МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
федеральное государственное бюджетное образовательное учреждение
высшего образования
«УЛЬЯНОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

Т. Е. Родионова

ТЕХНОЛОГИИ ПРОГРАММИРОВАНИЯ

учебное пособие
для студентов направления 01.03.04

Ульяновск
УлГТУ
2018

УДК 004.4(075)
ББК 32.973 – 018.2я7
Р 60

Рецензенты: заведующий кафедрой «Механика и теория управления» УлГУ, доктор физико-математических наук, профессор А. С. Андреев; директор Ульяновского филиала «Института радиотехники и электроники им. В.А. Котельникова РАН» доктор технических наук, доцент В.А. Сергеев

Утверждено редакционно-издательским советом университета в качестве учебного пособия

Родионова, Татьяна Евгеньевна
Р60 Технологии программирования : учебное пособие для студентов направления 01.03.04 / Т. Е. Родионова. — Ульяновск : УлГТУ, 2018. — 115 с.

ISBN 978-5-9795-1789-6

Пособие предназначено для подготовки студентов направления 01.03.04 по дисциплине «Технологии программирования». Составлено в соответствии с учебным планом специальности. Разработано на кафедре прикладной математики и информатики. В пособии рассматривается история развития языков программирования и технологий программирования. В работе приведены этапы жизненного цикла программного обеспечения и описание документации на программный продукт. Подробно изложены принципы объектно-ориентированного программирования. В учебном пособии приведено большое количество примеров для иллюстрации излагаемого материала. Кроме того, пособие содержит задание на лабораторный практикум по объектно-ориентированному программированию. Пособие может быть использовано для самостоятельного изучения студентами программирования на языке С++.

Предназначено для студентов вузов дневной формы обучения.

УДК 004.4(075)
ББК 32.973 – 018.2я7

ISBN 978-5-9795 -1789-6

Оглавление

Введение

С момента появления первых электронно-вычислительных машин разработка программного обеспечения прошла длинный путь: от возможности написать простую программу для вычисления формулы до осознания того, что именно от технологии разработки программного обеспечения зависит прогресс в вычислительной технике.

Сначала развитие вычислительной техники было сосредоточено на решении технических проблем, основное внимание уделялось аппаратуре, а программирование зависело от заинтересованных в нем лиц.

По мере того как вычислительные машины становились мощнее и надежнее, повышалось значение программного обеспечения. Важным этапом считают появление языков программирования высокого уровня (в конце 50-х годов). Они упростили процесс программирования и существенно расширили круг задач, которые стали решаться с помощью вычислительной техники.

Но затем создатели программного обеспечения стали отмечать, что имеющихся в их арсенале средств и приемов недостаточно для успешного решения поставленной задачи.

Развитие объектно-ориентированного подхода в технологии программирования подтолкнуло развитие сред визуального программирования.

Каждое новое достижение в аппаратном либо в программном обеспечении обычно приводит к попыткам расширить сферу применения ЭВМ, тем самым ставит новые задачи, для решения которых нужны новые возможности.

История развития технологии программирования

Этапы развития технологии программирования

Исходя из обычного значения слова «технология» под термином «технологией программирования» понимают совокупность производственных процессов, которая обеспечивает создание требуемого программного средства, а также документирование этой последовательности процессов. Таким образом, под технологией программирования будем понимать процесс разработки программных средств, включая все стадии, начиная с момента появления идеи этого средства; причем каждый из этапов создания программного продукта опирается на использовании определенных методов.

В технической литературе можно встретить и другие определения данного понятия, например, близкое к нему понятие *программной инженерии*, которое определяется как систематический подход к разработке, эксплуатации, сопровождению и изъятию из обращения программных средств. Отличие этих понятий в следующем: технология программирования делает акцент на изучении процессов разработки программы (технологических процессов) и инструментальных средствах их разработки, которые задействованы в этих процессах. А в программной инженерии изучаются различные методы, подходы, средства разработки с точки зрения достижения основной цели программирования – создания программного обеспечения для решения задачи. Изучаемые программной инженерией приемы могут использоваться в разных технологических процессах (и в разных технологиях программирования).

Другое близкое понятие – *методология программирования*. В технологии программирования методика создания программ рассматривается с точки зрения организации ее технологических

процесса, а в методологии программирования нас интересуют основы построения программных продуктов. Методологию программирования многие авторы определяют как совокупность механизмов, применяемых в процессе разработки программного обеспечения и объединенных одним общим философским подходом.

При рассмотрении процесса развития технологии программирования можно выделить следующие этапы.

1. Стихийное программирование. Этот этап характеризуется отсутствием какой-либо технологии, программирование на то момент было, по сути, искусством. Данный этап продолжался от появления первых вычислительных машин до середины 60-х годов 20-го века. Развитие языков программирования шло от машинных языков и ассемблеров к первым алгоритмическим языками высокого уровня (такие как Fortran, Algol). Появилась возможность повторного использования фрагментов программ (функций, подпрограмм), это значительно повысило производительность труда программиста. Для разработки программ использовался принцип «снизу вверх», при котором сначала создавались наиболее простые подпрограммы, и затем из них собиралась более сложная система. Конец данного этапа ознаменовался кризисом программирования, который выразился в значительных сроках создания программных продуктов и, следовательно, значительному повышению их стоимости.

2. Структурный подход. Этот подход сложился в 60–70-е годы 20-го века и означает набор определенных приемов разработки программ и их сопровождение на всех этапах жизненного цикла программного продукта. Основой данного подхода является декомпозиция или разделение сложной системы на небольшие задачи. Результатом решения каждой задачи будут подпрограммы, которые в дальнейшем объединяться управляющим модулем. Разделение большой задачи на части, требующие отдельного решения, можно

представить в виде иерархического дерева. Проектирование программного продукта в этом случае осуществляется «сверху вниз» и подчинено решению общей идеи. Одна из проблем при этом состояла в разработке интерфейсов подпрограмм.

На данном этапе появились ограничения на используемые в алгоритмах конструкции. Появился метод пощаговой детализации алгоритма, формальные модели описания задачи. Принципы структурного программирования были реализованы в процедурных языках программирования , таких как PL/1, Pascal, C.

Естественным продолжением структурного подхода стала технология модульного программирования, в ней фрагменты программы (функции, подпрограммы), которые используют одни и те же глобальные данные, объединялись в отдельно компилируемые модули. Программисты на практике убедились, что структурный подход совместно с модульным программированием позволяет получить достаточно надежные программы. Основной проблемой по-прежнему оставались межмодульные интерфейсы, так как ошибки в них сложно обнаружить из-за раздельной компиляции модулей.

3. Объектный подход. Начало использования объектного подхода к созданию программ – это середины 80-х годов 20-го века. Объектно-ориентированное программирование (ООП) можно описать как технологию создания сложных программных продуктов, которая базируется на использовании в программе совокупности объектов, каждый из которых принадлежит определенному типу (классу). Классы, существующие в программе, образуют иерархию с возможностью наследования свойств от базового класса. Взаимодействие объектов в программе происходит за счет передачи между ними сообщений.

Основное достоинство объектно-ориентированного подхода по сравнению с модульным программированием – понятная

декомпозиция программной системы на составляющие ее элементы, и, следовательно, облегчение ее разработки и тестирования. Главным преимуществом объектно-ориентированного подхода можно считать принципиально новые способы организации программ, которые базируются на наследовании и полиморфизме. Это дает возможность повторного использования кодов, а кроме того, позволяет создавать библиотеки классов для их дальнейшего применения.

Развитие объектно-ориентированного подхода в технологии программирования подтолкнуло развитие сред визуального программирования. Были разработаны и стали активно использоваться такие языки, как C++ Builder, Visual C++, C#.

4. Компонентный подход и CASE-технологии. Начало данного этапа относят к середине 90-х годов 20-го века. Этот подход характеризуется составлением (сборкой) программного обеспечения из отдельных компонентов, которые взаимодействуют между собой через стандартизованные двоичные интерфейсы. Новым является также то, что объекты-компоненты можно собирать в динамически вызываемые библиотеки и использовать их в любом языке программирования, поддерживающем данную технологию. Основы компонентного подхода были разработаны компанией Microsoft, начиная с технологии OLE (Object Linking and Embedding – связывание и внедрение объектов).

Также здесь можно отметить и OLE-automation (технология создания программируемых приложений), которая призвана обеспечить доступ к внутренним службам этих приложений. Современная технология ActiveX создана на основе OLE-automation и предназначена для написания современного программного обеспечения, как сосредоточенного на одном компьютере, так и распределенного.

Этапы развития языков программирования

Поговорим далее о роли технологии программирования на разных этапах развития программирования. История развития языков программирования начинается с машинных языков. Программы для первых компьютеров были написаны в машинных кодах, а перфокарты и перфоленты были основными носителями информации. Любой программист обязан был досконально знать архитектуру машины, для которой писалась программа. Сами программы были достаточно простыми, это объясняется ограниченными возможностями тех вычислительных машин, достаточно большой сложностью разработки, а главное, отладка программ производилась непосредственно на машинном языке.

Повышения мощности компьютеров, развитие средств и методологии программирования позволяло увеличить сложность решаемых на компьютерах задач. Резкое удешевление стоимости компьютеров (аппаратного обеспечения) привело к их широкому внедрению практически во все сферы человеческой деятельности. Человеческий фактор стал играть решающую роль в процессе создания программных продуктов. Сформировалось требования к качеству программного продукта, надежности, его эффективности и удобству работы с ним для пользователей. Широкое распространение персональной вычислительной техники и, как следствие, появление и активное использование компьютерных сетей привело к развитию распределенных вычислений, потребовалась реализация дистанционного доступа к информации и электронной почты для обмена информацией между людьми. Если ранее компьютерная техника рассматривалась как средство решения отдельных задач, то сейчас она является средством информационного моделирования реального мира. Если рассматривать процесс развития

программирования по десятилетиям, то можно выделить для каждого из них следующие значимые черты:

1. **50-е годы.** Это компьютеры первого поколения, они характеризовались небольшой мощностью и программированием в машинном коде. Решались, главным образом, научно-технические задачи, представляющие собой счет по формулам. Использовалась интуитивная технология программирования: при получении заказа программист сразу приступал к написанию программы, и изменения в исходном задании приводили к многократному переписыванию исходного кода (что сильно увеличивало время и без того итерационного процесса составления программы). Документация на программный продукт оформлялась уже после его создания и отладки. К этому периоду относят появление фундаментальной для технологии программирования концепции модульного программирования, которая позволяла преодолеть ограничения и трудности программирования в машинном коде. В это же время появились первые языки программирования высокого уровня.

2. **60-е годы.** Их можно описать как время бурного развития и, как следствие, широкого использования языков программирования высокого уровня. У программистов появилась надежда, что новые языки решат все проблемы, возникающие в процессе разработки больших программных комплексов. Мощность компьютеров этого периода была достаточно высока и позволяла решать сложные задачи, для которых требовалось сложное программное обеспечение. Стало понятно, что важно не только то, на каком языке мы программируем, но и какими подходами во время создания программного продукта мы пользуемся. Это привело к тому, что профессионалы задумались над методологией и технологией программирования. Появление в компьютерах 2-го поколения прерываний, как элемента обработки поступающей информации, привело к появлению

мультипрограммирования и разработке больших программных комплексов. Стала внедряться коллективная разработка сложных программных систем.

3. **70-е годы**. Получили широкое распространение информационные системы и базы данных. Началось интенсивное внедрение нисходящей методологии разработки и структурного программирования, развитие абстрактных типов данных. Исследуются проблемы надежности программного средства и его мобильности.

4. **80-е годы.** Это время характеризуется широким внедрением персональных компьютеров во все сферы человеческой деятельности и, следовательно, значительным увеличением количества пользователей программных продуктов. Это подтолкнуло развитие дружественных пользовательских интерфейсов, и, с другой стороны, потребовала создания концепции качества. Разрабатываются требования технологии программирования. Идет стандартизации технологических процессов и, прежде всего, документации, создаваемой в этих процессах. На лидирующие позиции выходит объектный подход к разработке программ. Разрабатываются новые инструментальные среды для написания, тестирования и отладки программ. Развивается программное обеспечение компьютерных сетей.

5. **90-е годы.** Характеризуются широким охватом всего человечества международной компьютерной сетью и подключением к ней персональных компьютеров. Это привело к появлению проблем технологического, юридического и этического характера для регулирования прав доступа к информации в компьютерных сетях. Остро встала задача защиты информации, хранящейся на компьютерах и передаваемых по сети сообщений. Началось развитие CASE-технологий разработки программных средств.

6. **Начало 21-го века.** Широкое внедрение цифровых технологий: развитие сети Интернет и значительный рост ее роли в жизни общества, распространение мобильных телефонов, цифровых камер. Компьютеры на сверхсложных микропроцессорах с параллельно-векторной структурой, одновременно выполняющих десятки последовательных инструкций программы. Они позволяют строить системы обработки данных и знаний, моделировать архитектуру нейронных биологических систем, распознавать сложные образы. Новые языки программирования оперируют абстракциями все более высокого уровня. Интерфейсы программ подстраиваются под вас автоматически, запоминая ваш выбор и последовательность действий.

Сейчас некоторые разработчики программного обеспечения уже отходят от классических методов разработки программ и применяют так называемые приемы экстремального программирования. Это творческий, неструктурированный процесс, который использует свои специальные приемы и методы, и в котором пропускаются многие этапы классической последовательности проектирования. С другой стороны, быстрая разработка позволяет сократить время создания программного продукта (увеличить скорость написания и тестирования исходного кода программистом). Но пока не ясно, позволит ли такой подход улучшить качество создаваемых программных систем.

Возможно, что совсем скоро вы просто сможете наговаривать компьютеру, что вы хотите сделать, а он сам будет преобразовывать это в программы.

Современной этап развития технологий программирования характеризуется активным использованием компьютерных технологий создания и сопровождения программных систем на всех этапах их жизненного цикла. Эти так называемые CASE-технологий

(Computer Aided Software/System engineering), они позволяют автоматизировать не только процесс разработки программного обеспечения с помощью большого набора библиотек, но и автоматически документировать разрабатываемое программное обеспечение. Существующие CASE-технологии поддерживают структурный, объектно-ориентированный и компонентный подходы к программированию.

К настоящему времени, по мнению специалистов, существует уже более восьми с половиной тысяч различных языков программирования. И это число постоянно увеличивается за счет появления новых версий и потомков активно применяемых языков. Современные алгоритмические языки позволяют обеспечить высокую производительность работы программиста и создавать эффективные прикладные и системные программы (в том числе операционные системы), отвечающие требованиям пользователя.

Императивные языки программирования

Строение любого языка программирования соответствует определенной парадигме программирования. Большая часть современных языков ориентирована на императивную модель, которая задается фон неймановской архитектурой ЭВМ (классическая архитектура). Кроме императивной модели существуют и другие подходы: это языки программирования со стековой вычислительной моделью; функциональные языки, такие как Lisp; языки логического программирования (Prolog). В настоящее время активно развиваются проблемно-ориентированные, декларативные и языки визуальные программирования.

Программы, создаваемые на императивных языках программирования, представляют собой пошаговое описание процесса реализации поставленной задачи. При этом существуют два

основных подхода, являющихся воплощением данной идеи: функциональное и логическое программирование.

Функционального программирования рассматривает и, соответственно, реализует программу как набор, последовательность математических функций. Функция рассматривается как общий механизм представления и анализа решений сложных задач. При этом активно используются рекурсии. При реализации функций в исходном коде программы практически не используют обычные присваивания и возможности низкоуровневого управления вычислениями.

Функциональный стиль программирования ориентирован на реализацию задач посимвольной обработки входной информации. Известно, что любые данные для обработки на вычислительной технике можно трактовать как последовательный набор символов. Входные символы имеют определенный смысл, который можно восстановить по заранее известным алгоритмам их описания (правила формирования исходных данных).

Особенности функциональное программирование можно выразить тремя его основными принципами:

1. Природа данных. Все данные представляются в форме символьных выражений. Системы функционального программирования используют для хранения исходных и промежуточных данных всю доступную память компьютера, поэтому программист при написании исходного кода не заботиться о распределении и освобождении памяти под отдельные блоки данных.

2. Рекурсивная обработка символьных выражений. Следующая черта функционального программирования: для программы любая входная информация это последовательность символов, которая обрабатывается рекурсивными функциями. Такие

программы могут обработать и обычные данные, только затем необходимо преобразовать их в значения.

3. Сходство с машинными языками. Система функционального программирования разрешает интерпретировать программу, представленную в виде структур данных. Это демонстрирует сходство приемов функционального программирования с приемами низкоуровневого подхода и является отличаем от классической методики языков высокого уровня. Эта особенность наиболее ярко проявляется в языке программирования Lisp.

Методы логического программирования позволяют создавать программы как набор формул математической логики. Во время работы такой программы компилятор пытается реализовать эту логику и получить следствие из них. К этой группе относится популярный язык логического программирования Prolog (первая версия его относится к 1971). У него уже есть потомки – это язык Parlog, который специализируется на параллельных вычислениях.

Следующий класс языков программирования, который активно развивается в настоящее время – это языки поддержки параллельных вычислений. Программы, составленные на данных языках, описывают процессы, которые могут выполняться одновременно в реальном времени или работать в псевдопараллельном режиме, имитирующем одновременную обработку данных. В этом случае программа использует прием разделения времени, при котором время на обработку конкретных данных, поступающих от процессов, выделяется по мере необходимости и с учетом приоритетности текущих операций. Языки параллельных вычислений позволяют достичь заметного выигрыша при обработке больших массивов информации, поступающих от одновременно работающих пользователей. Другой областью применения языков параллельных

вычислений являются системы реального времени, в которых пользователю необходимо получить ответ от системы непосредственно после запроса. Обычно такие системы отвечают за жизнеобеспечение.

Рейтинг языков программирования

Итак, технология программирования во многом определяется выбранным языком программирования для решения задачи. В языке могут быть заложены средства, влияющие на технологичность и архитектуру разрабатываемой системы (например, объектная ориентированность, модульность и т. п.).

В таблице приводится рейтинг распространенности языков программирования, составляемый фирмой TIOBE (Голландская компания TIOBE Software BV). Рейтинг составляется по определенной методике, основанной в основном на упоминаниях в Интернете. Индекс TIOBE – индекс, демонстрирующий популярность языков программирования среди профессионалов. Индекс обновляется раз в месяц и основывается на количестве поисковых запросов на таких ресурсах, как Google, Bing, Yahoo!, Wikipedia, Amazon, YouTube and Baidu. Необходимо отметить, что, по задумке создателей, TIOBE демонстрирует не самый «лучший» язык, а самый «популярный» язык за тот или иной промежуток времени. В таблице приводится первая двадцатка языков с наибольшими рейтингами на январь 2017 года и указанием тенденций в их местах по сравнению с январем 2016. Как видно, ведущие места занимают С-подобные языки. Язык Java прочно удерживает первую позицию в течение последних лет.

Январь 2017	Январь 2016	Изменение	Язык программирования	Рейтинг	Изменение %
1	1		Java	17.278%	-4.19%
2	2		C	9.349%	-6.69%
3	3		C++	6.301%	-0.61%
4	4		C#	4.039%	-0.67%
5	5		Python	3.465%	-0.39%
6	7	^	Visual Basic .NET	2.960%	+0.38%
7	8	^	JavaScript	2.850%	+0.29%
8	11	^	Perl	2.750%	+0.91%
9	9		Assembly language	2.701%	+0.61%
10	6	⌄⌄	PHP	2.564%	-0.14%
11	12	^	Delphi/Object Pascal	2.561%	+0.78%
12	10	⌄	Ruby	2.546%	+0.50%
13	54	^^	Go	2.325%	+2.16%
14	14		Swift	1.932%	+0.57%
15	13	⌄	Visual Basic	1.912%	+0.23%
16	19	^	R	1.787%	+0.73%
17	26	^^	Dart	1.720%	+0.95%
18	18		Objective-C	1.617%	+0.54%
19	15	⌄⌄	MATLAB	1.578%	+0.35%
20	20		PL/SQL	1.539%	+0.52%

Инструментальные средства разработки программы

Рассмотрим состав основных инструментальных программных средствах, которые позволяют создавать, отлаживать, тестировать, и выполнять программы на языках высокого уровня. Существующие компиляторы с языков программирования высокого уровня преобразуют исходный текст программы не сразу в выходные коды, а в промежуточную форму, которая называется *объектным кодом*. После получения в процессе компиляции объектного кода следующий этап обработки – сборка (*редактированием связей*), здесь из объектных кодов отдельных частей программы собирается вся программа в кодах целиком. Также на данном этапе происходит подключение стандартных библиотечных подпрограмм и преобразование оставшихся имен объектов в адреса.

Таким образом, программист сначала с помощью редактора текстов набирает исходный код своей программы, сохраняет его в файлы на носителях, затем с помощью другой программы (компилятора) преобразует его в объектный коды. Далее с помощью третьей программы (редактора связей или сборщик) получает программу в кодах (*загрузочный файл).* После выполнения этой последовательности действий наш загрузочный файл можно выполнить средствами операционной системы. Вышеперечисленные инструментальные средства составляют минимальный набор, который позволит программисту разрабатывать программы. Рассмотрим полный набор существующих инструментальных средств.

Редактор — программа для ввода и редактирования исходного текста программы.

Компилятор — программа, преобразующая исходный текст программы в выполняемый код. Как правило, компилятор порождает так называемый *объектный файл* — машинный код с незаполненными внешними ссылками, хотя могут генерировать и исполняемые файлы. По *способу генерации кода* компиляторы делятся на обычные и оптимизирующие. Оптимизирующие компиляторы работают гораздо дольше обычных, но генерируют код, оптимизированный по скорости или по объему.

Компоновщик или редактор связей — программа, собирающая из совокупности объектных файлов и библиотек исполняемую программу. По *способу генерации кода* компоновщики делятся на обычные и инкрементальные. Существуют инкрементальные редакторы связей, которые позволяют ускорить процесс компоновки за счет учета информации о предыдущей компоновке.

Отладчик — программа, позволяющая контролировать код и данные программы во время ее выполнения с целью обнаружения ошибок. В настоящее время распространены символьные отладчики,

позволяющие кроме чисел отображать код на ассемблере либо на исходном языке высокого уровня.

Профилировщик — программа, предназначенная для оценки эффективности выполнения различных участков кода. Профилировщики встраивают в программу специальный код, позволяющий подсчитать частоту вызова различных подпрограмм, время их работы с учетом и без учета вызываемых подпрограмм и большое количество других параметров. Накопленную информацию профилировщики отображают в численном виде, а наиболее совершенные из них — в виде графиков и диаграмм.

Библиотекарь — программа, позволяющая использовать объектные файлы стандартной библиотеки.

Библиотека — набор стандартных подпрограмм и функций**.** Библиотеки по *способу использования* делятся на статические (static library) и динамически связываемые (dynamic link library – DLL). Статические используются только при компоновке программы, при которой из них извлекаются используемые модули. Динамические библиотеки постоянно находятся в памяти, вследствие чего их модулями могут пользоваться различные программы одновременно. Для этого используется специальный механизм динамического связывания.

Современные интегрированные среды программирования поддерживают популярную технологию визуального программирования – это технология быстрой разработки приложений (Rapid Application Development – RAD). Она позволяет быстро создавать графический интерфейс пользователя из существующих стандартных элементов, настраивая их свойства. Так как изначально разработка графического интерфейса пользователя была объемной и сложной задачей, применение RAD-технологии позволяет уменьшить затраты на разработку приложения.

В то же время технология RAD не может ускорить создание прикладной части программы, которая занимается собственно обработкой данных. Эта творческая часть работы, и она в любом случае ложится на программистов.

Жизненный цикл программного продукта

Под жизненным циклом программного продукта (software life cycle) понимают весь период его разработки и эксплуатации, начиная от момента возникновения замысла программного продукта, определения его целевого назначения и заканчивая выводом его из эксплуатации. Понятие «жизненный цикл» охватывает весь процесс создания и использования программного продукта. Этот процесс может быть организован по-разному в зависимости от особенностей коллектива разработчиков и разрабатываемого программного продукта.

Среди систем, предопределяющих условия и требования, предъявляемые к таким процессам, сегодня можно назвать только три основных:

- ГОСТ 34.601-90 (Информационная технология. Комплекс стандартов на автоматизированные системы. Автоматизированные системы. Стадии создания);
- ISO/IEC 12207:2008 (Systems and software engineering – Software life cycle processes – стандарт ISO, описывающий процессы жизненного цикла программного обеспечения);
- Oracle CDM (описание методологических основ и инструментальных средств для автоматизации процессов разработки сложных прикладных систем, ориентированных на интенсивное использование баз данных).

Для второго международного стандарта имеется российский аналог. Это ГОСТ Р ИСО/МЭК 12207-2010, отвечающий за системную и программную инженерию. Но жизненный цикл

программного обеспечения, описываемый в обоих правилах, является идентичным.

В настоящее время можно выделить следующие основные стратегии конструирования программного продукта.

• Каскадный (или водопадный) подход. При таком подходе процесс разработки состоит из цепочки этапов, выполняемых последовательно друг за другом. На каждом этапе создается документация, используемая на последующем этапе. Это классический подход к созданию программного продукта. Достоинством такого подхода является возможность создания четкого плана и временного графика работ, планирование используемых ресурсов. Недостатки данного подхода: реальные проекты часто требуют отклонения от стандартной последовательности шагов; цикл основан на точной формулировке исходных требований к программе (реально в начале проекта требования заказчика определены частично); результаты проекта доступны заказчику только в конце работы.

• Итеративные (или инкрементальные) модели. Заключается в разбиение создаваемой программной системы на набор частей. Каждая часть реализуется путем нескольких последовательных проходов последовательности всех работ. На первой итерации разрабатывается независимая часть системы. При этом чаще всего проводится полный цикл работ. Затем оцениваются полученные результаты. Следующая итерация заключается либо в исправление недочетов первой части, либо в проектировании следующего фрагмента, зависящего от первого, либо в добавлением новых функций. В результате на каждой итерации можно анализировать промежуточные результаты работ, учитывать мнение конечных пользователей и вносить изменения на следующих итерациях. Каждая

итерация в процессе своего выполнения может содержать полный цикл видов деятельности, начиная от анализа требований.

• Продолжением идеи итераций является спиральная модель жизненного цикла программного обеспечения. В данной модели разработки каждая итерация начинается с анализа целей, разработки основных альтернатив, определения необходимых ограничений, оценки рисков и т. д. Результатом каждой итерации будет оценка проведенных в ее рамках работ. Следует отметить общую структуру действий на каждой итерации — планирование, определение задач, ограничений и вариантов решений, оценка предложенных решений и рисков, выполнение основных работ итерации и оценка их результатов. Название спиральной эта модель получила из-за изображения хода работ в «полярных координатах», в которых угол соответствует выполняемому этапу в рамках общей структуры итераций, а удаление от начала координат — затраченным ресурсам.

• Исследовательское программирование. Этот подход предполагает быструю (насколько это возможно) реализацию рабочих версий программ, выполняющих основные функции. После опытного применения созданных программ переходят к их модификации с целью устранения недочетов и приближении их к реальным требованиям пользователей. Такой подход соответствовал ранним этапам развития программирования. В настоящее время этот подход применяется для разработки систем искусственного интеллекта.

• Прототипирование. Этот подход моделирует начальную фазу работы вплоть до создания рабочих версий программ с целью установить (уточнить) требования заказчика к программному продукту. Основная цель создания прототипа — снять неопределенности в требованиях заказчика. В дальнейшем обычно следует разработка программы по установленным требованиям в

рамках какого-либо другого подхода (например, классического). Достоинством данного подхода является определение полных требований к программному продукту. Недостатки: когда заказчик видит уже работающую версию программного продукта, возникает соблазн принять его за готовую версию и оставить нерешенными вопросы качества и удобства сопровождения. При этом заказчик не может адекватно оценить объем будущих работ и требует получить рабочий продукт в нереально короткие сроки. Следует также отметить, что для быстрого получения работающего макета программист может идти на определенные компромиссы, например, использование не самых подходящих языков программирования и операционных систем, или для демонстрации функциональных возможностей может применяться неэффективный алгоритм.

• Сборочное программирование. Этот подход заключается в конструировании программ из стандартных компонент, которые уже существуют в разработанных библиотеках. Эти библиотеки компонент достаточно хорошо разработаны и многократно используются при создании сложных программных систем, имеющие типовые фрагменты обработки данных. Такой процесс разработки состоит скорее из сборки программ из компонент, чем из их программирования .

Весь жизненный цикл программного продукта можно разбить на три крупные фазы:

1) разработка;
2) эксплуатация;
3) сопровождение и разработка новых версий.

В фазе разработки программный продукт разрабатывается, документируется, тестируется и выпускается.

В фазе эксплуатации разработанный программный продукт используется на практике конкретными потребителями, при этом

могут выявляться ошибки, недоработки, изменяться требования к пользовательскому интерфейсу.

В *фазе сопровождения* программный продукт изменяется в соответствии с поступившими требованиями, и появляются его принципиально новые версии.

Фаза эксплуатации, в идеале, должна начинаться сразу после выпуска (поставки) программного продукта, но часто уже работоспособная версия поставляется заказчику еще до завершения полного цикла разработки. Это происходит по договоренности с заказчиком для опытной проверки программы и выявления несоответствий требованиям. Этап эксплуатации включает процессы хранения, внедрения, сопровождения программного продукта, а также транспортировки и применения его по своему назначению (использование программного продукта для решения практических задач на компьютере).

Сопровождение − это процесс сбора информации о качестве программного продукта во время эксплуатации, устранения обнаруженных в нем ошибок, его доработки и модификации, а также извещения пользователей о внесенных в него изменениях. Основой продолжающейся разработки является требование заказчиков на разработку новых версий или желание самих разработчиков улучшить хороший коммерческий продукт.

Сопровождение осуществляется на основе юридической договоренности между разработчиками и пользователями (заказчиками).

Фазу разработки обычно разделяют на следующие *логические этапы*:

1) анализ требований;
2) проектирование;
3) программирование (кодирование);

4) отладка и тестирование;
5) документирование;
6) выпуск.

На этапе *анализа требований* для заказчика обосновывается необходимость нового программного продукта, выявляются наиболее общие требование к нему. Результатом системного анализа является выработка *спецификации требований* на программный продукт, содержащая указанные требования в формальном виде. Этот документ является описанием поведения программы с точки зрения ее будущего пользователя и с фиксацией требований относительно его качества. По единой системе программной документации (ЕСПД) такая спецификация называется *техническим заданием*.

На этапе *проектирования* сформулированные общие требования к будущему программному продукту последовательно реализуются в рабочий проект. Он в деталях описывает структуру программной системы, применяемые форматы данных и алгоритмы, внешний вид интерфейса пользователя. Результатом проектирования является технический проект.

Этап *кодирования* при наличии детального технического проекта является достаточно рутинным. По сути, кодирование является автоматическим процессом реализации предложенных алгоритмов обработки на конкретном языке программирования с использованием конкретной методологии и инструментария. Результатом данного этапа являются программы в исходном тексте и выполняемые файлы. Этап кодирования сопровождается процессом документирования. Этапы проектирования и кодирования часто перекрываются, иногда довольно сильно.

Этап *отладки и тестирования* предназначен для выявления и устранения ошибок и недочетов в программах. Результатом данного этапа являются отлаженные программы, для которых тестированием

установлено (насколько это возможно) соответствие запрашиваемым требованиям.

На этапе *документирования* к созданной программной системе подготавливается пакет документации, описывающей создаваемый программный продукт. Каждый документ ориентируется на конкретный тип пользователей: конечных пользователей, системных администраторов, прикладных программистов и т. д.

На этапе *выпуска* программный продукт подвергается итоговым испытаниям по утвержденной методике. После этого программный комплекс продается или внедряется на фирме, заказавшей его.

Этапам кодирования, отладки, тестирования и документирования в ЕСПД соответствует этап *рабочего проекта*. Таким образом, процесс разработки является итерационным. Только самые простые программы проходят все шаги без итераций.

Тестирование и отладка программ

Тестирование программ осуществлялось еще при их написании в машинных кодах, но к концу 60-х годов 20-го века эта работа стала упорядочиваться, и к середине 70-х были сформулированы основные приемы организации тестирования отдельных программ и программных комплексов. Эти методы используются и в настоящее время. Методы тестирования включают в себя модульное тестирование, внешнее тестирование, нисходящее и восходящее тестирование, анализ передачи управления в программе при тестировании, принципы проектирования и разработки надежных программных систем.

В это же время появилась потребность в теории доказательства правильности программ. Был предложен следующий подход: если есть формальное описание семантики всех конструкций языка программирования, то возможно на основе анализа текста строго

математически вывести заключение о правильности или неправильности программы.

Отсюда следует, что сначала необходимо создать строгое описание синтаксиса языка и его семантики (смысла его конструкций). Для описания синтаксиса языка проблема была решена с помощью использования формальной теории грамматики языка Хомского и способа записи грамматики Бэкуса — Каура. Для описания семантики языка было предложено несколько методов описания: W-грамматики, аксиоматический подход, денотационный метод, атрибутные грамматики, Венский метод описания и ряд других. С помощью этих методов было описано несколько реальных языков программирования: PL/1, Algol-68, Pascal.

Методы доказательства правильности программы продолжали развиваться, и они позволяли доказать корректность небольших программ, сгенерированных для тестирования данной теории. Но прежде всего необходимо было четко сформулировать понятие «корректная» программа, и, с другой стороны, как оценить сложность реальных программ. Для большинства реальных программ строгое описание того, что программа должна делать, существенно больше по объему самой программы и требует очень высокой математической квалификации программиста. Для большого количества программ такое подробное описание в принципе невозможно. При разработке и использовании программного продукта мы имеем дело с многократно преобразованием (переводом) информации из одной формы представления в другую.

Как показывает опыт программирования, несмотря на тщательное проведение этапов проектирования и использование современных технологий программирования, не удается разработать полностью безошибочную программу. Основными активными

методами поиска и устранения ошибок являются тестирование и отладка.

Тестирование – это процесс выявления имеющихся в программе ошибок, а отладка – это процесс их устранения.

При тестировании проверяется, работает ли программа и все ее ветви в соответствии с заявленными требованиями. Чтобы убедиться в правильности функционирования программы и обеспечить полный и эффективный контроль выполнения всех ее ветвей, сначала выбирается стратегия тестирования.

У процесса тестирования программы есть свои особенности:

– не существует эталона программы, с которым можно сравнить результаты тестирования;

– не существует теста для полностью исчерпывающей проверки;

– отсутствуют формализованные критерии процесса тестирования;

– обычно пишутся тесты для поиска ошибок в программе, а не для доказательства корректности ее работы;

– обычно программу тестирует программист, который ее написал.

Наиболее характерными объектами тестирования являются: требования и спецификации, отдельные программные модули, группы программ, законченные функциональные задачи.

Существуют три основных подхода к тестированию программ: нисходящее, восходящее и раздельное. Они отличаются очередностью написания отдельных модулей и процедурой передачи данных тестируемым компонентам. Как и при проектировании программ, наибольший эффект достигается при совместном использовании этих методов.

При стратегии *восходящего тестирования* сначала обычно пишутся и тестируются физические модули нижнего уровня, затем

переходят к вызывающим модулям и так вплоть до главного модуля. Основные трудности такого подхода состоят в необходимости постоянного обновления тестов при подсоединении каждого нового модуля более высокого уровня. При этом достоинством можно считать, что все модули нижнего уровня тестируются детально и независимо, это отлавливает значительное количество ошибок в них. Главный модуль (модуль верхнего уровня иерархии) программируется и тестируется последним.

Передача данных и имитация вызывающих модулей выполняется при использовании специальных программ-тестировщиков. Эта программа имитирует работу вызывающих модулей и обеспечивает передачу данных в тестируемый физический модуль и из него. Все исходные, промежуточные и конечные данные выводятся на печать. Возможна реализация автоматической проверки полученных и ожидаемых результатов.

Такой подход ориентирован на испытание программ модульной структуры и требует работы целой команды квалифицированных программистов.

Алгоритм *нисходящего тестирования* предписывает начинать проверку с главного модуля, к которому последовательно подключаются модули более низких иерархических уровней, при этом активно используются программы-заглушки. При использовании этого подхода есть возможность сохранения исходных тестов и их дополнение по мере подключения новых модулей, т. е. с самого начала тестирования могут использоваться реальные исходные данные, которые расширяются по мере работы. Недостаток: более сложный, чем в предыдущем подходе, поиск ошибок в подключаемых модулях.

Раздельное тестирование — этот метод применяется только для раздельно компилируемых физических модулей. Относится к

приемам восходящего тестирования. Он применяется в случаях, когда программу невозможно полностью описать или когда существует критический модуль, на функционирование которого накладываются определенные ограничения, и он должен быть испытан в самом начале тестирования. В этом случае процедура тестирования разбивается на два этапа:

– независимая разработка каждого физического модуля с имитацией вызывающего модуля и использованием модулей-заглушек;

– совместное редактирование связей уже проверенных модулей и их комплексное тестирование.

Проектирование тестов следует начинать сразу же после создания документации на программный продукт. Существуют два основных подхода к выбору стратегии проектирования тестов. Метод «черного ящика» заключается в том, что тесты проектируются только на основании изучения внешнего описания программного продукта, описания его архитектуры и отдельных модулей. Внутренняя структура модулей при этом никак не учитывается. Фактически такой подход заключается в полном переборе входных данных, так как в противном случае некоторые ветви алгоритма могут не работать, и это не определится при пропуске теста. Эта ситуация ведет к тому, что не все ошибки будут выявлены. Однако тестирование полным набором входных данных практически неосуществимо и требует огромного времени. Метод «белого ящика» заключается в том, что тесты проектируются на основании изучения логической структуры программы с целью протестировать все возможные пути выполнения. Следует учесть, что большинство программ содержит циклы с переменным числом повторений и сложную логику. Следовательно, различных ветвей выполнения программы может оказаться

достаточно много, и их тестирование также будет практически неосуществимо.

Для выбора оптимальной стратегии проектирования тестов можно использовать следующий принцип: для каждого программного модуля, входящего в состав программного продукта, необходимо проектировать собственные тесты для обнаружения в нем ошибок. Это будет соответствовать требованиям разработки надежных программ. Кроме этого, необходим хотя бы один тест :

− на каждую область и на каждую границу изменения какой-либо входной величины ;

− на каждую особую (исключительную) ситуацию, указанную в спецификациях.

Вообще, тестирование программ не дает гарантий их качества, так как невозможно однозначно гарантировать отсутствие ошибок в реальной системе. Математическое обоснование надежности программ основано на формальной *верификации*, которая посредством формального доказательства позволяет устанавливать присутствие требуемых свойств программы для всех допустимых этой программой выполнений. Основой верификации является логика, формальный язык логики, а также формальные модели и методы. Верификация программ обычно сводит анализ их свойств к доказательству истинности условий корректности в виде логических формул. При этом исследуется (верифицируется) не сама программа, а ее спецификация (формальная модель). Примером распространенных формальных моделей являются сети Петри. Несмотря на результаты применения верификации, практическое значение для обоснования надежности программного продукта имеет тестирование. Это связано с трудоемкостью и высокими затратами процесса верификации.

Отладка — это процесс исправления ошибок, обнаруженных на этапе тестировании программы. Обычно отладку представляют в виде циклического повторения трех процессов: тестирования, которое позволяет выявить ошибку, локализация ошибки и исправление программы (и документации).

Отладка = Тестирование + Поиск ошибок + Редактирование.

Если в результате тестирования полученные результаты отличаются от эталонных (ожидаемых), то необходимо определить местоположение ошибки. Для этого можно использовать контрольные точки и слежение за значениями переменных. Во многих системах программирования на языках высокого уровня перечисленные выше возможности включены в отладчики, позволяющие вести отладку в интерактивном режиме.

Известен феномен — по мере роста числа обнаруженных и исправленных ошибок в программном продукте растет также относительная вероятность существования в нем необнаруженных ошибок. Поэтому используются следующие принципы тестирования:

Принцип 1. Тестирование является основной задачей разработки программного продукта, поэтому ее необходимо поручать квалифицированным программистам; нежелательно тестировать свою собственную программу.

Принцип 2. Необходимо разрабатывать тесты, для которых высока вероятность обнаружить ошибку.

Принцип 3. Необходимо сгенерировать тестовые наборы как для правильных, так и для неправильных исходных данных.

Принцип 4. Рекомендуется документировать результат каждого теста для дальнейшего детального анализа.

Принцип 5. Каждый модуль подключается к программе только один раз.

Принцип 6. Если в программу были внесены изменения (например, в результате устранения ошибки), проводите заново все тесты, связанные с проверкой ее работы или ее взаимодействия с другими программами.

Классификация ошибок

Для каждого языка программирования существует свое характерное подмножество ошибок. Рассмотрим классификацию ошибок с точки зрения причины их появления:

1. Синтаксические ошибки. В основном это ошибки в ключевых словах и конструкциях языка программирования. Легко выявляются транслятором, их устранение не вызывает особых трудностей.

2. Опечатки. Как правило, они вызваны невнимательностью программиста при механическом наборе или редактировании исходного текста (ошибка в операции, имя не той переменной и т. д.). В результате появляется синтаксически правильный, но абсолютно неверный логически участок программы.

3. Ошибки реализации алгоритма. Это ошибки, вызванные неверным программированием при верном алгоритме. Этот тип ошибок, по мнению многих авторов, является самым распространенным и наиболее трудно классифицируемым.

4. Ошибки алгоритма. Это логические ошибки в применяемом методе, которые приводят к ошибочной работе программы при правильной программной реализации. Такие ошибки очень трудно найти, так как предположение о их наличии делается обычно после проверки всех остальных альтернатив. К этому типу ошибок относят также отсутствие в алгоритме учета ограничений реализуемого им метода. Ограничения любого математического метода хорошо описаны в специализированной литературе, но часто существуют попытки применить его без учета его особенностей.

5. Ошибки метода. Каждый используемый в программе метод должен сопровождаться оценкой вычислительных погрешностей и перечнем ограничений, снижающих его универсальность.

Документирование и стандартизация

При разработке программного продукта создается большой объем документации. Она используется как средство передачи информации между разработчиками и как средство описания информации, которая необходима пользователям для применения программы. Также программная документация может быть использована для тестирования программы.

Документирование должно начинаться одновременно с разработкой продукта. Существуют следующие основные программные документы:

Текст программы — запись программы с необходимыми комментариями.

Описание программы — сведения о логической структуре и функционировании программы.

Программа и методика испытаний — требования, подлежащие проверке при испытании программы, а также порядок и методы их контроля.

Техническое задание — описанием поведения программы с точки зрения ее будущего пользователя и с фиксацией требований относительно его качества.

Пояснительная записка — содержит общее описание алгоритма, часто в виде граф-схем, схему функционирование и взаимосвязи программных модулей, а также обоснование принятых технических и экономических решений.

Раздел эксплуатационных документов включает в себя :

Руководство пользователя — сведения об области назначения программы, области ее применения, используемых при реализации методах, ограничениях алгоритма, конфигурации требуемых технических средств; описание интерфейса пользователя и допустимых к использованию функций. Создается на основе документов «Описание применения» и «Руководство оператора», описанных в ЕСПД.

Руководство системного администратора — сведения для обеспечения установки, функционирования и настройки программ на условиях конкретного применения. Создается на основе документа «Руководство системного программиста», описанного в ЕСПД.

Стандарты ЕСПД устанавливают требования, регламентирующие разработку, сопровождение, изготовление и эксплуатацию программ, что обеспечивает возможность:

– унификации программных изделий для взаимного обмена программами и применения ранее разработанных программ в новых проектах;

– снижения трудоемкости и повышения эффективности разработки, сопровождения, изготовления и эксплуатации программных изделий;

– автоматизации изготовления и хранения программной документации.

Далее перечисляются стандарты ЕСПД, которые чаще всего используются на практике при создании коммерческих программных продуктов.

– *ГОСТ (СТ СЭВ) 19.201-78 (1626-79).* ЕСПД. Техническое задание. Требование к содержанию и оформлению.

– *ГОСТ (СТ СЭВ) 19.101-77 (1626-79).* ЕСПД. Виды программ и программных документов. Устанавливает виды программ и

программных документов для вычислительных машин, комплексов и систем независимо от их назначения и области применения.

– *ГОСТ 19.102-77.* ЕСПД. Стадии разработки. Устанавливает стадии разработки программ и программной документации для вычислительных машин, комплексов и систем независимо от их назначения и области применения.

– *ГОСТ 19.103-77* ЕСПД. Обозначение программ и программных документов.

– *ГОСТ 19.105-78* ЕСПД. Общие требования к программным документам. Настоящий стандарт устанавливает общие требования к оформлению программных документов для вычислительных машин, комплексов и систем, независимо от их назначения и области применения и предусмотренных стандартами Единой системы программной документации (ЕСПД) для любого способа выполнения документов на различных носителях данных.

– *ГОСТ 19.402-78* ЕСПД. Описание программы. Состав документа «Описание программы» в своей содержательной части может дополняться разделами и пунктами, почерпнутыми из стандартов для других описательных документов и руководств: *ГОСТ 19.404-79* ЕСПД. Пояснительная записка*, ГОСТ 19.502-78* ЕСПД. Описание применения, *ГОСТ 19.503-79* ЕСПД. Руководство системного программиста, *ГОСТ 19.504-79* ЕСПД. Руководство программиста, *ГОСТ 19.505-79* ЕСПД. Руководство оператора.

– *ГОСТ 19.701-90* ЕСПД. Схемы алгоритмов, программ, данных и систем. Обозначения условные графические и правила выполнения. Он устанавливает правила выполнения схем, используемых для отображения различных видов задач обработки данных, и средств их решения и полностью соответствует стандарту ИСО 5807:1985.

– *ГОСТ 19.301-79* ЕСПД. Программа и методика испытаний, которая (в адаптированном виде) может использоваться для

разработки документов планирования и проведения испытательных работ по оценке готовности и качества программной системы.

Инструкция пользователя (user documentation) объясняет пользователям, как они должны действовать, чтобы применить данный программный продукт. К такой документации относятся документы, которыми должен руководствоваться пользователь при инсталляции программы, при использовании программы для решения своих задач и при взаимодействии с другими программными продуктами. В связи с этим выделяют две категории пользователей программ: конечные пользователи и администраторы программ.

Конечный пользователь использует программу для решения своих предметных задач. Он может и не знать многих деталей работы компьютера или принципов программирования. Администратор (system administrator) управляет использованием программы конечным пользователями и осуществляет его сопровождение, не связанное с модификацией программ.

Обычно конечному пользователю требуются либо документы для изучения программного продукта (использование в виде инструкции), либо для уточнения некоторой информации (использование в виде справочника).

Исходя из этого, в состав пользовательской документации для крупных программных продуктов входит:

• Общее функциональное описание. Дает краткую характеристику функциональных возможностей ПС.

• Руководство по инсталляции. Предназначено для системных администраторов. Оно должно детально предписывать, как устанавливать системы в конкретной среде, в частности, требования к минимальной конфигурации аппаратуры.

• Инструкция по применению. Предназначена для конечных пользователей. Содержит необходимую информацию по применению программы.

• Справочник по применению. Содержит необходимую информацию по применению программы, в удобном для избирательного поиска отдельных деталей виде.

• Руководство по управлению. Предназначено для системного администратора. Оно должно описывать сообщения, генерируемые, когда программы взаимодействует с другими системами, и как должен реагировать администратор на эти сообщения.

Интерфейсом пользователя называют совокупность способов и правил взаимодействия программы с пользователем.

Во время проектирования современных программных систем разработке пользовательского интерфейса уделяется особое внимание. От удобного внешнего вида системы зависит ее популярность и, следовательно, ее коммерческий успех. Даже такие программы, которые должны работать без вмешательства человека, например, управляя искусственным спутником Земли, снабжаются интерфейсом пользователя для обеспечения возможностей их установки, контроля и настройки. Исключением, пожалуй, являются встраиваемые микропрограммы, которые функционируют полностью автономно.

В настоящее время в прикладном программировании широко используются визуальные интерфейсы, предоставляющие пользователю набор отображаемых стандартизованных элементов управления программой. Для визуального интерфейса существует ряд общепринятых требований. Это прежде всего интуитивно понятный интерфейс, независимость от конкретных устройств ввода-вывода, простота установки и настройки под личные требования.

Структурное программирование

Большие программные системы, как правило, разрабатываются по частям, которые называются *программными модулями*. Такой метод разработки программ называют *модульным программированием*. Программный модуль − это любая логически завершенная часть программы, которая оформляется как самостоятельный программный продукт. Таким образом, каждый программный модуль кодируется, компилируется, тестируется и отлаживается отдельно от других модулей программы. Следовательно, можно считать, что модули программы физически разделены. Модульный подход в программировании можно считать средством борьбы со сложностью и с дублированием в программах. При создании каждого модуля следует учитавать, что программа должна быть понятной не только компьютеру, но и человеку. Программист, кодирующий модуль, тестировщик, готовящий тесты для отладки модуля, вынуждены будут многократно разбирать логику его работы.

В современных языках программирования достаточно средств, чтобы запутать логику программы, тем самым сделать модуль труднопонимаемым и, как следствие этого, сделать его ненадежным.

Создателем структурного подхода к программированию считается Э. Дейкстра. Фактически это первый законченный метод программирования, предлагающий путь от задачи до ее воплощения в программе. Этот метод применялся очень широко в практическом программировании и по сей день не потерял своего значения для определенного класса задач.

Дейкстра предложил строить программу как композицию из нескольких типов управляющих конструкций (структур), которые позволяют сильно повысить понимаемость логики работы программы. Программирование с использованием только таких конструкций назвали *структурным*.

К основными конструкциями структурного программирования относятся: следование, ветвление и циклическое повторение. Эти конструкции состоят либо из операторов, либо из более крупных фрагментов программы (функции, блоки). Каждая из этих конструкций имеет по управлению только один вход и один выход. Используемый в стандартной конструкции сложный фрагмент также имеет только один вход и один выход. Весьма важно также, что эти конструкции являются уже математическими объектами, что и объясняет причину успеха структурного программирования. Для структурированных программ можно математически доказывать некоторые свойства, что позволяет обнаруживать в программе некоторые ошибки. Структурное программирование иногда называют еще «программированием без GO TO», данный оператор затрудняет ясность программы. Хотя данный оператор можно использовать для выхода из цикла или процедуры по особому условию. Такое «досрочное» завершение работы структурной единицы локально нарушает общую структурированность программы.

Структурное программирование содержит рекомендации о корректном построении текст модуля программы и тем самым улучшает понимание логики его работы. Часто программирование модуля начинают с построения его блок-схемы, описывающей в общих чертах логику его работы.

Структурный подход базируется на двух основополагающих принципах: первый – это использование процедурного стиля программирования, второй – это последовательная декомпозиция алгоритма задачи сверху вниз.

Объектно-ориентированное программирование

Основные принципы ООП

Объектно-ориентированное программирование (ООП) родилось и получило широкое распространение ввиду осознания следующих трех важнейших проблем :

1. Развитие языков и методологий программирования уже не соответствовало потребностям разработчиков больших программных систем. Единственным методом ускорения разработки программного обеспечения оставался метод многократного использования разработанных ранее модулей, хотя приемы процедурного программирование позволяли использовать разрабатываемые функции как блоки для построения программ. Существенного ускорения процесса массового программирования это не дало.

2. Второй проблемой являлась необходимость упрощения сопровождения и модификации разработанных систем, которые требуют не меньших усилий, чем собственно разработка.

3. Основной недостаток структурного программирования заключается в том, что не все задачи поддаются алгоритмическому описанию и алгоритмической декомпозиции. Необходим подход, в котором структура программы наиболее близка к структуре поставленной задачи, и понятия, в которых сформулирована задача, соответствуют терминам используемого языка программирования.

Считается, что объектно-ориентированная технологии позволяет преодолеть перечисленные недостатки, и это является ее основными достоинствами.

Объектные модели базируются, прежде всего, на процессах абстрагирования, инкапсуляции, модульности, иерархии.

Абстрагирование – это выделения существенных характеристик объекта, которые отличают его от всех других объектов и являются наиболее важными для пользователя и программиста.

Инкапсуляция – это объединение элементов абстракции в определенную структуру, включая не только описательные характеристики, но и ее поведение.

Модульность – разделение системы на *модули*, под которыми понимаются единицы кода, служащие блоками физической структуры системы.

Иерархия – определяет упорядоченность выделенных абстракций. Типичный пример иерархии – это наследование. Кроме этого, иерархическая структура присуща также модулям и другим частям системы.

Под ООП понимается методология программирования, при которой программа организуется как совокупность взаимодействующих объектов, каждый из которых является экземпляром какого-либо класса, а классы образуют иерархию наследования. При этом классы считаются статичными, а объектам присуща динамика, что выражается в полиморфизме.

Ниже приведены основные признаки объектно-ориентированного подхода в создании программ:

1. **Объект.** Основным понятием ООП является **класс** – это тип, описывающий множество предметов реального мира, имеющих одинаковые основные характеристики (данные) и правила поведения (методы обработки данных). Объект – это типичный представитель (экземпляр, абстрактный представитель) своего класса.

2. **Инкапсуляция.** Это объединение в единую целостную структуру данных и процедур (функций), которые обрабатывают эти данные.

3. **Наследование**. Это определение объекта и затем использование его для построения иерархии производных объектов, причем каждый производный объект («потомок») наследует доступ к коду и данным всех своих «родителей». Создать новые классы можно,

наследуя уже существующие (иерархическое наследование).

4. **Ограничение прав доступа**. При описании класса часть методов и свойств разрешается спрятать внутри реализации класса, так что обратиться к этим характеристикам и методам можно будет только из методов данного класса. При выполнении наследования также задаются права, которые определяют доступ к свойствам базовых классов.

5. **Полиморфизм.** Некоторому действию придается одно имя, которое совместно используется объектами всей иерархии, причем каждый объект иерархии реализует это действие своим собственным, подходящим для него, образом.

6. **Абстрагирование.** Реализовано в механизме создания абстрактных классов, имеющих не описанные (абстрактные) функции. Эти классы используются в качестве базовых для формирования потомков, имеющих тот же набор методов, но уже переопределенных.

7. **Устойчивость.** Под устойчивостью понимают время существования объектов в системе. Это свойство реализуется в основном в языках объектно-ориентированных баз данных.

Важнейшей характеристикой объекта является описание того, как он может взаимодействовать с окружающим миром. Это описание называется *интерфейсом объекта*. Принято считать, что объекты взаимодействуют между собой с помощью сообщений. Принимая сообщение, объект выполняет соответствующее действие. Эти действия называются *методами*.

Интерфейс – это внешнее описание объекта. При разработке описания объекта программист определяет, будет ли данное свойство класса необходимо другим объектам. Если это так, то необходимо объекту добавить метод, который передает значение данного поля другим объектам. Этот метод и будет составлять интерфейс объекта. Аналогично решается вопрос и с другими атрибутами.

Помимо интерфейса у объекта могут быть методы или атрибуты, предназначенные только для использования в самом объекте. В этом случае внутренняя структура объекта скрыта (свойство инкапсуляции данных). Поэтому внутреннюю структуру объекта становится возможным изменять независимо от других взаимодействующих с ним объектов.

В классах используются следующие виды методов:

- конструктор (Constructor) – метод, имя которого совпадает с именем класса. Он выполняется при создании объекта. Поэтому конструктор обычно содержит инструкции по инициализации переменных объекта;

- деструктор (Destructor) – метод, выполняемый при уничтожении объекта. Не во всех объектно-ориентированных языках есть деструкторы. Обычно они применяются для освобождения системных ресурсов, например, оперативной памяти, занимаемых объектом;

- методы чтения (Accessors) – еще их называют get-методами, они возвращают значение закрытого атрибута объекта. Именно так внешние объекты обычно получают доступ к инкапсулированным данным;

- методы изменения (Mutators), set-методы, устанавливают новое значение атрибута. Таким образом, внешние объекты изменяют инкапсулированные данные.

Прочие методы, определенные в классе, зависят от назначения класса, то есть от действий, которые он призван выполнять.

Представителем объектно-ориентированного подхода в программировании является язык C++. Язык C был разработан в 1972 году сотрудником Bell Laboratories Денисом Ричи для использования в разработанной им совместно с Кеном Томпсоном операционной системой Unix. На языке C написан компилятор этого

языка, а также операционная система Unix для мини-ЭВМ PDP-11. Этот язык получил развитие в виде созданного фирмой Borland для семейства микропроцессоров 8086, 8088 и операционной системы MS DOS языка программирования под названием «Turbo-C». Затем в начале 80-х годов также сотрудник Bell Laboratories Бьерн Страуструп на основе языка С разработал значительно более мощный язык С++, реализующий объектно-ориентированный подход к программированию. В конце XX века С++ приобрел статус стандартного языка программирования. В начале XXI века появился еще один преемник языка С – это С# (произносится: си шарп). Этот язык предложен фирмой Microsoft как конкурент языка Java и представлен как язык компонентной сборки. В настоящее время язык С++ является мощным и эффективным средством разработки разнообразного программного обеспечения.

Итак, язык программирования С++ создан на основе нескольких базовых принципов, которые в настоящее время характерны для современных языков.

1. Используется структурное программирование – это метод проектирования программ в виде последовательной структуры функционально законченных блоков, исполняемых один за другим.

2. Реализован принцип проектирования: «Сверху – вниз». Это значит, что сначала проект разбивается на несколько простых задач, решаемых отдельно, для каждой задачи создается отдельный программный модуль. Затем следует сборка модулей в единый программный продукт.

3. Применяются концепции объектно-ориентированного программирования.

4. В язык С++ встроена возможность расширения базовых конструкций языка за счет использования большого количества внешних библиотек.

5. Предусмотрены способы переносимости программ с небольшими изменениями на другую операционную или аппаратную среду. Основное назначение языка C++ – это системное программирование. Поэтому считается, что это относительно низкоуровневый язык. По объему и скорости выполнения программы, написанные на C++, приближаются к программам, написанным на ассемблере.

В отличие от процедурного подхода к программированию, когда описание алгоритма представляет собой последовательность действий, объектно-ориентированный подход к программированию предлагает описывать программные системы в виде взаимодействия объектов.

Как уже отмечалось, каждый объект характеризуется методами (функции, подпрограммы), они описывают действия, которые может выполнять объект, и свойствами.

Свойство – характеристика объекта, его параметр. Все объекты наделены определенными свойствами, которые в совокупности выделяют объект из множества других объектов.

Метод – это программа, описывающая допустимые действия над объектом или над его свойствами. С помощью методов изменяется поведение объекта.

Объект характеризуется набором реализованных методов обработки, они либо пишутся программистом, либо заимствуются из стандартных библиотек. Методы вызываются при наступлении заранее определенных событий, например, однократное нажатие левой кнопки мыши, вход в поле ввода, выход из поля ввода, нажатие определенной клавиши.

По мере развития систем обработки данных создаются стандартные библиотеки методов, в состав которых включаются типизированные методы обработки объектов определенного класса, которые можно заимствовать для различных объектов.

Событие – изменение состояния объекта. **Внешние события** генерируются пользователем (например, клавиатурный ввод или нажатие кнопки мыши, выбор пункта меню, запуск макроса); **внутренние события** генерируются системой.

Программный продукт, созданный с помощью технологии объектно-ориентированного программирования, всегда имеет экранную форму, с объектами управления. Через данную форму вызываются методы обработки при наступлении определенных событий. Экранные формы также используются для перехода от одной части программного продукта к другой.

В настоящее время в области развития и повышения эффективности программирования существуют два основных подхода: структурный и объектно-ориентированный. Каждый из этих подходов поддерживается приблизительно равными по мощности языками. Выбор того или другого подходов сложен. Отсюда идет противопоставление структурного и объектно-ориентированного программирования. Это связано с недостаточно ясным пониманием тонкостей технологии этих двух подходов. Классический структурный анализ, предусматривающий расчленение процесса на функциональные модели, легко воспринимается участниками всех уровней любого бизнес-процесса. Однако такие понятия ООП, как наследование, инкапсуляция, полиморфизм и др. понимает далеко не каждый. Хотя очевидно, что объектно-ориентированная модель наиболее адекватно отражает реальный мир, представляющий собой в целом совокупность взаимодействующих посредством обмена сообщениями объектов.

В объектно-ориентированном программировании имеется несколько соглашений об именовании объектов. Хотя никто не заставляет вас называть свои классы, атрибуты и методы именно так,

следование общепринятым правилам обеспечит взаимопонимание с другими программистами:

- имена классов начинаются с заглавной буквы, за которой следуют строчные. Если имя класса состоит из нескольких слов, то они разделяются либо символом подчеркивания, например Maximum_item, либо внутренними заглавными буквами, например Maximumltem;

- имена атрибутов (свойств) и методов начинаются со строчной буквы и могут содержать заглавные или строчные буквы, а также цифры. Если имя атрибута или метода состоит из нескольких слов, то они разделяются либо символом подчеркивания, например proizv_num, либо внутренними заглавными буквами, например proizvNum;

- имена методов чтения начинаются со слова get, а за ним следует имя атрибута, значение которого считывается. Например, метод для получения кодового номера объекта может называться getNumber;

Определение класса

Класс – это тип, определенный пользователем, содержит данные и функции для работы с ними. Память выделяется только тогда, когда класс используется для создания объекта.

Посмотрим, как можно представить в языке С (структурный подход к созданию программ) понятие даты, используя для этого тип структуры и набор функций, работающих с переменной этого типа.

```
struct date {int month, day, year;};// описание структуры
date today; // описание переменной типа date
void set_date (date*, int, int, int); //описание функций для работы с датой
void next_date (date*);
void print_date (date*);
```

В данном фрагменте программы никакой явной связи между функциями и структурой date нет. Ее можно установить если описать функции как члены структуры:

```
struct date {
int month, day, year;
void setData (int, int, int;);  //метод для установки даты
void next_date ();
void print_date ();
};
```

Таким образом, используя структуру, получено описание нового типа данных – класса. В описанные методы созданного классе нет необходимости передавать параметр **date***, так как атрибуты month, day, year доступны всем методам класса по определению.

В языке C++ класс описывается с помощью ключевого слова class. Описание класса должно быть до его использования и на внешнем уровне (вне функций программы).

Пример описания класса:

```
class Date   // это описание нового типа данных, его имя date
{int day, month, year; //  это данные – члены класса
public:
 void setDate (int, int, int);  //  функции – члены  класса (методы класса)
void print_date ( );
void next_date ();
};
```

К полям day, month, year могут обратиться только методы класса. Обращение извне к ним невозможно.

Опишем переменную соответствующего типа (объект):

```
Date today;
Date my_birthday;
```

Описанные методы класса можно вызвать только через имя объекта, используя стандартную запись обращения к члену структуры:

my_birthday.setDate (30, 12, 2000);

today.set (1,09,2017);

my_birthday.print_date ();

today.next_date ();

Можно описать указатель на данный тип и использовать его для обращения к членам класса:

Date *ptr_d;

ptr_d → print_date( );

При описании класса могут использоваться метки:

1) private – это закрытые поля и функции, они могут использоваться только методами данного класса;

2) public – это открытая часть класса, она обеспечивает связь объекта с внешней программой;

3) protected – это защищенная часть класса**,** аналогична private, но в отличие от нее может наследоваться производным классом.

К закрытым полям и к закрытым методам можно обращаться только из функций-членов данного класса. Открытые части класса используются для связи объектов класса с программой, в которой они существуют.

По принципу умолчания, при описании класса словом class, его члены по умолчанию считаются закрытыми, если не использованы другие метки. Эти метки можно использовать в описании класса несколько раз и в любом порядке. Также класс в языке C++ можно описать через ключевые слова: struct (структура), union (объединение). Все члены класса, описанного через struct, считаются открытыми, но их можно закрыть, используя метку private. Члены класса, описанного через union, могут быть только открытыми.

Описание методов класса может содержаться в описании класса или вне его. В этом случае при определении метода нужно указать имя класса:

```
void date::next_date ( ) //описание тела функции вне класса
{       if (++date>28) {   // описание действий
        }
}
```

void – это тип возвращаемого значения описываемого метода;

Date – имя класса, к которому относится метод;

:: – оператор расширения области действия;

next_date – имя описываемого метода.

В теле функции имена полей (свойств) класса можно использовать без указания имени объекта. В таком случае имя относится к тому члену объекта, для которого была вызвана функция.

```
void date::print( ) // печать даты в принятом в России виде
{
        cout <<day<<.<<month<<.<<year;
}
```

От функций, не являющихся членом класса date, наши данные ограждены. Допустим, в следующем примере мы хотим обратиться к полю day в теле функции backdate :

```
void backdate( )
{
        day--; // ошибка, так как эта функция не является методом
}                       //  класса Date
```

Аналогично если мы захотим вызвать функцию backdate для объекта типа Date. Например,

```
Date today; // описание объекта
today. backdate( ); // ошибка, так как в классе Date не описан
//    данный метод, это просто внешняя  функция
```

Есть ряд преимуществ в том, что доступ к полям данных ограничен явно указанным списком функций. Любая ошибка в дате (например. December, 36, 1985) могла быть внесена только методом класса, поэтому первая стадия отладки – локализация ошибки – происходит даже до первого пуска программы.

Это только частный случай общего правила: любое изменение в поведении типа Date может и должно вызываться изменениями в его членах. Другое преимущество в том, что потенциальному пользователю класса для работы с ним достаточно знать только определения открытых методов (интерфейса класса). Защита частых данных основывается только на ограничении использования имен членов класса.

В методе класса можно непосредственно использовать имена членов того объекта, для которого она была вызвана. Например, в следующем классе **X** есть поле **m**, значение которого использует метод **readm ( )**:

```
class X
{ int m;
public:
int readm ( ) {return m}
};
void func1 (X aa, X bb)
{ int a=aa.readm ( );
  int b=bb.readm ( );
}
```

Внешняя функция **func1** получает два аргумента – объекты типа **X** и для каждого из них вызывает метод **readm ( )**. При первом вызове readm( ) вызывается для объекта **aa,** а при втором – для объекта **bb**.

Пример программы, которая работает со строками. Описан класс, который хранит в закрытой части содержимое строки (поле **s**) и

ее длину (поле **len**). В открытой части классе определены методы **assign** и **print** :

```
#include<stdio.h>
#include<string.h>
class WiseString
{const char *s;
  int len;
public:
void assign(const char *string) // описание метода
                {strcpy(s,string);
                 len=strlen(s);
                 }
void print( ) const; // описание прототипа метода
}; //конец описания класса
void main ( ) //начало главной функции
{
WiseString str1, str2; // создаются два объекта
str1.assign (“ПМИд-21”); //вызов метода assign для первого объекта
str2.assign (“4-й семестр”); //вызов метода assign для второго объекта
str1.print( );
str2.print( );
} //конец главной функции
//описание метода вне класса
void WiseString :: print( ) const
{const char *ft=”Это строка\n Ее содержание: %s\n Длина %d символов”;
printf(ft, s, len);
}
```

В результате работы данной программы на экране будет напечатано:

Это строка

Ее содержание: ПМИд-21

Длина 7 символов

Это строка

Ее содержание: 4-й семестр

Длина 11 символов

Ввод/вывод данных

В стандартном языке C++ существуют два основных пути ввода-вывода информации: с помощью потоков, реализованных в STL (Standard Template Library) и посредством традиционной системы ввода-вывода, унаследованной от языка C. Для использования традиционного ввода-вывода в программу необходимо включить заголовочный файл <stdio.h>. Следует учесть, что компилятор должен иметь доступ к соответствующей объектной библиотеке для правильной сборки исполняемого файла. Библиотека *stdio* предоставляет необходимый набор функций для ввода и вывода информации как в текстовом, так и в двоичном представлении. Можно отметить, что в отличие от классической C-библиотеки, в современных библиотеках имеются более безопасные аналоги «классических» функций. Как правило, они имеют такое же имя, к которому добавлен суффикс *_s*. В программах рекомендуется использовать именно эти безопасные функции.

При запуске программы на выполнение (консольного приложения) неявно открываются три потока: *stdin* – для ввода с клавиатуры, *stdout* – для буферизованного вывода на монитор и *stderr* – для небуферизованного вывода на монитор сообщений об ошибках. Эти три имени определены в бибилиотеке stdio.h. Для

консольного ввода-вывода предусмотрена отдельная группа функций. Однако они, как правило, являются обертками для аналогичных функций файлового ввода-вывода, для которых аргумент типа *FILE* задан по умолчанию.

Ниже приведены некоторые популярные функции из *stdio,* которые обычно используют в программах на языке C++:

```
FILE *fopen(const char *filename, const char *mode)   // открытие файла
int fclose(FILE *stream)     // закрытие файла
int printf(const char *format, ...)   // фоматированный консольный вывод
int fprintf(FILE *stream, const char *formatб, ...) // форматированный ввод из файла
int scanf(const char *format, ...)     // фоматированный консольный ввод
int fscanf(FILE *stream, const char *format, ...)  // фоматированный ввод из файла
int fgetc(FILE *stream)     // читает символ из файла
char *fgets(char *s, int n, FILE *stream)     // читает строку из файла
int fputs(const char *s, FILE *stream)     // записывает строку в файл
int getchar(void)     // читает символ из stdin
int putchar(int c)     // записывает символ в stdout
int puts(const char *s)     // записывает строку в stdout
```

Для использования объектно-ориентированного консольного ввода-вывода с помощью потоков (stream) STL в программу необходимо включить заголовочный файл <iostream>, а для файлового еще и <fstream>. (Разумеется, компилятор должен иметь доступ к соответствующей объектной библиотеке для правильной сборки исполняемого файла).

При запуске консольного приложения неявно открываются четыре потока: *cin* – для ввода с клавиатуры, *cout* – для буферизованного вывода на монитор, *cerr* – для небуферизованного

вывода на монитор сообщений об ошибках и *clog* – буферизованный аналог *cerr*. Эти четыре имени определены посредством <iostream>. Потоки *cin*, *cout* и *cerr* соответствуют потокам *stdin*, *stdout* и *stderr* соответственно. Какой механизм использовать – вопрос предпочтений программиста,

Использование механизма потоков считается более безопасным. Но, как известно, плохую программу можно написать на любом языке программирования. Это также относится и к использованию библиотек.

В C++ имеется ряд манипуляторов для потоков. Некоторые из них приведены в таблице 1.

Таблица 1. Манипуляторы для потоков

Манипулятор	**Описание действия**
endl	Помещение в выходной поток символа конца строки '\n
width(ширина)	Устанавливает ширину поля вывода
precision(точность)	Устанавливает количество значащих цифр в числе (или после запятой) в зависимости от использования fixed
fixed	Показывает, что установленная точность относится к количеству знаков после запятой
scientific	Выводит число в экспоненциальной форме
get()	Ожидает ввода символа
getline(указатель, количество)	Ожидает ввода строки символов. Максимальное количество символов ограничено полем «количество»

Пример. Фрагмент программы ввода-вывода значения переменной в C++:

```
int n;
cout << "Введите n:";
cin >> n;
cout << "Значение n равно: " << n << endl;
double a = -112.234;
```

```
double b = 4.3981;
int c = 18;
cout << endl << "double number:" << endl;
cout <<"width(10)" << endl;
cout.width(10);
cout << a << endl << b << endl;
cout.precision(5);
cout <<"precision(5)" << endl << a << endl << b << endl;
cout << "fixed" << endl << fixed << a << endl << b << endl;
cout << "scientific" <<endl << scientific << a << endl << b << endl;
cout << endl << "int number:" << endl;
cin.get();
```

Помимо чтения с клавиатуры и вывода на экран, библиотека iostream поддерживает чтение и запись в файлы. Для этого предназначены следующие классы:

- ifstream, производный от istream, связывает ввод программы с файлом;
- ofstream, производный от ostream, связывает вывод программы с файлом;
- fstream, производный от iostream, связывает как ввод, так и вывод программы с файлом.

Чтобы использовать часть библиотеки iostream, связанную с файловым вводом/выводом, необходимо включить в программу заголовочный файл:

```
#include <fstream>
```

Файл fstream уже включает в себя описание библиотеки iostream, так что включать оба файла обычно необязательно.

Рассмотрим пример открытия файла для чтения и записи в него квадратов чисел от 1 до 10.

```
#include <fstream>
```

```
using namespace std;
const char *filename = "data.txt";
int main() {
   // создание потока ostr, открытие файла для записи в текстовом
режиме,
   ofstream ostr;
   ostr.open(filename);
   if (ostr) {  // запись данных в файл
      for (int i = 0; i < 16; i++) {
         ostr << i*i << endl;
      }       }
      ostr.close();// закрытие файла.
   }
   else { // вывод сообщения об ошибки открытия  файла
        cerr << "Output file open error \"" << filename << "\"" << endl;
        return 1;
        }
   // открытие файла  для чтения в текстовом режиме,
   int data;
   int counter = 0;
   ifstream istr(filename);
   if (istr) {  // проверка на то, что файл открыт
      while (!(istr >> data).eof()) {   // чтение данных до конца файла
         cout.width(8);//  форматированный вывод на экран
         cout << data;
         }
      istr.close();// закрытие файла.
      }
   else {  // вывод сообщения об ошибки открытия  файла
      cerr << "Input file open error \"" << filename << "\"" << endl;
```

```
        return 2;
    }
    return 0;
}
```

Инициализация и удаление объектов

Инициализация объектов класса с помощью таких функций, как set_date() – неэффективное решения. Поскольку явно не было указано, что объект требует инициализации, программист может либо забыть это сделать, либо сделать дважды, что может привести к столь же катастрофическим последствиям. Лучше дать программисту возможность описать функцию, явно предназначенную для инициализации объектов. Поскольку такая функция конструирует значение данного типа, она называется *конструктором*. Эту функцию легко распознать – она имеет то же имя, что и ее класс:

```
class Date {
//…
Date (int, int, int);
};
```

Если в классе есть конструктор, все объекты этого класса будут проинициализированы. Конструктор автоматически вызывается при создании объекта. Если конструктору требуются параметры, их надо указывать в круглых скобках после имени объекта:

```
Date today = date (23, 6, 1983);
Date xmas (25, 12, 0); // краткая форма
Date my_birthday; // неправильно, нужен параметр
```

Часто бывает удобно указать несколько способов инициализации объекта. Для этого нужно описать несколько конструкторов:

```
class date {
        int month, day, year;
```

```
public:
//…
date (int, int, int); // день, месяц, год
date (int, int); // день, месяц и текущий год
date (int); // день и текущие месяц, год
date(); // стандартное значение: текущие день, месяц и год
date (const char*); // дата в строковом представлении
};
```

В описании класса необходимо дать определение каждого конструктора.

Параметры конструкторов подчиняются тем же правилам о типах параметров, что и все остальные функции. Конструкторы должны различаются по типам своих параметров или их количеству. Транслятор способен правильно выбрать конструктор по данным признакам. Например:

```
Date today (4);
Date july (“July 4. 2000”);
Date guy (8,3);
Date now; // инициализация стандартным значениям
```

Объект класса без конструктора может инициализироваться присваиванием ему другого объекта этого класса. Это не запрещается и в том случае, когда конструкторы описаны:

```
Date d = today; // инициализация присваиванием,
```

При этом объект today уже должен быть ранее создан и инициализирован.

Пользовательские типы чаще имеют, чем не имеют конструкторы, которые проводят инициализацию.

Для многих типов требуется и обратная операция – деструктор, гарантирующая правильное удаление объектов этого типа. Деструктор класса X обозначается ~X («дополнение конструктора»).

В частности, для многих классов используется свободная или динамическая память, выделяемая конструктором и освобождаемая деструктором.

В классе может быть только один деструктор. Он должен располагаться в открытой части класса и не иметь параметров. Деструктор вызывается автоматически при выходе из модуля, который содержит описание класса.

Основная память чаще всего освобождается автоматически при выходе программы из блока, в котором определена переменная типа класса.

Пример программы, работающей со стеком:

```
class stack
{int size;    //    стек состоит из  символов
char* top;   //    указатель стека
public:
stack (int sz)    // конструктор
{top=s=new char [sz];  // выделение памяти и занесение ее в s и
top
}
~stack ( ) {delete s;}   // деструктор  удаляет память
};
void f ( )
{stack f (100);   // вызов конструктора для 100 объектов
      …
 }
```

В момент закрытия объекта автоматически вызывается деструктор, в него можно заключить все действия, связанные с завершением работы над объектом (например, очистить память, закрыть все файлы, с которыми работала программа, выдать какое-нибудь сообщение).

Оператор расширения области видимости

Оператор :: мы уже использовали при описании метода вне класса, чтобы показать принадлежность его определенному типу. Рассмотрим применение этого оператора для уточнения имени объекта:

1. Используется для обращения к глобальной переменной, закрытой локальной переменной.

```
int i=0;  // –  глобальная переменная
int f ( )
{…
int i=0; // –  локальная переменная с тем же именем
i++; // –  увеличиваем на 1 локальную переменную
:: i++; // –  увеличение на 1 глобальной переменной
   …
}
```

2. Используется для указания явного различия между именем членов класса и прочими именами.

Пример 1: Уточнение имени переменной **m**

```
class x
{int m;
public:…
void set (int m)
{x:: m=m;
// x:: m – это уточненное имя,   m – это имя аргумента функции
}
};
```

Пример 2: Уточнение имени функции

```
class my_file
{
```

```
public:
int open (char*, char*) //определена своя функция open, которая
};                      //  «закрывает» стандартную
int my_file :: open (char*name, char*spec) // при описании своей
{…                      //  функции open для обращения к
if(::open(name, flag))  // стандартной open используем оператор ::
{…
}
}
```

Ссылки

Для связи между функциями вместо значения самой переменной можно передать ее адрес (указатель) или ссылку. Если у нас есть переменная типа **t**, то **t&** означает ссылку на данный тип.

```
int  t;  // описание переменной
int& pt = t;  //описание ссылки pt и ее связь с переменной t
pt =10; //в переменную t записано значение 10
```

Ссылка – это второе имя объекта, то есть когда мы передаем функции ссылку на объект, она получает адрес объекта и тем самым может его модифицировать. Для ссылки не требуется дополнительного пространства в памяти, она является псевдонимом переменной.

Рассмотрим пример передачи данных между функциями (в этих функциях x и y меняются местами) с использованием указателя (функция swap1) и ссылки (функция swap2) :

```
void swap1 (int*x, int*y)  //  передача аргументов через указатель
{int z=*y; *y=*x;*x=z;    }
void swap2 (int &x, int &y)  // передача аргументов через ссылки
{int z=y; y=x; x=z; }
```

Воздействуя на формальные параметры внутри функций, мы изменяем значения аргументов, с которыми функция была вызвана.

Для вызова функции swap1 необходимо описать исходные данные и указатели на них:

```
int A,B;
int  *pA=&A, *pB=&B;
swap1(pA,pB);
```

Для вызова функции swap2 необходимо описание только исходных данных:

```
int A,B;
Swap2(&A,&B);
```

Передача параметров через ссылки удобна еще и тем, что внутри функции для обращения к аргументам не нужно использовать дополнительные операции адресации.

Дружественные функции класса

Предположим, для реализации математических операций необходимо описать два класса, Vector и Matrix (класс, описывающий работу над векторами, и класс, описывающий работу над матрицами). Описание свойств данных классов находится в закрытой части (private). Если нам необходимо реализовать умножение матрицы на вектор, то лучше описать эту функцию как внешнюю, не являющуюся методом ни одного из данных классов. Пусть вектор состоит из четырех элементов (которые индексируются 0...3),следовательно, для выполнения операции умножения матрица должна состоять из 4 столбцов. Пусть матрица будет размером 4 строки на 4 столбца. В каждом классе опишем метод elem(), который осуществляет доступ к элементам вектора и соответственно к элементам матрицы. Опишем внешнюю (глобальную) функцию mult():

```
Vector mult(Matrix& m, Vector& v);
{
Vector rez; // создадим локальный объект для хранения результата
```

```
for (int i = 0; i<3; i++) {
rez.elem(i) = 0;
for (int j = 0; j<3; j++)
rez.elem(i) += m.elem(i,j) * v.elem(j);
}
return rez;
}
```

Данная функция возвращает в то место, откуда был вызов объект типа Vector. Это один из способов реализации рассматриваемой математической операции. Проанализируем эффективность данного алгоритма: при каждом обращении к функции mult() метод elem() будет вызываться 4*(1+4*3) раза.

Если сделать функцию mult() методом класса Vector, то сможем обойтись без проверки индексов при обращении к элементу вектора, а если описать mult() методом класса Matrix, то мы сможем обойтись без проверки индексов при обращении к элементу матрицы. Однако членом двух классов функция быть не может. Но нам необходимо право доступа к закрытой части обоих классов. Функция не член, получившая право доступа к закрытой части класса, называется *другом класса* (friend). Функция становится другом класса после описания как friend в теле класса. В качестве аргументов функции будет использовать ссылки на соответствующие объекты. Например:

```
class Matrix; //описание прототипа класса matrix
class Vector {//описание  класса vector
float v[4];
// ...
friend Vector mult(Matrix&, Vector&);
};
class Matrix {
```

```
float  matr[4][4];
// ...
friend Vector mult(Matrix&, Vector&);
};
```

Функция friend не имеет никаких особенностей, помимо права доступа к закрытой части класса. Описание дружественной функции с ключевым словом friend должно быть помещено в тело класса. Оно может располагаться или в закрытой, или в открытой части описания класса; где именно, значения не имеет. Основное описание тела функции должно быть на внешнем уровне и без ключевого слова friend:

```
Vector mult(Matrix& m, Vector& v);
{ Vector rez;
for (int i = 0; i<3; i++) { rez.v[i] = 0;
     for (int j = 0; j<3; j++)
     rez.v[i] += m.matr[i][j] * v.v[j];
}
return rez;
}
```

Метод одного класса может быть другом другого. Например:

```
class X {
// ...
void function();
};
class Y {
// ...
friend void X::function();
};
```

Все методы одного класса могут являются друзьями другого. Для этого есть даже более краткая запись:

```
class X {
friend class Y;
// ...
};
```

Такое описание делает все методы класса **Y** друзьями класса **X**.

Переопределение операторов

Одна из возможностей класса – определение в нем *перегруженных* методов, это методы с одинаковым именем, но с разными входными данными. Поскольку типы данных различны, то различен и открытый интерфейс таких методов.

Переопределения операторов, заложенных в C++, позволяет программисту в дополнение к арифметическим и логическим операторам и операторам отношения переопределить операторы вызова ( ) и индексации [ ], а также переопределить операторы присваивания и инициализации.

Часто программы работают с объектами, которые являются конкретными представлениями абстрактных понятий. Например, тип данных int в C++ вместе с операциями +, –, *, / и т. д. предоставляет реализацию (ограниченную) математического понятия целых чисел. Такие понятия обычно включают в себя множество операций, которые кратко, удобно и привычно представляют основные действия над объектами. Классы дают возможность в C++ описать действия над неэлементарными объектами. Это позволяет обеспечить более общепринятую и удобную запись для манипуляции над объектами. Например, есть класс матриц и необходимо реализовать операции поэлементного сложения и вычитания матриц.

```
class Matrix {
    float matr[4][4];
public:
```

```
    //…
    friend Matrix operator+( Matrix&, Matrix&);
    friend Matrix operator-( Matrix&, Matrix&);
};
```

Этот класс достаточно понятно определяет операции над матрицами с помощью двух дружественных функций, реализованных как перегруженные операторы. Теперь в главной программе есть возможность использовать общепринятую запись сложения и вычитания, которая обеспечит лучшее понимание текста программы:

```
void main()
{
Matrix M1();
Matrix M2();
Matrix R1();
Matrix R2();
R1=M1+M2;
R2=M1-M2;
}
```

При использовании переопределенных операций выполняются обычные правила приоритетов.

Можно описывать функции, определяющие значения следующих операций:

```
+ - * / % ^ & | ~ !
= < > += -= *= /= %= ^= &=
|= << >> >>= <<= == != <= >= &&
|| ++ -- [] () new delete
```

Последние четыре оператора – это индексирование, вызов функции, выделение свободной памяти и освобождение свободной памяти. Изменить приоритеты перечисленных операций невозможно, как невозможно изменить и синтаксис выражений. Нельзя, например,

определить операцию «%» как унарную или операцию «!» как бинарную. Невозможно также придумать новые имена (обозначения) операций.

Формат описания переопределенной функции: пишется ключевое слово operator (то есть операция), за ним указывается знак операции, например, operator<<. Данная функция описывается и вызываться так же, как любая другая функция. Использование перегруженного оператора это лишь сокращенная запись явного вызова функции операции. Например:

```
void main()
{
Matrix M1();
Matrix M2();
Matrix R1();
Matrix R2();
R1=M1+M2; // краткая запись вызова оператора
R1= operator-(M1,M2);// полная запись вызова оператора
 }
```

Бинарная операция может быть определена или как функция–член класса, получающая один параметр, или как функция–друг, получающая два параметра. Рассмотрим следующие примеры:

```
class X {
// друзья
friend X operator-(X);      // унарный минус
friend X operator-(X,X);    // бинарный минус
friend X operator-();       // ошибка: нет операндов
friend X operator-(X,X,X);  // ошибка: несоответствие числа
                                    // аргументов
X* operator&(); // унарное & (взятие адреса)
X operator&(X); // бинарное & (операция И)
```

```
X operator&(X,X);   // ошибка: несоответствие числа аргументов
};
```

Когда операции ++ и — перегружены, префиксное использование и постфиксное различить невозможно.

Перегруженная операция может быть методом класса, и в этом случае в качестве одного из ее аргументов используются поля класса. Если переопределяются операции new и delete, функция обязательно должна быть членом класса (методом).

Если первый аргумент переопределяемой операции не является объектом, функция не может быть определена как член класса. Например, рассмотрим операцию умножения матрицы на константу. Вызов M1*2, при соответствующем описании метода может быть интерпретирован как M1.operator*(2). Но если вызов сделан как 2*M1, то это неверная операция, так как в классе int нет реализации умножения на наш объект. Даже если бы такой тип был, то для того, чтобы обработать и 2+aa и aa+2, понадобилось бы два различных метода. Для реализации вызова 2*M1 вполне подойдет внешняя дружественная функция.

Все допустимые в языке программирования операции по определению перегружены. Функция операция задает новый смысл операции в дополнение к встроенному определению, и может существовать несколько функций операций с одним и тем же именем, если в типах их параметров имеются отличия, различимые для компилятора, чтобы он мог различать их при обращении.

Рассмотрим пример описания объекта, состоящего из текстовой строки, ее длины и двух переопределенных операций: сложения и вывода (<<).

```
class string {public: char s [80];
                      int str_len;
string operator + (string, string);
```

```
friend ostream & operator << (ostream &, string &);
 };
string string :: operator + (string s1, string s2)
{ string temp;
  strcpy (temp, s1.s);
  strcut (temp, s2.s);
  temp.str_len=s1.str_len+s2.str_len;
  return temp;
}
ostream & operator << (ostream & st, string & x)
{return (st << "Строка:" << x.s << "Длина=" << x.str_len << '\n');
 }
void main ( )
{ string st1, st2, st3;
   strcpy (st1.s, «Это 1 строка»);
   strcpy (st2.s, «Это 2 строка»);
   st1.str_len=strlen (s1.s);
   st2.str_len=strlen (s2.s);
   st3=st1+st2;
   cout<<st3;
}
```

Наследование

Наследование – это повторное использование уже работающих классов с внесенными необходимыми дополнениями.

Наследование заключается в приеме некоторым производным классом компонентов базового класса. Формат описания производного класса следующий:

Class Tag: public Base {…};

Tag – имя производного класса;

Base – имя базового класса;

public – указывает право наследования;

операция «:» означает базируется на.

Компонентами производного класса являются все компоненты базового класса, за исключением конструктора, деструктора и операции «=». К ним добавляются те компоненты, которые описаны для производного класса.

Таблица 2. Соотношение атрибутов доступа в базовом и производном классе

Наследование с правом доступа	**Доступ в базовом классе**	**Доступ в производном классе**
public	public protected private	public protected недоступно
protected	public protected private	protected protected недоступно
private	public protected private	private private недоступно

Исходя из вышесказанного, можно сделать следующие выводы:

1) открытые элементы базового класса (public) полностью употребляются в производном классе;

2) закрытые элементы базового класса (private) в производном классе недоступны;

3) защищенные элементы базового класса (protected) могут полностью использоваться в производном классе.

Если в производном классе определен компонент с тем же именем, что и в базовом, то к нему можно обратиться, используя «::» – оператор разрешения области видимости.

Пример описания класса Person, строящегося на основе базовых классов Job и Name:

struct Name {char*first name;

```
                char*second name;     //Ф.И.О.
                char*surname;
                        …
               }
struct Job {char*Company; // организация
            char*Position;  // должность
                    …
           }
struct Person : Name, Job {int age;  // возраст
                               char*sex;  // пол
                                    …
                              }
```

При создании объекта производного класса сначала вызываются конструкторы базового класса, а затем конструкторы производного класса.

При разрушении объекта сначала вызываются деструкторы производного класса, а затем деструкторы базового класса.

Чтобы передать конструкторам базовых классов параметры, их необходимо указать в определении конструктора производного класса после «:».

```
struct Name { char*first name;
                char*second name;
                char*surname;
Name (char*FN, char*SN, char*Sur N)  // конструктор
{first name=FN; second name=SN;
surname=Sur N;}
~Name ( ) {…}
}
struct Job {char*Company;
            char*Position;
```

```
        Job (char*C, char*P)  // конструктор
{Company=C, Position=P}
 };
struct Person: Name, Job
{int age;
 char*sex;
Person (char*I first name, char*I second name, char*I surname, char*I Company, char*I Position, int I age, char*I sex):
Name (I first name, I second name, I surname),
Job (I Company, I Position)  // конструктор производного класса
{age=I age;
 sex=I sex;
}
void main ( )
{  Person Pl (″Иван″, ″Иванович″, ″Иванов″, ″УлГТУ″,
″Инженер″, 50,  ″муж″);
}
```

Если существует указатель на базовый класс, то его можно использовать для обращения к производному классу. Это связано с тем, что все наследуемое идет в производный класс первым.

Считаем, что у нас есть базовый класс Base1 и есть производный класс Deriv. Рассмотрим примеры описания и использования указателей на базовый и производный классы:

```
1) Base1 *p; // p — указатель на базовый класс
   p=new Base1; // проинициализировали указатель
   p=new Deriv;
2) Deriv d;
   Base1 * b_ptr=&d; // неявное преобразование указателей
3) Deriv *d_ptr;
   d_ptr=(Deriv*) b_ptr; // явное преобразование указателей
```

Объекты класса конструируются снизу вверх: сначала базовый, потом члены, а потом сам производный класс. Уничтожаются они в обратном порядке: сначала сам производный класс, потом члены, а потом базовый.

Виртуальные функции

Полиморфизм в языке программирования C++ обеспечивается за счет использования производных классов и виртуальных функций. Виртуальная функция — это функция, объявленная с ключевым словом virtual в базовом классе и переопределенная в одном или в нескольких производных классах. Виртуальные функции являются особыми функциями, потому что при вызове во время исполнения программы объекта производного класса с помощью указателя (ссылки) компилятор C++ определяет, какую функцию вызвать, основываясь на типе объекта, а не на типе используемого указателя. Для разных объектов могут вызываются разные версии одной и той же виртуальной функции. Класс, содержащий одну или более виртуальных функций, является полиморфным классом.

Виртуальная функция объявляется в базовом классе с использованием ключевого слова virtual. Когда она переопределяется в производном классе, повторять ключевое слово virtual нет необходимости, хотя и в случае его повторного использования ошибки не возникнет.

Рассмотрим пример использования виртуальной функции:

```
// небольшой пример использования виртуальных функций
#include <iostream.h>
class Base { // определение базового класса
public:
virtual void who() { // определение виртуальной функции
cout << *Base\n";
}
```

```
};
class first_d: public Base { // определение производного класса
public:
// переопределение виртуальной функции who() применительно к
first_d
void who() {
cout << "First class\n";
}
};
class second_d: public Base { // определение производного класса
public:
// переопределение виртуальной функции who() применительно к
second_d
void who() {
cout << "Second class\n*";
}
};
void main()// начало главной функции
{
Base base_obj;
Base *p; // определение указателя на базовый класс
first_d first_obj;
second_d second_obj;
p = &base_obj;
p->who(); // доступ к who класса Base
p = &first_obj;
p->who(); // доступ к who класса first_d
p = &second_ob;
p->who(); // доступ к who класса second_d
```

```
return 0;
}
```

Программа выдаст следующий результат:

```
Base
First class
Second class
```

Проанализируем эту программу, чтобы понять, как происходят вызовы виртуальных функций. Как мы видим, в объекте Base функция who() объявлена как виртуальная. Это означает, что эта функция может быть переопределена в производных классах. В каждом из классов first_d и second_d функция who() переопределена. В функции main() определены три объекта. Первым описан объект с именем base_obj, который относится к классу Base. Затем описан указатель **p** на класс Base, и далее созданы два объекта производных классов first_obj и second_obj. Затем указателю **p** присваивается адрес объекта base_obj и через указатель производится вызов функции who(). Поскольку эта функция объявлена как виртуальная, то компилятор C++ на этапе исполнения программы определяет, какую из версий функции who() использовать, в зависимости от того, на какой объект указывает указатель **p**. В данном случае им является объект типа Base, поэтому исполняется версия функции who(), объявленная в классе Base. Затем указателю **p** присваивается адрес производного объекта first_obj. (Как известно, указатель на базовый класс может быть использован для любого производного класса.) После того как функция who() была вызвана, C++ снова анализирует тип объекта, на который указывает **p**, для того, чтобы определить версию функции who(), которую необходимо вызвать. Поскольку **p** указывает на объект типа first_d, то используется соответствующая версия функции who(). Аналогично, когда указателю **p** присвоен

адрес объекта second_obj, то используется версия функции who(), объявленная в классе second_d.

Наиболее распространенным способом вызова виртуальной функции служит использование параметра функции. Например, рассмотрим следующую модификацию предыдущей программы:

```
/* Здесь ссылка на базовый класс используется для доступа к виртуальной функции */
#include <iostream.h>
class Base {
public:
virtual void who() { // определение виртуальной функции
cout << "Base\n";
}
};
class first_d: public Base {
public:
void who () { // определение who() применительно к first_d
cout << "First class\n";
}
};
class second_d: public Base {
public:
void who() { // определение who() применительно к second_d
cout << "Second class\n*";
}
};
// описание внешней функции
//использование в качестве параметра функции ссылки на базовый класс
void show_who (Base &r) {
```

```
r.who();  // вызов виртуальной функции для объекта r
}
void main()
{
Base base_obj;
first_d first_obj;
second_d second_obj;
show_who (base_ob j) ; // доступ к who класса Base
show_who(first_obj); // доступ к who класса first_d
show_who(second_obj); // доступ к who класса second_d
return 0;
}
```

Приведенная выше программа выдаст на экран такие же результаты, что и предыдущая. Отличие между ними в том, что описанная внешняя функция show_who() принимает в качестве исходного параметра ссылку на класс Base. В основной функции main() создаются три объекта, при этом вызов виртуальной функции who() осуществляется через внешнюю функцию show_who().В этом случае версия функции who(), которая должна быть вызвана в каждом конкретном случае, определяется типом объекта, на который ссылается параметр при вызове функции.

Основной особенностью применения виртуальных функций для обеспечения полиморфизма времени исполнения является то, что используется указатель именно на базовый класс. Полиморфизм времени исполнения достигается только при вызове виртуальной функции с использованием указателя или ссылки на базовый класс.

Можно отметить, что механизм виртуальных функций напоминает механизм перегрузки функции. Тем не менее, между ними имеются существенные различия. Как уже отмечалось, перегруженные функции имеют одинаковое имя, но количество и тип

аргументов должны быть различными. А виртуальные функции в каждом классе имеют одинаковые аргументы. Следующее отличие состоит в том, что виртуальная функция обязательно должна быть методом, а не другом класса.

Если функция была объявлена как виртуальная, то она остается таковой вне зависимости от количества уровней в иерархии классов. Например, если класс second_d получен из класса first_d, а не из класса Base, то функция who() останется виртуальной и будет вызываться корректная ее версия, как показано в следующем примере:

```
// этот класс производный от first_d, а не от Base
class second_d: public first_d {
public:
void who() { // определение who() применительно к second_d
cout << "Second class\n*";
}
};
```

Если в производном классе виртуальная функция не переопределяется, то тогда используется ее версия из базового класса.

```
#include <iostream.h>
class Base {
public:
virtual void who() {
cout << "Base\n"; }
};
class first_d: public Base {
public:
void who() {
cout << "First class\n"; }
};
class second_d: public Base {
```

```
// who() не переопределяется
};
void main()
{
Base base_obj;
Base *p;
first_d first_obj; ,
second_d second_obj;
p = &base_obj;
p->who(); // доступ к who класса Base
p = &first obj;
p->who(); // доступ к who класса first_d
p = &sepond_ob;
p->who(); /* доступ к who() класса Base, поскольку second_d не
переопределяет */
return 0;
}
```

Итак, если класс не переопределяет виртуальную функцию, язык C++ использует первое из определений, которое он находит, следуя от потомков к предкам. Рекомендации:

- Если в базовом классе есть функция с ключевым словом virtual, то такой класс называется *полиморфным*, а все функции внутри потомков (которые предполагается переопределить) называются *виртуальными*.
- То, какая виртуальная функция будет вызвана определяется во время выполнения программы.
- Виртуальная функция должна быть членом класса, для которого определяется, а не его «другом», но в то же время виртуальная функция может быть другом иного класса**.**

- Функциям деструкторов разрешается быть виртуальными, а функциям конструкторов нет.

Лабораторный практикум по ООП

1 Описание класса

Вариант 1. Опишите класс для формирования массива вещественных чисел. Конструктор класса читает числа из символьной строки. Функция разбора упорядочивает их по возрастанию. Общий интерфейс описания класса может выглядеть так:

```
class mas
{ float massiv[20]; // описание массива для хранения вещественных чисел
  int N; // переменная для хранения размера массива чисел
//…
public:
        mas (char*);
int razbor ( );
void print ( );
};
```

Конструктор mas::mas ( ) имеет параметр-строку и переписывает данные из строки в массив вещественных чисел. Функция mas::razbor ( ) упорядочивает элементы по возрастанию.. Функция mas::print( ) выводит на экран элементы массива. Вывести необходимо как исходный, так и отсортированный массив. Примеры описания объекта и вызова методов класса приведены ниже:

```
mas x("1, 3, 4 ,4.3"); // конструктор, элементы массива вводятся в строчку
                       //через запятую
x.razbor( ); //вызов метода сортировки
x.print();   //вызов метода для вывода элементов массива
```

Программа должна выполнить работу с тремя различными объектами.

Вариант 2. Опишите класс для формирования массива целых чисел. Конструктор класса читает числа из символьной строки. Функция сортировки упорядочивает их по убыванию. Общий интерфейс описания класса может выглядеть так:

```
class mas_int
```

```
{ int massiv[20]; // описание массива для хранения целых чисел
  int N; // переменная для хранения размера массива чисел
//…
public:
mas_int (char*);
void sort( );
void print( );
};
```

Конструктор mas_int::mas_int ( ) имеет параметр-строку и переписывает данные из строки в массив целых чисел. Функция mas_int::sort ( ) упорядочивает элементы по убыванию. Функция mas_int::print( ) выводит на экран элементы массива. Вывести необходимо как исходный, так и отсортированный массив. Примеры описания объекта и вызова методов класса приведены ниже:

```
mas_int x("1,3,5,4,4,7"); // конструктор, элементы массива вводятся в строчку
                                  //через запятую
x.sort( ); //вызов метода сортировки
x.print ( );    //вызов метода для вывода элементов массива
```

Программа должна выполнить работу с тремя различными объектами.

Вариант 3. Опишите класс для формирования матрицы целых чисел. Конструктор класса читает числа из символьной строки. Первый столбец матрицы должен быть упорядочен по возрастанию. Общий интерфейс описания класса может выглядеть так:

```
class matr
{ int massiv[10][10]; // описание матрицы для хранения целых чисел
  int N,M; // переменные для хранения размера матрицы чисел
//…
public:
matr (char*);
int razbor1( );
int razbor2 ( );
void sort( );
void print ( );
};
```

конструктор matr::matr( ) имеет параметр-строку, задающую матрицу, функция matr::sort( ) упорядочивает элементы первого столбца, функция matr::razbor1( ) возвращает число строк, а функция matr::razbor2( ) возвращает число столбцов в матрице. Функция matr::print() выводит на экран элементы матрицы.

Использовать эти функции можно так:

```
matr x("(1, 3, 4) (4, 3, 12)"); // элементы массива вводятся в строчку ,
                                 //через запятую, строки заключаются в скобки
x.razbor1( );
x.razbor2( );
x.sort( ); //вызов метода сортировки
x.print ( ); //вызов метода для вывода элементов матрицы
```

Программа должна выполнить работу с тремя различными объектами.

Вариант 4. Опишите класс для формирования, разбора и вывода простых арифметических выражений, состоящих из целых констант и операций +, –. Общий интерфейс описания класса может выглядеть так:

```
class expr
{char *stroka;  // переменная, которая хранит исходное выражение
 int kol_const, kol_oper; // переменные для хранения количества констант
                          // и операций в выражении
//…
public:
        expr (char*);
        void razbor ( );
        void print( );
};
```

Конструктор expr::exper( ) имеет параметр-строку, задающую выражение. Функция expr::razbor ( ) определяет число операндов и операций в выражении, функция expr::print( ) выводит выражение на экран.

Использовать эти функции можно так:

```
expr x("123+4+123-4-3"); // вызов конструктора с параметром – строкой
x.razbor( ); // вызов метода для разбора выражения
x.print( ); // вызов метода для вывода на экран
```

Программа должна выполнить данные с тремя различными объектами.

Вариант 5. Опишите класс для работы со строкой символов. Конструктор должен «отфильтровать» (т. е. удалять из строки) символы, отличные от заглавных латинских букв или цифр. Далее необходимо подсчитать количество четных цифр в полученной строке. Общий интерфейс описания класса может выглядеть так:

```
class string
{char *stroka;  // переменная, которая хранит отфильтрованную строку
 int kol_digit; // переменная для хранения количества четных цифр
//…
public:
        string (char*);
        int count( );
        void print ( );
};
```

Конструктор string:: string ( ) читает исходную строку и фильтрует ее. Функция string::count ( ) определяет количество четных цифр в отфильтрованной строке, функция string::print( ) выводит строку на экран.

Использовать эти функции можно так:

```
string x( “DJaasVH446KS^*()*05#!”);
// вызов конструктора с параметром – строкой
 x.count();//вызов метода для подсчета количества четных цифр
x.print( ); // вызов метода для вывода на экран
```

Программа должна выполнить работу с тремя различными объектами.

Вариант 6. Описать класс для подсчета количества «длинных» слов (т. е. длина которых превышает 10 символов) в переданной фразе. Общий интерфейс описания класса может выглядеть так:

```
class mas_string
{char *stroka;  // переменная, которая хранит исходную строку
char *slova[10];  // массив для хранения длинных слов (не более 10)
 int kol_slov; // переменная для хранения количества длинных слов
//…
public:
mas_string (char*);
int count( );
void( );
```

```
};
```

Конструктор mas_string:: mas_string ( ) читает исходную строку и выбирает из нее длинные слова. Функция mas_string::count ( ) определяет количество длинных слов в исходной строке, функция mas_string::print( ) выводит длинные слова на экран.

Использовать эти функции можно так:

```
mas_string x( “Dsadsedddddd hhh, nhnjhbnjhygt gg”); // конструктор
// разделители слов – пробел или запятая
x.count( ); //вызов метода для подсчета количества длинных слов
x.print( ); // вызов метода для вывода на экран
```

Программа должна выполнить работу с тремя различными объектами.

Вариант 7. Опишите класс для формирования массива случайных целых чисел в заданном количестве и с заданным распределением. В процессе формирования конструктор должен «отфильтровать» (т. е. удалять из массива) все нечетные числа. Общий интерфейс класса может выглядеть так:

```
class mas_random
{int massiv[20]; // описание массива для хранения целых чисел
  int N; // переменная для хранения размера массива чисел
//…
public:
        mas_random (int, int);
        int count( );
void print( );
};
```

Конструктор mas_random:: mas_random ( ) имеет два параметра – целых числа. Первое число задает количество чисел в массиве, второе – максимальное число для диапазона генерации (например, mas_random(10,50) означает, что генерируется 10 чисел в диапазоне от 0 до 50). Конструктор переписывает данные в массив целых чисел. Функция mas_random:: count ( ), возвращающая количество чисел, сумма разрядов у которых больше 10. Функция mas_random::print( ) выводит на экран элементы массива. Примеры описания объекта и вызова методов класса приведены ниже.

```
mas_random x( 20, 100); //вызов конструктора
 x.count( ); //подсчет количество чисел, сумма разрядов у которых больше 10
```

x.print( ); // вывод массива на экран

Программа должна выполнить работу с тремя различными объектами.

Вариант 8. Опишите класс для формирования матрицы случайных целых чисел заданной размерности и с заданным распределением. В процессе формирования конструктор должен «отфильтровывать» (т. е. заменять на ноль) все нечетные числа. Общий интерфейс класса может выглядеть так:

```
class mass_random
{ int massiv[10][10]; // описание матрицы для хранения целых чисел
  int N,M; // переменные для хранения размера матрицы чисел
//…
public:
        mass_random (int, int, int);
        int count( );
void print( );
};
```

Конструктор mass_random:: mass_random ( ) имеет три параметра – целых числа. Первые два числа задают количество строк и столбцов в массиве, третье – максимальное число для диапазона генерации (например, mass_random(5,5,50) означает, что генерируется матрица размером 5 на 5 с диапазоном значений от 0 до 50). Конструктор переписывает данные в матрицу целых чисел. Функция mass_random:: count ( ), возвращающая количество нулевых элементов в матрице. Функция mas_random::print( ) выводит на экран элементы массива. Примеры описания объекта и вызова методов класса приведены ниже

```
matr_random x( 5, 5, 100); //вызов конструктора
 x.count( ); // вызов метода для подсчета количества нулей в матрице
x.print( );  // вызов метода для вывода матрицы на экран
```

Программа должна выполнить работу с тремя различными объектами.

Вариант 9. Определите класс для формирования случайного двоичного вектора заданной длины (данный вектор – это одномерный массив, значениями которого могут быть только числа 0 и 1). После генерации вектора осуществляется его кодирование, т. е. к вектору добавляется один контрольный разряд, такой, чтобы общее число единичных разрядов в коде было четным. Общий интерфейс класса может выглядеть так:

```
class kod_chet
{int vect[30];
```

```
// описание массива для хранения вектора не более 30 разрядов
// пусть нулевой элемент массива будет контрольным разрядом
int N; // переменная, которая хранит размер массива
//…
public:
        kod_chet (int); // конструктор класса
        void kod( ); // кодирование вектора
        void print ( ); // вывод вектора
};
```

Конструктор kod_chet:: kod_chet(int) генерирует с помощью датчика случайных чисел массив заданного размера (например, kod_chet(10) должен создать массив из 10 чисел). Метод kod( ) проверяет вектор на четность (количество единиц в массиве – четное) и в зависимости от результата проверки дописывает в контрольный разряд 0 или 1.

Общий интерфейс класса может выглядеть так:

```
kod_chet x(20); // вызов конструктора
x.kod( ); // вызов метода для проверки на четность
x.print( );// вывод вектора на экран
```

Программа должна выполнить работу с тремя различными объектами.

Вариант 10. Определите класс для формирования случайного двоичного вектора заданной длины (данный вектор – это одномерный массив, значениями которого могут быть только числа 0 и 1). После генерации вектора осуществляется его кодирование, т. е. к вектору добавляется один контрольный разряд, такой, чтобы общее число единичных разрядов в коде было нечетным. Общий интерфейс класса может выглядеть так:

```
class dekod_nechet
{int vect[30];
// описание массива для хранения вектора не более 30 разрядов
// пусть нулевой элемент массива будет контрольным разрядом
int N; // переменная, которая хранит размер массива //…
public
        dekod_nechet (int); // конструктор класса
        int dekod( ); // кодирование вектора
        void print( ); // вывод вектора
};
```

Конструктор dekod_nechet:: dekod_nechet(int) генерирует с помощью датчика случайных чисел массив заданного размера (например, dekod_nechet(10) должен создать массив из 10 чисел). Метод dekod( ) проверяет вектор на нечетность (количество единиц в массиве – нечетное) и в зависимости от результата проверки дописывает в контрольный разряд 0 или 1.

Общий интерфейс класса может выглядеть так:

```
dekod_nechet x(20); // вызов конструктора
x.dekod( ); // вызов метода для проверки на нечетность
x.print( ); // вывод вектора на экран
```

Программа должна выполнить работу с тремя различными объектами.

Вариант 11. Определите класс для формирования случайного двоичного вектора заданной длины (данный вектор – это одномерный массив, значениями которого могут быть только числа 0 и 1). Конструктор читает вектор из строки символов. После ввода вектора осуществляется его кодирование, т. е. к вектору добавляется один контрольный разряд, такой, чтобы общее число единичных разрядов в коде было четным. Общий интерфейс класса может выглядеть так:

```
class kod_chet
{int vect[30];
// описание массива для хранения вектора не более 30 разрядов
// пусть нулевой элемент массива будет контрольным разрядом
int N; // переменная, которая хранит размер массива
//…
public:
        kod_chet (char*); // конструктор класса
        void kod( ); // кодирование вектора
        void print ( ); // вывод вектора
};
```

Конструктор kod_chet:: kod_chet(char*) читает из строки символов массив заданного размера. Метод kod( ) проверяет вектор на четность (количество единиц в массиве – четное) и в зависимости от результата проверки дописывает в контрольный разряд 0 или 1.

Общий интерфейс класса может выглядеть так:

```
kod_chet x(“011001111100”); // вызов конструктора
x.kod( ); // вызов метода для проверки на четность
x.print( );// вывод вектора на экран
```

Программа должна выполнить работу с тремя различными объектами.

Вариант 12. Определите класс для формирования случайного двоичного вектора заданной длины (данный вектор – это одномерный массив, значениями которого могут быть только числа 0 и 1). Конструктор класса читает вектор из строки символов и записывает его в числовой массив. После ввода вектора осуществляется его кодирование, т. е. к вектору добавляется один контрольный разряд, такой, чтобы общее число единичных разрядов в коде было нечетным. Общий интерфейс класса может выглядеть так:

```
class dekod_nechet
{int vect[30];
// описание массива для хранения вектора не более 30 разрядов
// пусть нулевой элемент массива будет контрольным разрядом
int N; // переменная, которая хранит размер массива //…
public
        dekod_nechet (char*); // конструктор класса
        int dekod( ); // кодирование вектора
        void print( ); // вывод вектора
};
```

Конструктор dekod_nechet:: dekod_nechet(char*) вводит из строки массив. Метод dekod( ) проверяет вектор на нечетность (количество единиц в массиве – нечетное) и в зависимости от результата проверки дописывает в контрольный разряд 0 или 1.

Общий интерфейс класса может выглядеть так:

```
dekod_nechet x(“11110000111000”); // вызов конструктора
x.dekod( ); // вызов метода для проверки на нечетность
x.print( ); // вывод вектора на экран
```

Программа должна выполнить работу с тремя различными объектами

Вариант 13. Опишите класс для формирования матрицы случайных целых чисел заданной размерности и с заданным распределением. В процессе формирования конструктор должен «отфильтровывать» (т. е. заменять на ноль) все четные числа. Общий интерфейс класса может выглядеть так:

```
class mass_random
{ int massiv[10][10]; // описание матрицы для хранения целых чисел
  int N,M; // переменные для хранения размера матрицы чисел
//…
```

```
public:
        mass_random (int, int, int);
        int count( );
void print( );
};
```

Конструктор mass_random:: mass_random ( ) имеет три параметра – целые числа. Первые два числа задают количество строк и столбцов в массиве, третье – максимальное число для диапазона генерации (например, mass_random(5,5,50) означает, что генерируется матрица размером 5 на 5 с диапазоном значений от 0 до 50). Конструктор переписывает данные в матрицу целых чисел. Функция mass_random:: count ( ), возвращающая количество элементов в диапазоне от 0 до 9. Функция mas_random::print( ) выводит на экран элементы массива. Примеры описания объекта и вызова методов класса приведены ниже

```
matr_random x( 5, 5, 100); //вызов конструктора
 x.count( ); // вызов метода для подсчета количества чисел из диапазона
x.print( );  // вызов метода для вывода матрицы на экран
```

Программа должна выполнить работу с тремя различными объектами.

Вариант 14. Опишите класс для формирования массива случайных целых чисел в заданном количестве и с заданным распределением. В процессе формирования конструктор должен «отфильтровать» (т. е. удалять из массива) все четные числа. Общий интерфейс класса может выглядеть так:

```
class mas_random
{int massiv[20]; // описание массива для хранения целых чисел
  int N; // переменная для хранения размера массива чисел
//…
public:
        mas_random (int, int);
        int count( );
void print( );
};
```

Конструктор mas_random:: mas_random ( ) имеет два параметра – целые числа. Первое число задает количество чисел в массиве, второе – максимальное число для диапазона генерации (например, mas_random(10,50) означает, что генерируется 10 чисел в диапазоне от 0 до 50). Конструктор переписывает данные в массив целых чисел. Функция mas_random:: count ( ), возвращающая

количество чисел, сумма разрядов у которых меньше 8. Функция mas_random::print( ) выводит на экран элементы массива. Примеры описания объекта и вызова методов класса приведены ниже

```
mas_random x( 20, 100); //вызов конструктора
x.count( ); //подсчет количество чисел, сумма разрядов у которых меньше 8
x.print( ); // вывод массива на экран
```

Программа должна выполнить работу с тремя различными объектами.

Вариант 15. Опишите класс для работы со строкой символов. Конструктор должен «отфильтровать» (т. е. удалять из строки) символы, отличные от заглавных латинских букв. Далее необходимо подсчитать количество коротких слов в полученной строке (длина слова меньше 4). Отсортировать слова в строке по возрастанию их длины. Общий интерфейс описания класса может выглядеть так:

```
class mas_string
{char *stroka;  // переменная, которая хранит отфильтрованную строку
 int kol_digit; // переменная для хранения количества слов
// ...
public:
mas_stririg (char*);
int count( );
void sort( );
void print( );
};
```

Конструктор mas_string:: mas_string (char*) читает исходную строку и фильтрует ее. Функция mas_string::count ( ) определяет количество коротких слов в отфильтрованной строке, функция mas_string:: sort ( ) сортирует слова по возрастанию их длины, функция mas_string::print( ) выводит строку на экран.

Использовать эти функции можно так:

```
mas_strlng x(“DDH JKTYRR fgdr KLJHYT”); //вызов конструктора
x.count( ); //
x.sort( ); //
x.print( ); //
```

Программа должна выполнить работу с тремя различными объектами.

Вариант 16. Опишите класс для формирования, сортировки и вывода на экран массива вещественных чисел. Конструктор читает числа из строки

символов и записывает их в числовой массив. Далее полученный массив упорядочивается по убыванию. Общий интерфейс класса может выглядеть так:

```
class mas_real
{ float massiv[20]; // описание массива для хранения вещественных чисел
  int N; // переменная для хранения размера массива чисел
//…
public:
        mas_real(char*);
        void sort( );
        void print( );
};
```

Конструктор mas_real::mas_real ( ) имеет параметр-строку и переписывает данные из строки в массив вещественных чисел. Функция mas_real::razbor ( ) упорядочивает элементы по убыванию. Функция mas::print( ) выводит на экран элементы массива. Вывести необходимо как исходный, так и отсортированный массив. Примеры описания объекта и вызова методов класса приведены ниже:

```
mas x("1.4, 3.7, 4.6 ,4.3"); // конструктор, элементы массива вводятся в строчку
                             //через запятую
x.razbor( ); //вызов метода сортировки
x.print();    //вызов метода для вывода элементов массива
```

Программа должна выполнить работу с тремя различными объектами.

Вариант 17. Опишите класс для формирования массива случайных целых чисел в заданном количестве и с заданным распределением. В процессе формирования конструктор должен «отфильтровать» (т. е. удалять из массива) все нечетные числа. Общий интерфейс класса может выглядеть так:

```
class mas_random
{int massiv[20]; // описание массива для хранения целых чисел
  int N; // переменная для хранения размера массива чисел
//…
public:
        mas_random (int, int);
        int count( );
void print( );
};
```

Конструктор mas_random:: mas_random ( ) имеет два параметра – целые числа. Первое число задает количество чисел в массиве, второе – максимальное число для диапазона генерации (например, mas_random(10,50) означает, что генерируется 10 чисел в диапазоне от -50 до 50). Конструктор переписывает данные в массив целых чисел. Функция mas_random:: count ( ), возвращающая количество чисел, сумма разрядов у которых больше 5. Функция mas_random::print( ) выводит на экран элементы массива. Примеры описания объекта и вызова методов класса приведены ниже

```
mas_random x( 20, 100); //вызов конструктора
 x.count( ); //подсчет количество чисел, сумма разрядов у которых больше 5
x.print( ); // вывод массива на экран
```

Вариант 18. Опишите класс для формирования, перевода в десятичную форму и вывода на дисплей целых шестнадцатеричных чисел. В процессе формирования конструктор должен «отфильтровывать» (т. е. удалять из массива) все символы, отличные от представления шестнадцатеричного разряда (т. е. символы отличные от 0-9 и A-F). Общий интерфейс класса должен выглядеть примерно так:

```
class num_16
{char *num; // строка для хранения шестнадцатеричного числа
int N; // переменная для хранения длины строки
//…
public
        num_16 (char*);
        void to_10( ); // метод для перевода число в десятичное
        void print( );
};
```

Конструктор num_16::num_16() читает из параметра – строки символы и отфильтровывает те, которые не могут быть в записи шестнадцатеричного числа. Метод num_16:: to_10( ) переводит исходное число в десятичную систему счисления. Метод num_16:: print() предназначен для вывода на экран шестнадцатеричного числа.

Использовать эти функции можно примерно так:

```
num_16 x(“95axz5”); // вызов конструктора
x.to_10( ); // вызов метода перевода число в десятичную систему счисления
```

x.print( ); //вывод числа на экран

Программа должна выполнить работу с тремя различными объектами.

Вариант 19. Опишите класс для формирования, перевода в десятичную форму и вывода на дисплей целых восьмеричных чисел. В процессе формирования конструктор должен «отфильтровывать» (т. е. удалять из массива) все символы, отличные от представления восьмеричного разряда (т. е. символы отличные от 0-7). Общий интерфейс класса должен выглядеть примерно так:

```
class num_8
{char *num; // строка для хранения восьмеричного числа
int N; // переменная для хранения длины строки
//…
public
        num_8 (char*);
        void to_10( ); // метод для перевода число в десятичное
        void print( );
};
```

Конструктор num_8::num_8() читает из параметра – строки символы и отфильтровывает те, которые не могут быть в записи восьмеричного числа. Метод num_8:: to_10( ) переводит исходное число в десятичную систему счисления. Метод num_8:: print() предназначен для вывода на экран восьмеричного числа.

Использовать эти функции можно примерно так:

num_8 x(“495811”); // вызов конструктора

x.to_10( ); // вызов метода перевода число в десятичную систему счисления

x.print( ); //вывод числа на экран

Программа должна выполнить работу с тремя различными объектами.

Вариант 20. Опишите класс для работы со строкой символов. Конструктор должен «отфильтровать» (т. е. удалять из строки) символы, отличные от строчных латинских букв или цифр. Далее необходимо подсчитать количество слов в полученной строке. Отсортировать слова в строке по убыванию их длины. Общий интерфейс описания класса может выглядеть так:

```
class mas_string
{char *stroka;  // переменная, которая хранит отфильтрованную строку
 int kol_digit; // переменная для хранения количества слов
```

```
// ...
public:
mas_stririg (char*);
int count( );
void sort( );
void print( );
};
```

Конструктор mas_string:: mas_string (char*) читает исходную строку и фильтрует ее. Функция mas_string::count ( ) определяет количество слов в отфильтрованной строке, функция mas_string:: sort ( ) сортирует слова по убыванию их длины, функция mas_string::print( ) выводит строку на экран.

Использовать эти функции можно так:

```
mas_strlng x("zzxsd DSEgfd W324Dhgy"); //вызов конструктора
x.count( ); //вызов метода для подсчета количества слов
x.sort( ); // вызов метода для сортировки
x.print( ); //вызов метода для печати
```

Программа должна выполнить работу с тремя различными объектами.

2 Описание дружественных функций

Для созданного в задании 1 класса опишите новые функции, среди них обязательно должны быть дружественные.

Вариант 1. Класс mas дополнительно должен реализовать следующие операции:

(search) поиск элемента;

(insert) включение нового элемента в массив;

(delete) удаление элемента из массива.

Программа должна выполнить данные операции с несколькими объектами. Результаты всех операций вывести на экран.

Вариант 2. Класс mas_int дополнительно должен реализовать следующие операции:

(search) поиск элемента;

(insert) включение нового элемента в массив;

(delete) удаление элемента из массива;

(add) слияние двух упорядоченных массивов в один упорядоченный массив.

Программа должна выполнить данные операции с несколькими объектами. Результаты всех операций вывести на экран.

Вариант 3. Класс matr дополнительно должен реализовать следующие операции:

(search) поиск элемента;

(insert) включение новой строки;

(delete) удаление строки из массива.

Программа должна выполнить данные операции с несколькими объектами. Результаты всех операций вывести на экран.

Вариант 4. Класс expr дополнительно должен реализовать следующие операции:

(count) подсчитывает значение выражения;

(add) находит сумму значений двух выражений;

(mult) находит произведение значений двух выражений.

Объявите, какие можно, функции дружественными.

Программа должна выполнить данные операции с несколькими объектами. Результаты всех операций вывести на экран.

Вариант 5. Класс string дополнительно должен реализовать следующие операции:

(add) склеивание двух строк;

(minus) удаление из первой строки всех символов второй строки;

(mult) формирование строки, состоящей из символов, присутствующих одновременно как в первой строке, так и во второй.

Объявите, какие можно, функции дружественными.

Программа должна выполнить данные операции с несколькими объектами. Результаты всех операций вывести на экран.

Вариант 6. Класс mas_string дополнительно должен реализовать следующие операции:

(add) склеивание двух массивов строк;

(minus) удаление из первой массива всех слов второго массива;

(mult) формирование массива, состоящего из слов, присутствующих одновременно как в первой строке, так и во второй.

Объявите, какие можно, функции дружественными.

Программа должна выполнить данные операции с несколькими объектами. Результаты всех операций вывести на экран.

Вариант 7. Класс mas_random дополнительно должен реализовать следующие операции:

(form) формирование из заданного массива массива требуемой длины.

(add) сложение двух массивов (по правилу zi=xi+yi);

(minus) вычитание двух массивов (по правилу zi=xi-yi);

(mult) умножение скаляра на массив (по правилу zi=v+yi).

Объявите, какие можно, функции дружественными.

Программа должна выполнить данные операции с несколькими объектами. Результаты всех операций вывести на экран.

Вариант 8. Класс mass_random дополнительно должен реализовать следующие операции:

(add) сложение двух матриц;

(minus) вычитание двух матриц;

(mult) умножение скаляра на матрицу.

Объявите, какие можно, функции дружественными.

Программа должна выполнить данные операции с несколькими объектами. Результаты всех операций вывести на экран.

Вариант 9. Класс kod_chet дополнительно должен реализовать следующие операции:

(add) сложение двух векторов;

(rang) вычисление ранга вектора (c=a[1]+a[2]+…+a[n]);

(mult) умножение векторов(c[i]=a[i]*b[i]).

Объявите, какие можно, функции дружественными.

Программа должна выполнить данные операции с несколькими объектами. Результаты всех операций вывести на экран.

Вариант 10. Класс dekod_nechet дополнительно должен реализовать следующие операции:

(add) сложение двух векторов;

(rang) вычисление ранга вектора (c=a[1]+a[2]+…+a[n]);

(mult) скалярное умножение векторов(c=sum(a[i]*b[i])).

Объявите, какие можно, функции дружественными.

Программа должна выполнить данные операции с несколькими объектами. Результаты всех операций вывести на экран.

Вариант 11. Класс kod_chet дополнительно должен реализовать следующие операции:

(rang) вычисление ранга вектора (c=a[1]+a[2]+…+a[n]);

(mult) умножение векторов(c[i]=a[i]*b[i]);

(mult2) скалярное умножение векторов(c=sum(a[i]*b[i])).

Объявите, какие можно, функции дружественными.

Программа должна выполнить данные операции с несколькими объектами. Результаты всех операций вывести на экран.

Вариант 12. Класс dekod_nechet дополнительно должен реализовать следующие операции:

(ort) проверка взаимной ортогональности двух векторов;

(mult1) умножение векторов(c[i]=a[i]*b[i]);

(mult2) скалярное умножение векторов(c=sum(a[i]*b[i])).

Объявите, какие можно, функции дружественными.

Программа должна выполнить данные операции с несколькими объектами. Результаты всех операций вывести на экран.

Вариант 13. Класс mass_random дополнительно должен реализовать следующие операции:

(add) сложение двух матриц;

(minus) вычитание двух матриц;

(mult) умножение скаляра на матрицу.

Объявите, какие можно, функции дружественными.

Программа должна выполнить данные операции с несколькими объектами.

Результаты всех операций вывести на экран.

Вариант 14. Класс mass_random дополнительно должен реализовать следующие операции:

(poisk) поиск одинаковых элементов в двух массивах;

(minus) вычитание двух массивах;

(max) поиск максимального элемента в массиве.

Объявите, какие можно, функции дружественными.

Программа должна выполнить данные операции с несколькими объектами. Результаты всех операций вывести на экран.

Вариант 15. Класс mas_string с дополнительно должен реализовать следующие операции:

(add) склеивание двух массивов строк;

(rshift) циклический сдвиг вправо на заданное растояние массива;

(lshift) циклический сдвиг влево на заданное растояние массива.

Объявите, какие можно, функции дружественными.

Программа должна выполнить данные операции с несколькими объектами. Результаты всех операций вывести на экран.

Вариант 16. Класс mas_real дополнительно должен реализовать следующие операции:

(add) слияние двух упорядоченных массивов в один упорядоченный;

(delete) удаление элемента из массива;

(mult) перемножение двух массивов по правилу скалярного произведения.

Объявите, какие можно, функции дружественными.

Программа должна выполнить данные операции с несколькими объектами. Результаты всех операций вывести на экран.

Вариант 17. Класс mas_random дополнительно должен реализовать следующие операции:

(minus) вычитание двух массивов;

(delete) удаление элемента из массива;

(mult) перемножение двух массивов по правилу скалярного произведения.

Объявите, какие можно, функции дружественными.

Программа должна выполнить данные операции с несколькими объектами. Результаты всех операций вывести на экран.

Вариант 18. Класс num_16 дополнительно должен реализовать следующие операции:

(minus) вычитание двух шестнадцатеричных чисел;

(add) сложение двух шестнадцатеричных чисел;

(binar) вывод заданного числа в двоичном виде.

Объявите, какие можно, функции дружественными.

Программа должна выполнить данные операции с несколькими объектами. Результаты всех операций вывести на экран.

Вариант 19. Класс num_8 дополнительно должен реализовать следующие операции:

(minus) вычитание двух восьмеричных чисел;

(add) сложение двух восьмеричных чисел;

(binar) вывод заданного числа в двоичном виде.

Объявите, какие можно, функции дружественными.

Программа должна выполнить данные операции с несколькими объектами. Результаты всех операций вывести на экран.

Вариант 20. Класс mas_string дополнительно должен реализовать следующие операции:

(add) склеивание двух массивов строк;

(minus) удаление из первой массива всех слов второго массива;

(mult) формирование массива, состоящего из слов, присутствующих одновременно как в первой строке, так и во второй.

Объявите, какие можно, функции дружественными.

Программа должна выполнить данные операции с несколькими объектами. Результаты всех операций вывести на экран.

3 Переопределение операторов

Вариант 1. Перепишите следующие функции класса mas:

(insert) замените на оператор + (включение нового элемента в множество)

(delete) замените на оператор – (удаление элемента из множества).

Дополнительно переопределите следующий оператор:

Сложение двух массивов (оператор*)

Программа должна выполнить данные операции с несколькими объектами. Результаты всех операций вывести на экран.

Вариант 2. Перепишите следующие функции класса mas_real:

(insert) замените на оператор+ (включение нового элемента в массив);

(delete) замените на оператор – (удаление элемента из массива);

(add) замените на оператор * (слияние двух упорядоченных массивов в один упорядоченный массив).

Дополнительно переделайте следующий оператор:

Нахождение суммы положительных элементов в массиве (унарный оператор !).

Программа должна выполнить данные операции с несколькими объектами. Результаты всех операций вывести на экран.

Вариант 3. Перепишите следующие функции класса matr:

(insert) замените на оператор + (включение новой строки);

(delete) замените на оператор – (удаление элемента из массива).

Дополнительно переопределите следующие операторы:

Сложение матриц – оператор *.

вычитание матриц – оператор %.

Программа должна выполнить данные операции с несколькими объектами.

Результаты всех операций вывести на экран.

Вариант 4. Перепишите следующие функции класса expr:

(add) замените на оператор + (находит сумму значений двух выражений);

(mult) замените на оператор * (находит произведение значений двух выражений).

Дополнительно переопределите следующие операторы:

Вычитание значений двух выражений (оператор –);

Деление значений двух выражений (оператор /).

Программа должна выполнить данные операции с несколькими объектами. Результаты всех операций вывести на экран.

Вариант 5. Перепишите следующие функции класса string :

(add) замените на оператор + (склеивание двух строк);

(minus) замените на оператор – (удаление из первой строки всех символов второй строки);

(mult) замените на оператор * (формирование строки, состоящей из символов, присутствующих одновременно как в первой строке, так и во второй).

Дополнительно переопределите следующие операторы:

Сравнивание двух строк (оператор ==).

Программа должна выполнить данные операции с несколькими объектами. Результаты всех операций вывести на экран.

Вариант 6. Перепишите следующие функции класса mas_string:

Новый класс дополнительно должен реализовать следующие операции:

(add) замените на оператор + (склеивание двух массивов строк);

(minus) замените на оператор – (удаление из первой массива всех слов второго массива);

(mult) замените на оператор * (формирование массива, состоящего из слов, присутствующих одновременно как в первой строке так и во второй).

Дополнительно переопределите следующие операторы:

Сравнивание двух массивов строк (оператор ==).

Программа должна выполнить данные операции с несколькими объектами. Результаты всех операций вывести на экран.

Вариант 7. Перепишите следующие функции класса mas_random:

(add) замените на оператор + (сложение двух векторов (по правилу zi=xi+yi));

(minus) замените на оператор – (вычитание двух векторов (по правилу

zi=xi-yi));

(mult) замените на оператор * (умножение скаляра на вектор (по правилу zi=v*yi)).

Дополнительно переопределите следующий оператор:

Возведение вектора в квадрат (оператор ^).

Программа должна выполнить данные операции с несколькими объектами. Результаты всех операций вывести на экран.

Вариант 8. Перепишите следующие функции класса mass_random:

Новый класс дополнительно должен реализовать следующие операции:

(add) замените на оператор + (сложение двух матриц);

(minus) замените на оператор – (вычитание двух матриц);

(mult) замените на оператор * (умножение скаляра на матрицу).

Дополнительно переопределите следующий оператор:

Возведение матрицы в заданную степень (оператор ^).

Программа должна выполнить данные операции с несколькими объектами. Результаты всех операций вывести на экран.

Вариант 9. Перепишите следующие функции класса kod_chet:

(add) замените на оператор + (сложение двух векторов);

(rang) замените на унарный оператор! (вычисление ранга вектора (c=a[1]+a[2]+…+a[n]));

(mult) замените на оператор * (умножение векторов(c[i]=a[i]*b[i])).

Дополнительно переопределите следующий оператор:

Получение инверсного кода (по правилу c[i]=1 – a[i]) (унарный оператор ^).

Программа должна выполнить данные операции с несколькими объектами. Результаты всех операций вывести на экран.

Вариант 10. Перепишите следующие функции класса dekod_nechet:

(add) замените на оператор + (сложение двух векторов);

(rang) замените на унарный оператор! (вычисление ранга вектора (c=a[1]+a[2]+…+a[n]));

(mult) замените на оператор * (скалярное умножение векторов(c=sum(a[i]*b[i]))).

Дополнительно переопределите следующий оператор:

Сравнение двух векторов (оператор ==).

Программа должна выполнить данные операции с несколькими объектами. Результаты всех операций вывести на экран.

Вариант 11. Перепишите следующие функции класса kod_chet :

(rang) замените на оператор ! (вычисление ранга вектора);

(mult) замените на бинарный оператор * (поразрядное умножение двух векторов);

(mult2) замените на бинарный оператор +(скалярное умножение векторов).

Дополнительно переопределите следующий оператор:

Удаление из одного вектора всех нулей (бинарный оператор –).

Программа должна выполнить данные операции с несколькими объектами. Результаты всех операций вывести на экран.

Вариант 12. Перепишите следующие функции класса dekodes_nechet :

(ort) замените на оператор ^ (проверка взаимной ортогональности двух векторов);

(mult1) замените на оператор * (поразрядное умножение векторов);

(mult2) замените на оператор % (скалярное умножение векторов).

Дополнительно переопределите следующий оператор:

Получении группы инверсных кодов (по правилу c[i]=1-a[i]) – унарный оператор ^).

Программа должна выполнить данные операции с несколькими объектами. Результаты всех операций вывести на экран.

Вариант 13. Перепишите следующие функции класса mass_random :

(add) замените на бинарный оператор+ (сложение двух матриц);

(minus) замените на бинарныйоператор – (вычитание двух матриц);

(mult) замените на оператор * (умножение матрицы на число).

Дополнительно переопределите следующий оператор:

Вычисление суммы четных элементов матрицы (оператор!).

Программа должна выполнить данные операции с несколькими объектами. Результаты всех операций вывести на экран.

Вариант 14. Перепишите следующие функции класса mass_random:

(poisk) замените на оператор == (поиск одинаковых элементов в двух массивах);

(minus) замените на оператор – (вычитание двух массивов);

(max) замените на унарный оператор ! (поиск максимального элемента в массиве).

Дополнительно переопределите следующий оператор:

Умножение двух массивов по правилу скалярного произведения (оператор*).

Программа должна выполнить данные операции с несколькими объектами. Результаты всех операций вывести на экран.

Вариант 15. Перепишите следующие функции класса mas_string:

(add) замените на оператор + (склеивание двух массивов строк);

(rshift) замените на оператор >> (циклический сдвиг вправо на заданное растояние массива);

(lshift) замените на оператор << (циклический сдвиг влево на заданное растояние массива).

Дополнительно переопределите следующий оператор:

Сравнение двух строк(оператор ==).

Программа должна выполнить данные операции с несколькими объектами. Результаты всех операций вывести на экран.

Вариант 16. Перепишите следующие функции класса mas_real:

(add) замените на оператор + (слияние двух упорядоченных массивов в один упорядоченный);

(delete) замените на оператор / (удаление элемента из массива);

(mult) замените на оператор * (перемножение двух массивов по правилу скалярного произведения).

Дополнительно переопределите следующий оператор:

Вычисление суммы отрицательных элементов массива (унарный оператор –).

Программа должна выполнить данные операции с несколькими объектами. Результаты всех операций вывести на экран.

Вариант 17. Перепишите следующие функции класса mas_random :

(minus) замените на оператор – (разность двух массивов);

(delete) замените на оператор / (удаление элемента из массива);

(mult) замените на оператор * (перемножение двух массивов по правилу скалярного произведения).

Дополнительно переопределите следующий оператор:

Сравнение двух массивов (оператор ==).

Программа должна выполнить данные операции с несколькими объектами. Результаты всех операций вывести на экран.

Вариант 18. Перепишите следующие функции класса num_16:

(minus) замените на оператор – (вычитание двух шестнадцатеричных чисел);

(add) замените на оператор + (сложение двух шестнадцатеричных чисел).

Дополнительно переопределите следующий оператор:

Перемножение двух шестнадцатеричных чисел (оператор *).

Программа должна выполнить данные операции с несколькими объектами. Результаты всех операций вывести на экран.

Вариант 19. Перепишите следующие функции класса num_8 :

(minus) замените на оператор – (вычитание двух восьмеричных чисел);

(add) замените на оператор + (сложение двух восьмеричных чисел).

Дополнительно переопределите следующий оператор:

Перемножение двух восьмеричных чисел (оператор *).

Программа должна выполнить данные операции с несколькими объектами. Результаты всех операций вывести на экран.

Вариант 20. Перепишите следующие функции класса mas_string:

(add) замените на оператор + (склеивание двух массивов строк);

(minus) замените на оператор – (удаление из первого массива всех слов второго массива);

(mult) замените на оператор + (вывод слов, присутствующих одновременно в обоих массивах строк).

Дополнительно переопределите следующий оператор:

Циклический сдвиг в строке на заданное количество слов (оператор !).

Программа должна выполнить данные операции с несколькими объектами. Результаты всех операций вывести на экран.

4 Наследование

Вариант 1. С помощью наследования создать класс mas2. Новый класс дополнительно хранит сумму и произведение нечетных чисел из исходного класса mas. Реализовать функцию добавления элемента в массив, соблюдая его упорядоченность.

Вариант 2. С помощью наследования создать класс mas_int2. Новый класс дополнительно хранит сумму и произведение четных чисел из исходного класса mas_int. Реализовать функцию добавления элемента в массив, соблюдая его упорядоченность.

Вариант 3. С помощью наследования создать класс matr2. Новый класс дополнительно хранит сумму и произведение строк матрицы из исходного класса matr. Реализовать функцию удаления строки, столбца из матрицы.

Вариант 4. С помощью наследования создать класс expr2. Новый класс дополнительно должен хранить количество каждой из операций (+ – * /). Реализовать функцию, определяющую операнды, используемые в выражении.

Вариант 5. С помощью наследования создать класс string2. Новый класс дополнительно должен хранить количество гласных и согласных букв, а также реализовать функцию, определяющую букву, имеющую максимальное количество повторений.

Вариант 6. С помощью наследования создать класс mas_string2. Новый класс дополнительно должен хранить количество гласных и согласных букв в «длинных» словах, а также реализовать функцию, определяющую букву, имеющую максимальное количество повторений в «длинных» словах.

Вариант 7. С помощью наследования создать класс mas_random2. Новый класс дополнительно хранит сумму и произведение чисел из вектора исходного класса. Реализовать функцию подсчета количеств четных и нечетных элементов вектора.

Вариант 8. С помощью наследования создать класс mass_random2. Новый класс дополнительно хранит сумму и произведение строк матрицы из исходного класса. Реализовать функцию удаления строки, столбца из матрицы.

Вариант 9. С помощью наследования создать класс kod_chet2. Новый класс должен хранить количество единиц и нолей в исходном векторе. Дополнительно реализовать функцию перевода двоичного вектора в десятичное число.

Вариант 10. С помощью наследования создать класс dekod_nechet2. Новый класс должен хранить количество единиц и нолей в исходном векторе. Дополнительно реализовать функцию перевода двоичного вектора в восьмеричное число.

Вариант 11. С помощью наследования создать класс kod_chet2. Новый класс должен хранить количество единиц и нолей в исходном векторе. Дополнительно реализовать функцию перевода двоичного вектора в шестнадцатеричное число.

Вариант 12. С помощью наследования создать класс dekods_nechet2. Новый класс должен хранить количество единиц и нолей в исходном векторе.

Дополнительно реализовать функцию перевода двоичного вектора в десятичное число.

Вариант 13. С помощью наследования создать класс mass_random2. Новый класс дополнительно хранит сумму и произведение столбцов матрицы из исходного класса. Реализовать функцию удаления строки, столбца из матрицы.

Вариант 14. С помощью наследования создать класс mass_random2. Новый класс дополнительно хранит сумму и произведение четных и, отдельно, нечетных элементов массива из исходного класса. Реализовать функцию удаления элемента из массива.

Вариант 15. С помощью наследования создать класс mas_string2. Новый класс должен хранить количество гласных и согласных букв в строке. Дополнительно реализовать функцию определения символа, который повторяется в строке максимальное число раз.

Вариант 16. С помощью наследования создать класс mas_real2. Новый класс должен хранить массив из целых частей исходного массива чисел и массив из дробных частей исходного массива. Дополнительно реализовать функцию поиска максимального и минимального элемента в массиве и их положения .

Вариант 17. С помощью наследования создать класс mas_random2. Новый класс должен хранить количество четных и нечетных чисел в исходном массиве. Дополнительно реализовать функцию поиска максимального и минимального элемента в массиве и их положения .

Вариант 18. С помощью наследования создать класс num_16_2. Новый класс должен хранить количество цифр $(0-9)$ и количество букв $(a-f)$ в записи исходного числа. Дополнительно реализовать функцию перевода шестнадцатеричного числа в двоичное и восьмеричное.

Вариант 19. С помощью наследования создать класс num_8_2. Новый класс должен хранить количество четных цифр и количество нечетных цифр в записи исходного числа. Дополнительно реализовать функцию перевода восьмеричного числа в двоичное и шестнадцатеричное.

Вариант 20. С помощью наследования создать класс mas_string2. Новый класс должен хранить количество гласных и согласных букв в строке. Дополнительно реализовать функцию определения символов, которые повторяются в строке четное число раз.

5 Виртуальные функции

В программу лабораторной работы 4, где описаны два класса (базовый и производный) добавить виртуальную функцию и ее вызовы для созданных объектов.

Вариант 1. Функция базового класса должна выводить упорядоченный массив и его размер. Функция производного класса должна выводить на экран упорядоченный массив, а также сумму и произведение его нечетных элементов.

Вариант 2. Функция базового класса должна выводить упорядоченный массив и его размер. Функция производного класса должна выводить на экран упорядоченный массив, а также сумму и произведение его четных элементов.

Вариант 3. Функция базового класса должна выводить исходную матрицу и ее размер. Функция производного класса должна выводить на экран исходную матрицу, а также сумму и произведение ее строк.

Вариант 4. Функция базового класса должна выводить исходное арифметическое выражение, количество операций и операндов. Функция производного класса должна выводить на экран исходное выражение, а также используемые в нем операции.

Вариант 5. Функция базового класса должна выводить отфильтрованную строку и количество четных чисел в ней. Функция производного класса должна выводить на экран отфильтрованную строку, а также количество повторений букв в ней.

Вариант 6. Функция базового класса должна выводить исходную строку и количество «длинных» слов в ней. Функция производного класса должна выводить на экран исходную строку, а также количество гласных и согласных букв в каждом «длинном» слове.

Вариант 7. Функция базового класса должна выводить массив и количество чисел, сумма разрядов у которых больше 10. Функция производного класса должна выводить на экран массив, а также сумму и произведение его элементов.

Вариант 8. Функция базового класса должна выводить матрицу и количество нулевых значений. Функция производного класса должна выводить на экран матрицу, а также сумму и произведение ее строк.

Вариант 9. Функция базового класса должна выводить двоичный вектор с контрольным разрядом и его ранг. Функция производного класса должна

выводить на экран двоичный вектор с контрольным разрядом, а также его значение в десятичной системе счисления.

Вариант 10. Функция базового класса должна выводить двоичный вектор с контрольным разрядом и его ранг. Функция производного класса должна выводить на экран двоичный вектор с контрольным разрядом, а также его значение в восьмеричной системе счисления.

Вариант 11. Функция базового класса должна выводить двоичный вектор с контрольным разрядом и его ранг. Функция производного класса должна выводить на экран двоичный вектор с контрольным разрядом, а также его значение в шестнадцатеричной системе счисления.

Вариант 12. Функция базового класса должна выводить двоичный вектор с контрольным разрядом и его ранг. Функция производного класса должна выводить на экран двоичный вектор с контрольным разрядом, а также его значение в десятичной системе счисления.

Вариант 13. Функция базового класса должна выводить фильтрованную матрицу и его размер. Функция производного класса должна выводить на экран матрицу, а также сумму и произведение ее столбцов.

Вариант 14. Функция базового класса должна выводить фильтрованный массив и количество чисел, сумма разрядов у которых меньше 8. Функция производного класса должна выводить на экран массив, а также сумму и произведение четных и, отдельно, нечетных элементов.

Вариант 15. Функция базового класса должна выводить отсортированную строку и количество «коротких» слов в ней. Функция производного класса должна выводить на экран отсортированную строку, а также количество гласных и согласных букв в ней.

Вариант 16. Функция базового класса должна выводить упорядоченный массив и его размер. Функция производного класса должна выводить на экран упорядоченный массив, а также значения максимального и минимального элемента в массиве и их положения.

Вариант 17. Функция базового класса должна выводить фильтрованный массив и количество чисел, сумма разрядов у которых больше 5. Функция производного класса должна выводить на экран массив, а также значения максимального и минимального элемента в массиве и их положения.

Вариант 18. Функция базового класса должна выводить исходное шестнадцатеричное число и его значение в десятичной системе счисления.

Функция производного класса должна выводить на экран исходное число, а также его значение в двоичной и восьмеричной системах счисления.

Вариант 19. Функция базового класса должна выводить исходное восьмеричное число и его значение в десятичной системе счисления. Функция производного класса должна выводить на экран исходное число, а также его значение в двоичной и шестнадцатеричной системах счисления.

Вариант 20. Функция базового класса должна выводить отсортированную строку и количество слов в ней. Функция производного класса должна выводить на экран отсортированную строку, а также количество символов, которые повторяются в строке четное число раз.

6 Создание списка объектов

Необходимо организовать линейный связанный список из структуры, указанной в задании. Со списком необходимо выполнить следующие действия:

- Чтение из файла и размещение в свободной памяти;
- Просмотр списка;
- Включение новой компоненты в заданное место списка. Варианты указания места включения:

a. В начало списка;

b. В конец списка;

c. Место указывается номером, который должна иметь компонента после включения.

- Удаление заданной компоненты. Варианты указания удаляемой компоненты;

a. Первая компонента;

b. Вторая компонента;

c. Задается номер удаляемой компоненты;

- Удаление всего списка.

Выполнение операции над списком должно осуществляться в интерактивном режиме.

Замечание 1. В скобках задаются соответственно вариант включения и вариант исключения компоненты.

Замечание 2. В четных вариантах заданий необходимо описать операцию и удаления соответственно с помощью функции insert и delete, а в нечетных с помощью операторов + и –.

Вариант 1. Личность (имя, фамилия, адрес) (аа)

Вариант 2. Работник (номер бригады, фамилия, разряд) (аб)

Вариант 3. Студент (номер группы, фамилия, размер стипендии) (ав)

Вариант 4. Фирма (название фирмы, адрес, количество сотрудников) (ба)

Вариант 5. Книга (название, автор) (бб)

Вариант 6. Преподаватель (фамилия, квалификация, стаж) (бв)

Вариант 7. Файл (название файла, количество байт) (ва)

Вариант 8. Программист (фамилия, отдел, язык программирования) (вб)

Вариант 9. Фирма (название, назначение, уставный капитал) (вв)

Вариант 10. Трамвай (номер трамвая, номер маршрута) (аа)

Вариант 11. Музыкант (муз. инструмент, муз. жанр, муз. образование) (аб)

Вариант 12. Спорт (вид спорта, массовость) (ав)

Вариант 13. Конфликт (повод, причина, результат) (ба)

Вариант 14. Статья (автор, заглавие, журнал, номер журнала, год выпуска) (бб)

Вариант 15. Оружие (калибр, тип, фирма изготовитель) (бв)

Вариант 16. Компьютер (марка, параметры, название) (ва)

Вариант 17. Животное (род, место обитания) (вб)

Вариант 18. Болезнь (инкубационный период, основной симптом) (вв)

Вариант 19. Наука (предмет исследования, методы исследования) (аа)

Вариант 20. Авиакомпания (название авиакомпании, капитал) (аб)

Заключение

Данное пособие ориентировано на знакомство с основными понятиями технологии программирования. Рассмотрены этапы развития технологий и языков программирования. Приведены основные этапы жизненного цикла программного продукта. В данном пособии на примерах рассматриваются различные алгоритмы, методы и примеры написания программ, рассматриваются вопросы качества и стиля программирования, а также процесс отладки и тестирования.

Весь материал разбит на разделы, в которых кратко описаны основы объектно-ориентированного программирования на языке C++.

В пособии приведен лабораторный практикум по программированию на языке C++. Задания позволяют на практике применить знание теоретического курса и последовательно добавлять в свою программу элементы, реализующие новые темы. Учебное пособие ориентировано на студентов, начинающих изучать объектно-ориентированное программирование, но уже знакомых с языком программирования C и со структурным подходом к созданию программ.

Библиографический список

1. Алексеев, Е. Р. Программирование на Microsoft Visual C++ и Turbo C++ Explorer / Е. Р Алексеев. – Москва : НТ Пресс, 2007.– 191 с. – Москва : Изд. центр ЕАОИ, 2009. – 351 с.
2. Волкова, И. А. Основы объектно-ориентированного программирования. Язык программирования C++ : учебное пособие для студентов 2 курса / И. А. Волкова, А. В. Иванов, Л. Е. Карпов. – Москва : Издательский отдел факультета ВМК МГУ, 2011. – 112 с.
3. Иванова, Г. С. Технология программирования : учебник для вузов / Г. С. Иванова. – Москва : Изд-во МГТУ им. Н.Э. Баумана, 2002. – 320 с.
4. Иванова, Г. С. Программирование : учебник для вузов / Г. С. Иванова. – 3-е изд., стер. – Москва : Кнорус, 2014. – 426 с. – (Бакалавриат).
5. Крылов, Е. В. Техника разработки программ: В 2 кн. / Е. В. Крылов, В. А. Острейковский, Н. Г. Типикин. – Москва : Высшая школа, 2007. – (Информатика и вычислительная техника). – Кн. 1: Программирование на языке высокого уровня : учебник для вузов. – 375 с.

6. Лаврищева, Е. М. Технология программирования и программная инженерия : учебник для вузов / Е. М. Лаврищева. – Москва : Издательство Юрайт, 2017. – 432 с. – (Серия : Бакалавр. Академический курс).
7. Лафоре, Р. Объектно-ориентированное программирование в С++: [перевод с английского] / Р. Лафоре. – 4-е изд. – Санкт-Петербург [и др.]: Питер, 2012. – 923 с.– (Классика computer science).
8. Объектно-ориентированное программирование : учебник / Г. С. Иванова, Т. Н. Ничушкина; под общ. ред. Г. С. Ивановой. – Москва : Изд-во МГТУ им. Н. Э. Баумана, 2014. – 455, [1] с.
9. Павловская, Т. А. С/С++. Программирование на языке высокого уровня : учебник для вузов / Т. А. Павловская. – Санкт-Петербург [и др.] : Питер, 2007. – 460 с.
10. Подбельский, В. В. Программирование на языке Си : учебное пособие для вузов / В. В. Подбельский , С. С. Фомин. – 2-е изд., доп. – Москва : Финансы и статистика, 2005. – 600 с.
11. Родионова, Т. Е. Программирование на языке СИ : учебное пособие / Т. Е. Родионова. – Ульяновск : УлГТУ, 2013. – 133 с.: Доступен также в Интернете http://venec.ulstu.ru/lib/disk/2015/7.pdf
12. Страуструп, Б. Язык программирования С++ / Б. Страуструп. – СПб.; Москва : «Невский проспект» : «Издательство БИНОМ», 2011. – 1136 с.
13. Иванова, Г. С. Технология программирования : учебник / Г. С. Иванова. – Москва : КНОРУС, 2011. – 336 с.
14. Технологии и методы программирования: учебное пособие / Л. А. Артюшина, А. А. Воронина ; Владим. гос. ун-т им. А. Г. им Н. Г. Столетовых. – Владимир: Изд-во ВлГУ, 2014. – 96 с.
15. Технологии программирования : учебное пособие / Е. А. Мирошниченко. – 2-е изд., испр. и доп. – Томск : Изд-во Томского политехнического университета, 2008. – 124 с.

16. Технология программирования : учебное пособие /Ю. Ю.Громов, О. Г. Иванова, М. П. Беляев, Ю. В. Минин. – Тамбов : Изд-во ФГБОУВПО «ТГТУ», 2013. – 172 с.
17. Тюгашев, А. А. Языки программирования : учебное пособие для вузов / А. А. Тюгашев. – Санкт-Петербург [и др.]: Питер, 2014. – 333 с. (Стандарт третьего поколения).
18. Фаронов, В. В. Программирование на языке C# : учебный курс / В. Фаронов. – Санкт-Петербург [и др.]: Питер, 2007. – 239 с.
19. Шилдт, Г.Полный справочник по C++ / Г. Шилдт. – 4-е изд.– Москва : «Вильямс», 2006. – 791 с.
20. Язык программирования C++ / А. Л. Фридман. – (2-е изд.) – Москва : НОУ "Интуит", 2016. – 218 с.

Интернет-ресурсы

1. Все для программистов. URL: http://www.codenet.ru/cat/Languages/ (дата обращения: 29.03.18)
2. Язык Си в примерах. URL: http://www.wikibooks.org/wiki/Язык_Си_в_примерах (дата обращения: 29.03.18)
3. Язык Си в примерах URL: http://www kpolyakov.spb.ru (дата обращения: 29.03.18)
4. Технологии программирования URL:https://studfiles.net/preview/2714199/ (дата обращения: 29.03.18)
5. Основы информационных технологий URL: https://www.intuit.ru/studies/courses/3481/723/info (дата обращения: 29.03.18)
6. Введение в программные системы и их разработку URL:https://www.intuit.ru/studies/courses/3632/874/lecture/14289 (дата обращения: 29.03.18)
7. Технологии программирования URL:http://bourabai.kz/alg/technology.htm (дата обращения: 29.03.18)

Учебное издание

РОДИОНОВА Татьяна Евгеньевна

Технологии программирования

Учебное пособие

Редактор Н.А.Евдокимова

ЛР №020640 от 22.10.97

Подписано в печать 09.04.2018. Формат 60×84/16.

Усл. печ. л. 6,75. Тираж 80 экз. Заказ 376. ЭИ № 1074.

Ульяновский государственный технический университет

432027, Ульяновск, ул. Сев. Венец, 32.

ИПК «Венец» УлГТУ, 432027, Ульяновск, ул. Сев. Венец, 32