процедура вычисления суммы}
procedure rac (x: mt; n: integer; var sumx:  real);
var s :  real;
begin
s := 0;
for i  :=  1 to  n  do
for j  :=  1 to  n  do
if x[i,j]  >  0  then  s  :=  s  +  x[i,j];
sumx  :=  s;
end;
```

```
{Основная программа}
begin
{Вызов процедуры ввода для матрицы A}
vvod (a, 2, ‘a’);
{Вызов процедуры ввода для матрицы B}
vvod (b, 3, ‘b’);
{Вызов процедуры вывода для матрицы A}
vivod (a, 2, ‘a’);
{Вызов процедуры расчета суммы для матрицы A}
rac (a, 2, suma);
writeln (‘Матрица A ‘, ‘Сумма=’,suma:5:1);
{Вызов процедуры вывода для матрицы B}
vivod (в, 3, ‘b’);
{Вызов процедуры расчета суммы для матрицы B}
rac (b, 3, sumb);
writeln (‘Матрица B ‘, ‘Сумма=’,sumb:5:1);
readln;
end.
```

2.9. Контрольные вопросы

1. Что понимают под стилем программирования?
2. Какие приемы программирования вы знаете, какие используете?
3. Для чего нужны комментарии в программах?
4. Как и для чего используют модульное программирование?
5. Что такое структурное кодирование?
6. На каких принципах построено защитное программирование?
7. Какую программу можно считать эффективной?
8. Как проводится отладка программы?

2.10. Содержание и порядок выполнения работы

1. Ознакомиться с теоретическими разделами. Самостоятельно изучить средства отладки в среде TurboPascal.

2. По заданию преподавателя разработать модульную, эффективную программу в соответствии с изученными приемами программирования. Провести отладку программы средствами TurboPascal.

2.11. Общие требования к выполнению задания 1

Разработать структурную схему программы, разбитой на модули (процедуры). Создать для каждого модуля отдельную структурную схему. Структурные схемы оформить, используя редактор Word.

Разработать и отладить программу. Программа должна состоять из:

основной программы;

процедуры ввода массива;

процедуры вывода массива;

процедур реализации основных вычислений; вывод результатов осуществить в основной программе.

Снабдить текст программы комментариями.

Разработать контрольный пример, позволяющий проверить все модули (процедуры программы).

Оформить отчет, который должен содержать условие задачи, структурные схемы, текст программы, контрольный пример.

Отметить фрагменты программы, в которых можно выполнить оптимизацию, таким образом, чтобы количество «медленных» операций было бы сведено к минимуму. Вставить в отчет переписанные фрагменты кода, указав, почему представленная форма записи оказывается более экономной (использование инициализации данных, определенных типов данных, схемы Горнера, программирование вложенных цик-

лов, замена инвариантных фрагментов кода, замена операций умножения и деления и т.п.)

2.12. Варианты для выполнения задания №1

1. Если все элементы массива четные, вычислить их сумму. Если все элементы массива нечетные, вычислить их произведение. Если среди элементов массива есть и четные, и нечетные, определить количество нечетных элементов. Осуществить данные вычисления отдельно для массивов A(10) и B (12). Вывести массивы и полученные результаты.

2. Если все элементы массива положительные, вычислить их произведение. Если все элементы массива отрицательные или равные нулю, вычислить их сумму. Если среди элементов массива есть положительные, отрицательные и равные нулю, то определить количество положительных элементов. Осуществить данные вычисления отдельно для массивов C(8), K(14). Вывести массивы и полученные результаты.

3. Если все элементы массива кратны 5, то вычислить их сумму. Если все элементы массива не кратны 5, то вычислить их произведение. Если среди элементов массива есть кратные и не кратные 5, то определить количество кратных 5. Осуществить данные вычисления для массивов P(10), R(12). Вывести массивы и полученные результаты.

4. Если все элементы массива положительные или равные нулю, то найти максимальный элемент. Если все элементы массива отрицательные, то найти минимальный элемент. Если среди элементов массива есть положительные, отрицательные и нулевые элементы, то определить их сумму. Осуществить данные вычисления для массивов Q(12), W(15). Вывести массивы и полученные результаты.

5. Если все элементы вещественного массива имеют целые значения, то найти наибольший по модулю элемент. Если все элементы вещественного массива имеют значения с дробной частью, то найти наименьший по модулю элемент. Если

среди элементов массива есть элементы с целым значением, элементы с дробным значением, то определить количество целых и количество элементов с дробной частью. Осуществить данные вычисления для массивов F(10), G(12). Вывести массивы и полученные результаты.

6. Если все элементы массива больше заданного числа A, то найти минимальный среди них. Если все элементы массива меньше заданного числа A, то найти максимальный среди них. Если среди элементов массива есть элементы большие A, меньшие A и равные A, то найти среднее арифметическое элементов массива. Осуществить данные вычисления для массивов X(12), Y(15). Вывести массивы и полученные результаты.

7. Если все элементы массива положительные, то сформировать новый массив, элементы которого есть квадратные корни из соответствующих элементов исходного массива. Если все элементы отрицательные или равные нулю, то сформировать новый массив, элементы которого есть квадраты соответствующих элементов исходного массива. Если все элементы массива имеют разные знаки, то определить сумму положительных и сумму отрицательных элементов. Осуществить данные вычисления для массивов S(10), H(12). Вывести массивы и полученные результаты.

8. Если все элементы массива кратны 3, то сформировать новый массив, элементы которого есть удвоенные соответствующие элементы исходного массива. Если все элементы массива не кратны 3, то сформировать новый массив, элементы которого есть утроенные соответствующие элементы исходного массива. Если элементы массива и кратны 3, и не кратны 3, то определить сумму элементов массива. Осуществить данные вычисления для массивов Z(12), X(15). Вывести массивы и полученные результаты.

9. Если все элементы матрицы положительные или равные нулю, то найти минимальный элемент, номер строки и столбца, где он находится. Если все элементы матрицы отрицательны, то найти максимальный элемент, номер строки и

столбца, где он находится. Если элементы матрицы имеют разные знаки, то определить сумму элементов матрицы. Осуществить данные вычисления для матриц A(3,3), B(4,4). Вывести матрицы и полученные результаты.

10. Если все элементы матрицы четные, вычислить их сумму. Если все элементы матрицы нечетные, вычислить их произведение. Если среди элементов матрицы есть и четные, и нечетные, определить количество четных элементов. Осуществить данные вычисления отдельно для матриц F(2,2) и P (3,3). Вывести матрицы и полученные результаты.

11. Если все элементы матрицы кратны 5, то вычислить их произведение. Если все элементы матрицы не кратны 5, то вычислить их сумму. Если среди элементов массива есть кратные и не кратные 5, то определить количество кратных 5. Осуществить данные вычисления для матриц P(4,4), R(3,3). Вывести матрицы и полученные результаты.

12. Если все элементы матрицы больше заданного числа B, то найти минимальный среди них. Если все элементы матрицы меньше заданного числа B, то найти максимальный среди них. Если среди элементов матрицы есть элементы большие B, меньшие B и равные B, то найти среднее арифметическое элементов матрицы. Осуществить данные вычисления для матриц X(2,2), Y(3,3). Вывести матрицы и полученные результаты.

13. Если все элементы матрицы четные, вычислить их произведение. Если все элементы матрицы нечетные, определить их сумму. Если среди элементов матрицы есть и четные, и нечетные, определить количество четных элементов. Осуществить данные вычисления отдельно для матрицы X(2,2) и Y(3,3). Вывести матрицы и полученные результаты.

14. Если все элементы матрицы положительные, то все минимальные элементы заменить на 100 (если их несколько). Если все элементы матрицы отрицательные, то все максимальные элементы заменить на 0 (если их несколько). Если среди элементов матрицы есть и положительные, и отрицательные,

умножить элементы матрицы на 2. Осуществить данные вычисления отдельно для матрицы H(2,2) и G(3,3). Вывести матрицы до и после преобразования.

15. Если все элементы матрицы отрицательные, то найти максимальный элемент на главной диагонали. Если все элементы матрицы положительные, то найти минимальный элемент на главной диагонали. Если среди элементов матрицы есть и положительные, и отрицательные, то найти сумму элементов на главной диагонали. Осуществить данные вычисления для матриц D(4,4), R(3,3). Вывести матрицы и полученные результаты.

16. Если все элементы второй строки матрицы одинаковы и четны, то определить сумму элементов матрицы. Если все элементы второй строки матрицы одинаковы и нечетны, то определить произведение элементов матрицы. Если все элементы второй строки матрицы различны, то определить сумму элементов матрицы, взятых по модулю. Осуществить данные вычисления для матриц Q(3,3), W(4,4). Вывести матрицы и полученные результаты.

17. Если все элементы матрицы больше заданного числа B, то найти минимальный элемент. Если все элементы матрицы меньше заданного числа B, то найти максимальный элемент. Если среди элементов матрицы есть и большие B, и меньшие B, то найти максимальный по модулю элемент. Осуществить данные вычисления для матриц A(3,3), B(4,4). Вывести матрицы и полученные результаты.

18. Если все элементы массива упорядочены по возрастанию, то найти среднее значение. Если все элементы массива упорядочены по убыванию, то найти максимальный по модулю элемент. Если элементы массива не упорядочены, то определить произведение элементов массива. Осуществить данные вычисления отдельно для массивов K(15), L(20). Вывести массивы и полученные результаты.

19. Если все элементы матрицы кратны 5, то вычислить сумму элементов матрицы. Если все элементы матрицы не

кратны 5, то вычислить их произведение. Если среди элементов матрицы есть кратные и не кратные 5, то определить количество элементов кратных 5. Осуществить данные вычисления отдельно для матриц F(2,3), S(3,4). Вывести матрицы и полученные результаты.

20. Если все элементы на главной диагонали положительные, то найти сумму элементов матрицы. Если все элементы на главной диагонали отрицательные, то найти сумму элементов матрицы без элементов на главной диагонали. Если элементы на главной диагонали имеют разные знаки, то определить сумму элементов матрицы, взятых по модулю. Осуществить данные вычисления отдельно для матриц A(3,3), S(4,4). Вывести матрицы и полученные результаты.

21. Если в массиве больше положительных элементов, чем отрицательных, то определить минимальный положительный элемент. Если в массиве больше отрицательных элементов, чем положительных, то определить максимальный отрицательный элемент. Если положительных и отрицательных элементов поровну, то определить наибольший по модулю элемент. Осуществить данные вычисления отдельно для массивов F(10), D(15). Вывести массивы и полученные результаты.

22. Если все элементы матрицы положительные или равные нулю, то найти максимальный элемент на главной диагонали. Если все элементы матрицы отрицательны, то найти минимальный элемент на главной диагонали. Если элементы матрицы имеют разные знаки, то найти сумму элементов на главной диагонали. Осуществить данные вычисления отдельно для матриц P(3,3), R(4,4). Вывести матрицы и полученные результаты.

23. Если все элементы вещественной матрицы имеют целые значения, то найти их среднее значение. Если все элементы вещественной матрицы имеют значения с дробной частью, то найти их сумму. Если среди элементов матрицы есть элементы с целым и дробным значениями, то определить количе-

ство элементов с целым значением. Осуществить данные вычисления отдельно для матриц H(2,2), G(3,3). Вывести матрицы и полученные результаты.

24. Если все элементы матрицы больше заданного числа X, то найти минимальный элемент. Если все элементы матрицы меньше заданного числа X, то найти максимальный элемент. Если среди элементов матрицы есть большие X, меньшие X и равные X, то найти максимальный по модулю элемент. Осуществить данные вычисления отдельно для матриц T(3,4), Y(2,3). Вывести матрицы и полученные результаты.

25. Если все элементы матрицы положительные, то сформировать новую матрицу, элементы которой есть квадратные корни из соответствующих элементов исходной матрицы. Если все элементы матрицы отрицательные, то сформировать новую матрицу, возведя в квадрат каждый элемент исходной матрицы. Если элементы матрицы имеют разные знаки, то умножить элементы матрицы на 2. Осуществить данные вычисления отдельно для матриц A(3,3), C(2,2). Вывести матрицы до и после изменений.

26. Если все элементы матрицы кратны 3, то сформировать новую матрицу, удвоив элементы исходной матрицы. Если все элементы матрицы не кратны 3, то сформировать новую матрицу, утроив элементы исходной матрицы. Осуществить данные преобразования отдельно для матриц V(3,3), W(4,4). Вывести матрицы до и после изменений.

27. Если максимальный элемент находится над главной диагональю, то определить сумму элементов матрицы. Если максимальный элемент находится под главной диагональю, то определить сумму модулей элементов матрицы. Если максимальный элемент находится на главной диагонали, то определить сумму диагональных элементов. Осуществить данные вычисления отдельно для матриц Z(3,3), X(4,4). Вывести матрицы и результаты вычислений.

28. Если все элементы массива удвоенные четные, то создать новый массив, разделив исходный на 2. Если это не так,

то создать новый массив, умножив исходный массив на 2. Осуществить данные преобразования для массивов P(15), T(12). Вывести исходные и новые массивы.

29. Если все элементы второго столбца матрицы одинаковы, то найти сумму всех элементов матрицы, кроме второго столбца. Если все элементы второго столбца разные, то найти сумму элементов второго столбца. Осуществить данные вычисления отдельно для матриц S(3,3), R(4,4). Вывести матрицы и результаты вычислений.

30. Если все элементы матрицы одинаковые, то определить их сумму. Если все элементы матрицы разные, то вычислить среднее арифметическое. Осуществить данные вычисления отдельно для матриц D(3,4), F(2,3). Вывести матрицы и результаты вычислений.

31. Если в матрице две первые строки одинаковы, то элементы этих строк умножить на 2. Если элементы разные, то ко всем элементам матрицы прибавить 1. Осуществить данные вычисления отдельно для матриц B(3,3), C(4,4). Вывести матрицы до и после изменений.

32. Если в матрице больше положительных элементов, чем отрицательных, то определить минимальный положительный элемент. Если в матрице больше отрицательных элементов, чем положительных, то определить максимальный отрицательный элемент. Если положительных и отрицательных элементов поровну, то определить наибольший по модулю элемент. Осуществить данные вычисления отдельно для матриц A(3,3), D(4,4). Вывести матрицы и полученные результаты.

33. Если все элементы на главной диагонали одинаковы, то найти их сумму. Если это не так, то найти среднее арифметическое элементов матрицы. Осуществить данные вычисления отдельно для матриц H(3,3), K(4,4). Вывести матрицы и полученные результаты.

34. Если все элементы массива одинаковые, то умножить их на три. Если все элементы массива разные, то умножить их на четыре. Вывести массивы до и после изменений.

35. Если в массиве больше четных элементов, чем нечетных, то прибавить ко всем элементам максимальный элемент. Если в массиве меньше четных элементов, чем нечетных, то вычесть из всех элементов максимальный. Вывести массивы до и после изменений.

3. ТЕСТИРОВАНИЕ ПРОГРАММНЫХ СИСТЕМ (ПС)

Тестирование – это выполнение программы с целью обнаружения ошибки. Тестирование является частью отладки ПС. *Отладка* подразумевает также *обнаружение, локализацию, исправление* ошибок. Ошибки в программе появляются в результате:

- неправильного понимания алгоритма;
- неправильного кодирования;
- отказов, сбоев аппаратуры, операционной системы;
- неправильной постановки задачи и т.п.

В большинстве случаев источником ошибок является разработчик ПС.

Проверяются программы с помощью специальных *тестов*. Тест должен содержать:

- набор исходных данных;
- условия для запуска программы;
- набор ожидаемых результатов.

Процесс отладки можно представить в виде диаграммы (рис. 3.1).

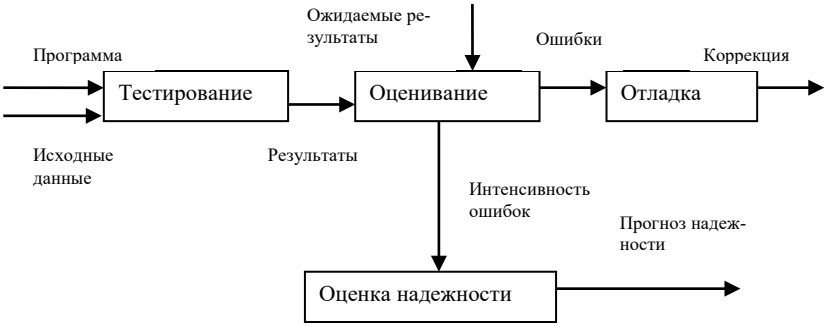


Рис.3.1. Процесс отладки программы

Для полной проверки программы необходимо провести *исчерпывающее тестирование*, т.е. проверить все наборы и комбинации исходных данных, что потребует огромного количества времени.

3.1. Принципы тестирования

Различают функциональное (по методу «черного» ящика) и структурное (по методу «белого» ящика) тестирование ПС.

При функциональном тестировании исследуется работа каждой функции ПС на всей области определения данных. Внутренняя логическая структура программ во внимание не принимается.

При структурном тестировании исследуются внутренние блоки программ (логические условия, ветки, циклы и т.д.) и связи между ними.

3.2. Тестирование по методу «черного» ящика

Проверяется работа каждой функции программы на всей области определения данных. Каждая функция – это своеобразный «черный» ящик, для которого известны входные/выходные данные, но не известен алгоритм их преобразо-

вания. При тестировании необходимо подобрать входные данные, ведущие к аномальному поведению программы, и выходные данные, демонстрирующие дефекты программы.

Техника состоит в *сокращении числа тестов и выявлении классов ошибок*, а не отдельных ошибок.

Для сокращения числа тестов можно поступить двумя способами.

1. Разбить данные на классы эквивалентности и исследовать по одному набору данных из каждого класса.

2. Проводить исследования на граничных значениях данных (анализировать граничные условия).

3.2.1. Разбиение на классы эквивалентности

По первому способу вся область данных (входных и выходных) разбивается на классы эквивалентности. Для каждого класса разрабатывается 1 тест.

Под *классом эквивалентности* здесь понимается набор данных с общими свойствами, при которых программа ведет себя одинаково. Т.е. в этом случае задействованы одни и те же операторы и связи между ними.

Следует сразу же выделить *допустимые и недопустимые данные*, внутри которых проводится разделение на классы.

Правила формирования классов эквивалентности таковы.

1. Если ввод предполагает диапазон значений, то выделяют 4 класса эквивалентности:

- допустимый (например, n<=x<=m),
- недопустимый 1 (например, x<n),
- недопустимый 2 (например, x>m),
- недопустимый 3 (например, x – не число).

2. Если предполагается ввод конкретного значения, то выделяют 1 допустимый и 3 недопустимых класса.

- допустимый - это правильное значение,
- недопустимый – значение, большее допустимого,

- недопустимый – значение, меньшее допустимого,
- недопустимый – значения других типов.

3. Если ввод предполагает множество значений {a, b, c}, то выделяют 2 класса:

- допустимый {a, b, c},
- недопустимый (x<>a) And (x<>b) And (x<>c).

4. Если вводится булево значение, исследуются 3 класса:

- значение true,
- значение false,
- значения других типов.

3.2.2. Анализ граничных условий

Существуют несколько правил для определения наборов тестов.

1. Если исходные данные представлены диапазоном значений n..m, то определяются наборы:

- n,
- m,
- левее n,
- правее m,
- значения других типов.

2. Если исходные данные – дискретное множество, то тесты представляются наборами:

- минимальное значение,
- максимальное значение,
- меньше минимума,
- больше максимума,
- других типов.

3. Правила 1 и 2 применяются к выходным данным.

4. Проверяются исходные данные на границах.

5. Если входные и выходные данные – упорядоченные множества, то тестируются 1-й и последний элементы этого множества, а также данные других типов.

3.3. Тестирование методом «белого» ящика

Структурное тестирование является более сложным и трудоемким, поэтому ему подвергаются наиболее сложные и значимые блоки программ.

Считается известной структура (алгоритм) каждой программы и ее блоков. Проверяются все независимые маршруты программы (или блока), проходятся все ветки логических условий, выполняются все циклы, анализируется вся внутренняя структура данных.

Для *циклов с n-повторениями* строится один из наборов тестов:

- прогон всего цикла;
- 1 проход цикла;
- 2 прохода цикла;
- m проходов цикла, m<n;
- n-1, n, n+1 проходов цикла.

Покрытие операторов – подбираются такие тесты, при которых каждый оператор выполняется хотя бы 1 раз.

Например, в блоке реализуется алгоритм поиска минимума из трех значений Y=min(a, b, c). Структурная схема алгоритма представлена ниже (рис. 3.2).

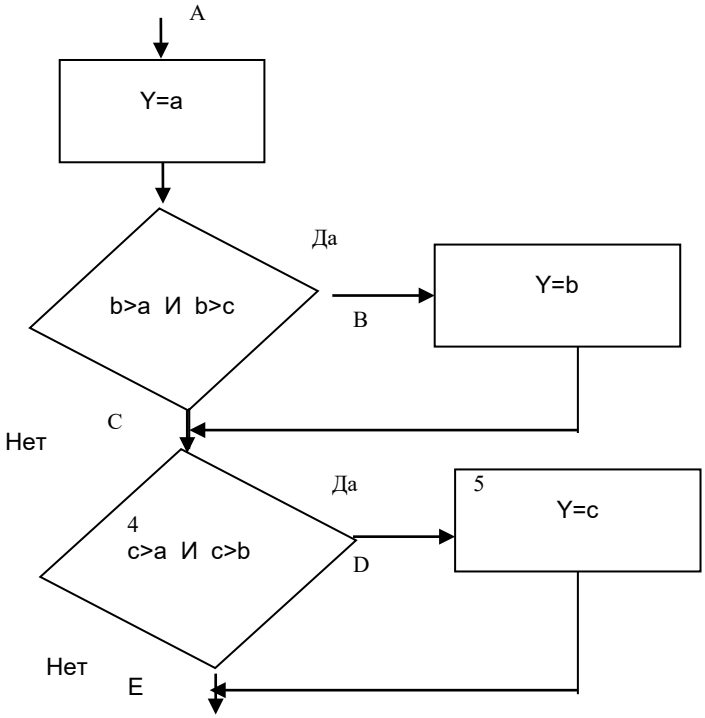


Рис. 3.2. Алгоритм поиска минимума трех чисел

В этом примере покрывают операторы следующие тесты:

1) a=5, b=1, c=2 – покрыты операторы 1, 2, 4;

2) a=1, b=3, c=2 - покрыты операторы 1, 2, 3, 4;

3) a=2, b=4, c=6 – покрыты операторы 1, 2, 4, 5.

Покрытие переходов – составляются такие тесты, совокупность которых обеспечивает проход каждого направления при ветвлении хотя бы 1 раз.

В примере покрывают переходы следующие тесты:

1) a=1, b=3, c=2 - покрыты переходы A, B, E;

2) a=2, b=4, c=6 – покрыты переходы A, C, D.

Покрытие условий – составляется совокупность тестов, проверяющая результаты каждого элементарного условия хотя бы 1 раз.

В примере покрывают все возможные результаты каждого элементарного условия следующие тесты:

1. a<b;
2. b<=a;
3. b>c;
4. b<=c;
5. c>a;
6. c<=a;
7. c>b;
8. c<=b.

Комбинаторное покрытие условий – подбирается такая совокупность тестов, при которой хотя бы 1 раз проверяются все возможные комбинации результатов условия и все точки входа.

Для рассмотренного примера это следующие тесты:

1. b>a, b>c;
2. b>a, b<=c;
3. b<=a, b>c;
4. b<=a, b<=c;
5. c>a, c>b;
6. c>a, c<=b;
7. c<=a, c>b;
8. c<=a, c<=b.

Наиболее надежным является комбинаторное покрытие условий.

3.4. Контрольные вопросы

1. В чем состоит отладка программ?
2. Что такое тестирование программы?
3. Сколько требуется тестов для исчерпывающего тестирования?
4. Какие способы сокращения числа тестов существуют?

5. В чем состоит методика тестирования по «черному» ящику?

6. В чем состоит методика тестирования по «белому» ящику?

7. Почему комбинаторное покрытие условий более надежно по сравнению с покрытием операторов или переходов?

3.5. Задание №2

1. Ознакомиться с теоретическими разделами. Ответить на контрольные вопросы.

2. Протестировать программу из задания № 1 по методам «черного» и «белого» ящиков. Стратегии сокращения числа тестов выбрать самостоятельно.

3. Отчет должен содержать цель работы, задание, текст программы, описание и обоснование выбранных стратегий сокращения числа тестов, тесты, результаты работы программы на тестах, выводы.

4. Для тестирования по методу «белого» ящика представить структурную схему алгоритма решаемой задачи.

Результаты тестирования оформить в виде табл. 3.1.

Таблица 3.1

Результаты тестирования

Способ тестирования	№ теста	Исходные данные	Выходные данные (ожидаемые результаты)

Выбор варианта контрольной работы осуществляется с помощью табл. 3.2.

Таблица 3.2

Таблица выбора вариантов контрольных заданий

По-след. цифра номера студ. билета	Предпоследняя цифра номера студенческого билета									
	0	1	2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8	9	10
1	11	12	13	14	15	16	17	18	19	20
2	21	22	23	24	25	26	27	28	29	30
3	31	32	33	34	35	1	2	3	4	5
4	6	7	8	9	10	11	12	13	14	15
5	16	17	18	19	20	21	22	23	24	25
6	26	27	28	29	30	31	32	33	34	35
7	1	2	3	4	5	6	7	8	9	10
8	11	12	13	14	15	16	17	18	19	20
9	21	22	23	24	25	26	27	28	29	30

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Орлов С.А. Технологии разработки программного обеспечения / С.А. Орлов. СПб: Питер, 2003.

2. Ганцева Е.А. Технология программирования: учеб. Пособие / Е.А. Ганцева. Воронеж: ВГТУ, 2007.

3. Проектирование пользовательского интерфейса на персональных компьютерах. Стандарт фирмы IBM / Под ред. М. Дадашова. DBS Ltd., 1991.

4. ГОСТ 19.003-80. Схемы алгоритмов и программ. Обозначения условные графические.

СОДЕРЖАНИЕ

ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНЫХ СИСТЕМ

МЕТОДИЧЕСКИЕ УКАЗАНИЯ
к выполнению контрольной работы
по дисциплине «Технология программирования»
для студентов специальности 230101
«Вычислительные машины, комплексы, системы и сети»
заочной и заочной сокращенной форм обучения

Составители: Гребенникова Наталия Ивановна
Сергеева Татьяна Ивановна

В авторской редакции

Подписано в печать 19.02.2008.
Формат 60x84/16. Бумага для множительных аппаратов.
Усл. печ. л. 2,7. Уч.-изд. л. 2,5. Тираж 100 экз. «С» 40.
Заказ № 46.

ГОУВПО «Воронежский государственный
технический университет»
394026 Воронеж, Московский просп., 14