8. Принципы объектно-ориентированного программирования

Технология программирования в этих условиях становится одной из важнейших современных областей науки и технологий, затрагивающей интересы миллионов людей. Круг ее проблем и методов стремительно расширяется.

Программирование – это, прежде всего, коллективный интеллектуальный труд, и его технология сильно отличается от других промышленных технологий. Появляются новые, все более сложные абстрактные модели функционирования программ и основанные на них языки и системы программирования. Важнейшую роль в технологии играют инструментальные средства программирования. Естественно, отбрасываются устаревшие приемы и инструменты, и в то же время происходит накопление фундаментальных знаний, оправдавших себя технологий, методов и средств.

Важную часть такого фундамента составляет технология модульного программирования, методы которой в основном сложились в 60-х и 70-х годах двадцатого века [32, 34, 35, 40, 41, 65]. К этому же периоду относится появление первых идей объектно-ориентированного программирования, которое стало дальнейшим развитием этой технологии [37-39, 12].

8.1. Парадигмы программирования

Рассмотрим кратко основные этапы эволюции методологии программирования на языках высокого уровня, результатом которой стало появление объектно-ориентированного программирования (ООП).

Каждый из рассмотренных далее методов не был отброшен, а органично вошел во все последующие.

Любая программа состоит из обрабатываемых данных и алгоритма:

Программа = Данные + Алгоритм

При программировании в машинных кодах и на языке ассемблера (50-е гг.) эти компоненты программы мало структурированы. В *процедурном программировании* на языках Fortran, Algol-60 и др. (60-е гг.) в основном структурируется алгоритм. В нем выделяется иерархия подпрограмм (процедур). Структуры данных мало развиты: скалярные (одиночные) значения и массивы из них. Этого было достаточно для решавшихся в основном в то время числовых задач.

Создание трансляторов, операционных, управляющих и проектирующих систем, машинная графика потребовали обработки более разнообразных и сложных типов данных: символьных (текстовых), графических, звуковых и т.п., мало поддерживаемых традиционными процедурно-ориентированными языками.

Концепция баз данных, ориентированная на хранение и обработку больших объемов сложноорганизованной информации во внешней памяти,

напротив, преимущественное внимание уделяет данным, в определенной степени отрывая их от алгоритмов.

Увеличение размера программ и сложности данных привело к *модульному программированию*, при котором функционально самостоятельные части программы оформляются в виде модулей. Модуль предоставляет пользователю (другому модулю) некоторые функции, скрывая от него детали их реализации. Такое сокрытие называется *инкапсуляцией*. Модульное программирование поддерживается языками Modula-2, C и др.

Развитие модульного программирования в этом направлении привело к языкам *объектно-ориентированного программирования*: Simula-67 (1967 г.), Smalltalk (1981 г.), , C++ (1983 г.), объектный Pascal и др.

Методы *модульного программирования* [39, 41, 70], основаны на идее разбить разрабатываемую программу на сравнительно небольшие независимые части - (программные) модули, т. е. на основном принципе борьбы со сложностью - "разделяй и властвуй". Они составляют фундамент более сложных и современных направлений технологии программирования.

Рекомендации и примеры разработки программ по модульной технологии приведены в книгах [23, 29, 30, 36, 37, 39 – 41, 45, 70].

8.2.1. Проектирование программы

Можно рассматривать реализацию следующих этапов решения задачи на компьютере [20].

1. Постановка (точная формулировка) задачи - уточнение терминологии, исходных данных и результатов, недостаточных и излишних данных, сделанных допущений, разрешимости задачи и др.

2. Построение математической модели решения - выбор математического аппарата для формализации и решения задачи и структур данных.

Дайте точную математическую формулировку для выбранного метода решения задачи и используемых понятий. Определите структуры входных и выходных данных, их величин, единицы их измерения, области изменения и зависимости между ними.

3. Разработка алгоритма. Проверка (доказательство) правильности алгоритма.

Проведите предварительный анализ и выбор алгоритма решения поставленной задачи, определив основные возникающие проблемы и наметив возможные методы их решения.

При разработке алгоритма обязательно доопределите задачу, включая случаи отсутствия решения. Найдите примеры, иллюстрирующие основные варианты решения задачи. Эти примеры в дальнейшем послужат тестами для отладки программы.

5. *Проектирование программы.* Программная реализация алгоритма. Анализ алгоритма - оценка сложности алгоритма по времени и памяти и ее экспериментальная проверка.

7. Тестирование и отладка программы.

8. Документирование программы.

Остановимся на этапе проектировании программы. Процесс *проектирования программы* включает внешнее проектирование, проектирование структуры и проектирование модулей.

Внешнее проектирование - это разработка *архитектуры программы*, т.е. точное описание ее поведения с точки зрения пользователя.

Классическим примером является архитектура часов – циферблат, стрелки и головка завода. Одна и та же архитектура может иметь разную *реализацию* – внутренний механизм, например, механические, электрические или башенные часы.

Другими словами, внешнее проектирование программы - это проектирование ее *внешнего интерфейса* (способа взаимодействия с пользователем), включающего все выполняемые для пользователя функции программы и детальное описание ее входных и выходных данных.

Архитектура программы описывается в виде ее *внешней спецификации*, которая служит исходной информацией для проектирования программы и разработки ее эксплуатационной документации. Описание входных и выходных данных включает их перечень, структуру и местонахождение. Описание данных иллюстрируется примерами. Удобно использовать примеры из раздела постановки задачи.

Необходимо доопределить результаты работы программы для всех возможных недопустимых данных.

Программа обязательно должна проверять правильность входных данных. При обнаружении ошибок программа обычно сообщает о них, а затем, если возможно, каким-либо образом исправляет их и продолжает работу. Худшее, что может сделать программа, - принять неправильные входные данные и выдать бессмысленный, но правдоподобный результат!

При описании выходных данных приведите перечень всех возможных сообщений программы. Отдельно укажите сообщения об ошибках. Желательны также сообщения о начале и конце работы программы, об особых ситуациях при решении задачи, об отсутствии решения, обо всех ситуациях, требующих вмешательства пользователя, и т. п.

Например, сообщение "n > 10. Таблица переполнена" неудачно, т.к. пользователю вряд ли понятно, о каком n и таблице идет речь (скорее всего, это внутренние данные программы). Неясно также, хорошо это или плохо, что будет делать программа и как поступить самому пользователю.

Подобное сообщение лучше сформулировать, например, так:

"Ошибка: количество вершин > 10. Повторите ввод".

Программа может иметь входные и выходные файлы, имена которых указываются ей в команде запуска. Эти имена передаются по правилам языка и конкретной системы программирования. Например, C/C++ программа получает их как параметры функции main [26, 51, 52, 61]. Для входного файла vhod и выходного файла rez, в операционной системе UNIX, Windows или MS DOS программу можно запустить, например, командой:

```
pkrc  vhod  rez
```

где pkrc - имя файла, содержащего исполняемый модуль программы.

В языках множества C, можно не указывать в программе в явном виде имена файлов. В этом случае подразумевается стандартный входной файл, обычно вводимый с клавиатуры, и стандартный выходной файл, обычно выводимый на экран, которые при запуске программы из командной строки можно переключить на любые файлы с заданными именами с помощью переадресации ввода-вывода [67]. Если, например, исходные данные вместо клавиатуры необходимо взять из файла vhod, а результат вместо экрана должен выводиться в файл rez, программа вызывается следующей командой операционной системы UNIX, Windows или MS DOS:

```
pkrc.exe  <vhod  >rez
```

где pkrc - имя файла с исполняемым модулем программы.

Обрабатываемые программой данные делятся на входные, промежуточные и выходные. В отличие от промежуточных данных, которые хранятся только в оперативной памяти в виде удобного для программы *внутреннего представления*, входные и выходные данные имеют еще и видимое пользователю *внешнее представление*, которое должно быть удобным, понятным и естественным для человека. При вводе данных программа преобразует их из внешнего представления во внутреннее, при выводе - из внутреннего представления во внешнее. Поэтому, если внутреннее представление данных выбирается только из соображений эффективности (экономии памяти и времени работы) программы, то внешнее представление данных должно быть достаточно удобным и для программы, и, главным образом, для человека.

Для графа, например, можно рекомендовать внешнее представление в виде перечня ребер (дуг), перед которым указывается количество вершин. Каждое ребро задается парой номеров вершин. При необходимости указывается вес ребра или дуги.

Другое внешнее представление графа - матрица смежности, вводимая по строкам или по столбцам. Она более удобна при большом количестве ребер.

На этапе внешнего проектирования полезно методами черного ящика разработать тесты для отладки программы (см. ниже раздел 8.2.2). Это позволит лучше понять задачу, уточнить ее постановку и проверить полноту и качество внешнего проекта программы. Если возникают затруднения при определении реакции программы на входные данные какого-либо теста,

необходимо соответствующим образом уточнить описание архитектуры программы.

Проектирование структуры программы

Проектирование (модульной) структуры - это разбиение программы на небольшие независимые модули (подпрограммы) с целью упрощения разработки программы и ее изменений. В этом состоит основная идея модульного программирования [39].

Модуль представляет собой часть программы, выполняющую самостоятельные функции. Обычно модуль оформляется как подпрограмма или несколько взаимосвязанных подпрограмм, либо как объект (или класс объектов) в объектно-ориентированном программировании (см. раздел 8.3). Модуль часто хранят в отдельном файле и транслируют отдельно от других модулей.

Модуль имеет три основных атрибута: он выполняет одну или несколько функций, обладает некоторой логикой и используется в разных контекстах. *Функция* - это внешнее описание модуля (что он делает, но не как это делается). *Логика* определяет внутренний алгоритм модуля (как он выполняет свою функцию). *Контекст* описывает конкретное применение модуля. Функция модуля может рассматриваться как композиция (объединение) его логики и функций всех *подчиненных* (вызываемых им) *модулей*. Другими словами, функция модуля включает не только то, что он делает непосредственно сам, но и функции подчиненных ему модулей. Желательно, чтобы разработанные в программе модули, имеющие самостоятельное значение, можно было использовать как библиотечные программы в других задачах. Поэтому модули преобразования данных обычно сами не занимаются их вводом-выводом: входные и выходные данные передаются в виде параметров. Это также облегчает изменения программы.

Ввод и вывод данных, в частности, вывод сообщений об ошибках, производятся либо в главной программе, либо оформляются как отдельные модули. Модули обычно сообщают об ошибках в исходных данных или о невозможности решить задачу не человеку на экран, а вызывающей программе с помощью кода завершения, передаваемого как выходной параметр или как значение функции.

Обычно нулевой код завершения свидетельствует об успешном завершении программы, а ненулевое значение рассматривается как номер ошибки (см., например, решения задач 2.2 – 2.4).

.

Модульная структура программы представляется в виде схемы взаимодействия модулей, в которой прямоугольники соответствуют модулям, а стрелки показывают сопряжения (интерфейсы) модулей и соединяют вызывающий модуль с вызываемыми им модулями.

На рис. 8.1 приведен вариант структуры программы поиска в заданном графе цикла минимальной длины.

Соответствующий каждому сопряжению набор входных и выходных данных вызываемого модуля указывается в *таблице сопряжений*. Сопряжения модулей программы из рис. 8.1 показаны в табл. 8.1.

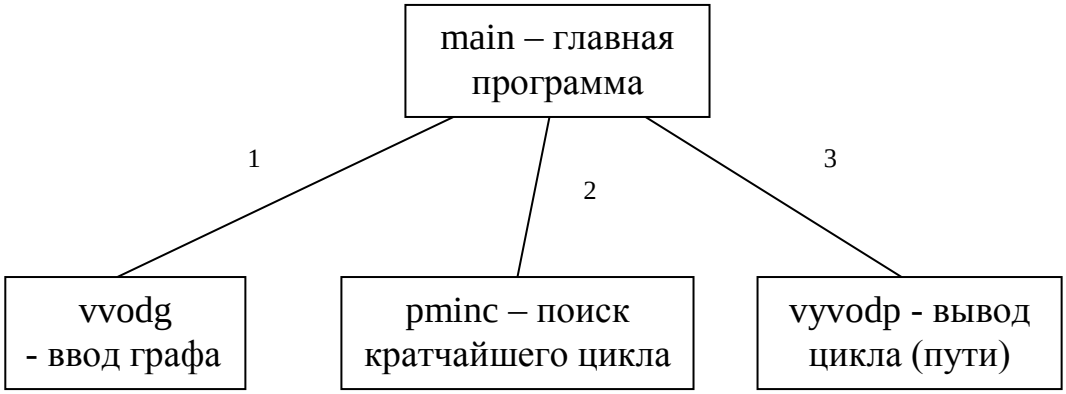

Рис. 8.1. Модульная структура программы

Выходные данные сопряжений 1 и 2 содержат код завершения (успеха), т.к. при вводе графа возможны ошибки (например, отсутствие входного файла), а поиск закончится неудачно, если в графе отсутствуют циклы.

Таблица 8.1. Сопряжения модулей программы

Номер	Вход	Выход
1		Граф, код успеха
2	Граф	Номера вершин цикла, код успеха
3	Номера вершин цикла (пути), длина пути	

Первоначальная структура программы уточняется затем в процессе проектирования модулей. Разработка обычно выполняется сверху вниз: функция текущего модуля (решаемая задача) разбивается на более мелкие функции (подзадачи), которые, в свою очередь, подвергаются разбиению и т.д., пока каждый модуль не будет иметь очевидную логическую структуру и длину приблизительно 10 - 50 строк, чтобы его исходный текст (или хотя бы исполняемые операторы) размещался на одной странице и был поэтому более наглядным.

Цель проектирования структуры - разбить программу на модули так, чтобы обеспечить максимальную степень их независимости друг от друга. Для этого необходимо, чтобы *прочность* (степень внутренних связей) каждого модуля была наибольшей, а *сцепление* между модулями (связь по данным) - как можно слабее [39].

Наилучшим по прочности является *функционально-прочный модуль*, выполняющий одну четко определенную функцию.

Нежелательно возлагать на модуль несколько различных функций, кроме случая *информационно-прочного модуля*, который выполняет все функции (операции) над некоторой структурой данных и соответствует объекту в объектно-ориентированном программировании. На языках C и Pascal информационно-прочный модуль реализуется в виде набора определений функций и данных, объединенных в отдельно транслируемый файл.

Второй способ достижения независимости модулей - ослабление сцепления между ними. В зависимости от характера данных и способа их передачи между модулями можно выделить следующие виды сцепления модулей (перечислены от лучшего к худшему).

1. Два модуля *сцеплены по данным*, если один из них вызывает другого и все входные и выходные данные передаются между ними только в виде параметров, причем скалярного типа, т. е. одиночных значений, не массивов и не структур (записей).

2. Группа модулей *сцеплена по формату*, если они используют не глобальные структуры данных одинакового строения и поэтому чувствительны к его изменению. Это могут быть, например, параметры или локальные переменные в виде массивов или структур (записей) одинакового типа.

3. Два модуля *сцеплены по управлению*, если один явно управляет работой другого, например, указывает ему код выполняемой функции, т. е. имеет информацию об его логике, что уменьшает их независимость.

4. Группа модулей, использующая общую (глобальную или внешнюю) скалярную переменную, *сцеплена по внешним данным*. Недостатки глобальных переменных заключаются в следующем: любой модуль может "испортить" значение глобальной переменной и эту ошибку трудно найти, т. к. невозможно контролировать доступ к данным; модули связаны общим именем переменной и их трудно использовать в других программах; трудно понять программу, т. к. из вызова модуля без анализа логики его и всех подчиненных ему модулей нельзя определить, какие глобальные переменные он изменяет.

5. Группа модулей, использующая общую (глобальную или внешнюю) структуру данных, *сцеплена по общей области*. К недостаткам глобальных переменных (пункт 4) добавляется зависимость всех модулей от строения общей области (пункт 2).

Избегайте глобальных переменных, передавайте данные между модулями с помощью параметров!

Проектирование модуля программы

Проектирование каждого модуля можно разбить на следующие шаги.

1. ***Внешнее проектирование*** - определение сопряжения модуля с вызывающими его модулями. Результатом внешнего проектирования всех модулей является *внутренний интерфейс* (межмодульные сопряжения) программы. Напишите для модуля заголовок подпрограммы, задав тем самым имя модуля, имена, типы и порядок записи параметров и определив тип возвращаемого модулем значения функции (либо установив отсутствие возвращаемого значения).

Определите выполняемую модулем функцию, его входные и выходные данные. Предусмотрите, каким кодом завершения модуль сообщит о неудаче при выполнении своей функции, если такое возможно.

Перечисленную информацию о модуле (его спецификацию) необходимо указать в его описании в пояснительной записке и в виде вводного комментария перед программным текстом модуля. В дальнейшем тщательно проверяйте, соответствует ли каждый вызов модуля его спецификации! Как и для всей программы, в процессе внешнего проектирования модуля составьте тесты черного ящика для его отладки (см. раздел 8.2.2).

2. ***Выбор алгоритма и структуры данных***. Прежде всего, поищите в литературе методы решения требуемой задачи. Для комбинаторных задач можно рекомендовать книги [35, 7, 13, 15 – 24, 48, 49, 59]. В отсутствие специальных методов комбинаторную задачу приходится решать методом исчерпывающего поиска - перебором вариантов. Особое внимание обратите на выбор подходящих структур данных и способов их внутреннего представления.

3. ***Описание данных***. Определите перечень участвующих в алгоритме величин, их имена и типы, напишите их определения.

4. ***Пошаговая детализация алгоритма и данных***. Проанализируйте последовательность действий при решении задачи и опишите ее в виде алгоритма. Если его трудно сразу разбить на операции выбранного языка программирования, то не углубляйтесь преждевременно в детали!

Запишите сначала алгоритм на *псевдокоде* (сочетании языка программирования с естественным языком) через более крупные операции, обозначая их предложениями естественного языка или другим удобным способом. Затем подобным же образом детализируйте эти операции до тех пор, пока весь алгоритм не будет записан на языке программирования [20, 23, 37, 67].

Укрупненный алгоритм на псевдокоде является частью описания модуля. Параллельно детализируются структуры данных. Допускается использование схем вместо псевдокода, но они менее технологичны. Детализацию алгоритма полезно сопровождать хотя бы частичным доказательством его правильности.

5. ***Ручное тестирование***. Проверьте алгоритм модуля, выполнив его вручную для простых тестов, охватывающих основные ситуации в его работе,

с записью на бумаге трассировочной таблицы, показывающей процесс изменения значений переменных [67].

6. ***Программирование***. Запишите алгоритм модуля на выбранном языке программирования, соблюдая правила хорошего стиля программирования [29 - 31, 39, 41, 67, 70].

Стиль программирования – это совокупность приемов использования средств языка и оформления программы, чтобы сделать ее легкой для чтения и внесения изменений, как самим автором, так и другими людьми. Напомним основные принципы.

1. Единообразие – выработайте единые правила и старайтесь поступать одинаково в сходных ситуациях, хотя бы в рамках одной программы. Для начала желательно заимствовать стиль опытных профессионалов [29 – 31, 41].

2. Имена переменных, функций и других объектов должны отражать их смысл. Более длинные имена используются для глобальных объектов, особенно в больших программах, а короткие – для локальных.

Желательно использовать какие-либо единые правила (собственные стандарты) выбора имен. Например, имя константы в языке С по традиции пишут заглавными буквами, часто в начале или в конце имени переменной вставляют буквы, указывающие на ее тип, например: n – целое число, p - указатель, c - символ, s – строка символов, sz – строка с нулевым байтом в конце, psz – указатель строки с нулевым байтом в конце и т. п.

3. Соблюдайте правила структурного программирования! Записывайте ветвления и циклы с помощью операторов **if**, **while**, **do-while**, **switch**, **break**, **continue** и т. п. без использования оператора **goto** [37, 67].

4. Форматируйте программу: используйте ступенчатую запись (отступы) для выявления структуры вложенности операторов. Выделяйте пустыми строчками подпрограммы и самостоятельные фрагменты программы. Используйте пробелы и скобки для наглядной записи операторов и выражений. Разбивайте сложные выражения и операторы на более простые части.

5. Пишите понятнее: избегайте запутанных приемов (“трюков”), в том числе правил умолчания; не используйте переменную в том же выражении, где она изменяется, и другие побочные эффекты. Применяйте общепринятые в данном языке способы, шаблоны (своего рода идиомы) для оформления циклов, многовариантных ветвлений и других типовых конструкций.

Например, обнуление массива из n элементов в языке С обычно пишут так:

```
for (j=0; j<n; j++)    x[j] = 0;
```

В многовариантном выборе ключевые слова else лучше не сдвигать вправо на уровень соответствующего слова if , а выравнивать по вертикали:

```
if (условие_1) {
    Ветка_1
} else if (условие_2) {
    Ветка_2
```

```
. . .
} else if (условие_n) {
    Ветка_n
} else {
    Ветка_по умолчанию
}
```

6. Комментируйте смысл всех данных, особенно глобальных, назначение каждой подпрограммы и фрагмента программы, используемые методы, инварианты циклов и другие ключевые идеи программы. Не поясняйте то, что очевидно из текста программы (например, что X++ увеличивает X на 1). Исправляйте комментарий при изменениях программы. Сдвигайте комментарий, чтобы он не мешал читать программу. Не пытайтесь исправить комментарием плохо написанную программу, лучше перепишите ее.

Написав программу, просмотрите ее и проверьте, нет ли в ней типичных ошибок. Для этого удобно использовать *контрольный список* часто встречающихся ошибок. Подобные контрольные списки распространенных ошибок полезны для проверки всех этапов разработки [41, 67]. Желательно дополнить их характерными для вас ошибками.

Пример. . В качестве примера рассмотрим этапы 1 - 6 проектирования модуля поиска цикла (замкнутого пути) минимальной длины в заданном графе (см. рис. 8.2 и табл. 8.1).

1. Внешнее проектирование. Заголовок подпрограммы модуля:

```
/*************************************************************/
/* Поиск минимального цикла в графе с количеством вершин n,  */
/* матрицей смежности g. Значение функции: 0 - цикл найден,  */
/* 1 - граф не имеет циклов. Номера вершин найденного цикла  */
/* - в векторе cmin, его длина - dcmin (3..n)                */
/*                    Д.Г. Хохлов  13.03.95                  */
/*************************************************************/
```

int pminc (**int** n, **int** g[][NMAX], **int** *dcmin, **int** cmin[])

Все входные и выходные данные модуля pminc передаются параметрами, однако, параметр g является массивом (с – адрес массива, т. е. скалярный параметр). Поэтому pminc сцеплен с вызывающими модулями по формату (соответствующий g фактический параметр должен иметь такое же строение). Здесь это приемлемо, т. к. для матрицы g требуется указать хотя бы размер строки и поэтому не удается избежать зависимости от формата, как сделано для одномерного массива cmin.

2. Выбор алгоритма и структуры данных. В литературе [33, 44, 54] не удалось найти методы поиска кратчайшего цикла, но описаны три подхода, применимые к данной задаче: поиск с возвращением, поиск в ширину и

получение степеней матрицы смежности (раздел 4). Ниже использован первый вариант.

Как часто бывает, рекурсивный алгоритм проще итеративного, но тратит больше времени и памяти за счет рекурсивных вызовов. Итеративный алгоритм предпочтительнее, если он не слишком сложен. Поэтому для поиска минимального цикла используем итеративный алгоритм 8.1.

Алгоритм 8.1. Итеративный поиск с возвращением всех решений задачи

```
k = 0;
do
if (существует еще не рассмотренный вариант v очередного шага решения)
{  e[k] = v;                /*  Вперед:   в стек                          */
   if (e[0] … e[k] - решение)
      Использовать решение e[0] … e[k];
    k++;
} else k--;                 /* Назад: удаление последнего элемента  стека */
while (k ≥ 0);              /* Стек не пуст                               */
```

3. Описание данных (представление стека в виде вектора).

```
int v[NMAX+1];          /* стек с номерами вершин текущего пути       */
int k;                  /* указатель стека v                          */
```

4. Пошаговая детализация алгоритма и данных. Выбранный алгоритм 8.1 конкретизирован в алгоритме 8.2 для задачи поиска кратчайшего цикла как перебор всех возможных путей обходом графа в глубину.

Алгоритм 8.2. Поиск с возвращением кратчайшего цикла графа

```
/*              Поиск кратчайшего цикла обходом графа в глубину            */
dcmin = n + 1;                      /* длина минимального цикла (3..n)   */
for (v[0]=0; v[0]<n; v[0]++)        /* начальная вершина: 0..n-1          */
{  /*   Обход в глубину дерева путей, начинающихся с v[0]                 */
  k = 1;  v[1] = 0;                 /* начальный номер преемников v[0] */
  do
  { Найти номер вершины vn очередного преемника v[k-1];
    if (есть vn)                    /* есть путь вперед                   */
    {  v[k] = vn;                   /* вперед: vn - в стек                */
       v[k+1] = 0;                  /* начальный номер преемников v[k] */
       if (v[0]...v[k] - цикл)
          if (k < dcmin)
          {    dcmin = k;
               c[0]...c[k] = v[0]...v[k];  /* запомнить цикл              */
          }
```

```
        k++;
    }
    else                          /* назад                          */
    {  v[k]++;                    /* следующий преемник v[k-1]      */
       k--;                       /* удалить v[k] из стека          */
    }
  while (k > 0);                  /* стек не пуст                   */
}
```

Перебор путей по алгоритму 8.2 можно сократить, как показано в алгоритме 8.3, используя следующие правила – *эвристики* (в скобках приведена соответствующая формальная запись).

1) Прекратить поиск, обнаружив цикл длиной 3 (dcmin == 3).

2) Отвергать пути длиннее минимального из найденных циклов или равные ему (i >= dcmin). Поэтому, найдя цикл, можно удалить из стека две вершины.

3) Отвергать пути, ведущие в вершину, ранее использованную в качестве начальной (vn < v[0]), т.к. все проходящие через нее циклы уже просмотрены. По этой причине перебор по возрастанию возможных номеров очередной вершины пути начинать не с 0, а с начальной вершины текущего пути (v[к+1] = v[0]).

Алгоритм 8.3. Алгоритм модуля pminc

```
int v[NMAX+1];                /* стек с номерами вершин текущего пути */
int к;                        /* указатель стека v                    */

/*          Поиск кратчайшего цикла обходом графа в глубину           */
dcmin = n + 1;                /* длина минимального цикла (3..n)      */
for (v[0]=0; v[0]<n && dcmin>3; v[0]++)      /* начальная вершина     */
{  /*        Обход в глубину дерева путей, начинающихся с v[0]        */
   k = 1; v[1] = v[0]+1;      /* начальный номер преемников v[0]      */
   do
     Найти номер вершины vn очередного преемника v[k-1];
     if (есть vn && k<dcmin)  /* есть путь вперед                     */
     {  v[k] = vn;            /* вперед: vn - в стек                  */
        v[k+1] = v[0];        /* начальный номер преемников v[k]*/
        if (v[0]...v[k] - цикл)
        {  dcmin = k;
           c[0]...c[k] = v[0]...v[k];  /* запомнить цикл              */
           k = k - 3;         /* назад: удалить 2 вершины пути        */
           v[k+1]++;          /* следующий преемник v[k]              */
        }
        k++;
```

```
      } else                          /* назад                          */
      {   v[k]++;                     /* следующий преемник v[k-1]      */
          k--;                              /* удалить v[k] из стека
        */
      }
   } while (k > 0 && dcmin>3);        /* стек не пуст                   */
}
```

5. Ручное тестирование позволило обнаружить две ошибки в первоначальной версии алгоритма 8.3.

6. Программирование. Текст программы модуля pminc поиска кратчайшего цикла графа приведен в Лекции 4 (алгоритм 4.8).

Там же рассмотрен другой вариант решения этой задачи обходом графа в ширину (алгоритм 4.9). Оба модуля имеют одинаковый интерфейс и поэтому взаимозаменяемы.

8.2.2. Отладка программы

Как бы тщательно ни проектировалась программа, она практически всегда содержит ошибки. *Отладка* (обнаружение, поиск и устранение ошибок) - самый трудоемкий и наименее изученный этап разработки программы.

Основным методом обнаружения ошибок является *тестирование*, т. е. выполнение программы со специально подобранными тестами для выявления ошибок или анализа ее работы. *Тест* состоит из исходных данных и ожидаемого правильного результата их обработки.

Полезным дополнением к тестированию являются активно разрабатываемые сейчас математические методы *верификации* (доказательства правильности) программ, которые, однако, тоже не гарантируют отсутствия ошибок.

Не рассчитывайте на отсутствие ошибок в программе! Планируйте ее отладку с самого начала разработки! Отладка имеет три основных аспекта: планирование последовательности отладки и объединения модулей, проектирование тестов, поиск и устранение ошибок.

1. Планирование отладки

План отладки программы составляется на основании ее модульной структуры, пример которой показан на рис. 8.2.

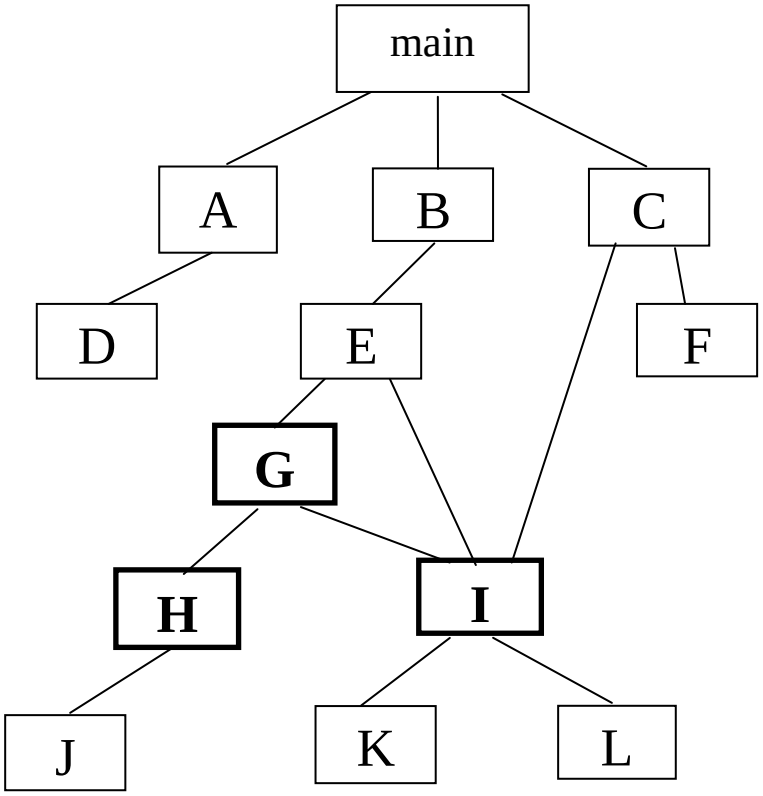

Рис. 8.2. Модульная структура программы

Для облегчения отладки перед тестированием программы в целом автономно отлаживают ее отдельные модули. Это упрощает поиск и устранение ошибок, позволяет более полно проверить модули и организовать их параллельное тестирование.

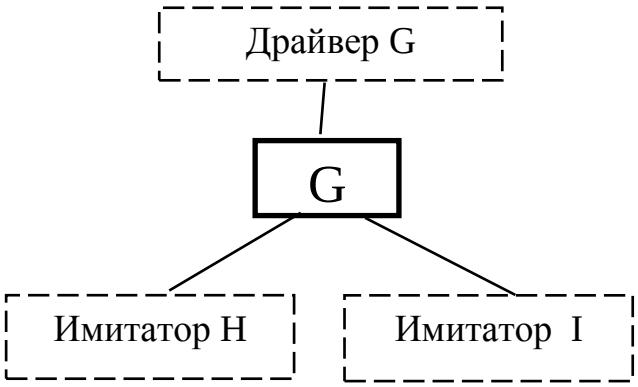

Рис. 8.3. Программа (проект) для тестирования модуля G

Для *автономного тестирования* модуля или группы модулей в общем случае требуется специальный драйвер и столько имитаторов, сколько модулей вызывается из тестируемого модуля или группы модулей.

Например, для отладки модуля G нужно собрать программу или программный проект в составе самого модуля G, написанного для него драйвера, а также имитаторов вызываемых из G модулей H и I (рис. 8.3).

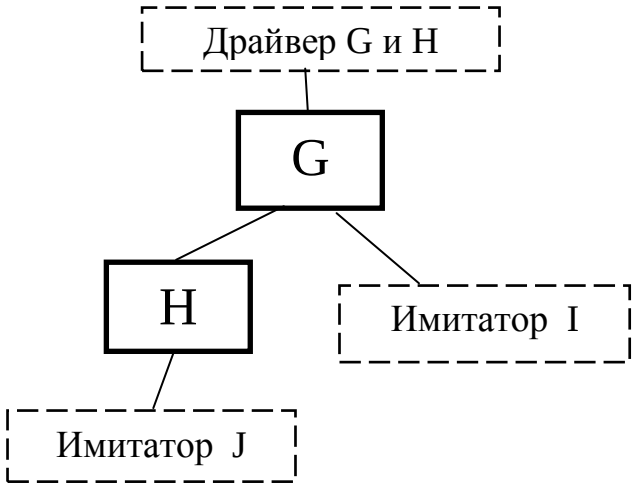

Рис. 8.4. Программа для тестирования модулей G и H

На рис. 8.4 показана программа для совместного тестирования модулей G и H, включающая имитаторы вызываемых из них модулей I и J, а также драйвер модулей G и H, который может отличаться от драйвера для одного модуля G.

Драйвер – это отладочная программа (функция main на языке C), которая запускает тестируемые модули и создает необходимую обстановку для их автономной работы. Для выполнения отлаживаемых модулей в тестовом режиме работу всех вызываемых из этих модулей подпрограмм упрощенно воспроизводят их *имитаторы*.

Драйвер состоит из нескольких обращений к тестируемому модулю, каждое из которых реализует один тест и обычно включает следующие действия:

- вывод номера и входных данных теста;
- вызов тестируемого модуля с передачей ему входных данных теста в виде параметров и/или глобальных переменных;
- вывод полученных результатов (или сообщения об итогах их сравнения с эталонным ответом).

Имитатор (заглушка) – это отладочная программа, имитирующая работу модуля, вызываемого из тестируемого модуля. В зависимости от конкретной ситуации можно использовать разные виды заглушек, например:

- пустая подпрограмма, состоящая из заголовка функции и пустого блока (полное отсутствие вызываемой функции, даже если она не нужна для работы тестируемого модуля, недопустимо из-за ошибок трансляции);
- пустая подпрограмма, выводящая сообщение о том, что она отработала, что позволяет проследить последовательность вызовов модулей;
- подпрограмма, возвращающая всегда один и тот же постоянный результат (иногда этого достаточно для проверки правильности функционирования отлаживаемого модуля);
- подпрограмма, возвращающая заранее заготовленный результат, вводимый из файла или хранящийся в массиве;

- подпрограмма, выполняющая имитируемую функцию по упрощенному алгоритму, например, более медленному или с меньшей точностью.

Возможно использование какой-либо комбинации приведенных вариантов.

Заглушка имитирует тестовую ситуацию для отлаживаемого модуля и поэтому зависит от тестов, которые частично или целиком встраиваются в нее в виде констант либо вводятся ею. Для разных тестов могут потребоваться разные заглушки.

Разработка имитаторов - более творческая задача, чем составление драйверов, имеющих обычно простую структуру даже при большом размере. Примеры заглушек и драйверов приведены в Лекции 9, разделе 9.6.

Большое значение имеет порядок тестирования модулей и их присоединения к программе. Традиционное *монолитное тестирование* (или метод *"большого скачка"*) предполагает сначала автономное тестирование каждого модуля с последующим объединением всех модулей сразу в единую программу для *комплексной отладки* связей между модулями. Комплексная отладка требует трудоемкого поиска ошибок в сопряжениях модулей, много заглушек и драйверов для автономной отладки модулей и по длительности часто превышает автономную отладку.

Лучше использовать *пошаговый метод*, при котором каждый модуль для тестирования присоединяется к набору уже отлаженных модулей, и программа собирается из модулей постепенно с более ранней отладкой сопряжений (интерфейса). Сложные и ответственные модули перед присоединением к программе отлаживаются автономно.

В зависимости от конкретной ситуации следует гибко сочетать восходящую и нисходящую стратегии пошагового метода, стараясь быстрее подключить к программе сложные модули, чтобы раньше обнаружить возможные ошибки, и модули ввода-вывода, чтобы использовать их для ввода и вывода отладочных данных.

При *восходящей стратегии* (снизу вверх) сначала автономно отлаживаются модули самого нижнего уровня, не имеющие вызовов других модулей. Затем к ним постепенно (в предельном случае по одному) присоединяются модули более высоких уровней (все подпрограммы которых уже присоединены), кончая главным модулем. В этом случае не требуются заглушки. При *нисходящей стратегии* (сверху вниз) программа наращивается вниз, начиная с главного модуля, не требуются драйверы.

В качестве примера рассмотрим план отладки программы поиска минимального цикла в графе (см. раздел 8.2.1).

В данной программе затруднительно применение в чистом виде рекомендуемой выше стратегии тестирования - возможно более быстрого присоединения к программе сложных модулей с их предварительной автономной отладкой. Дело в том, что наиболее сложный модуль pminc имеет громоздкие входные и выходные данные. Для его автономной отладки эти данные пришлось бы передавать в этот модуль и из него с помощью

имитаторов и драйверов, близких по возможностям и сложности к модулям main, vvodg и vyvodp. Поэтому pminc удобно отлаживать после модулей main, vvodg и vyvodp или совместно с ними.

Работа модулей main и vvodg тесно переплетена: main сообщает об ошибках, обнаруженных модулем vvodg. При автономной отладке каждого из них, особенно при вводе ошибочных данных, драйвер или имитатор практически должен дублировать работу другого модуля. Поэтому их удобно отлаживать совместно. Модуль vyvodp прост и не требует автономной отладки.

Из приведенных соображений можно принять следующий план отладки в три этапа.

1. Тестирование модуля vvodg с драйвером dvvodg для тестов с правильными входными данными. Для проверки правильности ввода требуется модуль вывода матрицы смежности, который полезно использовать на всех этапах отладки.
2. Совместное тестирование модулей pminc, vyvodp и main с модулем vvodg (т. е. всей программы) на "правильных" тестах.
3. Тестирование всей программы для совместной отладки модулей main и vvodg на "неправильных" тестах.

2. Проектирование тестов

Методы и приемы проектирования тестов и отладки простых программ рассмотрены в первой части учебника [67]. Поэтому ограничимся напоминанием основных принципов. Их можно использовать для отладки модулей.

Тестирование может доказать наличие ошибок, но не их отсутствие!

Проектирование тестов – проблема экономическая: полное тестирование невозможно, и необходимо небольшим, но представительным набором тестов обнаружить основные ошибки, что является *творческой* задачей. Полная формализация здесь невозможна!

Нужны тесты с высокой вероятностью обнаружения ошибок. При коллективной работе нежелательно, чтобы тесты составлял автор программы. Во-первых, ему психологически трудно настроиться на придирчивое отношение к своей программе, и, во-вторых, неверно поняв задачу, он может сделать в тестах такие же ошибки, как и в программе, и они не будут обнаружены.

Основные тесты должны включать систематически подобранные, а не случайные входные данные!

Для отдельного модуля или небольшой программы (например, при курсовом и дипломном проектировании) рекомендуется сначала разработать

тесты на основе спецификации программы (*методы черного ящика*), а затем дополнить их тестами на основе логики программы (*методы белого ящика*), чтобы набор тестов покрывал основные пути выполнения алгоритма (все пути перебрать невозможно). В этом случае разработка тестов состоит из двух этапов.

1. **Методы черного ящика**. На основе спецификации программы готовится тест для каждой возможной ситуации (комбинации условий) во входных и выходных данных. Учитывается каждая область допустимых и недопустимых значений входных данных и область изменения выходных данных программы.

Основную роль здесь играет *метод эквивалентного разбиения*. Его идея – разбить *пространство тестов* (т.е. пространство входных данных и соответствующих результатов программы) на классы эквивалентных тестов и использовать по одному представителю из каждого класса.

Два или несколько тестов считаются *эквивалентными*, если с их помощью обнаруживаются одинаковые ошибки и не обнаруживаются одинаковые ошибки. Строго говоря, разные тесты не могут быть эквивалентными (всегда можно придумать программу, реагирующую на них по-разному). Поэтому данный метод является не формальным, а эвристическим.

Необходимы тесты для каждой области недопустимых входных данных. Такие *“неправильные” тесты* зачастую более эффективно выявляют ошибки в программе, чем *“правильные” тесты*. “Правильный” тест может охватить несколько тестовых ситуаций. “Неправильный” тест должен содержать не более одной ошибки в данных, т. к., обнаружив первую ошибку, программа изменяет режим работы, может не найти другие ошибки, и правильность реакции на них останется не проверенной.

Дополнительно используются *тесты граничных условий* в районе границ областей эквивалентности (проверка на краевой эффект). Полезен метод *предположений об ошибке* – создать такую тестовую ситуацию для программы, которая могла не учитываться разработчиком.

При наличии сложных связей входных и выходных данных хорошим систематическим методом построения тестов является *метод функциональных диаграмм* [40, 45].

Тесты черного ящика полезно иметь до разработки алгоритмов на этапе составления внешней спецификации программы. Это способствует более четкой постановке задачи, служит критерием качества спецификации. Если затруднительно определить реакцию программы на исходные данные (результат ее работы для этого случая), то необходимо уточнить спецификацию.

Тесты нужны не только для отладки, но и для оценки характеристик программы: времени работы, объема памяти и др. Для этого удобны и тесты со случайными данными, которые могут также создать ситуации, упущенные при систематической разработке тестов. Полезно также *стрессовое тестирование* –

ввод большого объема сгенерированных автоматически исходных данных (в том числе случайных и некорректных).

Как пример, рассмотрим построение тестов для программы нахождения кратчайшего цикла в графе (см. раздел 8.2.1). В этом случае удобен метод эквивалентного разбиения. Рассмотрим каждую входную и выходную величину, как один из элементов (координату) многомерного пространства данных и выделим границы областей эквивалентности по этой координате.

Каждая область эквивалентности рассматривается как тестовая ситуация, которую необходимо создать для отлаживаемой программы хотя бы одним тестом. Такие тестовые ситуации приведены в табл. 8.3.

“Неправильные” классы эквивалентности, соответствующие “неправильным” тестам, выделены в таблице в отдельный столбец, поскольку каждый тест может создавать несколько “правильных” тестовых ситуаций, но не более одной “неправильной”.

Первой входной величиной является количество вершин графа n, которое должно находиться в диапазоне от 1 до NMAX. Для этой величины имеются три области эквивалентности (в скобках указаны их номера) - одна “правильная”: $1 \le n \le NMAX$ (1) и две “неправильные”: $n < 1$ (2) и $n > NMAX$ (3). Желательно поэтому, кроме других значений, тестировать программу для граничных n = 0, 1, NMAX, NMAX+1. Аналогично выявляются тестовые ситуации с номерами до 16.

Каждое сообщение программы должно появляться хотя бы один раз при ее отладке и поэтому образует самостоятельную тестовую ситуацию (от 17 до 26), причем сообщениям об ошибках соответствуют “неправильные” ситуации (от 20 до 26).

Таблица 8.3.

Области входных / выходных данных тестов программы

Входное / выходное условие (значение)	"Правильные классы эквивалентности	"Неправильные" классы эквивалентности
Количество вершин n	1..NMAX (1)	<1 (2), >NMAX (3)
Количество ребер	0..n*(n-1) (4)	> n*(n-1) (5)
Кол. вершин в ребре	2 (6)	< 2 (7), > 2 (8)
Номер вершины	0..n-1 (9)	< 0 (10), > n-1 (11)
Наличие петель	нет (12)	есть (13)
Наличие циклов	есть (14), нет (15)	
Минимум длины цикла	3..n (16)	
Сообщения программы	1 (17), 2 (18), 3 (19),	4 (20), 5 (21), 6 (22), 7 (23), 8 (24), 9 (25), 10 (26)
Имя входного файла	Дано (27), не дано (28)	
Входной файл	Существует (29),	не существует (30)

	не пустой (31)	пустой (32)
Имя выходного файла	Дано (33), не дано (34)	
Выходной файл	Существует (35), не существует (36), не пустой (37), пустой (38)	

Ситуации с номерами от 27 до 38 возникают при различных способах записи команды запуска отлаживаемой программы и разных состояниях входного и выходного файлов.

Затем был разработан по возможности минимальный набор из 12 тестов, обеспечивающий создание каждой из полученных тестовых ситуаций (табл. 8.4). Сначала использовались тесты, составленные еще при постановке задачи, охватывающие несколько тестовых ситуаций, которые указаны в правом столбце табл. 8.4. Затем выявлялись не охваченные ими ситуации и подбирались тесты для этих ситуаций до тех пор, пока не были охвачены все ситуации.

После этого методами белого ящика проверяется полнота разработанного набора тестов.

2. **Методы белого ящика**. Набор тестов строится так, чтобы он удовлетворял выбранному критерию покрытия алгоритма, т. е. в достаточной мере охватывал логику программы – возможные пути выполнения алгоритма.

Таблица 8.4

Тесты черного ящика для отладки программы

N	Вход	Выход	Основные ситуации
1	n=5 (рис. 2.1, 2.3) Ребра: 0-2 4-2 2-3 4-3 2-0 1-3 1-0 pkrc <t1 >rt1	Min цикл длиной 3 2, 3, 4, 2 Сообщения: 1, 3, 10	1, 4, 6, 9, 12, 14, 16, 17, 19, 25, 27, 29, 31, 33, 36
2	n=0 pkrc	Сообщения: 5, 8	2, 21, 24, 28, 34
3	n=NMAX+1 pkrc <t3 >rt3 - пуст	Сообщения: 5,8	3, 21, 24, 27, 29, 31, 33, 35, 38
4	n=2 ребра: 0-0 0-1 1-0 1-1	Сообщения: 2, 3, 10	5, 6, 9, 13, 15, 18, 19, 26
5	n=NMAX ребра: 0-1 1-2 2	Сообщения: 3, 7, 8	1, 4, 7, 9, 12, 19, 23, 24
6	n=5 ребра: 2-1-3 4-0	Сообщения: 3, 7, 8	1, 4, 8, 9, 12, 19, 24
7	n=4 ребра: -1-3 2-0	Сообщения: 3, 6, 8	1, 4, 6, 9, 10, 19, 22, 24
8	n=4 ребра: 4-1 2-0	Сообщения: 3, 6, 8	1, 4, 6, 9, 11, 19, 22, 24
9	pkrc <f - не существует	DOS: file not found	27, 30, 34
10	pkrc <t10 - пуст	Сообщения: 3, 4, 8	19, 20, 24, 27, 32, 34

11	¦ n=1	Сообщения: 2, 3	1, 4, 14 и др.
12	n=4 Ребра: 0-1 1-2 2-3 3-0 pkrc >rt12 - не пуст	Min цикл длиной 4: 0, 1, 2, 3, 0 Сообщения: 1, 3	1, 4, 6, 9, 12, 14, 16, 17, 19, 29, 31, 35, 37

Примечание: ребра записаны через тире для наглядности, входные данные не содержат тире.

Критерий *покрытия операторов* требует, чтобы каждый оператор программы выполнялся при ее отладке хотя бы один раз.

Более сильным критерием является *покрытие решений* (или *переходов*), т.е. выполнение во время отладки каждого условного перехода программы (разветвления алгоритма) в каждом возможном направлении. Он включает в себя покрытие операторов.

Критерий *комбинаторного покрытия условий* требует, чтобы тестировались все возможные комбинации значений составных условий циклов и ветвлений. Этот критерий включает оба предыдущих критерия.

Тесты должны охватывать основные пути выполнения алгоритма. Каждый цикл желательно (если это возможно) тестировать с нулевым, единичным и максимальным числом повторений. Желательно проверить чувствительность алгоритма к особым значениям данных (нулевым, отрицательным и т. п., пустым файлам), а также выход за верхние и нижние границы индексов и числовых значений.

Таблица 8.5

Комбинаторное покрытие условий тестами черного ящика

Модуль	Элементарное условие	Номера тестов	
		Истина	Ложь
main	kz > 2	2,3,5 и др.	1, 4 и др.
main	kz == 3	2, 3	1,4,5 и др.
main	kz == 1	1, 4	12
main	i < n	1, 4, 12	1,4,12
main	pminc (n, g, &dcmin, cmin)	4, 11	1, 12
pminc	v[0]<n	1, 4, 11, 12	4,11
pminc	*dcmin>3	1, 12	1,4,11,12
pminc	vn<n	1, 12	1,12
pminc	g[j][vn]==0	1, 12	1
pminc	k>1	1, 12	1,12
pminc	vn==v[k-2]	1, 12	1,12
pminc	vn<n	1, 12	1,12
pminc	k<*dcmin	1, 12	1
pminc	v[0]==v[k]	1, 12	1,12
pminc	k>0	1, 12	1,12
pminc	j<=k	1, 12	1,12

pminc	k > 0	1, 12	1
pminc	*dcmin>3	1, 12	1,12
pminc	*dcmin > n	1, 12	4,11
vvodg	feof(stdin)	10	остальные
vvodg	*n<1	2	1,3,4 и др.
vvodg	*n>NMAX	3	1,2,4 и др.
vvodg	i<*n	1, 4, 5 и др.	1,4,5 и др.
vvodg	j<*n	1, 4, 5 и др.	1,4,5 и др.
vvodg	! feof(stdin)	1, 4, 5 и др.	1,4,11,12
vvodg	i<0	7	1,4,5 и др.
vvodg	i>=*n	8	1,4,5 и др.
vvodg	feof(stdin)	5, 6	1,4,5 и др.
vvodg	j<0	-	1,4,5 и др.
vvodg	j>=*n	-	1,4,5 и др.
vvodg	g[i][j]	1	1,4,5 и др.
vyvodp	i<kvp	1,12	1,12

Проиллюстрируем эти правила на примере тестов для программы поиска кратчайшего цикла графа. В программе имеются составные условия. Поэтому для разработки тестов белого ящика использован критерий комбинаторного покрытия условий (см. табл. 8.5). В таблицу выписаны все элементарные условия из всех условных операторов и циклов каждого модуля программы.

Например, в тексте модуля pminc (см. алгоритм 4.8) имеется фрагмент

```
for (vn=v[k];  vn<n && (g[j][vn]==0 || k>1 && vn==v[k-2]);  vn++) ;
if (vn<n && k<*dcmin)            /* можно вперед                    */
```

содержащий шесть элементарных условий:

vn<n, g[j][vn]==0, k>1, vn==v[k-2], vn<n, k<*dcmin,

каждое из которых выписано в отдельную строку таблицы. Для каждого условия в таблице указаны номера тестов, при пропуске которых это условие принимает значения “истина” и “ложь”. При некоторых тестах в разные моменты условие может принимать оба этих значения. Необходимо убедиться в том, что при отладке каждое элементарное условие программы принимает оба возможных значения хотя бы по одному разу.

Из табл. 8.5 видно, что в модуле vvodg два условия: j<0 и j>=*n во время отладки ни в одном из разработанных тестов черного ящика не принимают значение “истина”. Поэтому для покрытия значения “истина” этих условий дополнительно разработаны тесты 13 и 14, приведенные в таблице 8.6.

Таблица 8.6

Тесты белого ящика для отладки программы

N	Вход	Выход	Основные ситуации
13	n=4 ребра: 3- -1 2-0	Сообщения: 3, 6, 8	1, 4, 6, 9, 10, 19, 22, 24
14	n=4	Сообщения: 3, 6, 8	1, 4, 6, 9, 11, 19, 22,

	ребра: 1-4 2-0		24

К отладочным средствам, кроме тестов, относятся драйверы, заглушки, программы вывода отладочной информации, вспомогательные файлы и др. Перечень таких средств зависит от последовательности тестирования модулей и определяется планом отладки.

3. Поиск и устранение ошибок

Необходимо убедиться в наличии информации о входных, промежуточных и выходных данных, достаточной для выявления и локализации (поиска) ошибок. Для обнаружения ошибок следует тщательно проверять результаты программы! Ищите малейшие отклонения от ожидаемого результата - симптомы ошибок. Ошибки могут быть не только в программе, но и в тестах. Выявив все симптомы, можно приступать к поиску причины и местоположения ошибки в программе.

Ищите ошибки, прежде всего, методами логических рассуждений! В методе *индукции* выявляют закономерности в симптомах, на их основе выдвигают гипотезу об ошибке и объясняют симптомы с помощью этой гипотезы. В процессе *дедукции*, наоборот, сначала выдвигаются несколько гипотез о причинах ошибки, а затем путем анализа симптомов исключаются неверные гипотезы.

Эффективный метод локализации ошибки - *обратная прокрутка* программы: от места получения неверного результата – назад, туда, где программа сбилась в первый раз. Для дополнительной информации придумываются специальные тесты.

Если логический анализ не помог найти ошибку, вставьте в программу отладочный вывод промежуточных данных и снова пропустите тот же тест. Позаботьтесь о наглядности вывода. Менее эффективны автоматические отладочные средства, например, пошаговое исполнение программы или контрольные точки. Их эффект выше при предварительном ручном тестировании программы, когда разработчик хорошо знаком с деталями ее функционирования.

Отладочные средства: тесты, заглушки, драйверы, операторы отладочного вывода и т. п., необходимо сохранять после отладки, чтобы облегчить сопровождение программы! При исправлении ошибки остерегайтесь внести новые ошибки! После любого изменения программу необходимо снова тщательно тестировать.

8.3. Объектно-ориентированное программирование

8.3.1. Основные концепции

Объектно-ориентированное программирование (ООП) основано на понятии абстрактного типа данных [14, 36, 42 – 44, 77].

Абстрактным типом данных называют тип данных, определяемый функционально: только через операции над объектами этого типа без описания способа представления их значений.

В ООП абстрактные типы данных называют *классами*. Определение класса перечисляет его члены (элементы): *члены-данные* для представления объектов этого класса и *члены-функции* - операции над такими объектами. Класс – это обобщение понятия структуры и его определение в C++ синтаксически подобно определению структуры и объединения (смеси) - union в языке C. Данные объекта определяют его состояние, а набор допустимых операций описывает поведение объекта. Класс соответствует понятию информационно-прочного модуля [39] или “исполнителя” [36].

Тремя основными концепциями ООП являются: инкапсуляция, наследование и полиморфизм.

Инкапсуляция – это сокрытие описания реализации объекта (данных, программ) от использующих объект модулей, предоставление им строго ограниченного набора операций (функций) над объектом.

Наследование представляет собой механизм для определения новых (производных) типов данных (классов) на основе существующих (базовых). *Производный класс (потомок)* наследует все свойства *базового (родительского) класса*, т. е. его характеристики (члены-данные) и набор операций (члены-функции).

Пример. Из базового типа "животные" с характеристиками: вес, размер, цвет и т. п. и действиями: передвижение, размножение, питание и т. п. можно породить типы: "млекопитающие", "рыбы", "птицы", имеющие кроме базовых, свои характеристики: ноги у млекопитающих, плавники у рыб, крылья у птиц и т. д.

Полиморфизм ("многоформенность") - это использование одной и той же функции или операции для выполнения по-разному действий над параметрами или операндами разных типов.

Например, все "животные" выполняют операцию "передвижение", но по-разному: млекопитающие бегают, рыбы плавают, птицы летают.

Перегрузка - это использование одинакового обозначения для разных операций (или функций), различающихся количеством и типами операндов или параметров. Для реализации полиморфизма необходима перегрузка. Перегрузка же служит не только для полиморфизма. Она является обычным явлением любого языка программирования. Например, в выражении с двумя операндами X-Y знак "-" обозначает операцию вычитания, а в выражении с одним операндом -X этот же знак обозначает операцию изменения знака числа. Таким образом, этот знак перегружен.

Объектно-ориентированное программирование позволяет:

- упростить развитие разработанных программ;
- упростить использование предыдущих разработок (определений классов) в новых проектах благодаря наследованию.

Объектно-ориентированное программирование - это сложная технология, оправданная для достаточно сложных проектов. Для знакомства с ООП далее рассмотрен небольшой пример.

8.3.2. Объектно-ориентированные средства языка C++

Для первого знакомства с ООП на языке C++ рассмотрим его на примере задачи вычисления постфиксного выражения (см. раздел 7.2.2).

Пример 8.1. C++ программа "Класс Стек. Интерпретация постфиксного выражения". Задача: описать класс "Стек". Использовать его для вычисления постфиксного выражения. Решение этой задачи дано в программе 8.5.

Алгоритм 8.5.

```
//              Пример 8.1. C++ программа: Класс "Стек"                1
const int N=20;              // Максимальный размер стека              3
//           Определение класса Стек                                    4
typedef int T;               // Тип элементов стека                    5
class TStack                 // Стек                                   6
{  T    st[N];               // Отображающий вектор стека              7
   int  ist;                 // Указатель стека                        8
   public:                   //                                        9
   TStack (void) {ist=-1;};      // Конструктор                        10
   ~TStack () {};                // Деструктор (необязателен)          11
   int push (T z);               //      Стек <== z                    12
   int pop  (T &z);              //      Стек ==> z                    13
};                               //                                    14
//                Операции над стеком                                  15
int TStack :: push (T z)   // Вталкивание в стек величины z            16
{  if (ist < N-1)                    // Есть место                     17
   {  ist++;  st[ist]=z;  return 0; }    //  Стек <== z               18
   else  return 1;                   //  Переполнение стека           19
}                                    //                                20
int TStack :: pop (T& z)       // Выталкивание из стека величины z      21
{  if (ist >= 0)                          // Есть элементы             22
   {  z=st[ist];  ist--;  return 0; }     //  Стек ==> z               23
   else  return 1;                       // Стек пуст                  24
}                                        //                            25
//                Пример использования стека                           26
// Вычисление постфиксного выражения с числами не более 9              27
```

```
#include <stdio.h>                          //                                  28
int main (void)                            //                                  29
{ int t;                                   //                                  30
  T   x, y;                                //                                  31
  TStack oprn;                             // Стек операндов                   32
  while ((t=getchar())!='\n' && t!=EOF)    //                                  33
  {  if (t>='0'&& t<='9')          // t - операнд: втолкнуть в стек            34
       oprn.push (t-'0');          //                                          35
     else                          // t - операция: выполнить в стеке          36
     {  oprn.pop (y);  oprn.pop (x);       //                                  37
        switch (t)                         //                                  38
        {  case '+': x += y;  break;       //                                  39
           case '-': x -= y;  break;       //                                  40
           case '*': x *= y;  break;       //                                  41
           case '/': if (y)  x /= y;  else  x = 0;   //                        42
        }                                  //                                  43
        oprn.push (x);                     //                                  44
     }                                     //                                  45
  }                                        //                                  46
  oprn.pop (x);                            //                                  47
  printf ("= %d\n", x);                    //                                  48
  return 0;                                //                                  49
}                                          //                                  50
```

Пояснения к программе (алг. 8.5). В скобках указаны номера строк.

1. *Определение класса* подобно определению структуры языка С, но может содержать не только данные, но и функции (строки 6-25). Класс TStack содержит два члена данных: отображающий вектор стека st и указатель стека ist, и две функции: push() - вталкивание элемента, pop() - выталкивание элемента.

Определение класса (как и структуры) задает новый тип и не выделяет память. Память выделяется при создании объектов, например, при определении переменной oprn класса TStack (32).

2. *Метки режима доступа* к членам класса (используются многократно в любом порядке, действуют до следующей метки или конца определения):

- **public** (открытый, общедоступный) - доступ разрешен всем функциям (для интерфейса объектов с программой);
- **private** (закрытый, частный) - доступ разрешен только функциям-членам и функциям-друзьям данного класса (9);
- **protected** (защищенный) - доступный функциям-членам и функциям-друзьям данного класса и производных из него классов.

При определении класса с помощью слова **class** его члены по умолчанию **private**, при использовании слова **struct** - его члены по умолчанию **public**, но

могут определяться как **private**; при использовании **union** его члены могут быть только **public**. Обычно данные объявляют как **private**, функции - **public**.

3. Конструктор и деструктор. Кроме других функций, каждый класс имеет конструктор и деструктор (явно или неявно).

Конструктор – это функция-член класса, автоматически вызываемая при создании объектов этого класса для их инициализации; может также динамически выделять память для объекта; имя конструктора совпадает с именем класса.

Деструктор – это функция-член класса, автоматически вызываемая при уничтожении объектов этого класса и освобождении выделенной конструктором памяти; имя деструктора имеет вид: ~имя-класса (знак ~ называется “тильда”).

Конструктор и деструктор не имеют возвращаемого значения и тип их значения не указывается. Конструктор может перегружаться и иметь параметры (т. е. класс может иметь несколько конструкторов). Деструктор не имеет параметров.

Другой вариант программы использует перегрузку операций. *Перегрузка операций* **(operator)**. Действия над классами можно обозначать не только функционально, но и в форме операции, перегрузив имеющуюся в C++ операцию (кроме: . .* :: ?:). При этом сохраняется количество операндов, приоритеты операций, и смысл операций для встроенных типов данных. Хотя бы один операнд перегруженной операции должен быть объектом (иметь тип класса) или элементом класса.

В программе алг. 8.5 можно заменить имена функций push и pop на operator << и operator >> (изменения в строках 12-47). Измененный вариант представлен в программе алг. 8.6.

```
//              Алгоритм 8.6.  C++ программа
// Класс "Стек" с использованием перегрузки операций          1
// Файл: stckvpf0.cpp              Д.Г. Хохлов 28.04.97       2
const int N=20;              // Максимальный размер стека    3
//           Определение класса Стек                          4
typedef int T;               // Тип элементов стека          5
class TStack                 // Стек                         6
{  T    st[N];               // Отображающий вектор стека    7
   int  ist;                 // Указатель стека              8
   public:                   //                              9
   TStack (void) {ist=-1;};      // Конструктор              10
   ~TStack () {};                // Деструктор (необязателен) 11
   int operator << (T  z);       //                          12
   int operator >>  (T &z);      //                          13
};                               //                          14
//                Операции над стеком                        15
int TStack::operator<<(T z)     //Вталкивание в стек величины z  16
```

```
{  if (ist < N-1)                  // Есть место                    17
   {  ist++;  st[ist]=z;  return 0; }        //                    18
   else  return 1;                 // Переполнение стека            19
}                                  //                               20

int TStack::operator>>(T& z)  //Выталкивание из стека величины z 21
{  if (ist >= 0)                           // Есть элементы         22
   {  z=st[ist];  ist--;  return 0; }      //                       23
   else  return 1;                         // Стек пуст             24
}                                          //                       25

//              Пример использования стека                          26
// Вычисление постфиксного выражения с  числами не более 9         27
#include <stdio.h>                         //                       28
int main (void)                            //                       29
{ int t;                                   //                       30
  T   x, y;                                //                       31
  TStack oprn;                             // Стек операндов        32
  while ((t=getchar())!='\n' && t!=EOF)    //                       33
  {  if (t>='0'&& t<='9')          // t - операнд: втолкнуть в стек 34
       oprn << (t-'0');            //                               35
     else                          // t - операция: выполнить в стеке 36
     {  oprn >> y;  oprn >> x;               //                     37
        switch (t)                           //                     38
        {  case '+': x += y;  break;         //                     39
           case '-': x -= y;     break;      //                     40
           case '*': x *= y;    break;       //                     41
           case '/': if (y)  x /= y;  else  x = 0;   //             42
        }                                    //                     43
        oprn << x;                           //                     44
     }                                       //                     45
  }                                          //                     46
  oprn >> x;                                 //                     47
  printf ("= %d\n", x);                      //                     48
  return 0;                                  //                     49
}                                            //                     50
```

Шаблоны классов (другие названия: генераторы классов, обобщенные классы, параметризованные типы) описывают структуру семейства классов, по которой компилятор создаст классы в зависимости от параметров настройки.

Например, в программе алг. 8.5 можно заменить строки 4 - 6, 16 - 32 для использования шаблонов класса. Измененный вариант приведен в программе алг. 8.7.

```
//              Алгоритм 8.7. C++ программа
// Класс "Стек" с использованием шаблонов классов                   1
```

```
// Файл: stckvpf0.cpp              Д.Г. Хохлов 28.04.97                2
const int N=20;                  // Максимальный размер стека          3
//            Определение шаблона класса Стек                          4
template <class T>               // Тип элементов стека                5
class TStack                     //                                    6
{  T    st[N];                   // Отображающий вектор стека          7
   int  ist;                     // Указатель стека                    8
   public:                       //                                    9
   TStack (void) {ist=-1;};      // Конструктор                        10
   ~TStack () {};                // Деструктор (необязателен)          11
   int push (T z);               //                                    12
   int pop  (T &z);              //                                    13
};                               //                                    14
//                Операции над стеком                                  15
template <class T>
int TStack :: push (T z)         // Вталкивание в стек величины z      16
{  if (ist < N-1)                // Есть место                         17
   {  ist++;  st[ist]=z;  return 0; }        //                        18
   else  return 1;                           // Переполнение стека     19
}                                            //                        20
template <class T>
int TStack :: pop (T& z)  // Выталкивание из стека величины z          21
{  if (ist >= 0)                             // Есть элементы          22
   {  z=st[ist];  ist--;  return 0; }        //                        23
   else  return 1;                           // Стек пуст              24
}                                            //                        25
//                Пример использования стека                           26
// Вычисление постфиксного выражения с  числами не более 9            27
#include <stdio.h>                           //                        28
int main (void)                              //                        29
{ int t;                                     //                        30
   T   x, y;                                 //                        31
   TStack <int> oprn;            // Стек операндов                     32
   while ((t=getchar())!='\n' && t!=EOF)     //                        33
   {  if (t>='0'&& t<='9')       // t - операнд: втолкнуть в стек      34
        oprn.push (t-'0');       //                                    35
      else                       // t - операция: выполнить в стеке    36
      {  oprn.pop (y);  oprn.pop (x);        //                        37
         switch (t)                          //                        38
         {  case '+': x += y;  break;        //                        39
            case '-': x -= y;  break;        //                        40
            case '*': x *= y;  break;        //                        41
            case '/': if (y)  x /= y;  else  x = 0;   //               42
         }                                   //                        43
```

```
      oprn.push (x);                    //          44
    }                                   //          45
  }                                     //          46
  oprn.pop (x);                         //          47
  printf ("= %d\n", x);                 //          48
  return 0;                             //          49
}                                       //          50
```

В случае использования шаблонов классов также можно использовать перегрузку операций. Например, в программе алг. 8.7. для использования операций << и >> вместо функций push и pop, достаточно заменить строки 16 и 21 на следующие строки:

```
template <class T>
int TStack<T>::operator<<(T  z)     // Вталкивание в стек величины z     16
template <class T>
int  TStack<T>::operator>>(T&  z)        //  Выталкивание  из  стека
величины z  21
```

8.4. Упражнения и задачи

8.1. *"Формулы"*

Требуется составить программу, которая вводит выражение и упрощает его описанным ниже способом. В выражении могут использоваться: целые числа, имена переменных, операции: сложение "+", вычитание "-", умножение "*", целочисленное деление "/", не более 10 уровней круглых скобок "(" и ")". Выражение записывается без пробелов. Два знака операции подряд не допускаются. Имя переменной состоит из заглавных и/или строчных латинских букв и/или цифр и начинается с буквы. Строчная и заглавная буквы считаются различными. Операции с одинаковым приоритетом выполняются слева направо. Длина имен и чисел, в том числе результатов операций, не превышает 9 символов.

Требуется реализовать следующие преобразования (каждый пункт включает все предыдущие пункты как частные случаи).

1. Вычислить значение выражения, не содержащего переменных.

Примеры входного файла	Ожидаемый выходной файл
25/6	4
10-2*(100-200/(18-5*2))+1	-139

2. Упростить выражение, содержащее переменные и числа, выполнив операции с числовыми операндами. Полученное выражение вывести без лишних скобок.

Примеры входного файла	Ожидаемый выходной файл
10+(3*6)-(X+5-1)/2	28-(X+5-1)/2

Операция 5-1 не выполнена, поскольку, по условию задачи, сначала должно выполняться сложение!

(10+3*6)-(X+(5-1))/2	28-(X+4)/2

3. К перечисленным выше операциям добавить унарную (с одним операндом) операцию изменения знака "-", имеющую наивысший приоритет, т. е. выполняемую раньше других операций.

Пример входного файла	Ожидаемый выходной файл
-(-(2*5))+X	10+X

4. К перечисленным выше операциям добавить операцию сравнения "=" с самым низшим приоритетом. Результат сравнения равен 1, если операндами являются равные числа или эквивалентные выражения, и равен 0 при сравнении неравных чисел. Сравнение не выполняется, если операндами являются неэквивалентные выражения.

Примеры входного файла	Ожидаемый выходной файл
25=5*5	1
2*3=2+3	0
-(-(2*5))+X=10+X	1

-(-(2*5))+X=X+10	10+X=X+10

5. Реализовать сравнение выражений с учетом коммутативности (перестановочного закона) для сложения и умножения.

Пример входного файла	Ожидаемый выходной файл
-(-(2*5))+X=X+10	1

6. Упростить выражение, используя тождества:

X+0 = 0+X = X, X*1 = 1*X = X, X*0 = 0*X = 0.

Пример входного файла	Ожидаемый выходной файл
10-5*2+A=A	1

7. Перевести выражение в последовательность команд на языке ассемблера простого компьютера. Компьютер имеет один сумматор и 10 регистров: R0, R1, R2, ..., R9. Каждая команда начинается с новой строки и содержит код операции из одной или двух букв, за которым через один пробел может следовать операнд. Операндом может быть либо регистр либо имя переменной (не совпадающее с регистром). Имеются следующие команды.

Команда	Смысл команды
L операнд	сумматор = операнд
A операнд	сумматор = сумматор + операнд
S операнд	сумматор = сумматор - операнд
M операнд	сумматор = сумматор * операнд
D операнд	сумматор = сумматор / операнд
N	сумматор = - сумматор
ST регистр	регистр = сумматор

Со второй строки вывести в выходной файл полученную ассемблерную программу, которая вычисляет значение выражения и помещает его в сумматор. Программа должна содержать все операции выражения (экономия операций не допускается).

Пример входного файла	Ожидаемый выходной файл
A*B-C*(B+X)	A*B-C*(B+X) L A M B ST R0 L B A X ST R1 L C M R1 ST R1 L R0 S R1

Указание. Модули (подпрограммы) удобно разрабатывать и отлаживать поэтапно: сначала получение и вывод дерева выражения, затем его преобразования.