Учреждение образования Федерации профсоюзов Беларуси
«Международный университет «МИТСО»

Факультет экономический
Кафедра информационных технологий

СОГЛАСОВАНО	СОГЛАСОВАНО
Заведующий кафедрой	Декан факультета
_____О.Б.Хорошко	_____А.В.Ковтунов
30.10 2025 г.	16.12 2025 г.

ЭЛЕКТРОННЫЙ УЧЕБНО-МЕТОДИЧЕСКИЙ КОМПЛЕКС ПО УЧЕБНОЙ ДИСЦИПЛИНЕ

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОЕКТИРОВАНИЕ И ПРОГРАММИРОВАНИЕ

для специальности 6-05-0611-01 Информационные системы и технологии

Составители: Пекарь А.С., преподаватель кафедры информационных технологий учреждения образования Федерации профсоюзов Беларуси «Международный университет «МИТСО»
Пархимович А.В., преподаватель кафедры информационных технологий учреждения образования Федерации профсоюзов Беларуси «Международный университет «МИТСО»

Рассмотрено и рекомендовано к утверждению на заседании кафедры информационных технологий учреждения образования Федерации профсоюзов Беларуси «Международный университет «МИТСО»
30.10.2025 г., протокол № 3

Утверждено на заседании научно-методического совета учреждения образования Федерации профсоюзов Беларуси «Международный университет «МИТСО»
16.12.2025 г., протокол № 2

РЕЦЕНЗЕНТЫ:
Кафедра микропроцессорных систем и сетей Института информационных технологий УО «Белорусский государственный университет информатики и радиоэлектроники»;
Гришко Н.И., доцент кафедры экономики и менеджмента учреждения образования федерации профсоюзов Беларуси «Международный университет «МИТСО», кандидат технических наук, доцент

Регистрационный № УД-013-26/7

Регистрационное свидетельство № 1202645889 от 03.02.2026

АКТУАЛИЗИРОВАН
Рассмотрено на заседании кафедры информационных технологий учреждения образования Федерации профсоюзов Беларуси «Международный университет «МИТСО»
_____2025 г., протокол № ___

Утверждено на заседании научно-методического совета учреждения образования Федерации профсоюзов Беларуси «Международный университет «МИТСО»
_____2025 г., протокол № ___

ОГЛАВЛЕНИЕ

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

Электронный учебно-методический комплекс (далее – ЭУМК) подготовлен в соответствии с учебной программой Учреждения образования Федерации профсоюзов Беларуси «Международный университет «МИТСО» по дисциплине «Объектно-ориентированное проектирование и программирование» для направления специальности 6-05-0611-01 Информационные системы и технологии.

Цель ЭУМК – обеспечение максимальной доступности и конкретности изучаемого материала по дисциплине. Практическая значимость состоит в том, что данный ЭУМК позволяет конкретизировать и спланировать учебную деятельность студентов, разнообразить формы усвоения достаточно сложного материала.

Основными структурными элементами ЭУМК являются:

• учебная программа по дисциплине;

• теоретический раздел, включающий конспект лекций для теоретического изучения учебной дисциплины;

• практический раздел, содержащий учебно-методические материалы для поведения лабораторных работ;

• раздел контроля знаний, в котором представлены контрольные вопросы по дисциплине, вопросы для самостоятельного изучения по всем предусмотренным учебной программой темам дисциплины;

• вспомогательный раздел, в котором приведены рекомендации по освоению дисциплины.

Рекомендации по организации работы с ЭУМК. Для лучшего усвоения получаемых знаний слушателям рекомендуется сначала ознакомиться с учебной программой по дисциплине «Объектно-ориентированное проектирование и программирование». Усвоение материала по каждой из тем рекомендуется начинать с изучения соответствующей лекции («Теоретический раздел»), после чего следует ответить на вопросы для самоконтроля и выполнить задания лабораторных работ («Практический раздел»). Проверка знаний проводится демонстрацией созданных программ, ответов на вопросы из перечней вопросов для опросов и вопросов к экзамену «Раздел контроля знаний». Студентам рекомендуется использовать материалы ЭУМК для осуществления активной самостоятельной работы при изучении дисциплины, подготовки к аудиторным занятиям и экзамену.

Учреждение образования Федерации профсоюзов Беларуси
«Международный университет «МИТСО»

УТВЕРЖДАЮ

Проректор по учебной работе
Учреждения образования
Федерации профсоюзов Беларуси
«Международный университет «МИТСО»
_____Ю.Л.Шевцов
_____2023

Регистрационный № УД-172/01 – 23/уч

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОЕКТИРОВАНИЕ И ПРОГРАММИРОВАНИЕ

Учебная программа учреждения высшего образования по учебной дисциплине для специальности

6-05-0611-01 Информационные системы и технологии

2023 г.

Учебная программа составлена в соответствии с Образовательным стандартом высшего образования ОСВО 6-05-0611-01-2023 (общее высшее образование), для специальности 6-05-0611-01 «Информационные системы и технологии», типовой учебной программой «Объектно-ориентированное проектирование и программирование», утвержденной 03.06.2022, регистрационный № ТД-I.1579 и учебным планом учреждения образования Федерации профсоюзов Беларуси «Международный университет «МИТСО» для специальности 6-05-0611-01 «Информационные системы и технологии»

СОСТАВИТЕЛЬ:
А.П. Жалов, старший преподаватель кафедры информационных технологий учреждения образования Федерации профсоюзов Беларуси «Международный университет «МИТСО»

РЕКОМЕНДОВАНА К УТВЕРЖДЕНИЮ:
Кафедрой информационных технологий учреждения образования Федерации профсоюзов Беларуси «Международный университет «МИТСО»
(протокол № 11 от 28.06.2023);

Научно-методическим советом учреждения образования Федерации профсоюзов Беларуси «Международный университет «МИТСО»
(протокол № 8 от 14.07.2023)

Нормоконтроль
ведущий специалист
УМУ
Т.Ф. Лагунович

Зав. библиотекой
О.О. Бабарикина

I. ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

Актуальность изучения учебной дисциплины «Объектно-ориентированное проектирование и программирование» обоснована необходимостью подготовки специалистов, владеющих фундаментальными знаниями и практическими навыками в области объектно-ориентированного анализа, программирования и элементов проектирования при решении практических задач.

В рамках образовательного процесса по учебной дисциплине «Объектно-ориентированное проектирование и программирование» студент должен приобрести не только теоретические и практические знания, умения и навыки по специальности, но и развить свой ценностно-личностный, духовный потенциал, сформировать качества патриота и гражданина, готового к активному участию в экономической, производственной, социально-культурной и общественной жизни страны

Цель учебной дисциплины: теоретическая и практическая подготовка, обеспечивающая получение базовых знаний по основам объектно-ориентированного проектирования и программирования.

Задачи учебной дисциплины:

освоение возможностей, предоставляемых современными компьютерными технологиями;

изучение принципов проектирования, создания, масштабирования объектно-ориентированных приложений;

овладение методами, подходам, принципами создания объектно-ориентированных приложений;

приобретение знаний и навыков проектирования и создания объектно-ориентированных приложений;

формирование навыков программирования с использованием объектно-ориентированных подходов;

приобретение навыков работы в интегрированной среде разработки приложений.

Базовыми учебными дисциплинами по курсу «Объектно-ориентированное проектирование и программирование» являются «Иностранный язык», «Дискретная математика», «Технологии разработки программного обеспечения», «Основы алгоритмизации и программирования». В свою очередь, учебная дисциплина «Объектно-ориентированное проектирование и программирование» является базой для таких учебных дисциплин, как «Скриптовые языки программирования», «Программирование сетевых приложений», «Программирование мобильных информационных систем».

В результате изучения учебной дисциплины «Объектно-ориентированное проектирование и программирование» формируются следующие **компетенции:**

универсальные:

владеть основами исследовательской деятельности, осуществлять поиск, анализ и синтез информации;

быть способным к саморазвитию и совершенствованию в профессиональной деятельности;

проявлять инициативу и адаптироваться к изменениям в

профессиональной деятельности;

базовая профессиональная:

применять фундаментальные методы и свойства объектно-ориентированного проектирования и программирования для разработки проектных и программных решений задач в рамках объектно-ориентированной парадигмы.

В результате изучения учебной дисциплины обучающийся должен:

знать:

базовые понятия и синтаксис объектно-ориентированного языка программирования, технологию объектно-ориентированного проектирования и приемы разработки программ;

методы создания и использования основных объектов и конструкций объектно-ориентированного языка программирования;

технологию создания, организации и использования иерархии классов, предопределенных классов и типов данных, методы ограничения доступа и обработки исключительных ситуаций;

методы параметризации классов и их использование для решения практических задач;

методы применения шаблонов и контейнерных абстракций;

методы работы с потоками ввода/вывода и разработки интерактивных приложений;

уметь:

определять абстракции, модули, строить иерархию классов для реализации программ;

использовать принципы типизации, инкапсуляции, наследования, полиморфизма для разработки программных продуктов;

использовать возможности стандартных библиотек объектно-ориентированного языка программирования;

использовать механизм исключений для создания устойчивых приложений; создавать собственные и использовать предоставляемые стандартные библиотеки динамических структур данных;

использовать технологию объектно-ориентированного проектирования для разработки сложных и масштабируемых программ и систем;

владеть:

методами, инструментальными средствами и системами разработки объектно-ориентированных программ;

техникой создания объектно-ориентированных программных компонент и организацией их взаимодействия в программных проектах.

Распределение аудиторных часов по видам занятий и семестрам.
Виды и формы аттестации

Семестр	Количество академических часов							Форма текущей аттестации
	Всего	Аудит.	Из них					
			Лекции	Лабор. занятия	Практ. занятия	Семи-нары	УСР[1]	
Очная (дневная) форма получения высшего образования								
2	120	68	24	32			12	зач.
3	110	52	18	22			12	экз.
Всего	230	120	42	54			24	

[1] Управляемая самостоятельная работа

II. СОДЕРЖАНИЕ УЧЕБНОГО МАТЕРИАЛА

Раздел 1. Концепция и особенности объектно-ориентированного проектирования и программирования

Тема 1. Концептуальные основы объектно-ориентированного проектирования. Сравнение принципов объектно-ориентированного проектирования с другими парадигмами

Предмет учебной дисциплины и ее содержание. Связь учебной дисциплины с другими дисциплинами учебного плана. Парадигмы программирования и проектирования, их особенности. Основные направления в программировании и проектировании программных продуктов. Возникновение объектно-ориентированного программирования и особенности использования в нем принципов объектно-ориентированного проектирования. Базовые принципы объектно-ориентированного программирования.

Тема 2. Фундаментальные методы, подходы, свойства объектной модели, ее преимущества, недостатки, особенности использования

Основные положения объектной модели. Ее составные элементы, свойства, преимущества, недостатки. Абстрагирование. Модульность. Иерархия. Типизация. Взаимосвязь основных элементов объектно-ориентированной парадигмы.

Раздел 2. Базовые абстракции объектно-ориентированного проектирования и программирования

Тема 3. Базовые конструкции объектно-ориентированных программ. Абстрагирование как один из основных принципов в объектно-ориентированном проектировании и программировании. Особенности использования абстракции для выделения основных элементов проектируемой системы

Классы и объекты в объектно-ориентированном проектировании и программировании. Компоненты класса: поля и методы. Инициализация и разрушение объектов класса. Использование конструкторов и деструкторов класса. Конструктор по умолчанию, конструктор копирования. Перегрузка и переопределение методов класса.

Тема 4. Инкапсуляция как один из основных принципов в объектно-ориентированном проектировании и программировании. Методы и принципы реализации инкапсуляции и организации корректного доступа к элементам объекта

Атрибуты доступа к компонентам класса. Область действия класса и доступ к компонентам класса. Управление доступом к компонентам класса.

Тема 5. Структурные элементы класса, методы взаимодействия объектов классов. Особенности создания корректных связей между классами

Организация внешнего доступа к локальным компонентам класса. Понятие интерфейса в объектно-ориентированном проектировании и программировании. Дружественные методы как способ доступа к содержимому класса. Статические и константные компоненты класса. Особенности использования статических полей и методов класса. Сравнение использования статических компонент класса и статических переменных (локальных и глобальных). Вложенные классы, особенности организации доступа к ним. Перегрузка операторов и методов класса. Преобразование типов данных (явное и неявное). Использование указателей и ссылок на объекты. Операторы динамического выделения и освобождения памяти при работе со встроенными и пользовательскими типами данных. Организация ввода/вывода данных. Статические и динамические массивы объектов пользовательских типов данных.

Раздел 3. Методы и механизмы разработки объектно-ориентированных программ

Тема 6. Наследование как один из основных принципов в объектно-ориентированном проектировании и программировании. Механизмы наследования типов и определения собственных типов данных. Принципы и подходы при повторном использования кода

Базовые и производные классы как основа повторного использования кода. Основные правила и принципы построения базовых и производных классов. Атрибуты доступа при наследовании. Работа конструкторов и деструкторов при наследовании. Связь композиции, агрегации, наследования (обобщения) при проектировании программных продуктов. Переопределение методов базового класса в производном. Простое и множественное наследование. Особенности и проблемы, возникающие при реализации множественного наследования в объектно-ориентированных языках программирования.

Тема 7. Полиморфизм как один из основных принципов в объектно-ориентированном проектировании и программировании. Основные проявления, механизмы, способы реализации полиморфизма

Понятие раннего и позднего связывания. Использование виртуального механизма вызова методов при реализации принципа полиморфизма. Перегрузка операторов и методов как полиморфизм этапа компиляции. Переопределение методов как полиморфизм этапа выполнения программы. Виртуальные методы класса и механизм их использования. Абстрактные классы: их назначение, свойства, необходимость создания при проектировании объектно-ориентированной системы. Возможные пути решения неоднозначности при множественном наследовании.

Тема 8. Использование параметризированных классов в объектно-ориентированном проектировании и программировании. Особенности использования обобщенного проектирования и программирования в объектно-ориентированном

Параметризированные классы и методы: их свойства, особенности использования. Совместное использование параметризации и принципов наследования. Особенности использования параметров типов в объектно-ориентированном языке программирования. Организация внешнего доступа к компонентам параметризированных классов. Параметризированные классы и статические элементы класса. Создание специализированной версии параметризированного класса. Задание значений параметров класса по умолчанию.

Тема 9. Управление совместным использованием ресурсов. Создание собственных механизмов и использование встроенных компонент для реализации управления ресурсами

Реализация концепции RAII как принцип управления ресурсами. Особенности создания и использования умных указателей и механизма транзакций при управлении используемыми ресурсами.

Тема 10. Особенности возникновения и обработки исключительных ситуаций

Основы возникновения, создания, обработки исключительных ситуаций. Альтернативные способы обработки исключительных ситуаций. Генерация исключений как способ описания возникших исключительных ситуаций. Перехватывание исключений. Повторная генерация исключения. Обработка неожидаемых типов исключительных ситуаций. Генерация исключений в конструкторах, особенности создания объектов при возникновении исключительных ситуаций. Взаимосвязь возникновения исключений и иерархии классов. Спецификация исключений. Классы исключений стандартной библиотеки. Создание собственных типов исключительных ситуаций.

Тема 11. Использование потоков ввода/вывода как основа создания интерактивных программных средств. Объектно-ориентированное проектирование в библиотеках, реализующих ввод/вывод данных

Понятие потока ввода/вывода данных. Особенности создания и закрытия потоков ввода/вывода. Организация ввода данных из потока и вывода данных в поток. Особенности перегрузки операторов при использовании потоков ввода/вывода. Контроль состояния потока, установка битов ошибок, исправление ошибок, возникающих при вводе/выводе данных. Неформатированный ввод/вывод данных. Стандартные и определяемые пользователем манипуляторы потоков как способ управления вводом/выводом данных.

Тема 12. Использование потоков файлового ввода/вывода как основа создания программных средств с возможностью долговременного хранения данных. Объектно-ориентированное проектирование в библиотеках, реализующих ввод/вывод данных в/из файлы(ов)

Файловые потоки ввода/вывода данных. Общие свойства потоков ввода/вывода данных. Режимы открытия файловых потоков. Реализация последовательного и произвольного доступа к содержимому файла. Организация ввода/вывода данных переменных примитивных типов и объектов классов.

Тема 13. Контейнерные типы данных как возможность делигирования ответственности выделения динамической памяти. Особенности применения стандартных библиотек классов коллекций

Введение в стандартную библиотеку шаблонов (классов коллекций), основные понятия, концепции. Классы контейнеры и итераторы. Их взаимосвязь, особенности использования. Типы контейнерных классов, адаптеры контейнеров. Алгоритмы библиотеки классов: их типы, особенности использования с контейнерными классами.

Тема 14. Использование паттернов проектирования при разработке объектно-ориентированных приложений. Особенности и основные принципы применения объектно-ориентированного проектирования при разработке прикладных программ

Особенности использования паттернов проектирования при проектировании и программировании объектно-ориентированных программ. Основные виды паттернов проектирования, особенности использования, решаемые задачи. Взаимосвязь паттернов проектирования и принципов объектно-ориентированного проектирования и программирования.

III. ТРЕБОВАНИЯ К КУРСОВОМУ ПРОЕКТУ

Цель курсового проекта: освоение практических навыков проектирования и создания объектно-ориентированных приложений, разработки алгоритмов, их практической реализации в виде законченных, отлаженных и протестированных программных продуктов.

Курсовой проект представляет собой решение по проектированию и программированию прикладного объектно-ориентированного программного обеспечения.

Курсовой проект выполняется индивидуально. В проекте студент должен продемонстрировать умение применять все знания, которые были получены в ходе лекционных и лабораторных занятий.

Организация выполнения курсового проекта (курсовой работы) осуществляется в соответствии с Методическими рекомендациями по подготовке, оформлению и защите научных работ студентов, утвержденными Приказом ректора Университета от 06.04.2023 № 124 по порядку организации курсового проекта (курсовой работы) в Университете.

По согласованию с преподавателем студент может выбрать тему курсового проекта, отвечающую вышеприведенным цели и требованиям.

IV. УЧЕБНО-МЕТОДИЧЕСКАЯ КАРТА ДИСЦИПЛИНЫ

Очная (дневная) форма получения высшего образования

Номер темы	Название темы	Количество аудиторных часов				Количество часов УСР	Форма контроля знаний
		Лекции	Практические занятия	Семинарские занятия	Лабораторные занятия		
1	2	3	4	5	6	7	8
1.	**Раздел 1. Концепция и особенности объектно-ориентированного проектирования и программирования**	**4**				**4**	
1.1	Тема 1. Концептуальные основы объектно-ориентированного проектирования. Сравнение принципов объектно-ориентированного проектирования с другими парадигмами	2				2	УО, РПЗ, Р
1.2	Тема 2. Фундаментальные методы, подходы, свойства объектной модели, ее преимущества, недостатки, особенности использования	2				2	УО, РПЗ, Р
2.	**Раздел 2. Базовые абстракции объектно-ориентированного проектирования и программирования**	**12**			**18**	**4**	
2.1	Тема 3. Базовые конструкции объектно-ориентированных программ. Абстрагирование как один из основных принципов в объектно-ориентированном проектировании и программировании. Особенности использования абстракции для выделения основных элементов проектируемой системы	4			6	2	УО, РПЗ, Р
2.2	Тема 4. Инкапсуляция как один из основных принципов в объектно-ориентированном проектировании и программировании. Методы и принципы реализации инкапсуляции и организации корректного доступа к элементам объекта	4			6	2	УО, РПЗ, Р
2.3	Тема 5. Структурные элементы класса, методы взаимодействия объектов классов. Особенности создания корректных связей между классами	4			6		УО, РПЗ
3.	**Раздел 3. Методы и механизмы разработки объектно-ориентированных программ**	**26**			**32**	**14**	
3.1	Тема 6. Наследование как один из основных принципов в объектно-ориентированном проектировании и программировании. Механизмы наследования типов и определения собственных типов данных. Принципы и подходы при повторном использования кода	4			6	2	УО, РПЗ
3.2	Тема 7. Полиморфизм как один из основных принципов в объектно-ориентированном проектировании и программировании. Основные проявления, механизмы, способы реализации полиморфизма	4			8	2	УО, РПЗ
	Итого за 2 семестр:	**24**			**32**	**12**	**зач.**
3.3	Тема 8. Использование параметризированных классов в объектно-ориентированном проектировании и программировании.	2			4	2	УО, РПЗ, Р

1	2	3	4	5	6	7	8
	Особенности использования обобщенного проектирования и программирования в объектно-ориентированном						
3.4	Тема 9. Управление совместным использованием ресурсов. Создание собственных механизмов и использование встроенных компонент для реализации управления ресурсами	2			4	2	УО, РПЗ, Р
3.5	Тема 10. Особенности возникновения и обработки исключительных ситуаций	2			2	2	УО, РПЗ, Р
3.6	Тема 11. Использование потоков ввода/вывода как основа создания интерактивных программных средств. Объектно-ориентированное проектирование в библиотеках, реализующих ввод/вывод данных	2			2	2	УО, РПЗ, Р
3.7	Тема 12. Использование потоков файлового ввода/вывода как основа создания программных средств с возможностью долговременного хранения данных. Объектно-ориентированное проектирование в библиотеках, реализующих ввод/вывод данных в/из файлы(ов)	2			2	2	УО, РПЗ, Р
3.8	Тема 13. Контейнерные типы данных как возможность делегирования ответственности выделения динамической памяти. Особенности применения стандартных библиотек классов коллекций	4			4		УО, РПЗ
3.9	Тема 14. Использование паттернов проектирования при разработке объектно-ориентированных приложений. Особенности и основные принципы применения объектно-ориентированного проектирования при разработке прикладных программ	4			4	2	УО, РПЗ, Р
	Итого за 3 семестр:	**18**			**22**	**12**	экз.
	Итого:	**42**			**54**	**24**	

V. ИНФОРМАЦИОННО-МЕТОДИЧЕСКАЯ ЧАСТЬ

ОСНОВНАЯ ЛИТЕРАТУРА

1. Шилдт, Г. C++: полное руководство / Герберт Шилдт. – М. : Вильямс, 2018. – 796 с.

2. Прата, С. Язык программирования C--+.Лекции и упражнения : пер. с англ. / Стивен Прата. – 6-е изд. – М. [и др.] : Вильямс, 2018. – 1244 с.

3. Страуструп, Б. Программирование. Принципы и практика с использованием C++ : пер. с англ. / Б. Страуструп. – 2-е изд. – М. : ООО «И. Д. Вильямс», 2018. – 1328 с.

4. Маклафлин, Б. Объектно-ориентированный анализ и проектирование : пер. с англ. / Б. Маклафлин, Г. Поллайс, Д. Уэст. – СПб. [и др.] : Питер : Питер Пресс, 2018. – 601 с.

5. Эккель, Б. Философия Java : пер. с англ. / Брюс Эккель. – 4-е полное изд. – СПб. [и др.] : Питер : Питер Пресс, 2018. – 1165 с.

ДОПОЛНИТЕЛЬНАЯ ЛИТЕРАТУРА

6. ГОСТ 19.701-90 – Единая система программной документации – Схемы алгоритмов, программ, данных и систем – Условные обозначения и правила выполнения.

7. Шилдт, Г. Полный справочник по C++ / Г. Шилдт ; пер. с англ. – М. : Вильямс, 2016. – 800 с.

8. Страуструп, Б. Программирование. Принципы и практика с использованием C++ : пер. с англ. / Бьярне Страуструп. – 2-е изд. – М. [и др.] : Вильямс, 2018. – 1328 с.

СРЕДСТВА ДИАГНОСТИКИ РЕЗУЛЬТАТОВ ОБРАЗОВАТЕЛЬНОЙ ДЕЯТЕЛЬНОСТИ

Диагностика результатов образовательной деятельности обучающихся осуществляется в ходе проведения всех видов занятий, самостоятельной работы и текущей аттестации по учебной дисциплине.

Основными формами контроля знаний по учебной дисциплине являются:

устный опрос	УО;
реферат	Р;
решение практических задач	РПЗ;
курсовой проект	курс.пр.;
зачет	зач.;
экзамен	экз.

МЕТОДЫ (ТЕХНОЛОГИИ) ОБУЧЕНИЯ

Основные методы (технологии) обучения, отвечающие целям и задачам дисциплины:

элементы проблемного обучения (проблемное изложение, вариативное изложение, частично-поисковый метод), реализуемые на лекционных занятиях;

элементы учебно-исследовательской деятельности, творческий подход, реализуемые на лабораторных занятиях;

проектные технологии, используемые при проектировании конкретного объекта, реализуемые на практических занятиях.

ОРГАНИЗАЦИЯ САМОСТОЯТЕЛЬНОЙ РАБОТЫ СТУДЕНТОВ

При изучении учебной дисциплины рекомендуется использовать следующие формы самостоятельной работы:

подготовка к лекциям, практическим и лабораторным занятиям;

реферирование статей, отдельных разделов монографий;

изучение учебников и учебных пособий;

изучение системной документации и стандартов;

изучение тем и проблем, не выносимых на лекции, лабораторные и практические занятия;

выполнение контрольных работ;

написание тематических докладов, рефератов и эссе на проблемные темы;

аннотирование и конспектирование монографий или их отдельных глав статей;

участие студентов в составлении тестов;

выполнение исследовательских и творческих заданий;

составление библиографии и реферирование по заданной теме;

создание наглядных пособий по изучаемым темам;

решение проблемных задач, проведение расчетов и др.

ПРИМЕРНЫЙ ПЕРЕЧЕНЬ ТЕМ ЛАБОРАТОРНЫХ ЗАНЯТИЙ

1. Разработка набора классов и их объектов. Реализация корректных связей между классами.
2. Создание массива объектов класса.
3. Организация ввода/вывода.
4. Динамическое выделение памяти.
5. Дружественные функции и классы.
6. Перегрузка операций.
7. Наследование. Простое наследование.
8. Принцип полиморфизма. Виртуальные функции.
9. Абстрактные классы.
10. Множественное наследование. Виртуальное наследование.
11. Параметризация в объектно-ориентированном проектировании и программировании. Реализация шаблонов классов.
12. Практические приемы использования шаблонов типов и иерархии классов.
13. Генерация и обработка исключительных ситуаций.
14. Потоки ввода/вывода.
15. Организация работы с файлами.
16. Последовательные классы-контейнеры.
17. Адаптеры контейнеров.
18. Ассоциативные классы-контейнеры.
19. Классы-итераторы библиотеки Standard Template Library.
20. Особенности использования объектно-ориентированного проектирования и программирования и паттернов проектирования.

ПРИМЕРНЫЙ ПЕРЕЧЕНЬ ЗАДАНИЙ И КОНТРОЛЬНЫХ МЕРОПРИЯТИЙ УСР

№ темы	Тема УСР	Кол-во часов	Форма контроля
1 курс, 2 семестр (12 часов)			
1.1	Тема 1. Концептуальные основы объектно-ориентированного проектирования. Сравнение принципов объектно-ориентированного проектирования с другими парадигмами	2	Реферат, доклад, сообщение
1.2	Тема 2. Фундаментальные методы, подходы, свойства объектной модели, ее преимущества, недостатки, особенности использования	2	Реферат, доклад, сообщение
2.1	Тема 3. Базовые конструкции объектно-ориентированных программ. Абстрагирование как один из основных принципов в объектно-ориентированном проектировании и программировании. Особенности использования абстракции для выделения основных элементов проектируемой системы	2	Реферат, доклад, сообщение
2.2	Тема 4. Инкапсуляция как один из основных принципов в объектно-ориентированном проектировании и программировании. Методы и принципы реализации инкапсуляции и организации корректного доступа к элементам объекта	2	Реферат, доклад, сообщение
3.1	Тема 6. Наследование как один из основных принципов в объектно-ориентированном проектировании и программировании. Механизмы наследования типов и определения собственных типов данных. Принципы и подходы при повторном использования кода	2	Реферат, доклад, сообщение
3.2	Тема 7. Полиморфизм как один из основных принципов в объектно-ориентированном проектировании и программировании. Основные проявления, механизмы, способы реализации полиморфизма	2	Реферат, доклад, сообщение
2 курс, 3 семестр (12 часов)			
3.3	Тема 8. Использование параметризированных классов в объектно-ориентированном проектировании и программировании. Особенности использования обобщенного проектирования и программирования в объектно-ориентированном	2	Реферат, доклад, сообщение
3.4	Тема 9. Управление совместным использованием ресурсов. Создание собственных механизмов и использование встроенных компонент для реализации управления ресурсами	2	Реферат, доклад, сообщение
3.5	Тема 10. Особенности возникновения и обработки исключительных ситуаций	2	Реферат, доклад, сообщение
3.6	Тема 11. Использование потоков ввода/вывода как основа создания интерактивных программных средств. Объектно-ориентированное проектирование в библиотеках, реализующих ввод/вывод данных	2	Реферат, доклад, сообщение
3.7	Тема 12. Использование потоков файлового ввода/вывода как основа создания программных средств с возможностью долговременного хранения данных. Объектно-ориентированное проектирование в библиотеках, реализующих ввод/вывод данных в/из файлы(ов)	2	Реферат, доклад, сообщение
3.9	Тема 14. Использование паттернов проектирования при разработке объектно-ориентированных приложений. Особенности и основные принципы применения объектно-ориентированного проектирования при разработке прикладных программ	2	Реферат, доклад, сообщение

ПРИМЕРНЫЙ ПЕРЕЧЕНЬ ВОПРОСОВ (ЗАДАНИЙ) ДЛЯ ПОДГОТОВКИ К ТЕКУЩЕЙ АТТЕСТАЦИИ

1. Основные принципы объектно-ориентированного программирования.
2. Базовые абстракции объектно-ориентированного программирования.
3. Понятие класса и объекта.
4. Структурные элементы класса и методы взаимодействия объектов. Конструкторы и деструкторы класса.
5. Разграничение доступа к атрибутам объектов Классы в программных модулях. Атрибуты доступа к элементам объектов. Термин «инкапсуляция».
6. Жизненный цикл объектов.
7. Методы и механизмы разработки объектно-ориентированных программ.
8. Механизм наследования. Производные классы. Свойства базового и производного класса.
9. Понятие виртуального метода. Перекрытие виртуального метода в производном классе. Абстрактный виртуальный метод. Механизм вызова виртуального метода. Методы обработки сообщений. Термин «полиморфизм».
10. Понятие ссылки на метод объекта. Понятие события. Применение ссылок на методы для расширения объектов.
11. Исключительные ситуации. Классы исключительных ситуаций. Создание и обработка исключительных ситуаций. Защита от утечки ресурсов в случае возникновения исключительных ситуаций.
12. Понятие интерфейса. Описание интерфейса. Поддержка интерфейса классом. Совместимость интерфейсов и классов. Механизм вызова метода объекта через интерфейс. Применение интерфейса для доступа к объекту динамически-подключаемой библиотеки.
13. Параметризация объектов. Параметризированные классы и методы, их свойства. Совместное использование параметризации и принципов наследования. Организация внешнего доступа к компонентам параметризированных классов.
14. Контейнерные типы и их применение.
15. Понятие компонента. Понятие визуального программирования. Инструментальные средства визуального компонентного программирования. Современные библиотеки компонентов.
16. Объектно-ориентированный анализ и проектирование программных средств.
17. Модели процесса разработки. Классификация методологий создания программных проектов. Выбор методологии для определенного типа программного проекта.
18. Виды отношений классов (ассоциация, агрегация, обобщение, зависимость, инстанцирование).
19. Назначение и условия применимости паттернов проектирования. Способ и результат применения. Классификация паттернов по стилю. Структурные и порождающие паттерны, паттерны поведения.
20. Архитектура и архитектурный подход. Архитектурные шаблоны в работе с уровнем данных. Основные шаблоны уровня данных (доступа к данным и базам данных).
21. Паттерны проектирования.

VI. ПРОТОКОЛ СОГЛАСОВАНИЯ УЧЕБНОЙ ПРОГРАММЫ УВО

Название дисциплины, с которой требуется согласование	Название кафедры	Предложения об изменениях в содержании учебной программы по изучаемой учебной дисциплине	Решение, принятое кафедрой, разрабатывающей учебную программу (с указанием даты и номера протокола)

I. ТЕОРЕТИЧЕСКИЙ РАЗДЕЛ

КОНЦЕПЦИЯ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ. ПОНЯТИЕ ОБЪЕКТА И ФУНДАМЕНТАЛЬНЫЕ ХАРАКТЕРИСТИКИ ООП (ИНКАПСУЛЯЦИЯ, НАСЛЕДОВАНИЕ, ПОЛИМОРФИЗМ)

Объектно-ориентированное программирование (ООП) – это новый подход к созданию программ. По мере развития вычислительной техники возникали разные методики программирования. На каждом этапе создавался новый подход, который помогал программистам справляться с растущим усложнением программ. Так язык программирования (даже высокого уровня), легко понимаемый в коротких программах, становился нечитабельным в более длинных. Ситуацию разрешило изобретение в 1960 году языков структурного программирования (к ним относятся Алгол, Паскаль и С). Структурное программирование подразумевает точно обозначенные управляющие структуры, программные блоки, отсутствие (минимальное использование) инструкций GOTO, автономные подпрограммы, в которых поддерживаются локальные переменные и рекурсия. Сутью структурного программирования является возможность разбиения программы на составляющие ее элементы. Хотя структурное программирование в свое время принесло выдающиеся результаты, чтобы написать более сложную программу, отвечающую современным требованиям, необходим новый подход к программированию. Объектно-ориентированное программирование позволяет разложить проблему на составные части, каждая из которых становится самостоятельным объектом. Каждый из объектов содержит свой собственный код и данные, которые относятся к этому объекту. Давайте обратимся к понятию “объект”. Технология ООП, прежде всего, накладывает ограничения на способы представления данных в программе. Любая программа отражает в них состояние физических предметов либо абстрактных понятий (назовем их объектами программирования) для работы, с которыми она предназначена. В традиционной технологии варианты представления данных могут быть разными. В худшем случае программист может «равномерно размазать» данные о некотором объекте программирования по всей программе. Например, имеется задача, условие которой гласит: «Определить принадлежит ли точка с заданными координатами заштрихованной области», и приводится рисунок. Объектами программирования такой задачи выступают «точка» и «заштрихованная область». Скорее всего, в решении этой задачи для структурного языка программирования как раз и наблюдается принцип “размазанности” данных: координаты точек приходиться хранить в независимых переменных X и Y, информация о заштрихованной области храниться на протяжении всей программы в виде операторов ветвления.

В противоположность такому подходу все данные об объекте программирования и его связях с другими объектами можно объединить в одну структурированную переменную. В первом приближении ее можно назвать объектом. Данные об объекте программирования объединены в большинстве случаев в структуру – структурированную переменную. Кроме того, с объектом связывается набор действий, иначе называемых методами. С точки зрения языка программирования набор действий или методов это функции, получающие в качестве обязательного параметра указатель на объект и выполняющие определенные действия с данными объекта программирования. Технология ООП запрещает работать с объектом иначе, чем через методы, таким образом, внутренняя структура объекта скрыта от внешнего пользователя. Описание множества однотипных объектов называется классом.

Объект - это структурированная переменная, содержащая всю информацию о некотором физическом предмете или реализуемом в программе понятии.

Класс - это описание множества объектов программирования (объектов) и выполняемых над ними действий.

Основные понятия объектно-ориентированного программирования: **инкапсуляция**, **наследование** и **полиморфизм.**

«Эпизодическое» использование технологии ООП заключается в разработке отдельных, не связанных между собой классов и использовании их как необходимых программисту базовых типов данных, отсутствующих в языке. При этом общая структура программы остается традиционной («от функции к функции»). Однако, строгое следование технологии ООП предполагает, что любая функция в про грамме представляет собой метод для объекта некоторого класса. Это не означает, что нужно вводить в программу какие попало классы ради того, чтобы написать необходимые для работы функции. Наоборот, класс должен формироваться в программе естественным образом, как только в ней возникает необходимость описания новых объектов программирования. С другой стороны, каждый новый шаг в разработке алгоритма также должен представлять собой разработку нового класса на основе уже существующих. В конце концов, вся программа в таком виде представляет собой объект некоторого класса с единственным методом run (выполнить). Именно этот переход (а не понятия класса и объекта, как таковые) создает психологический барьер перед программистом, осваивающим технологию ООП.

Программирование «от класса к классу» включает в себя ряд новых понятий. Прежде всего, это – инкапсуляция данных. ***Инкапсуляция*** – это механизм, который объединяет данные и код, манипулирующий с этими данными, а также защищает и то, и другое от внешнего вмешательства или неправильного использования.

В ООП код и данные могут быть объединены вместе (в так называемый «черный ящик») при создании объекта. Внутри объекта коды и данные могут быть закрытыми или открытыми. Закрытые коды или данные доступны только

для других частей того же самого объекта и, соответственно, недоступны для тех частей программы, которые существуют вне объекта. Открытые коды и данные, напротив, доступны для всех частей программы, в том числе и для других частей того же самого объекта.

Для примера, рассмотрим такую структуру хранения информации как стек. Пользователя интересует выполнение операций: push(), top(), empty(), pop(), reset(), full(). Внутренняя реализация (адреса ячеек памяти, значения регистров процессора, имена массивов и т.д.) пользователя не интересует, то есть она должна быть скрыта от него. Пользователю нужны только указанные операции, и он должен получить в свое распоряжение набор методов для их выполнения. Объектом программирования в этом случае является структура хранения информации типа стек. Соответственно, внутренняя организация объекта стек должна быть недоступна (закрыта) для пользователя, то есть все его попытки обратиться непосредственно к элементам, обеспечивающим функционирование структуры хранения данных, должны отвергаться (выдавать сообщение об ошибке). А вызов методов, напротив, должен приводить к выполнению желаемого действия и является общедоступным.

Вторым по значимости понятием является ***наследование***. Новый, или производный класс может быть определен на основе уже имеющегося, или базового класса. При этом новый класс сохраняет все свойства старого: данные объекта базового класса включаются в данные объекта производного, а методы базового класса могут быть вызваны для объекта производного класса, причем они будут выполняться над данными включенного в него объекта базового класса. Иначе говоря, новый класс наследует как данные старого класса, так и методы их обработки. Если объект наследует свои свойства от одного родителя, то говорят об одиночном наследовании. Если же объект наследует данные и методы от нескольких базовых классов, то говорят о множественном наследовании. Простой пример наследования - определение структуры, отдельный член которой является ранее определенной структурой.

Рассмотрим еще один пример. Например, базовый класс «животные» может иметь производные классы: «млекопитающие», «рыбы”, «птицы» и т.д. Они будут наследовать все характеристики базового класса, но каждый из них может иметь и свои собственные свойства: «рыбы – плавники», «птицы – крылья» и т.д.

Третьим по значимости понятием является полиморфизм. ***Полиморфизм*** - это свойство, которое позволяет один и тот же идентификатор (одно и то же имя) использовать для решения двух и более схожих, но технически разных задач. Целью полиморфизма, применительно к ООП, является использование одного имени для задания действий, общих для ряда классов объектов. Такой полиморфизм основывается на возможности включения в данные объекта также и информации о методах их обработки (в виде указателей на функции). Принципиально важно, что такой объект становится “самодостаточным”. Будучи доступным в некоторой точке программы, даже при отсутствии полной

информации о его типе, он всегда может корректно вызвать свойственные ему методы. Таким образом, полиморфная функция - это семейство функций с одним и тем же именем, но выполняющие различные действия в зависимости от условий вызова.

Например, нахождение абсолютной величины в языке С требует трех разных функций: *int abs(int); long labs(long); double fabs(double);*

Эти функции подсчитывают и возвращают абсолютную величину целых, длинных целых и чисел с плавающей точкой соответственно. С точки зрения полиморфизма, каждую из этих функций может быть названа abs(), а тип данных, который используется при вызове функции, определяет, какая конкретная версия функции действительно выполняется.

НЕ СВЯЗАННЫЕ С ОБЪЕКТАМИ РАСШИРЕНИЯ С++ ОТНОСИТЕЛЬНО С. НОВОЕ В ОПИСАНИИ ТИПОВ И ПЕРЕМЕННЫХ. ПЕРЕГРУЖАЕМЫЕ ФУНКЦИИ И ФУНКЦИИ С АРГУМЕНТАМИ ПО УМОЛЧАНИЮ

Язык С++ не появился на «голом» месте, на его синтаксические структуры определенное влияние оказали различные конструкции других языков программирования. В основу С++, безусловно, был положен язык программирования С, разработанный Денисом Ричи. Кроме того, предшественником С++ явился язык BCPL. Из языка Simula67 были позаимствованы концепция классов вместе с производными классами и функциями-членами. Перегрузка операций и свобода в расположении описаний взяты из Алгол68.

Название С++ появилось летом 1983 г. Более ранние версии этого языка использовались с 1979 г. под общим названием «С с классами». Название «С++» придумал Риск Маскитти (Rick Mascitti). Это название отражает эволюционный характер изменения языка С. Согласно одной из интерпретаций названия языка «++» – это оператор инкрементации в языке С.. Существуют и другие интерпретации названия, тем не менее язык не называется D, потому что он - расширение С. Фактически язык С++ представляет собой полный язык С с надстройкой для реализации идей ООП, а также некоторыми дополнительными синтаксическими структурами, включенными для удобства пользователей. Все основные операции, операторы, типы данных языка С присутствуют в С++. Некоторые из них усовершенствованы и добавлены принципиально новые конструкции, которые и позволяют говорить о С++ как о новом языке, а не просто о новой версии языка С. К настоящему времени язык С++ претерпел несколько существенных модернизаций, последняя из которых связана с процедурой стандартизации (1998 г.). Вследствие этого более ранние реализации языка отличаются друг от друга. При этом мы будем рассматривать те возможности языка С++, которые верны для большинства реализаций, и их принято считать общими, полагая что Standard С++ является надмножеством традиционного С++.

Дополнительные ключевые слова. С++ расширен следующими ключевыми словами:

class delete friend inline new operator private protected public template this virtual

Комментарии. Символы /* открывают комментарий, оканчивающийся символами */. Вся такая последовательность эквивалентна игнорируемому символу (например, пробелу). Это наиболее удобно для написания многострочных комментариев, и является единственным видом комментариев в языке С. Но как только что было сказано, все, что относится к С справедливо и

для С++. Кроме того, для написания коротких комментариев, в языке С++ используются символы

```
// int a; /* целая переменная */
float b; // вещественная переменная
```

Переменные. Язык С++ является блочно сконструированным. Переменные, описанные внутри блока, вне блока недоступны. Переменные внешнего блока доступны во внутреннем, если он их не переопределяет. Приведем несколько определений, касающихся идентификаторов переменных (объектов) С++. Контекстом идентификатора называется часть программы, в которой данный идентификатор может быть использован для доступа к соответствующему объекту. Если идентификатор объявлен в блоке, то он имеет контекст блока, который начинается в точке объявления и распространяется до конца блока. Поскольку тело функции представляет собой блок, то это правило распространяется на локальные переменные функции.

Параметры функции имеют контекст тела функции, например:

```
void func(char ch, int i) // начало контекста ch, i
{
        int x = 5;  // начало контекста x
        x++;
        int y = x * x + 8; // начало контекста y
. . . . . .
        int z = x + y; // начало контекста z
. . . . . .
 } // конец контекста ch, i, x, y, z
```

Если идентификатор объявлен вне всех блоков и функций, он имеет контекст файла, начинающийся в точке объявления и заканчивающийся в конце исходного файла. Объявленные таким образом переменные называются глобальными, например:

```
// файл my_func.c
float r; // r, d и h имеют контекст файл my_func.c
double d;
int h;
void func() {...}
```

Видимостью идентификатора называется область исходного текста программы, из которого возможен нормальный доступ к соответствующему объекту. Обычно видимость и контекст идентификаторов совпадают. Исключение составляют случаи, когда переменную внешнего (охватывающего) блока временно скрывает переопределение во внутреннем блоке переменной с тем же именем, например:

```
{
int i = 3; . . . . . .
for (int k = 0; k<5; k++)
{
        int i = k; // здесь i - новая переменная, изменяющаяся от 0 до 4,
// скрывает i = 3 внешнего блока
 . . . . . .
}
// здесь по-прежнему i = 3
}
```

Связанные с объявлением идентификаторов объекты характеризуются также продолжительностью. В C++ различают статическую (static), локальную (local) и динамическую (dynamic) продолжительности.

Объектам со статической продолжительностью память обычно выделяется в фиксированной области в начале выполнения программы и сохраняется до выхода из программы. К объектам со статической продолжительностью относятся все функции, независимо от того, где они определены, переменные с файловым контекстом и переменные, имеющие спецификаторы памяти static или extern.

Память объектам с локальной продолжительностью (или локальным переменным) выделяется в стеке при входе в ближайший охватывающий их блок и уничтожается при выходе из блока.

Объекты с динамической продолжительностью создаются и разрушаются в свободной области памяти (куче) путем вызова функций (malloc(), calloc(), free()) или с помощью операций new и delete в процессе выполнения программы, например:

```
float r;  // r, d и h имеют статическую продолжительность
double d;
int h;
void func(char ch, int i)
{
        int x = 5;  // ch, i и x - локальную
        char *str;
        str = (char*)malloc(81*(sizeof(char)));
. . . . . .
free(srt);  // srt - динамическую
}
```

В С переменные инициализируются константными выражениями. В С++ переменные можно инициализировать обычными выражениями. Последнее носит название динамической инициализации переменных, поскольку позволяет инициализировать значения переменных в процессе выполнения программы. Например:

```
void main(void)
 {
 char s[81];  …….
 int k = strlen(s);  // динамическая инициализация  ……..
 }
```

В С++ односимвольные константы имеют тип char (в С они автоматически преобразуются к типу int). С++ поддерживает также двухсимвольные константы типа int , например:

```
'AB', '\n\t', '\t\v'
```

при этом первый символ располагается в младшем байте, а второй - в старшем байте. Если в С++ символьной константе предшествует L, то она называется широкой символьной константой и имеет тип wchar_t, определенный в stddef.h, например:

```
x = L'abc';
```

Константы. Константы в языке С++ аналогичны константам в С. Отличие состоит в том, что для символьного представления константы в С использовалась директива препроцессора #define. В С++ для символьного представления константы рекомендуется использовать объявление переменной с начальным значением и ключевым словом const:

const <тип> <имя переменной>=<начальное значение>;

Например, *const int n=10;*

Область видимости константы такая же, как у обычной переменной. С помощью ключевого слова const можно объявить указатель на константу

const <тип> *<имя переменной>; Например,

*const int *m;*

m=&n; m - указатель на константу типа int.

Еще одна возможность const состоит в возможности создавать постоянный указатель на величину указанного типа

<тип> *const <имя переменной>=<значение>;

Например,

int i;

*int *const ptri=&i;*

Операции. В C++ введены следующие новые операции для работы с компонентами класса:

:: - операция доступа к контексту или разрешения контекста;

.* и -> * - операции обращения через указатель к компоненте класса;

new - динамическое выделение памяти объектам;

delelte - освобождение памяти, выделенной операцией new.

Операция :: доступа к контексту позволяет обращаться к глобальным переменным, когда они скрыты объявлением локального имени, например :

```
int x = 5;  // объявление глобальной переменной x main (void)
{
 int x = 3;  // объявление локальной переменной,   // скрывающей глобальную x
 ::x++; // изменение значения глобальной x
 ……….
}
```

В C++ предусмотрены специальные операции new и delete для динамического выделения и освобождения памяти объектам в свободной области памяти (куче). Эти операции могут использоваться для выделения памяти переменным и массивам, аналогично функциям malloc() и free(). Формат операций:

<указатель> = new <тип>;

или <указатель> = new <тип> (<инициализирующее_значение>);

delete <указатель>;

где указатель - переменная типа указатель на заданный тип;

тип - тип переменной, которой выделяется память;

нициализирующее_значение - константа, определяющая начальное значение переменной.

Операция new возвращает указатель на блок в свободной области памяти размером sizeof (<тип>). Если выделить память не удалось, то возвращается нулевой указатель. Созданный объект будет существовать в памяти до тех пор, пока память не будет освобождена с помощью операции delete или до окончания работы программы.

Например :

```
#include <stdio.h>
#include <stdlib.h>  main (void)
{  int *pi;
if ( !(pi = new int (5)) )  // выделение памяти
{   // и инициализация
 printf("Не хватает памяти для pi\n");  exit(1);
 }
printf ("pi= %d\n",*pi);  delete pi;   // освобождение памяти  }
```

В случае успеха будет выведено *5*, в противном случае появляется строка: *Не хватает памяти для pi*

Операция new отличается от функции malloc(), во-первых, тем, что операция new автоматически вычисляет размер типа и нет необходимости явно указывать операцию sizeof и, во-вторых, операция new позволяет инициализировать вновь создаваемый объект (в отличие от функции calloc(), которая только обнуляет память).

Важное замечание: если память объекту в программе отводилась с помощью операции new, то освобождаться она должна только с помощью операции delete (и ни в коем случае не функцией free() !). Нельзя также применять операцию delete для освобождения памяти, ранее выделенной функциями malloc() или calloc().

Ввод-вывод. В C++, как и в C, нет встроенных в язык средств ввода-вывода. В C для этих целей используется стандартная библиотека stdio.h, но она имеет несколько недостатков:

• функции ввода-вывода сложны в употреблении, что приводит к частым ошибкам; • затруднителен ввод-вывод абстрактных типов данных (например, структур - struct).

В C++ разработана новая библиотека ввода-вывода (iostream.h), написанная на C++, и использующая концепцию объектно-ориентированного программирования. Библиотека iostream.h определяет три стандартных потока:

cin стандартный входной поток (stdin в C) cout стандартный выходной поток (stdout в C)

cerr стандартный поток вывода сообщений об ошибках (stderr в C)

Для выполнения операций ввода-вывода переопределены две операции поразрядного сдвига:

>> получить из входного потока

<< поместить в выходной поток

Ввод значений переменной может быть осуществлен следующим способом:

cin >> идентификатор;

По этому выражению из входного потока читается последовательность символов до пробельного символа, затем эта последовательность преобразуется к типу <идентификатора> и получаемое значение помещается в <идентификатор>.

Вывод информации осуществляется с использованием потока cout:

cout << значение;

Здесь <значение> преобразуется в последовательность символов и выводится в выходной поток. Возможно многократное назначение потоков: *cin >> 'переменная1' >> 'переменная2' >>...>> 'переменная n'; cout << 'значение1' << 'значение2' << ... << 'значение n';*

Программа на С Программа на С++

```
# include <stdio.h> # include <iostream.h>
void main()  void main() { int n;  { int n; char str[80];   char str[80]; printf (“Ввести число\n”); cout << “Ввести число\n”;  scanf(“%d”,&n);  cin >> n; printf (“Ввести строку\n”);  cout << “Ввести строку\n”;
gets(str);   cin >> str;
printf(“n=%d\n”,n); cout << “n=“<<n<<endl;
printf(“str=%s\n”,str); cout << “str=“<<str <<endl;
}  }
```

Позднее мы еще вернемся к рассмотрению библиотеки ввода-вывода языка С++.

Ссылки. В С++ введен новый тип данных - ссылка. Ссылка позволяет определять альтернативное имя переменной. Объявляются ссылки добавлением символа & перед именем переменной. Формат объявления ссылки:

<тип> &<имя_ссылки> = <инициализатор>;

Ссылку нельзя объявить без инициализатора, т.е. некоторого объекта, связанного с сылкой, например :

```
main (void)
 {
  int x = 3;   // объявление переменной x int &rx = x;   // объявление ссылки на переменную x
  ……..
  rx = 5;   // изменяется значение x
  ……..
  }
```

Ссылка может использоваться везде, где используется переменная, на которую она указывает - от нее можно брать адрес, как от переменной, и т.д. В данном примере сылка rx используется как псевдоним переменной x и всякое обращение к rx равносильно обращению к x.

Ссылки могут также указывать на константу. В этом случае компилятор создает временный объект, инициализированный значением константы:

```
main (void)
 {
 int &ry = 7; // создается временный объект с именем ry
   // и значением 7
 ry = 8;  // изменяется значение ry
 ……..
 }
```

При новой инициализации временного объекта адрес его не изменяется. Необходимость создания временного объекта при инициализации ссылки константой вызвана тем, что большинство компиляторов оптимизируют код, создавая только одну копию одинаковых констант. Если допустить установку ссылки на константу, а не на временный объект, это может вызвать изменение констант в других частях программы, например:

```
int &rz = 1;  rz++;
int y = rz + 1;
```

вызовет инициализацию y значением 2 вместо 3.

Временный объект также создается, когда инициализатор представляет собой объект нессылочного типа, например:

```
main (void)
{
 float f = 1.5;  // переменная типа float  int  &rf = f;  // создается временный объект rf
 ……..
 rf = 2;  // изменяется только rf, а f остается    // без изменения
 ……..
}
```

В данном примере создается временный объект rf, поскольку тип float инициализатора f не совпадает с типом ссылки int, т.е. значение ссылки rf определяется инициализатором нессылочного типа.

Отметим некоторые особенности ссылок:

- ссылки не являются указателями;
- нельзя ссылаться на битовые поля, так как отдельные биты не имеют адресов;
- нельзя создать массив ссылок :

```
int &rm[5];  // недопустимо
```

- можно образовывать ссылку на ссылку, при этом временный объект не создается :

```
int &rx = 5;  int &ry = rx;
```

- можно также устанавливать указатель на ссылку:

```
int &rx = 5;
int *p = &rx;
```

Основное назначение ссылок - это передача аргументов функции «по ссылке», а не «по значению». Другими словами ссылки в аргументах функций C++ аналогичны параметру VAR в языке Pascal. При этом нет необходимости передавать функции в качестве аргументов адреса фактических параметров, а достаточно объявить формальные параметры функции как ссылки на соответствующий тип. Для примера рассмотрим функцию swap(), меняющую местами значения своих аргументов:

```
#include <stdio.h>
void swap(int &a, int &b)
{  int c;  c = a;  a = b;  b = c;
}
main(void)
{
 int i = 5, j = 7;  printf("Начальное значение: i = %d j = %d\n", i, j);  swap(i,j);
printf("После обмена: i = %d j = %d\n", i, j);
 }
```

Ссылки в качестве аргументов функции также полезны для сокращения области локальной памяти программы при передаче больших параметров. В этом случае функции будет передаваться не сам параметр, а ссылка на него. Чтобы предотвратить возможное изменение параметра в теле функции, он может быть описан со спецификатором const, например:

```
void func(const large& arg)  // здесь large - некоторый
  // пользовательский тип большого
  // объема
{
. . . . . // значение arg не изменяется
 }
```

Кроме того, в C++ функции могут не только принимать ссылку в качестве аргумента, но и возвращать ссылку на переменную. Выражение вызова такой функции может появиться в любой части операции присваивания. При этом необходимо учитывать, что если возвращаемое значение - указатель, то нельзя оператором return возвращать адрес локальной переменной.

Если возвращаемое значение - ссылка, то нельзя оператором return возвращать локальную переменную. (Так как после выхода из функции переменная не существует, и мы получим повисшую ссылку).

Таким образом, использовать ссылки нужно крайне осторожно. Чаще всего потребность в ссылках возникает при перегрузке операций.

Функции. Прототипы функций. Определение функции в программе выглядит следующим образом:

<заголовок_функции>
{
<тело функции>
}

Заголовок функции имеет следующий вид:

<тип_функции> <имя_функции> (<спецификация_формальных_параметров>)

Если функция не возвращает значения, то ее тип void. Если тип возвращаемого значения не указан, то по умолчанию используется тип int.

При обращении к функции, формальные параметры заменяются фактическими, причем соблюдается строгое соответствие параметров по типам. В отличие от своего предшественника - языка С С++ не предусматривает автоматического преобразования в тех случаях, когда фактические параметры не совпадают по типам с соответствующими им формальными параметрами. Говорят, что язык С++ обеспечивает <строгий контроль типов>. В связи с этой особенностью языка С++ проверка соответствия типов формальных и фактических параметров выполняется на этапе компиляции.

Строгое согласование по типам между формальными и фактическими параметрами требует, чтобы в модуле до первого обращения к функции было помещено либо ее определе- ние, либо ее описание (прототип), содержащее сведения о ее типе, о типе результата (то есть возвращаемого значения) и о типах всех параметров. Именно наличие такого прототипа либо полного определения позволяет компилятору выполнять контроль соответствия типов параметров. Прототип (описание) функции может внешне почти полностью совпадать с заголов- ком ее определения:

<тип_функции> <имя_функции> (<спецификация_формальных_параметров>);

Основное различие - точка с запятой в конце описания (прототипа). Второе отличие - необязательность имен формальных параметров в прототипе даже тогда, когда они есть в заголовке определения функции.

Функции. Значения формальных параметров по умолчанию. Спецификация формальных параметров - это либо пусто, либо void, либо список спецификаций отдельных параметров, в конце которого может быть поставлено многоточие. Спецификация каждого параметра в определении функции имеет вид: <тип> <имя_параметра>

<тип> <имя_параметра> = <значение_по_умолчанию>

Как следует из формата, для параметра может быть задано (а может отсутствовать) умалчиваемое значение. Это значение используется в том случае, если при обращении к функции соответствующий параметр опущен. При задании начальных (умалчиваемых) значений должно соблюдаться следующее соглашение. Если параметр имеет значение по умолчанию, то все параметры, специфицированные справа от него, также должны иметь начальные значения.

Пусть нужно вычислить n^k, где k чаще всего равно 2.

```
int pow(int n, int k=2)  // по умолчанию k=2
{  if( k== 2) return( n*n );  else return( pow( n, k-1 )*n );
}
```

Вызывать эту функции можно двумя способами:

```
t = pow(i+3); q = pow(i+3, 5);
```

Значение по умолчанию может быть задано либо при объявлении функции, либо при определении функции, но только один раз.

Функции. Перегрузка функций. Цель перегрузки функций состоит в том, чтобы функция с одним именем по-разному выполнялась и возвращала разные значения при обращении к ней с разными по типам и количеству фактическими параметрами. Например, может потребоваться функция, возвращающая максимальное значение элементов одномерного массива, передаваемого ей в качестве параметра. Массивы, использованные как фактические параметры, могут содержать элементы разных типов, но пользователь функции не должен беспокоиться о типе результата. Функция всегда должна возвращать значение того же типа, что и тип массива - фактического параметра. Для обеспечения перегрузки функций необходимо для каждого имени определить, сколько разных функций связано с ним, т.е. сколько вариантов сигнатур допустимы при обращении к ним. Предположим, что функция выбора максимального значения элемента из массива должна работать для массивов типа int, long, float, double. В этом случае придется написать четыре разных варианта функции с одним и тем же именем. Распознавание перегруженных функций при вызове выполняется по их сигнатурам. Перегруженные функции, поэтому должны иметь одинаковые имена, но спецификации их параметров должны различаться по количеству и (или) по типам, и (или) по расположению. При использовании перегруженных функций нужно с осторожностью задавать начальные значения их параметров.

Функции. Встраиваемые функции. В базовом языке С директива препроцессора #define позволяла использовать макроопределения для записи вызова небольших часто используемых конструкций. Некорректная запись макроопределения может приводить к ошибкам, которые очень трудно найти. Макроопределения не позволяют определять локальные переменные и не выполняют проверки и преобразования аргументов. Если вместо макроопределения использовать функцию, то это удлиняет объектный код и увеличивает время выполнения программы. Кроме того, при работе с макроопределениями необходимо тщательно проверять раскрытия макросов:

```
#define SUMMA(a, b) a + b
rez = SUMMA(x, y)*10;
```

После работы препроцессора получим:

```
rez = x + y*10;
```

В C++ для определения функции, которая должна встраиваться как макроопределение используется ключевое слово inline. Вызов такой функции приводит к встраиванию кода inlineфункции в вызывающую программу. Определение такой функции может выглядеть следующим образом:

```
inline double SUMMA(double a, double b)
{
  return(a + b);
}
```

При вызове этой функции *rez = SUMMA(x,y)*10;* будет получен следующий результат: *rez=(x+y)*10.*

При определении и использовании встраиваемых функций необходимо придерживаться следующих правил:

определение и объявление функций должны быть совмещены и располагаться перед первым вызовом встраиваемой функции.

Имеет смысл определять inline только очень небольшие функции.

Различные компиляторы накладывают ограничения на сложность встраиваемых функций. Компилятор сам решает, может ли функция быть встраиваемой. Если функция не может быть встраиваемой, компилятор рассматривает ее как обычную функцию. Таким образом, использование ключевого слова inline для встраиваемых функций и ключевого слова const для определения констант позволяют практически исключить директиву препроцессора #define из употребления.

АБСТРАКТНЫЕ ТИПЫ ДАННЫХ. ОБРАЩЕНИЕ К КОМПОНЕНТАМ КЛАССА. СТАТИЧЕСКИЕ ЧЛЕНЫ. ЗАЩИТА ЭЛЕМЕНТОВ КЛАССА И АТРИБУТЫ ДОСТУПА. КОНСТРУКТОРЫ И ДЕСТРУКТОРЫ

Объявление класса. *Синтаксически классы в C++ подобны структурам, компонентами которых помимо данных могут являться также и функции. При объявлении классов используется ключевое слово* class:

```
#include <iostream.h>
#include <stdlib.h>

class Array   // имя класса
{
  public:   // спецификатор степени доступа

 // компоненты-данные класса
 int *m;  //указатель на массив целых чисел
 int size; // размерность массива

 // компоненты-функции класса
 Array(int n = 0);  // описание конструктора
~Array() { delete m; }  // определение деструктора
int get(int i);  // извлечение элемента массива
void print(char *);  // функция вывода };

Array:: Array(int n)  // определение конструктора
{  size = n; m = new int[size];
 for (int i=0; i<size; m[i++]=0);
}

int Array:: get(int i)  //определение функции get()
{
 if (i < 0 || i >= size)
 {
 cout << "Выход за границы массива\n";
 exit(1);
 }
 return m[i];
}
```

```
void Array:: print(char *s)  // определение printf()
{
  cout << s << ": "; for (int i=0; i<size; cout << get(i++) << " ");
  cout << "\n";
}

main(void)
{
  Array x(5);  // определение объекта класса Array
Array *py;  // объявление указателя на объект
  py = new Array(10); // создание нового объекта     // и установка на него указателя
  // вызов компонент-функций:
  x.print("x");  // прямой
  py->print("y");  // косвенный
}
```

В этом примере опеределяется класс Array, который имеет две компоненты данных:

*m - указатель на массив целыхчисел и size - размерность массива.

Здесь же объявлены четыре компоненты-функции:

Array(), ~Array(), get() и print().

Компоненты-функции могут определяться как внутри класса (~Array()), так и вне его (Array() и print()). В последнем случае имени функции должно предшествовать имя класса с операцией доступа к контексту (::). Например, конструкция Array:: print() говорит о том, что далее следует описание функции print(), относящейся к классу Array.

Язык С++ допускает использование одинаковых имен компонент в различных классах. Чтобы иметь возможность доступа ко всякой компоненте произвольного класса в любой точке программы служит синтаксическая конструкция, называемая квалификационным именем, вида:

<класс>:: <компонента>

где класс - имя класса;

компонента - имя компоненты класса.

Напомним, что применение операции «::» перед некоторым именем без имени класса позволяет обращаться к глобальным переменным. В данном случае использование квалификационного имени позволяет большие описания компонент-функций выносить вне тела определения класса.

Описание класса определяет специальный тип данных, в котором наряду с собственно данными описываются также и функции для работы с этими данными. В последующем имя класса (Array) может использоваться подобно другим пользовательским типам для определения переменных, указателей,

ссылок и т.д. Каждый конкретный экземпляр (переменная) класса называется объектом.

Обращение к компонентам класса осуществляется так же, как к компонентам структур с помощью операций прямого (.) и косвенного (->) выбора компонент:

```
Array x(5),   // опеределение объектов
*py = new Array(10);
. . . . . .
= x.m[i];      // прямой выбор компоненты
= py->m[i];          // косвенный выбор компоненты
```

Обращение к компонентам-функциям осуществляется аналогично:

```
x.print();      // прямой выбор компоненты-функции
py->print(); // косвенный выбор компоненты-функции
```

Если обращение к компоненте осуществляется внутри тела другой компоненты, то обращение выполняется без указания объекта и, следовательно, без использования операторов прямого либо косвенного выбора, например:

```
void Array:: put(int i, int x)  // компонента-функция
{  // класса Array . . . . . .
  m[i] = x;    // обращение к компоненте
        // этого же класса
  print();   // … к компоненте-функции }
```

Встроенные функции и спецификатор inline. В программах на языке C++ большинство компонент-функций являются очень короткими, содержащими простейшие обращения к компонентам данных. В связи с этим возникает опасность низкой эффективности программы в целом из-за частых обращенийк функциям. Чтобы устранить этот недостаток в C++ предложен механизм встраиваемых функций. Компоненты-функции, определенные внутри тела класса, называются встраиваемыми. Компилятор вместо вызова таких функций в объектный модуль стремиться вставить непосредственно код этой функции. Указания компилятору на подобные действия можно сделать и для компонент-функций, описанных вне класса, с помощью спецификатора inline, например):

```
inline int Array:: get(int i)
{ if (i < 0 || i >= size)
 . . . . . .
 return m[i];
}
```

что эквивалентно

```
class Array {
. . . . . .
 int get(int i)
 {  if (i < 0 || i >= size)
 . . . . . .
 return m[i];
 }
. . . . . .
};
```

Указатель this. В функции - члене на данные - члены объекта, для которого она была вызвана, можно ссылаться непосредственно. Например:

```
class x {  int m;  public:  int readm() { return m; }
 }; x aa; x bb; void f() {
  int a = aa.readm();  int b = bb.readm();
  // ... }
```

В первом вызове члена readm() m относится к aa.m, а во втором - к bb.m.

Указатель на объект, для которого вызвана функция член, является скрытым параметром функции. На этот неявный параметр можно ссылаться явно как на this. В каждой функции класса x указатель this неявно описан как `x* this;`

и инициализирован так, что он указывает на объект, для которого была вызвана функция член. this не может быть описан явно, так как это ключевое слово. Класс x можно эквивалентным образом описать так:

```
 class x
{
        int m;
       public:
       int readm() {
              return this->m;
       }
};
```

При ссылке на члены использование this излишне. Главным образом this используется при написании функций членов, которые манипулируют непосредственно указателями.

Статические члены: функции и данные. Иногда необходимо реализовать класс таким образом, чтобы все объекты этого типа могли совместно использовать (разделять) некоторые данные. Предпочтительно, чтобы такие разделяемые данные были описаны как часть класса. Типичные случаи: требуется контроль общего количества объектов класса или одновременный

доступ ко всем объектам или части их, разделение объектами общих ресурсов. Тогда в определение класса могут быть введены (посредством ключевого слова static) статические элементы - переменные. Ключевое слово static может быть использовано при объявлении членов-данных и функций-членов. Такие члены классов называются статическими и, независимо от количества объектов данного класса, существует только одна копия статического элемента, поэтому к нему можно обращаться с помощью оператора разрешения контекста и имени класса

<имя_класса>::<имя_элемента>

Если x - статическое данное-член класса cl, то на него можно ссылаться cl::x, и при этом не имеет значения количество объектов класса cl. Аналогично можно обращаться к статической функции-члену:

```
class X {  public: int n;
 void memfunc( int n );
        static void statfunc(int n, X *ptr);  // статическая функция
        .....................
};

void prog()
{
  X obj;

  obj.memfunc(20)  // вызов для обычной функции
  X :: func(10, &obj);  // ... для статической функции
}
```

Статические данные-члены класса можно рассматривать как глобальную переменную класса. Но в отличии от обычных глобальных переменных на статические члены распространяются правила видимости private и public. Поместив статическую переменную в часть private, можно ограничить ее использование. Объявление статического члена в объявлении класса не является определением, то есть это объявление статического члена не обеспечивает распределения памяти и инициализацию. Статические члены-данные нельзя инициализировать в теле класса, а так же в функциях-членах. Статические члены должны инициализироваться аналогично глобальным переменным в области видимости файла:

```
class example
{ public:
 static int mask;
}
int example::mask = 0; // определение и инициализация
```

Защита компонент классов. Степень доступа компонент класса определяется с помощью меток (спецификаторов): public (общий), protected (защищенный) и private (скрытый) . Каждая метка определяет атрибут доступа на последующие описания компонент до тех пор, пока не встретиться другая метка. По умолчанию принимается атрибут private, например:

```
class X {
 int a;
char  ch;  // a, ch - private по умолчанию
  protected:
        float f;
        double d;  // f, d - protected

  public:
  void  func(void); // func - public
};
```

Здесь a и ch имеют по умолчанию атрибут доступа private, f и d - protected и func() - public. Все компоненты некоторого класса доступны любой компоненте-функции этого же класса. Например, функции func() доступны как компоненты a, ch, так и f,d.

Поясним теперь значение атрибутов доступа компонент:

- public определяет, что компонента может быть использована любой функцией;
- private - компонента может быть использована только компонентами-функциями и “друзьями” данного класса;
- protected - компонента может быть использована только компонентами-функциями и «друзьями» данного класса, а также компонентами-функциями и «друзьями» производных из него классов.

Поскольку данные объектов предостерегают от случайных изменений, а доступ и манипулирование данными стараются осуществлять через определенные функции, то на практике компоненты-данные классов обычно по умолчанию скрываются, а компоненты функции определяются с атрибутом public. Если предполагается некоторые компоненты данные использовать в производных классах, то им присваивается атрибут protected.

Дружественные функции. Иногда желательно, чтобы функция - не член класса, имела доступ к скрытым элементам класса. Это противоречит принципу инкапсуляции данных, поэтому вопрос об использование таких функций спорно. Основная причина использования таких функций состоит в том, что некоторые функции нуждаются в привилегированном доступе более, чем к одному классу. Такие функции получили название дружественных. Для того, чтобы функция - не член класса имела доступ к private-членам класса, необходимо в определение класса поместить объявление этой дружественной функции, используя ключевое слово friend. Объявление дружественной функции начинается с ключевого слова friend и должно находиться только в определении класса:

```
void func() {...}

class A
{
 //.....................
 friend void func();
 //.....................
 };
```

Дружественная функция, хотя и объявляется внутри класса (класс A в данном примере), функцией-членом не является. Поэтому не имеет значения, в какой части тела класса (private, public) она объявлена. Функция-член одного класса может быть дружественной для другого класса

```
class A
{.....................  int func();
  .....................
}; class B
{.....................
  friend int A :: func();
  .....................
};
```

Функция-член (func) класса A является дружественной для класса B. Если все функции-члены одного класса являются дружественными функциями второго класса, то можно объявить дружественный класс

friend class имя_класса;

Например,

```
class A
{
 .....................
}; class B
{.....................
 friend class A;
 .....................
};
```

Все функции-члены класса A будут иметь доступ к скрытым членам класса B.

Дружба классов не транзитивна: если X является другом класса Y, а Y - другом класса Z, это еще не означает, что X является другом класса Z.

Использование функций-друзей классов значительно увеличивает гибкость программ, разрабатываемых на языке C++. Они также удобны для перегрузки операций, например, только с помощью функций друзей можно перегрузить операции ввода-вывода.

Структуры и объединения. В C++ структуры (struct) и объединения (union) рассматриваютя как классы с умалчиваемым атрибутом доступа public, причем для объединений другие атрибуты запрещены, например:

```
struct X
{
int     a;
            char  ch;                   // a  и  ch  по
            private: float f;           умолчанию - public
            double d; protected:        // f и d - private
            void  func(void);           // func() - protected
};
```

Поскольку компоненты структур по умолчанию имеют атрибут public, то часто классы, имеющие только общие компоненты, описываются в виде структур. Например, описание

```
class X { public:
int k;
void funcX(int); };
```

эквивалентно

```
struct X
{ int k;
void funcX(int);
};
```

КОНСТРУКТОРЫ И ДЕСТРУКТОРЫ. ИНИЦИАЛИЗАЦИЯ ОБЪЕКТОВ. АВТОМАТИЧЕСКИЕ, ДИНАМИЧЕСКИЕ И СТАТИЧЕСКИЕ ОБЪЕКТЫ

Конструкторы и деструкторы. Среди компонент-функций класса важное место занимают конструкторы и деструкторы. Конструкторы определяют как объект создается и инициализируется, а деструктор - как объект разрушается. Если конструкторы или деструктор класса не определены явно, C++ генерирует их самостоятельно. При определении и разрушении объектов компилятор осуществляет вызов конструкторов и деструкторов автоматически. Имя конструктора совпадает с именем класса, а для деструктора спереди добавляется символ '~', например:

```
class X {
        int a; public:
        X (int i=0) {a = i; }  // конструктор класса X
         ~X() {  }  // деструктор класса X
};
```

Отметим некоторые особенности конструкторов и деструкторов:

- они не имеют возвращаемого значения (даже void);
- они не наследуются производным классом, хотя и могут быть вызваны из него;
- нельзя работать с их адресами;
- к конструкторам нельзя обращаться как к обычной функции, однако можно вызвать деструктор, указав полностью квалификационное имя:

```
main(void)
{
  X *p = new X(3);
 p -> X:: ~X();              // допустимый вызов деструктора
  p -> X:: X();              // недопустимый вызов конструктора
}
```

Автоматические, динамические и статические объекты. Конструктор вызывается всякий раз при создании объекта, а деструктор - при его разрушении. Поэтому важно знать какие объекты когда создаются и когда разрушаются. Объект может быть:

- автоматическим, когда он определяется в теле функции, уничтожается при выходе из блока;
- статическим, создается один раз при запуске программы и уничтожается при ее завершении;
- динамическим, создается в свободной области памяти с помощью операции new, уничтожается операцией delete.

Каждый конструктор начинает свою работу с проверки указателя this для определения находится объект в работе или нет. Если this равен null, то объект не в работе и конструктор вызывает new для выделения памяти объекту.

Статические объекты создаются путем вызова своих конструкторов до начала работы функции main(), автоматические - при входе в блок, в котором они определены. Компоненты базового класса образуются при создании объектов производного класса. Таким образом, зная момент создания объектов, программист может планировать действия, выполняемые даже до вызова функции main()!

Конструктор по умолчанию. Конструкторы, в отличие от деструкторов, могут принимать аргументы. Конструкторы, не имеющие параметров, называются конструкторами по умолчанию. Рассмотрим пример класса String, определяющего строку символов :

```
class String
{  char *str;  public:
  String()  // первый конструктор без аргументов
  {
  str = new char[1]; *str = 0;
  }
  String (char *); // объявление второго конструктора
  . . . . . .
};

// определение второго конструктора
String:: String (char *s)
{
  str = new char[strlen(s)+1];
  strcpy(str,s);
}

void main( void )
{
 String str1;
String str2("abc");
 .....
}
```

Первый конструктор в этом примере определяет пустую строку символов длиной 1, содержащую единственный символ конца строки. Второй конструктор позволяет создавать строку определенной длины и инициализирует ее константой или другой строкой.

Если конструктор класса не задан, компилятор автоматически формирует конструктор по умолчанию (без аргументов). Конструкторы, как и любые

функции языка С++, могут перегружаться (полиморфизм конструкторов). В случае определения в одном классе нескольких конструкторов вызов конкретного конструктора определяется компилятором из контекста вызова. Поэтому конструкторы должны отличаться либо количеством, либо типом своих аргументов.

Существует опасность неоднозначности вызова конструктора, когда используется передача аргументов по умолчанию, например:

```
class X
{
 . . . . . public:
 X(); // конструктор по умолчанию
 X(int i = 0); // конструктор с параметром
 . . . . . .
};

void main( void )
{
 X obj1; ... . . . .
}
```

Здесь явная неоднозначность, поскольку не ясно какой конструктор будет вызван:

- первый без аргументов или второй с аргументом i=0 ?

Конструктор копирования X(X &). В программах на языке С++ часто возникает необходимость копирования одного объекта в другой. Например, явное копирование объектов выполняется при использовании операции присваивания, а неявное копирование осуществляется, когда объект передается функции в качестве параметра, или возвращаемое функцией значение представляет собой некоторый объект. Если пользователем не определена процедура копирования, то компилятор осуществляет побитное копирование объектов:

```
class complex
{
 . . . . . .
 complex (double r=0, double i=0) // конструктор инициализации
 {
 re = x.re;
 im = x.im;
  }
 . . . . . .
};
```

```
main(void)
{
 complex x(2,1), // инициализация x списком
y = x;  // инициализация y
   // побитным копированием
……
}
```

Однако побитное копирование объектов, автоматически выполняемое компилятором, подходит далеко не для всех объектов. Рассмотрим класс String строк символов :

```
class String
{  char *str; public:
  String( char* s = “\0” )
  {
  str = new char[strlen(s)+1];
  strcpy(str,s);
  }
  ~String()
 {
  delete str;
  }
 void print(char *s)
 {
  cout << s << “: “ << str << “\n”;
  }
};
```

Выполнение следующей функции может привести к неприятностям: *void f()*

```
{
 String s1(“abc”),
 s2 = s1;
  s1.print(“s1”);
  s2.print(“s2”);
}
```

Здесь определяется строка s1 и инициализируется константой “abc”. Затем образуется вторая строка s2 нулевой длины. В результате инициализации, выполняемой компилятором побитно, значение указателя *str строки s2 станет равным указателю строки s1. При выходе из функции дважды будет вызван деструктор для s1 и s2, а поскольку указатели *str обеих объектов показывают на одну и ту же область памяти, то результат такого освобождения непредсказуем.

В подобных ситуациях пользователь должен сам определять способы копирования объектов. Отметим различие между присваиванием и инициализацией объекта. Присваивание, как правило, предполагает, что память объекту уже выделена и необходимо поместить туда некоторое значение. При инициализации в памяти образуется новый объект и определяется его значение. Здесь мы рассмотрим инициализацию объекта значением другого объекта, осуществляемую с помощью конструкторов копирования.

Конструктором копирования называют конструктор, который в качестве аргумента принимает ссылку на собственный класс. По этой причине для обозначения конструкторов копирования часто используется аббревиатура X(X&).

Для класса String конструктор копирования может быть определен следующим образом:

```
class String
{
 . . . . . .
 String(String& s)
 {
 str = new char[strlen(s.str)+1];
 strcpy(str,s.str);
  }
  ……….
};
```

В этом случае выполнение функции f() приведет к вызову двух конструкторов: для s1 - String(char*), а для s2 - String(String&). Причем конструктор для s2 предварительно отводит память нужного размера, куда и выполняется копирование строки s1. В результате каждый из указателей *str будет адресовать свою область памяти. Вызов деструкторов при завершении функции f() выполнится корректно.

Напомним, что конструкторы копирования вызываются компилятором неявно, когда объект является параметром функции или возвращаемым функцией значением, например: *main()*

```
{
  String s = “abc”;
 . . . . .  s = f(s);
 . . . . .
}
```

В данном примере вначале вызывается конструктор String(char*) для инициализации s строкой “abc”. Затем для копирования значения s в параметр x функции f() вызывается конструктор String(String&). Для возврата этой копии из

f() требуется еще один вызов конструктора String(String&), но на этот раз инициализируется временная переменная, которая затем присваивается s. Подобные временные переменные, автоматически формируемые компилятором, уничтожаются путем вызова деструктора при первой возможности. В нашем примере деструктор вызывается трижды перед завершением функции main().

Конструкторы копирования кроме аргумента X& могут также содержать и другие аргументы, однако аргумент X& должен быть первым, а для остальных аргументов обязательно должны быть определены их умалчиваемые значения.

Деструкторы. Противоположностью конструкторов являются деструкторы.

Деструкторы вызываются для освобождения членов объекта до разрушения самого объекта. Их главное назначение - освободить память, отводимую объекту и, возможно, выполнить некоторые другие действия.

Деструкторы не могут иметь параметров. Они вызываются автоматически при выходе переменной из объявленного контекста: для локальных переменных - когда перестает быть активным блок, в котором объявлена переменная; для глобальных переменных - при выходе из процедуры main(). Однако, это правило, не относится к указателям на объект: при выходе указателя объекта за пределы контекста автоматический вызов деструктора не производится. Для разрушения объекта в этом случае следует пользоваться операцией delete, например:

```
class complex
{ double *re, *im;
        public:
 complex(double r=0, double i=0)
 {
  re = new double(r);
  im = new double(i);
 }

 ~complex()
 {
 delete re;
        delete im;
 }
 void print(void) { ……}

};

void main()
{
 complex a(1.0);  // определение первого числа
 complex *pb = new complex(2.0);  // второго
```

```
 {
 complex c(3.0);   // третьего
 complex *pd = new complex(4.0); // четвертого
  c.print();
 pd->print();
 delete pd;  // косвенный вызов деструктора для pd
  }   // неявный вызов деструктора для c
a.print();
pb->print();
delete pb; // косвенный вызов деструктора для pb }     // неявный вызов деструктора для a
```

Здесь правило простое: если в программе объект образован с помощью операции new, то занимаемая им память должна освобождаться явным указанием операции delete.

При освобождении памяти, занимаемой объектами классов, следует обращать внимание на использование библиотечных функций abort() и exit(). Дело в том, что при выполнении функции abort() никакие деструкторы классов не вызываются, а при выполнении exit() вызываются деструкторы только глобальных переменных.

Имеется также два способа явного вызова деструктора объекта: прямо, задавая полное квалификационное имя деструктора, и косвенно, через операцию delete.

В качестве примера рассмотрим класс, определяющий двумерный массив целых чисел как массив указателей на одномерные массивы :

```
class ArrayTwo
{
  int **m;  int nstr, ncol;
  public:
  ArrayTwo (int n_str, int n_col); // объявление конструктора
  ~ArrayTwo ();   // объявление деструктора

  void print(char *);
};

// определение конструктора
ArrayTwo:: ArrayTwo (int n_str, int n_col)
{
 nstr = n_str;
 ncol = n_col;
 // выделение памяти массиву указателей
 m = (int **) calloc (n_str, sizeof (int *));
```

```
 // выделение памяти одномерным массивам
 for (int i=0; i < n_str; i++)
  m[i] = (int *) calloc (n_col, sizeof (int));
}

ArrayTwo:: ~ArrayTwo ()
{
 // освобождение памяти от одномерных массивов
 for (int i=0; i < n_str; i++)
 free (m[i]);
 // освобождение памяти от массива указателей free (m);
}

main(void)
{
  ArrayTwo a(2,4), b(6,3); // определение объектов

  a.print(“a”);
  a.ArrayTwo::~ArrayTwo();  // явный вызов деструктора

  b.print(“b”);
  b.ArrayTwo::~ArrayTwo(); // явный вызов деструктора }
```

В данном примере нельзя использовать деструктор, состоящий из одного вызова функции free(m), так как в этом случае память останется занятой одномерными массивами и доступ к ним будет потерян.

Может возникнуть вопрос: когда в программе необходимо определять деструктор класса явно и когда можно положиться на деструктор, формируемый компилятором автоматически? Здесь ответ таков: если размеры компонент-данных класса заранее известны и класс не содержит в качестве компонент данных указатели, которым в конструкторах выделяются блоки памяти, то можно принять деструктор по умолчанию. Во всех остальных случаях в программе необходимо явно определять деструктор класса, указав порядок освобождения памяти.

БАЗОВЫЕ И ПРОИЗВОДНЫЕ КЛАССЫ. ОГРАНИЧЕНИЕ ДОСТУПА. НАСЛЕДОВАНИЕ СВОЙСТВ И МОДИФИКАТОРЫ ДОСТУПА. МНОЖЕСТВЕННОЕ НАСЛЕДОВАНИЕ. КОНСТРУКТОРЫ БАЗОВЫХ И ПРОИЗВОДНЫХ КЛАССОВ

Наследование классов и производные классы. Наследование - это механизм создания нового класса на основе уже существующего.

В языке C++ для этого служит механизм наследования. Рассмотрим более общий формат определения класса:

class <имя_класса>[:<список_базовых_классов>] <описание_компонент_класса>

Если в определении класса присутствует список базовых классов, то такой класс называется производным (derived), а классы в базовом списке - базовыми (base) классами. В исходных предпосылках ООП предполагалась возможность наследования каждым производным классом только одного базового класса, что предполагает срогую иерархию классов. В языке C++ эти возможности расширены: каждый производный класс может наследовать несколько базовых классов, задаваемых списком базовых классов. При таком подходе к наследованию можно строить классы, отражающие структуру более сложную, чем простая иерархия.

Производные классы “получают наследство” - данные и методы своих базовых классов, и, кроме того, могут пополняться собственными компонентами (данными и собственными методами). Наследуемые компоненты не перемещаются в производный класс, а остаются в базовых классах. Сообщение, обработку которого не могут выполнить методы производного класса, автоматически передается в базовый класс. Если для обработки сообщения нужны данные, отсутствующие в производном классе, то их пытаются отыскать автоматически в базовом классе.

При наследовании некоторые имена методов (функций-членов) и (или) данных-членов базового класса могут быть по-новому определены в производном классе. В этом случае соответствующие компоненты базового класса становятся недоступными из производного класса. Для доступа из производного класса к компонентам базового класса, имена которых повторно определены в производном, используется операция разрешения контекста '::'.

Производному классу доступны все компоненты базовых классов, как если бы это были его собственные компоненты. Исключение составляют компоненты базового класса с атрибутами доступа private. При определении производного класса можно также влиять на атрибуты доступа компонент базовых классов. Для этого перед именем базового класса записываются спецификаторы доступа public или private. Спецификатор public используется для сохранения атрибутов доступа компонент базового класса в производном

классе без изменения. Спецификатор private делает недоступными компоненты базового класса для внешних функций, т.е. все компоненты базового класса с атрибутами public и protected принимают значение private в производном классе, и в следующем уровне иерархии классов не могут быть унаследованы другими производными классами. Использование спецификатора доступа protected перед именем базового класса запрещено.

Умалчиваемым значением спецификатора доступа для базовых классов считается private, а если в качестве базового класса выступает структура (struct), то умалчиваемым значением является public. Объединения не могут быть базовыми классами. Например:

```
class D : public B1, B2 {...};
```

В производном классе D компоненты класса B1 с атрибутами public и protected сохранят свое значение атрибута доступа, а компоненты класса B2 будут иметь атрибут доступа private и в последующем не смогут быть унаследованы другими производными классами.

Действие спецификаторов доступа в базовом списке можно скорректировать при помощи квалификационного имени в объявлениях производного класса, например:

```
class B {
        int a;  // a по умолчанию private
        public:
                int b, c; // b и c - public
};

class D: B  // b и c стали private, поскольку
{ // класс B по умолчанию private
        int d;
        public:
  B:: c;  // теперь c стала public
         int e;
};
```

Напомним, что все компоненты базовых классов с атрибутами доступа private недоступны и их атрибуты доступа не могут быть скорректированы в производном классе (для нашего примера это B::a).

Доступ к компонентам базовых классов из функций производного класса осуществляется так же, как если бы это были собственные компоненты, т.е. нет никаких различий в обращении к компонентам базовых или производного класса, например:

```
class B {
 protected:
int a;
 pubcic:
  B() { a = 5; }
};

class D: B
{
 int b;
 public:
  D() { b = 3; }
void print() ;
};

main(void)
{ D d;
  d.print();
}
```

Однако может возникнуть неоднозначность, если компоненты различных базовых классов имеют одинаковые имена. В этом случае для обращения к компонентам базовых классов следует использовать полное квалификационное имя, например :

```
class X
{
  public:
 int a;
  X() { a = 5; }
};

class Y {  public:
 double a;
 Y() { a = 10; }
};

class D: X,Y
{ public:
 void print(void) {...}
};

main(void)
{ D d;
  d.print();
}
```

C++ допускает одинаковые имена компонент в производном и базовых классах, однако следует быть внимательным при их использовании. Если возникают сомнения в правильности интерпретации имени компоненты, следует употреблять полное квалификационное имя. Компилятор, встретив любое имя в функции производного класса, в первую очередь ищет соответствующую компоненту в производном классе, а затем - в базовом.

Инициализация объектов при множественном наследовании Кроме рассмотренного ранее механизма инициализации объекта в теле конструктора, например:

```
class X
{  int a,b;
        public:
  X(int i, int j) { a = i; b = j; }
};
```

• язык C++ предусматривает также возможность присваивания значений компонентам

• класса через список инициализаторов. Список инициализаторов записывается перед телом конструктора, а каждый элемент списка представляет собой имя компоненты, за которым в круглых скобках следует инициализирующий параметр. Предыдущий пример в новой концепции может быть представлен следующим образом:

```
class X {
int a,b;
public:
  // инициализация компонент с помощью
  // списка инициализаторов
 X(int i, int j) a(i), b(j) {...}
};
```

В обоих случаях определение конструктора X инициализация объекта будет работать правильно, например выражение

```
 X x(1,2);
```

вызовет инициализацию x.a значением 1, а x.b - 2.

Эти два способа инициализации конструкторов можно комбинировать в пределах одного класса. Основное назначение второго способа (список инициализаторов) - обеспечить механизм передачи значений конструкторам базового класса из производного класса. В этом случае в список инициализаторов производного класса включаются конструкторы базовых классов (для обеспечения такой возможности последние должны быть

объявлены с атрибутами public или protected). Вначале инициализируются базовые классы в порядке их объявления, затем инициализируются компоненты производного класса, также в последовательности их объявления независимо от их расположения в списке инициализации. Рассмотрим пример :

```
class B1  // первый базовый класс
{  int x;  public:
  B1 (int i) { x = i;}  // инициализация присваиванием };

class B2 // второй базовый класс
{ int y; public:
  B2 (int j) y(j) { } // инициализация списком
   // инициализаторов
};

// производный класс class D: public B1, public B2
{  int a, b;  public:
 // комбинированная инициализация
 D(int i, int J): B1(i*3), B2(i+j), a(i)
 { b = j; } void print() {... }
};

main(void)
{
 D d(3,4);
  d.print();
}
```

Здесь конструктор класса B1 инициализируется по первому способу, а B2 - по второму. В инициализации конструктора класса D применяется комбинация этих способов: конструкторы B1, B2 и компонента a инициализируются по первому способу, а компонента b - по второму.

Если конструктор производного класса определяется вне тела класса, то список инициализаторов записывается при его определении.

Указатели на производные классы. C++ позволяет использовать указатель на базовый класс для обращения к компонентам базового класса, например:

```
class B        // базовый класс
{  int x;
 ...
};
```

```
class D: public B   // производный класс
{ int y;
 ..
};
. . . . . .
B *p; // указатель на базовый класс
D  d; // определение объекта производного
  // класса
p = &d;  // установка указателя базового класса
  // на объект производного класса,
  // такое допустимо в C++
. . . . . .
= p->x; // разрешенный доступ к компоненте
  // базового класса
= p->y; // так нельзя, поскольку y
  // компонента производного класса,
  // а p - указатель базового класса
```

Чтобы получить доступ к компонентам производного класса через указатель на базовый класс, необходимо выполнить преобразование типа вида:

```
j = ( (D *)p ) -> y; // теперь так можно
```

Таким образом, с помощью указателя на базовый класс можно обращаться к любым компонентам производного класса: как унаследованным из базового, так и специфичным для производного класса. Обратное утверждение неверно, т.е. нельзя с помощью указателя на производный класс обратиться к компонентам базового класса.

Отметим также, что инкремент и декремент указателя относится к его базовому классу, т.е. увеличивая указатель базового класса на единицу нельзя обратиться к следующему элементу производного класса.

Виртуальные базовые классы. В C++ запрещено множественное наследование одного и того же базового класса:

class B {...}; class D: B, B {...}; // недопустимо Однако это может произойти через косвенное наследование:

```
class B {...}; // базовый класс class X: B {...};
class Y: B {...};  // X и Y разные классы, однако они
 // наследуют один базовый класс B
class D: X, Y {...};  // производный от X и Y
```

В данном случае каждый объект класса D будет содержать два подобъекта класса B. Чтобы этого не происходило, класс B объявляют в классах X и Y как виртуальный со спецификатором virtual:

class B {...}; // базовый класс class X: virtual B {... }; // производный от B class Y: virtual B {...}; // производный от B class D: X, Y {...}; // производный от X и Y

Теперь объекты класса D будут содержать только один подобъект класса B.

В отношении виртуальных базовых классов в C++ имеется два ограничения:

- виртуальные базовые классы не могут точно инициализироваться списком инициализации конструктора, поэтому их конструкторы должны быть без аргументов, либо иметь умалчиваемые значения параметров;
- нельзя использовать указатель на виртуальный базовый класс для обращения к компонентам производного класса.

Конструкторы и деструкторы при наследовании. Базовый класс, производный класс или оба могут иметь конструкторы и/или деструкторы. Если и у базового и у производного классов есть конструкторы и деструкторы, то конструкторы выполняются в порядке наследования, а деструкторы - в обратном порядке. То есть если A - базовый класс, B - производный из A, а C - производный из B (A-B-C), то при создании объекта класса C вызов конструкторов будет иметь следующий порядок: конструктор класса A - конструктор класса B - конструктор класса C. Вызов деструкторов при разрушении этого объекта произойдет в обратном порядке: деструктор класса C – деструктор класса B - деструктор класса A. Понять закономерность такого порядка не сложно, поскольку базовый класс “не знает” о существовании производного класса, любая инициализация выполняется в нем независимо от производного класса, и, возможно, становится основой для инициализации, выполняемой в производном классе. С другой стороны, поскольку базовый класс лежит в основе производного, вызов деструктора базового класса раньше деструктора производного класса привел бы к преждевременному разрушению производного класса.

Конструкторы базовых классов не наследуются производным классом, однако могут из него вызываться. При образовании объектов производного класса первыми вызываются конструкторы базовых классов в порядке их объявления. Рассмотрим следующий фрагмент:

class D: X, Y {...};
. . . . . .
D one;

• для создания объекта one компилятором будет сформирована следующая последовательность вызовов конструкторов:

```
X();  // конструктор первого базового класса
Y();  // конструктор второго базового класса
D();  // конструктор производного класса
```

В свою очередь виртуальные классы имеют приоритет при вызове конструкторов базовых классов, например для следующего фрагмента

```
class B {...}; class X: virtual B {...}; class Y: virtual B {...}; class D: X, Y {...};
. . . . . .
D one;
```

будет сформирована последовательность вызовов:

```
B();  // конструктор виртуального базового класса
X();
Y();
D();
```

В случае множественных виртуальных базовых классов конструкторы запускаются в порядке их объявления, например:

```
class B1 {...}; class B2 {...}; class X: B2, virtual B1 { ...}; class Y: B2, virtual B1 {...}; class D: X, virtual Y {...};
. . . . . .
D one;
```

• образует следующий порядок вызова конструкторов:

```
B1();  // конструктор общего виртуального класса
B2();  //этот конструктор необходим для запуска Y()
Y(); //конструктор виртуального базового класса Y
B2();  //этот конструктор необходим для запуска X()
X(); //конструктор базового класса X
D(); //конструктор производного класса
```

В этом примере первым будет запускаться конструктор B1() как самый старший виртуальный базовый класс. Затем должен быть сформирован конструктор виртуального базового класса Y, но он содержит не виртуальный базовый класс B2, поэтому вначале запускается конструктор B2() для класса Y. Теперь может быть запущен конструктор Y(). Вновь запускается конструктор

B2(), но уже для класса X. И, наконец, последовательно запускаются конструкторы базового класса X и производного класса D.

Из этого примера можно также видеть, что конструкторы виртуальных классов (B1 и Y) запускаются только один раз, а конструкторы не виртуальных классов (B2) вызываются при создании каждого наследуемого объекта.

При разрушении объекта деструкторы вызываются аналогично конструкторам, но только в обратном порядке: вначале самый младший в иерархии деструктор, последним вызывается деструктор базового класса. Для нашего примера вызов деструкторов будет выглядеть следующим образом:

```
~D();
~X();
~B2();
~Y();
~B2();
~B1();
```

ПОЛИМОРФИЗМ. ПЕРЕГРУЗКА ОПЕРАЦИЙ. ОБЩИЕ ПРАВИЛА ПЕРЕОПРЕДЕЛЕНИЯ ОПЕРАЦИЙ. ДРУЖЕСТВЕННЫЕ ФУНКЦИИ И ОСОБЕННОСТИ ИХ ИСПОЛЬЗОВАНИЯ ДЛЯ ПЕРЕОПРЕДЕЛЕНИЯ ОПЕРАТОРОВ

Примером полиморфизма в языке C++ является перегрузка (overload) операций.

Она позволяет манипулировать объектами классов используя обычный синтаксис языка C. Для обеспечения такой возможности в перегружаемых операциях должно сохраняться количество аргументов соответствующей операции. Например, операция деления, обозначаемая “/”, в C имеет два аргумента, поэтому перегружаемая одноименная операция “/” также должна принимать два аргумента. Как правило, в качестве аргументов перегружаемых операций и возвращаемых ими значений выступают собственные классы. Так, если перегрузить операцию сложения для класса complex, то можно использовать естественный синтаксис языка C для их сложения:

```
class complex
{  double re, im; public:
. . . . . .
}
. . . . . .
complex  a, b, c;
. . . . . .
c = a + b;
```

Все перегружаемые операции имеют тот же приоритет и правила ассоциативности, что и предопределенные операции языка. По этой причине их можно использовать для построения цепочек операций, например: x = a + b + c + d;

Отметим, что в C++ нельзя перегрузить операции . :: .* и ?. Язык C++ не позволяет отличать постфиксную и префиксную формы перегруженных операций, поэтому, например, для перегруженной операции ++ выражения

++x и x++

имеют одно и то же значение. В общем случае предполагается, что перегруженные унарные операции используются в префиксной форме, поэтому для выражения “x++” компилятор выдаст предупреждение. Нельзя также определять новые лексические символы операций.

Язык C++ обеспечивает достаточную гибкость в перегрузке операций и их наследовании. Так, все перегруженные операции базовых классов, за исключением операции присваивания, наследуются производным классом. Перегруженная операция базового класса может затем вновь перегружаться в производном классе и т.д. Операции перегружаются с помощью функций-операторов, имеющих следующий формат:

<тип> operator <символ> (<список_аргументов>) {...}

где тип - это тип возвращаемого операцией значения, а символ - символьное обозначение перегруженной операции.

Функции-операторы могут вызываться таким же образом, как и любая другая функция. Использование операции - это лишь сокращенная запись явного вызова функцииоператора, например для комплексных чисел сокращенная запись

c = a + b;

- эквивалентна вызову функции

c = operator+(a,b);

В отношении действий, выполняемых перегружаемыми операциями, C++ не накладывает никаких ограничений. Можно, например, перегрузить операцию “+” для выполнения вычитания комплексных чисел, а “-” - для их сложения, но подобное использование механизма перегрузки операций не рекомендуется с точки зрения здравого смысла.

Сложные операции C++, равносильные комбинации нескольких операций, автоматически не раскрываются и, если пользователь их не определил, остаются компилятору неизвестными. Например, если для комплексных чисел перегружены операции сложения “+” и присваивания “=“, то запись вида

a += b;

- совсем не означает, что будет выполняться

a = a + b;

Чтобы можно было применять операцию «+=» с комплесными числами, она должна быть соответствующим образом перегружена в классе complex.

В C++ имеются различия при перегрузке операций с помощью функций-«друзей» и функций-компонент, поэтому рассмотрим их отдельно.

Перегрузка операций с помощью функций-«друзей» Функции-операторы, являющиеся “друзьями” класса, всегда имеют число аргументов, равное числу аргументов перегружаемой операции. Для унарных операций - это один аргумент, для бинарных - два и т.д. Отметим, что с помощью функций-”друзей” в Turbo C++ нельзя перегружать операции =, (), [] и ->. В качестве примера рассмотрим перегрузку операции сложения класса complex :

```
 class complex
{ double re, im;  public:
 complex (double r=0, double i=0)
  { re = r; im = i; } void print(char *s)
  {...); }
 friend complex operator + (complex, complex);
};
 complex operator + (complex a, complex b)
  { return complex(a.re + b.re, a.im + b.im); }
```

или :

```
class complex{...} complex operator + (complex a, complex b)
{  complex c;
  c.re = a.re + b.re;
  c.im = a.im + b.im;
  return c;
 }
```

Теперь для сложения комплексных чисел можно использовать естественный синтаксис языка C++:

```
  complex x(2.0,1.0), y(3.14,1.0), z;
 z = x + y;
```

Использование ссылок для перегрузки унарных операций

Если попытаться для класса complex аналогичным образом перегрузить унарную операцию инкремента следующим образом :

```
class complex
{
. . . . . public:
. . . . . .
 friend complex operator ++(complex);
};
complex operator ++(complex a)
 { return complex(a.re++, a.im++); }
```

или :

```
class complex{...}; complex operator ++(complex a)
{
 a.re++,
 a.im++;  return a;
}
```

• то можно убедиться, что перегруженная операция работать не будет. Дело в том, что в С++ аргументы передаются по значению и, следовательно, возвращаемая величина не изменяет своего значения. Можно использовать в качестве аргумента указатель :

```
class complex
{
. . . . . public:
 friend complex operator ++ (complex *);
};
 complex operator + (complex *p)
{ p->re++, p->im++; return *p;
}
```

Однако в этом случае компилятор выдаст ошибку, поскольку язык С++ требует, чтобы операнд операции “++” имел тип класса объекта. Выходом из этой ситуации является использование ссылок в качестве аргумента функции-оператора:

```
class complex
{
. . . . . public:
  friend complex operator ++(complex &);
};
 complex operator + (complex &r)
 {
 r.re++,
 r.im++; return r;
 }
```

Невозможность перегрузки унарных операций функциями «друзьями» послужило одной из причин введения ссылок в язык С++. Ссылки в качестве аргументов перегружаемых операций позволяют передавать реализующей ее функции не копии объектов, а их адреса, что позволяет изменять значения этих аргументов. Кроме того, использование ссылок в качестве аргументов для

больших объектов позволяет значительно сократить объем копируемой информации, что приводит к повышению эффективности программы.

Рассмотрим теперь вопрос каким должно быть возвращаемое значение функцииоператора. Оно не может быть ссылкой на автоматическую переменную, так как после выхода из функции возвращаемое значение указывает на несуществующую переменную. Оно также не может быть статической переменной. Допускается ссылка на переменную в “куче”, однако это усложняет программирование. Лучшим решением с точки зрения простоты реализации и эффективности считается копирование возвращаемого значения . Можно также возвращать ссылку, но только на существующий объект, например, когда эта ссылка передается функции-оператору в качестве параметра:

```
 class complex
{
 . . . . . public:
 . . . . . .
 friend complex& operator ++(complex& r);
};
complex& operator ++(complex& r)
{
 r.re++;
 r.im++; return r;
}
```

Перегрузка операций с помощью компонент-функций. Для функций-операторов, являющихся компонентами класса, первым аргументом по умолчанию является указатель this на тот объект, к которому она относится. Поэтому число аргументов для таких операторов на единицу меньше, по сравнению с числом аргументов перегружаемой операции. Например, для бинарных операций - один аргумент, для унарных - аргументы отсутствуют и т.д. Перегрузка операций + и ++ для класса complex, реализованная с помощью операторовкомпонент имеет вид:

```
class complex
{
 . . . . . public:
 complex operator + (complex b); complex operator ++ (void);
};
complex complex:: operator + (complex b) { return complex(re + b.re, im + b.im); };
 complex complex:: operator ++()
 { return complex(re++, im++); };
```

Отметим, что при перегрузке операций с помощью компонент функций широко используется указатель this. Однако при этом следует соблюдать особую осторожность. Соблазнительной может показастья идея использовать для получения результата вместо промежуточной переменной первый аргумент, например:

```
class complex
{
. . . . . public:
 complex operator +(complex b);
 complex operator ++(void);
};
complex complex:: operator + (complex b)
{ re += b.re; im += b.im; return *this;
};
complex complex:: operator ++ ()
{ re++; im++; return *this;
};
```

В последнем примере операция инкремента работает правильно, а операция сложения имеет побочный эффект, увеличивая значение первого операнда.

Напомним, что с помощью функций-операторов, являющихся компонентами класса, можно перегружать операции =, (), [] и ->, которые запрещены для перегрузки функциями «друзьями».

Может возникнуть вопрос в каких случаях при перегрузке операций следует использовать функции-«друзья», а в каких - функции-компоненты? Рассмотрим пример:

```
main(void)
{
  complex a(2.0,1.0), b(3.14,1.0), c; double d = 1.5; c = a + b; c = a + d; c = d
+ b;
. . . . .
}
```

Выражение *c = a + b* будет верным, поскольку a и b оба типа complex, для которого переопределена операция “+”. Выражение *c = a + d* также будет верным, поскольку оно интерпретируется как *a.operator+(complex(d))* Однако выражение *c = d + b* вызовет ошибку, если операция “+” перегружена с помощью функции-компоненты. Дело в том, что d + b по определению эквивалентно *d.operator+(b)*, но d не является объектом класса и не имеет компонент! В случае

же перегрузки “+” как функции-друга выражение верно, поскольку здесь будет вызван конструктор инициализации и *d + b* будет эквиваленто *complex(d) + b*

Отсюда можно сделать следующий вывод: в том случае, если предполагается неявное преобразование операндов перегружаемой операции, то реализующую ее функцию лучше сделать “другом”. К таким операциям можно отнести арифметические и логические операции

(+, -, || и т.д.).

Если же операция изменяет состояние объекта, а это можно сделать только для ранее созданного объекта, то реализующая операцию функция должна быть компонентой класса. К подобным операциям относятся =, *=, ++ и др.

При прочих равных условиях для перегрузки операций чаще используются функциикомпоненты. Это связано с тем, что они имеют на один аргумент меньше, допускают использование неявного параметра this, не чувствительны к модификациям функций преобразования типа и, наконец, описание функции-компоненты, как правило, короче описания функции-друга.

Многократная перегрузка операций

C++ позволяет несколько раз перегружать одну и ту же операцию в пределах одного класса. Важно только, чтобы однозначно определялся вызов необходимой операции, т.е.

функции-операторы должны различаться количеством или типом своих аргументов, например:

```
class String
{  char *str;  public:
  String (char * s = “\0”) // конструктор
  { str = new char [strlen (s) +1]; strcpy (str,s); }
 String operator + (String t) // первая перзагрузка +
 { return String(strcat(str,t.str)); }
 String operator + (char *s) // вторая перзагрузка +
 { return String(strcat(str,s)); }  void print(char* s)
 { cout << s << “: “ << str << “\n”; }
};
main(void)
{
  String a(“Минск”), b(“ - город”), c;
 c = a + b + “ -герой!”; c.print(“c”);
}
```

В данном примере дважды перегружена операция + для класса String. В первом случае она позволяет нам связывать объекты типа String между собой, а во втором - тип String со строкой символов. Это дает возможность в одном

выражении записывать как объекты типа String, так и обычные строки типа (char *).

Перегрузка операции присваивания =

Операция присваивания перегружается обычным образом. Особенностью этой операции является то, что она не может быть унаследована производным классом. Кроме того, в C++ операцию “=“ можно перегружать только с помощью функции-компоненты. Например:

```
class complex
{
. . . . . public:
  complex& operator =(complex &);
};
complex& complex:: operator =(complex& b)
{ re = b.re; im = b.im; return *this;
}
```

Здесь первый операнд передается через указатель this, принимает значение второго операнда, полученного в качестве аргумента, и возвращается с помощью выражения *this. Не смотря на то, что в этом примере изменяется значение первого аргумента, это не является ошибкой, поскольку его значение как раз и нужно изменить. Данный пример также демонстрирует те немногие случаи, когда в программе необходимо явно использовать указатель this.

Если для некоторого класса X операция присваивания не определена, то при необходимости она формируется компилятором C++ автоматически в виде:

```
X & X:: operator = (const X & <источник>)
{
  // покомпонентное присваивание
  return *this;
}
```

Рассмотрим на примере класса String более сложный случай перегрузки операции “=“, чем простое покомпонентное присваивание. Первый вариант перегрузки операции “=“ для класса String на первый взгляд вообще не возвращает никакого значения

```
class String
{
. . . . . public:
void operator = (Srting &t){
if (this == &t)  return;  // копирование в ту же строку  delete str; // удаляется старая строка  str = new char[strlen(t.str)+1]; // образуется новая strcpy(str,t.str); // в нее копируется значение
 }
}
```

Здесь вначале проверяется, чтобы строка не дублировала саму себя. Затем уничтожается старое значение строки, отводится память необходимого объема и туда копируется принимаемый аргумент.

Более корректной реализацией перегрузки операции “=“ будет явный возврат сформированного объекта с помощью указателя this:

```
class String
{
. . . . . public:
String& operator = (Srting &t)
{
if (this == &t)  return *this;
delete str; // удаляется старая строка str = new char[strlen(t.str)+1]; // образуется новая strcpy(str,t.str); // в нее копируется значение return *this;
 }
}
```

Если сравнить последнюю реализацию с конструктором класса String, то можно увидеть их значительное сходство. Поэтому можно воспользоваться уже готовым конструктором для формирования возвращаемого значения:

```
class String
{
. . . . . public:
String operator = (Srting &t)
{
if (this == &t) return *this; delete str; // удаляется старая строка return String(t.str);
 }
}
```

Заметим, что здесь изменился тип возвращаемого значения, поскольку конструктор String::String() возвращает объект, а не ссылку на него.

Таким образом, для простых типов (например, complex) операцию присваивания можно не перегружать, в этом случае она формируется

компилятором автоматически, осуществляя покомпонентное присваивание. Если объекты сложные и покомпонентное присваивание для них не работает, следует операцию присваивания перегружать явно. В качестве возвращаемого значения перегруженной операции присваивания чаще используется ссылка на объект . Возможно в возвращаемом выражении обращаться к конструктору класса. В последнем случае возвращаемым значением будет сам объект.

Перегрузка операций [], () и ->

Прежде всего отметим, что эти операции перегружаются только с помощью компонентфункций и их нельзя перегрузить функциями- “друзьями”.

Начнем с рассмотрения перегрузки операции []. Предположим, что мы хотим обращаться к действительной и мнимой частям некоторого комплексного числа h как к элементам массива, т.е. h.re соответствует h[0] и h.im соответствует h[1]. Для этого можно перегрузить операцию [] выделения элемента массива для класса complex следующим образом :

```
class complex
{ double re, im; public:
  . . . . . .
 double & operator [] (int i)
 { return *(&re + i);}
};
main(void)
{
  complex one(1.0, 0.0); cout << “one.re=“ << one[0] << “one.im=“ << one[1] << “\n”;
}
```

Перегрузка операции [] для класса X позволяет всякое выражение вида ob[i], где ob - объект класса X, интерпретировать как обращение к функции ob.operator[](i). Операция вызова функции вида:

<имя>(<список_аргументов>);

- в языке C++ рассматривается как бинарная операция, где первым операндом является <имя>, а вторым - <список_аргументов>. При перегрузке операции () список аргументов вычисляется и проверяется в соответствии с обычными правилами передачи аргументов. В общем случае список аргументов может быть пуст.

Перегрузка операции () позволяет рассматривать выражение вида

ob(<список_аргументов>)

- как обращение к функции

ob.operator()(<список_аргументов>).

Рассмотрим перегрузку операции () для класса Array. Иногда целесообразно считать, что индекс массива начинается не с нуля, а с единицы (подобно тому, как это реализовано в языке PL/1). Для этого перегрузим операцию () :

```
class Array
{
  . . . . . . . public:
  int operator()(int i) { return m[i-1]; }
};
void main()
{
 Array x(5); cout << "x= "; for (int i=1; i<=5; i++) cout << x(i) << " "; cout
<< endl;
}
```

Операцию () обычно перегружают для операций, требующих большого числа операндов, для классов с единственно возможной операцией, а также когда некоторая операция используется особенно часто. Перегружаемая операция () не может быть статической компонентой-функцией класса.

C++ предоставляет пользователю возможность выполнять некоторую предварительную обработку до обращения к компонентам классов. С этой целью используется перегузка операций ->, * и &. Операция -> рассматривается как унарная операция и ее перегрузка позволяет выражение вида

ob->m трактовать как

(ob.operator->())->m.

Причем функция-оператор, реализующая операцию -> должна либо возвращать указатель на объект данного класса, либо возвращать объект этого класса, т.е. ее описание для класса X должно иметь вид:

*X *operator -> () {...}* или

X operator -> () {...}.

Рассмотрим случай, когда возвращаемым значением перегуженной операции -> является тип этого же класса:

```
class X { public: int a;
 X(int i) { a = i; }
 X operator ->()
 {
 cout << "Доступ к компонентам класса X\n"; return *this;
  }
```

```
};
main(void)
{
  X x(5);
  X *px = new X(10); cout << "a= " << x->a << "\n"; // перегруженная операция cout << "a= " << px->a << "\n"; // предопределенная операция }
```

В данном примере перед обращением к компоненте в стандартный поток выводится сообщение.Наибольший интерес представляет случай, когда возвращаемым значением перегруженной операции -> класса X является указатель на некоторый другой класс Y. В этом случае операция -> вначале применяется к своему левому операнду для получения указателя p на класс Y, а затем указатель p используется как левый операнд бинарной операции -> для доступа к компоненте класса Y, например:

```
class Y {  public:
 int b;
 Y(int j) { b = j; }
}; class X { int a;  public:
 Y *p;
X(int i, int j)
 { a = i; p = new Y(j);
 }
 Y* operator ->()
 {
 cout << "Доступ к компонентам класса Y\n"; return p;
 }
};
main(void)
{
 X x(3,5);
 cout << "b= " << x->b << "\n"; // перегруженная операция cout << "b= " << x.p->b << "\n";  // базовая операция
}
```

Если класс Y в свою очередь имеет перегруженную операцию ->, то p будет использован в качестве левого операнда унарной опеации -> и вся процедура повторится для класса Y.

Унарные операции * и & перегружаются аналогично, причем сохраняется соответствующая семантика между операцией -> и операциями * и &.

Перегрузка операций new и delete

Язык C++ предоставляет две возможности для перегрузки операций new и delete. Они могут быть перегружены глобально, т.е. любое обращение в программе к new и delete будет вызывать перегруженные операции, или в

пределах класса. В последнем случае вызов перегруженных операций будет осуществляться только для объектов данного класса, а все другие обращения к new и delete будут вызывать переопределенные операции C++. Глобальная перегрузка операций new и delete имеет вид:

```
void * operator new (size_t size)
{
  // выполнение необходимых выделений памяти retutn p;
}
void * operator delete (void *p)
{
  // освобождение памяти, указываемой p
}
```

Здесь size_t - целый тип, определенный в stdlib.h; p - указатель на выделенную память.

Известно, что стандартная операция new при выделении память не обнуляет и там находится случайный “мусор”. Следующий пример показывает вариант глобальной перегрузки предопределенных операций new и delete, обнуляющих память при ее выделении:

```
extern void * operator new(size_t size)
{
 return calloc(1, size);
};
extern void operator delete(void *p)
{
 free ((char *)p);
};
main(void)
{
 int *m = new int[5]; cout << “m= “; for (int i=0; i < 5; i++)
 {
 cout << m[i] << “ “; cout << “\n”;
 }
}
```

Теперь всякое обращение к операции new будет возвращать область памяти, заполненную нулями, а операция delete позволяет освободить выделенный блок с помощью функции free().

Чтобы перегрузить операции new и delete только в отношении объектов некоторого класса X, они должны быть определены в теле этого класса. Перегруженные операции new и delete будут вызываться только для объектов

класса X. Для других объектов будут вызываться глобальные (либо собственные) new и delete.

Если в вызове операции new указан любой другой тип, отличный от X, то будет вызываться глобальная операция new, даже если вызов осуществляется в теле компонентыфункции класса X. Так, например, следующая попытка перегрузить операцию new для класса Array будет безуспешна, поскольку в конструкторе вызывается глобальное new :

```
class Array
{ int size; int *m;  public:
  Array (int n)   // конструктор
  { size = n; m = new int[size]; } // вызывается глобальная
   // new, которая и работает
  ~Array() { delete m; } // деструктор void * operator new(size_t n); void operator delete(void *p);
};
 // переопределение операции new
void *Array:: operator new(size_t n)
{ int *p;
 p = (int *) malloc(n*sizeo(int)); // выделяем и
 for (int i=0; i<n; i++)  // заполняем память  p[i] = i; return (void *)p;
}
 // переопределение операции delete
 void *Array:: operator delete(void *p)
{
 free(p);
}
main(void)
{
 Array a(5); // объект класса Array
 a.print();  // вывод заполненного массива Array *pb = new Array(7); // указатель на объект  pb->print();  // вывод массива }
```

В качестве примера перегрузки операции new только для отдельного класса рассмотрим класс Point, определяющий точку на экране дисплея. Каждой точке соответствует блок, представляющий собой структуру из двух целых чисел. Множество таких блоков объединяется в один массив blocks, который расположен в статической области памяти. Каждое обращение к Point:: new возвращает указатель на свободный блок памяти, определяемый переменной top:

```
const max = 512;
class Point
{ int x,y;
```

```
 static stract Block
{  // блок из двух точек
 int xy[2];
} blocks[max];  // массив блоков точек static int top;  // счетчик блоков
public:
Point(int a=0, int b=0) // конструктор
{ x=a; y=b; }
void * operator new(size_t n)
{
n = n; if (top < max) return blocks + top++; else  return 0;
 }
void print(char *s)
{
cout << s << "= (" << x << "," << y << ")\n";
 }
};
main(void)
{
 Point *pone = new Point(1,1); pone->print("one");
 Point *ptwo = new Point(2,2);  ptwo->print("two");
}
```

Отметим, что в C++ функции-операторы new и delete некоторого класса по умолчанию имеют спецификатор static, поэтому они не могут быть виртуальными функциями.

Преобразование типа

Механизм перегрузки операций можно применять для преобразования типов данных. Функции-операторы, обеспечивающие преобразование типов, должны быть компонентами класса и имеют следующий вид:

operator <тип>();

где тип - спецификация типа, которая является результатом преобразования.

В качестве примера введем в класс String функцию-оператор для преобразования объектов класса String к типу char*:

```
class String
{ char *str;  public:
. . . . . .
operator char *()
{   // объявление оператора // преобразования типа  char *p = new
char[strlen(str)+1];  strcpy(p,srt);  return p;
 }
};
void main()
```

```
{
 String *s("Минск - город-герой!\n"); char *t;  t = s;  // теперь так можно !
 cout << t;
}
```

Напомним, что преобразования типов могут выполняться в конструкторах, конструкторах копирования и в операциях присваивания. Поэтому при разработке классов сложных объектов следует предусматривать весь набор возможных преобразований типа. Для этого определяются конструктор копирования, перегружается операция присваивания и определяются необходимые преобразования типов.

ВИРТУАЛЬНЫЕ ФУНКЦИИ. ОСОБЕННОСТИ РАЗРАБОТКИ И ИСПОЛЬЗОВАНИЯ ВИРТУАЛЬНЫХ ФУНКЦИЙ. ЧИСТЫЕ ВИРТУАЛЬНЫЕ ФУНКЦИИ. АБСТРАКТНЫЕ КЛАССЫ

Виртуальные функции. Виртуальные функции - специальный вид функций-членов класса. Виртуальная функция отличается об обычной функции тем, что для обычной функции связывание вызова функции с ее определением осуществляется на этапе компиляции. Для виртуальных функций это происходит во время выполнения программы. Для объявлениявиртуальной функции используется ключевое слово virtual. Функция-член класса может быть объявлена как виртуальная, если

- класс, содержащий виртуальную функцию, базовый в иерархии порождения;
- реализация функции зависит от класса и будет различной в каждом порожденном классе.

Другими словами, виртуальная функция - это функция, которая определяется в базовом классе, а любой порожденный класс может ее переопределить. Виртуальная функция вызывается только через указатель или ссылку на базовый класс. Определение того, какой экземпляр виртуальной функции вызывается по выражению вызова функции, зависит от класса объекта, адресуемого указателем или ссылкой, и осуществляется во время выполнения программы. Этот механизм называется динамическим (поздним) связыванием или разрешением типов во время выполнения.

Указатель на базовый класс может указывать либо на объект базового класса, либо на объект порожденного класса. Выбор функции-члена зависит от того, на объект какого класса при выполнении программы указывает указатель, но не от типа указателя. При отсутствии члена порожденного класса по умолчанию используется виртуальная функция базового класса.

```
// Выбор виртуальной функции #include <iostream.h>
class X {  protected:  int i; public:
  virtual void print() {cout << “класс X: “ << i;}
};
class Y : public X
{  public:
  void print() {cout << “класс Y: “ << i;}
};
main()
{ X x;
*px=&x; // Указатель на базовый класс
y;
  x.i=10;
  y.i=15; px->print(); // класс X: 10 px=&y;
```

```
  px->print(); // класс Y: 15
}
```

В каждом случае выполняется различная версия функции print(). Выбор динамически зависит от объекта, на который ссылается указатель. В терминологии ООП <объект посылает сообщение print и выбирает свою собственную версию соответствующего метода>. Виртуальной может быть только нестатическая функция-член класса. Для порожденного класса функция автоматически становится виртуальной, поэтому ключевое слово virtual можно опустить.

Существует ограничение, связанное с применением виртуальных функций: деструкторы класса могут быть виртуальными, а конструкторы - нет.

Рассмотрим программу, которая вычисляет площади различных фигур. Каждая фигура может порождаться от базового класса figure

```
class figure
{ protected: double x, y;  public:
 virtual double area() {return(0);} // по умолчанию
};
class rectangle : public figure
{ double height, width;  public:
  double area() {return(height*width);}
};
class circle : public figure
{  double r;  public:
  double area() {return(PI*r*r);}
};
```

При такой иерархии порожденные классы соответствуют типам фигур. Вычисление площади является локальной ответственностью порожденных классов. Вычислить общую площадь фигур проекта можно следующим образом:

```
figure *p[N]; // массив указателей на базовый класс
  // элементы массива могут ссылаться на
  // различные производные классы
for(i=0; i<N; i++) double tot_area+=p[i]->area();
```

Код пользователя не нуждается в изменении, даже если к системе добавляются новые типы фигур.

Абстрактные базовые классы. Базовый класс иерархии типа обычно содержит ряд виртуальных функций, которые обеспечивают динамическую типизацию. Часто в самом базовом классе сами виртуальные функции фиктивны

и имеют пустое тело. Определенное значение им придается лишь в порожденных классах. Такие функции называются чистыми виртуальными функциями.

Чистая виртуальная функция - это функция-член класса, тело которой не определено. В базовом классе такая функция записывается следующим образом:

virtual прототип функции = 0;

Например

virtual void func() = 0;

Чистая виртуальные функции используются для того, чтобы отложить решение задачи о реализации функции на более поздний срок. В терминологии ООП это называется отсроченным методом. Класс, имеющий по крайней мере одну чистую виртуальную функцию, называется абстрактным классом. Для иерархии типа полезно иметь абстрактный базовый класс. Он содержит общие свойства иерархии типа, но каждый порожденный класс реализует эти свойства по-своему.

ПАРАМЕТРИЗАЦИЯ КЛАССОВ И ШАБЛОНЫ ФУНКЦИЙ. ОПЕРАТОР «TEMPLATE». МЕТОДЫ ИСПОЛЬЗОВАНИЯ ШАБЛОНОВ

Шаблоны функций.

Рассмотрим некоторую функцию, например, max(x,y), аргументы которой могут быть любого типа. Один путь решения этой задачи - перегузка функций:

int max(int x, int y) {retutn (x > y) ? x : y;}; long max(long x, long y) {retutn (x > y) ? x : y;};

. . . . . .

Однако это требует своего кода для каждой перегруженной функции. Общим здесь является только имя функции. Второй путь - использование макросов:

#define max(x,y) ((x>y)?x:y)

В этом случае не работает механизм проверки типов, что может привести к попытке сравнения несравнимых типов (например, int и struct). Далее, замена в исходном тексте макроса выполняется даже тогда, когда это не требуется, например:

```
class My_class
{
  . . . . . int max(int,int);  // синтаксическая ошибка !
  . . . . . .
}
```

Решением проблемы может являться использование шаблонов (template). При таком подходе тип аргументов функции обозначается некоторой буквой. В определении функции эта буква используется в качестве типа данных. В последующем компилятор автоматически сгенерирует необходимую функцию, соответствующую типу передаваемых аргументов :

#include <iostream.h> template <class T> // определение буквы шаблона T max(T x,T y) // определение функции-шаблона

```
{
  return (x > y) ? x : y;
}
void main()
{ int  a=3, b=5; float x=2.1, y=0.15; long  r=754, t=3561; cout << “max(a,b)= “
```

```
  << max(a,b) // генерируется max() для int
  << “max(x,y)= “
  << max(x,y) // генерируется max() для float
  << “max(r,t)= “
  << max(r,t) // генерируется max() для long
  << endl;
}
```

В приведенном примере функция-шаблон max() применима к любым типам, для которых определена операция “>“. Можно отвергнуть генератором функции-шаблона для специальных типов, явно определив функцию для конкретного типа:

```
char * max(char *s, char *t)
{
  return (strcmp(s,t) > 0) ? s : t; }
```

Теперь обращение к функции max(s,f) для строковых аргументов не будет генерировать новую функцию, а сформирует вызов необходимой функции:

```
#include <iostream.h> #include <string.h> template <class T>  // определение буквы шаблона T max(T x,T y)  // определение функции-шаблона
{
  return (x > y) ? x : y;
}
char * max(char *s, char *f) // определение max для (char*)
{
 return (strcmp(s,f) > 0) ? s : f;
}
void main()
{  int  a=3, b=5;  float  x=2.1, y=0.15;  long  r=754, t=3561;  char s[]=“Минск”,f[]=“Москва”;  cout << “max(a,b)= “
  << max(a,b) // генерируется max() для int
  << “max(x,y)= “
  << max(x,y) // генерируется max() для float
  << “max(r,t)= “
  << max(r,t) // генерируется max() для long
  << “max(s,f)= “
  << max(s,f) // вызывается max() для char*  << endl;
}
```

При использовании функций-шаблонов необходимо внимательно следить за типом передаваемых аргументов, поскольку для функций-шаблонов предопределенные преобразования типов не выполняются, например:

```
#include <iostream.h> template <class T>  // определение буквы шаблона T
max(T x,T y)  // определение функции-шаблона {
 return (x > y) ? x : y;
}
void main()
{ int i=5; char c=3; int x1,x2,x3,x4;
 x1=max(i,i);  // все в порядке x2=max(c,c);  // также все в порядке
x3=max(i,c);  // не работает, поскольку аргументы
  // разного типа
 x4=max(c,i);  // не работает по той же причине cout << "x1= " << x1 <<
"x2= " << x2
  << "x3= " << x3 << "x4= " << x4 << endl;
}
```

Чтобы допустить предопределенные преобразования типов при обращении к функции-шаблону, достаточно явно определить ее прототип:

```
#include <iostream.h> template <class T>  // определение буквы шаблона T
max(T x,T y)  // определение функции-шаблона
{
  return (x > y) ? x : y;
}
int max(int,int);  // определение прототипа max() void main()
{ int i=5; char c=3; int x1,x2,x3,x4;
 x1=max(i,i);  // все в порядке x2=max(c,c);  // также все в порядке
x3=max(i,c); // теперь уже работает x4=max(c,i); // теперь уже работает  cout
<< "x1= " << x1 << "x2= " << x2
 << "x3= " << x3 << "x4= " << x4 << endl;
}
```

Ограничением на использование функций-шаблонов является то, что для них нельзя указывать умалчиваемые значения аргументов.

Шаблоны классов

Шаблон класса (class template), называемый также обобщением классов (generic class) или генератором классов (class generator), задает образец для определений классов. Рассмотрим шаблон класса Array, который позволяет определить классы массивов с любым типом элементов:

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

```
template <class T> // определение буквы шаблона class Array
{   // определение шаблона классов Array  T *m;  // тип массива задается шаблоном  int size;  public:
 Array(int n=0)
 {
 size = n;
  m = new T[size]; // тип памяти задается шаблоном
  }
  ~Array() {delete m;} void set(int i, T x) { *(m+i) = x; } T & operator [] (int i) { return m[i]; } void print(char*);
};
// определение внешней функции template <class T> // шаблона классов void Array<T>:: print(char *s)
{
 cout << s << ":"; for (int i=0; i<size; cout << m[i++] << " "); cout << "\n";
};
```

Теперь можно определять объекты различных классов Array, задавая в качестве параметра шаблона конкретный тип элементов массива:

```
void main()
{
  Array<int>  mi(5); // определение класса Array для int Array<float> mf(3); // определение класса Array для float for (int i; i<5; i++)  mi.set(i,i);  mi.print("mi"); for (i; i<3; i++)  mf.set(i,(float)i);  mf.print("mf");
}
```

В общем случае определение шаблона классов имеет вид:

```
template <список_аргументов>
class имя_класса
{
  // определение компонент класса
};
```

Внешние функции шаблона классов определяются следующим образом:

```
template <список_аргументов> имя_класса<список_аргументов>:: имя_функции() {...};
```

Объекты конкретного класса задаются следующим образом: *имя_класса<параметры_шаблона> список_объектов;*

Аргументы шаблона классов могут быть двух видов: типовые и не типовые. Типовым аргументам предшествует ключевое слово class, они, как правило, обозначаются буквой и представляют параметр типа, т.е. изменяемые типы данных. Не типовые аргументы шаблона классов аналогичны параметрам конструктора класса, для них можно задавать умалчиваемые значения параметров, определяемые константными выражениями. При этом каждый генерируемый класс будет получать собственную копию статических компонент.

В следующем примере тип элементов массива и размер массива передаются в качестве параметров шаблона классов:

```
#include <iostream.h>
template <class T, int n=1> // определение шаблона class Array
{  // определение шаблона классов Array T *m;  // тип массива задается шаблоном  int size; public:
 Array()
 {
 size = n; // размер массива задается параметром
  // шаблона
  m = new T[size]; // тип памяти задается шаблоном
  }
  ~Array() {delete m;}
  T & operator [] (int i) {return m[i];} void print(char*);
};
void Array<T>:: print(char *s)
{
  cout << s << ":"; for (int i=0; i<size; cout << m[i++] << " "); cout << "\n";
};
void main()
{
  Array<int>  mi(5); // определение класса Array для int Array<float> mf(3); // определение класса Array для float for (int i; i<5; mi[i] = i++); mi.print("mi"); for (i; i<5; mf[i] = (float)i++); mf.print("mf"); }
```

Если автоматическое определение класса пользователя не устраивает, можно явно определить шаблон класса подобно тому, как это делается для шаблонов функций:

```
#include <iostream.h> #include <string.h>
template <class T> // определение буквы шаблона class Array
{  // определение шаблона классов Array
  . . . . . .
};
```

```
class Array<char*>
{  // определение класса Array для строк char  **m; // символов  int size;
public:
 Array<char*>(int n=0);
 ~Array();
  void set(int i, char *s) { strcpy(m[i],s); }  void print(char*);
};
Array<char*>:: Array(int n)
{
 size = n;
 m = new char*[size]; for (int i=0; i<size; i++) m[i] = new char[81];
}
Array<char*>:: ~Array()
{
  for (int i=0; i<size; i++)  delete m[i]; delete m;
}
void Array<char*>:: print(char *s)
{
 cout << s << ":"; for (int i=0; i<size; i++) cout << m[i] << " ";  cout <<
"\n";
};
void main()
{
 Array<int> mi(5); // определение класса Array для int
 Array<float> mf(3); // определение класса Array для float Array<char*>
mstr(2); // определение класса Array для char* for (int i=0; i<5; i++) mi.set(i,i);
mi.print("mi");      for  (i=0;  i<3;  i++)      mf.set(i,(float)i);  mf.print("mf");
mstr.set(0,"Минск"); mstr.set(1,"Москва");  mstr.print("mstr");
}
```

ПОТОКИ ВВОДА-ВЫВОДА. ИЕРАРХИЯ КЛАССОВ ВВОДА-ВЫВОДА. ОСНОВНЫЕ ФУНКЦИИ. ФОРМАТИРОВАННЫЙ И НЕФОРМАТИРОВАННЫЙ ВВОД-ВЫВОД. ФУНКЦИИ. ПОЛЯ УПРАВЛЕНИЯ ФОРМАТИРОВАНИЕМ, МАНИПУЛЯТОРЫ

Система ввода-вывода C++

Может появиться вопрос: в связи с чем возникла необходимость в языке C++ вводить собственную систему ввода-вывода, отличную от языка C ? Давайте обобщим уже известное. C++ позволяет оперировать объектами как обычными переменными языка программирования. Важное место здесь отводится механизму перегрузки операций. Было бы естественным предоставить такую же возможность и в отношении операций ввода-вывода. Для этого необходимо заменить стандартные функции ввода-вывода языка C (типа printf() и scanf()) операторами ввода-вывода.

Дальнейшее совершенствование системы ввода-вывода связано с ее реализацией через концепцию классов, задающих общий интерфейс ввода-вывода, зная который пользователь может создавать собственные библиотеки ввода-вывода.

Система ввода-вывода языка C++ основывается на концепции потока. Поток в C++ - это абстрактное понятие, относящееся к любому переносу данных от источника к приемнику. В качестве синонимов вывода данных используются термины извлечение, прием, получение, а синонимов ввода - вставка, помещение, заполнение. В языке C++ вся передаваемая/принимаемая информация рассматривается как последовательность символов, поскольку любое двоичное представление может быть рассмотрено как неотображаемая последовательность байт.

Перенос данных от источника к приемнику редко осуществляется непосредственно, а обычно связан с временным их хранением в некоторой области памяти, называемой буфером

Система ввода-вывода C++ состоит из четырех основных библиотек: iostream, fstream, strstream и constream. Библиотека iostream обеспечивает ввод-вывод, связанный со стандартными потоками. Библиотека fstream поддерживает работу с файлами, strstream - со строками символов, а constream обеспечивает работу с консолью. Все эти библиотеки позволяют выполнять форматный ввод-вывод с контролем типов как для предопределенных, так и для определеяемых пользователем типов данных с помощью перегруженных операций и других объектно-ориентированных методов.

Система ввода-вывода языка C++ включает два параллельных семейства классов. Классы одного семейства являются производными из базового класса streambuf, а другого - из класса ois. Базовые классы streambuf и ios являются низкоуровневыми и каждый предназначен для решения своего круга задач.

Класс streambuf обеспечивает общие правила буферизации и обработки потоков без форматирования данных. Реализация компонент- функций класса streambuf направлена прежде всего на поддержку максимальной эффективности ввода-вывода. Этот класс используется всеми четырьмя библиотеками системы ввода-вывода языка C++, он также доступен для построения пользовательских производных классов ввода-вывода. Доступ из семейства классов, построенных на основе ios, к классу streambuf осуществляется через указатель на класс streambuf.

Характерными для класса streambuf являются компоненты-функции подсоединения потока к некоторому буферу (setbuf()), извлечения и помещения отдельного символа (sgetc(), sbumpc(),snextc(), sputc()), нескольких символов (sgetn(), sputn()), перемещения потоковых указателей (seekoff(), seekpos(), stossc()), возврата символа в поток (sputbackc()), определения заполнения буферов (in_avail(), out_waiting). Класс streambuf содержит также ряд защищенных компонент- функций, доступных для производных классов ввода-вывода.

Из базового класса streambuf образуются три производных класса filebuf, strstreambuf и conbuf, соответственно специализирующих класс streambuf для работы с файлами, строками и экраном, которые определяются следующим образом:

class streambuf{...};
class filebuf: public streambuf{...};
class strstreambuf: public streambuf{...};
class conbuf: public streambuf{...};

Стандартные потоки ввода-вывода

Библиотека iostream объявляется с помощью файла заголовков iostream.h и имеет два класса: streambuf и ios.

Класс ios предназначен для форматного ввода-вывода через класс streambuf. Из него производятся три класса istream, ostream и iostream соответственно для ввода, вывода и одновременного ввода-вывода следующим образом:

class ios{...}; class istream: virtual public ios{...};
class ostream: virtual public ios{...};
class iostream: public istream, public ostream{...};

Кроме того, существуют три класса withassign, добавляющие в классы istream, ostream и iostream операцию присваивания:

class istream_withassign: public istream{...};
class ostream_withassign: public ostream{...};
class iostream_withassign: public iostream{...};

Последние классы обеспечивают четыре предопределенных стандартных потока C++: cin, cout, cerr и clog:

extern istream_withassign cin{...};
extern ostream_withassign cout{...};
extern ostream_withassign cerr{...};
extern ostream_withassign clog{...};

Эти потоки автоматически открываются при запуске всякой программы, содержащей файл iosrteam.h и соответствуют следующему:

- cin - стандартный ввод (аналогичен stdin);
- cout - стандартный вывод (аналогичен stdout);
- cerr - стандартный вывод ошибок (аналогичен stderr);
- clog - буфферизированная версия потока cerr, позволяющая сохранять протокол ошибок.

Названные стандартные потоки по умолчанию связаны с консолью.

Переназначение их на другие устройства или файлы обеспечивается обычными средствами языка С или операционной системы. Потоки можно также копировать с помощью операции присваивания, например:

cout = cerr;

В результате две переменные будут обращаться к одному потоку. Это позволяет использовать стандартные имена (cin, cout, cerr, clog) для обращения к другим потокам.

Два различных потока в языке C++ можно определить как связанные. Это означает, что использование одного потока воздействует на другой поток. Например, если потоки cin и cout связаны, то перед использованием cin разгружается поток cout. По умолчанию стандартные потоки cin, cerr и clog связаны с потоком cout. Для определения потока, связанного с данным, используется функция tie() без аргументов: ostream * tie(); возвращающая связанный поток, либо 0 в случае его отсутствия. Для привязки некоторого потока к данному служит функция tie() с аргументом:

ostream * tie(ostream * stream);

где stream - поток, привязанный к данному (для которого вызывается функция tie()), а возвращается предыдущий привязанный поток. Чтобы развязать поток вызывается tie(0).

Неформатный ввод-вывод

Компоненты-функции put() и write() класса ostream обеспечивают неформатный вывод данных в стандартные потоки.

Функция put(), описываемая как

ostream & put (char);

• позволяет вывод двоичных данных или отдельного символа, получаемого в качестве аргумента, в указанный поток, например:

```
int ch = 'a';
cout.put(ch);
```

• выводит символ 'a' в поток cout.

Большие по размерам объекты выводятся с помощью функции write():

```
ostream & write (char *p, int n);
```

где p указывает на выводимые данные; n - размер в байтах. В отличие от вывода строк с помощью оператора <<, функция write() не прекращает работу, встретив пустой символ, например:

```
cout.write((char *)&ch, sizeof(ch));
```

пошлет непреобразованное представление ch на стандартное устройство вывода.

Подчеркнем еще раз, что функции put() и write() не выполняют форматирование данных при выводе, они не могут вызывать разгрузку потоков, установку ширины поля и т.д. Их использование ограничено простыми примерами ввода-вывода, когда предъявляются жесткие требования к быстродействию и размеру абсолютного кода программы.

Для неформатного ввода данных из стандартного потока в C++ служит компонента-функция get() класса istream:

```
istream & get(char *str, int max, int term = '\n');
```

Функция get() считывает символы из входного потока в массив str до тех пор, пока не будет прочитано max-1 символов, либо пока не встретится символ, заданный терминатором (ограничителем) term. К прочитанным данным автоматически добавляется символ конца строки '\0'. По умолчанию значением терминатора является символ новой строки '\n', который в str не считывается и из istream не удаляется.

Для корректной работы программы массив str должен иметь размер не менее max символов, например:

```
char s[81];  cin.get(s,81);  // ввод строки с терминала
```

Функция get() в классе istream перегружена и имеет несколько реализаций, например,

istream & get(streambuf & buf, char term = '\n');

• позволяет читать символы в некоторый буфер потока buf. Следующие две функции

istream & get(char & ch); int get();

• позволяют читать из потока единственный символ независимо от того, является он пробельным или нет, например:

```
#include <iostream.h>
void main(){
char c1, c2;
cout.write("Введите два символа\n", 20);
        cin.get(c1);  // ввод символа через параметр
        c2 = cin.get(); // ввод символа через
        //возвращаемое значение
cin.get();  // выборка из потока лишнего символа
       cout.write("Были введены символы: ", 23);
        cout.put(c1);
       cout.put(' ');
       cout.put(c2);
        cout.put('\n');
}
```

Для неформатного ввода блока данных служит функция read(), определенная в классе istream:

*istream & read(char *p, int n);*

где p указывает на выводимые данные, а n - размер данных в байтах. Следующий пример позволяет вводить и выводить непреобразованное значение последовательности символов:

```
        void main(){  int s;
        cout << "Введите пять символов\n"; cin.read(s, sizeof(s)); cout << "Было
введено: ";
        cout.write(s, sizeof(s));
        }
```

Стандартные потоки cin и cout осуществляют последовательный символьный ввод-вывод. Функции же read() и write() наиболее эффективны при вводе-выводе двоичных блоков данных, поэтому их часто используют для работы с бинарными файлами В библиотеке iostream имеются еще несколько полезных функций неформатного ввода-вывода:

*istream & getline(char *str, int max, int term = '\n');*

- подобна функции get() за исключением того, что ограничивающий символ term извлекается, но не копируется в str (удобна для чтения строк);

int peek();

- возвращает следующий символ без извлечения из потока (просмотр вперед);

int gcount();

- возвращает число символов последнего извлечения;

istream & putback(char ch);

- возвращает обратно во входной поток символ ch;

istream & ignore(int n = 1, int term = EOF);

- пропускает n символов во входном потоке, останавливается, когда встретится term;

ostream & flush();

- разгружает (т.е. выводит содержимое) поток на связанное с ним устройство.

В качестве примера рассмотрим функцию компилятора для ввода индентификаторов языка C++

```
void getid(char *s)

{
char c = 0; // защита от конца файла
cin >> c;  // пропуск пробельных символов
      if (isalpha(c) || c == '_')  // если буква или '_'
      do{
*s++ = c; // введенный символ помещаем в s
            c = 0;  // защита от конца файла
            cin.get(c);  // ввод следующего символа
} while (isalnum(c) || c == '_'); // до тех пор, пока
 // буквы, цифры или '_'
*s = 0; // устанавливаем символ конца строки
            if (c)
cin.putback(c); // возвращаем лишний символ в поток
 }
```

Суть защиты от конца файла заключается в том, что если при вводе будет обнаружен конец файла, значение переменной с останется нулевым и функция getid() корректно завершит свою работу.

Форматный ввод-вывод

В языке C++ форматный вывод в поток выполняется с помощью перегруженной операции сдвига влево “<<“, называемой при выводе оператором вставки или помещения. Для форматного ввода используется перегруженная операция сдвига вправо “>>“, которая при вводе называется оператором извлечения или просто извлечением. Левый операнд оператора вставки представляет собой объект класса ostream, а оператора извлечения - istream. Правый операнд операторов << и >> может быть любого типа, для которого определен ввода-вывод потоком (стандартные типы языка C++, а также типы, определяемые пользователем и имеющие возможность вывода потоком). В библиотеке iostream ввод-вывод потоком определен для всех арифметических типов (char, short, int, long, float, double), строки символов (char *), значений указателей (void *), а также для буфера потока (streambuf *).

Операция << ассоциативна слева и возвращает ссылку на объект ostream, для которого она вызывалась. Это позволяет связывать операцию вставки в цепочки, например:

```
cout << “i= “ << i << “j= “ << j << “\n”;
```

• что равносильно следующей записи:

```
cout << “i= “;
 cout << i;
 cout << “j= “;
 cout << j;
cout << “\n”;
```

По умолчанию извлечение опускает пробельные символы ('\v', '\t', '\n' и пробел), а затем считывает символы, соответствующие типу объекта ввода. Как и в случае оператора вставки, извлечение ассоциативна слева и возвращает ссылку на объект istream, для которого она вызывалась. Это позволяет объединить в одном операторе несколько операций ввода, например:

```
int  k;
float y;
cin >> k >> y;
```

Последний оператор вызовет пропуск пробельных символов, считывает цифры со стандартного устройства ввода (консоли) до тех пор, пока не встретится символ, не соответствующий формату целого числа, преобразует введенное значение в двоичное представление целого и записывает в переменную k. Затем вновь пропускаются пробельные символы, считывается число с плавающей точкой, преобразуется во внутренний формат и записывается в переменную y.

Для других предопределенных типов С++ действие оператора извлечения аналогично: пропускаются пробельные символы, выполняется необходимое преобразование и осуществляется запись значения в указанную переменную. Строка символов считывается до первого встретившегося пробельного символа, затем добавляется нулевой символ '\0'.

Отметим, что в операторе извлечения указываются не адреса переменных (как это требует функция scanf()), а сами переменные. Следует также проявлять осторожность при считывании строки, так как контроль конца массива символов операция извлечения не выполняет. Например:

```
#include <iostream.h>
main(void)
{
char str[81];
        cout << "Ведите строку\n";
        cin >> str;
         cout << "Была введена строка: " << str << endl;
}
```

Поскольку при перегрузки операций приоритет их не изменяется, то приоритет операторов форматного ввода-вывода совпадает с приоритетом операций сдвига. Поэтому при выводе операнд, заданный выражением, берется в скобки только в том случае, когда выражение содержит операции более низкого приоритета.

Форматирование ввода и вывода определяется форматирующими флагами, представляющими собой биты числа типа long int, определенного в классе ios следующим образом:

```
        public: enum{ skipws = 0x0001, // пропуск при вводе пробельных
          // символов
 left  = 0x0002,  // левое выравнивание вывода  right = 0x0004,  // правое
выравнивание вывода  internal  = 0x0008, // заполнение пробелами поля между
 // знаком или основанием системы  // счисления и числовым значением  dec  =
0x0010,  // десятичное представление чисел  oct  = 0x0020,  // восьмеричное
представление чисел
 hex  = 0x0040, // шестнадцатеричное представление
 // чисел
 showbase  = 0x0080, // при выводе чисел показывается
 // основание системы счисления
         showpoint = 0x0100, // при выводе плавающих чисел
 // показывается позиция десятичной
 // точки
 uppercase = 0x0200, // вывод шестнадцатеричных значений
 // буквами верхнего регистра
 showpos  = 0x0400, // при выводе положительных целых
```

```
// чисел показывается знак “+”
scientific = 0x0800,  // плавающие числа отображаются
// в экспоненциальной форме
fixed = 0x1000,  // плавающие числа отображаются
// с десятичной точкой
unitbuf  = 0x2000,  // разгрузка всех потоков после
// вставки
stdio = 0x4000,  // разгрузка после вставки потоков
// stdout и stderr  }
```

Отметим, что по умолчанию символ экспоненты “e” для чисел с плавающей точкой и символ “x” для шестнадцатеричных чисел отображаются на нижнем регистре. Если ни один из флагов dec, oct и hex не установлены, то числа выводятся в десятичном представлении. Установка флага stdio позволяет разгружать выходной поток на физическое устройство после каждой операции вывода. Флаг unitbuf выполняет усовершенствование системы ввода-вывода C++ и по умолчанию всегда установлен.

При отсутствии специальных действий со стороны пользователя флаги в C++ устанавливаются так, чтобы обеспечивать ввод-вывод, соответствующий умалчиваемым форматам функций printf() и scanf().

Ряд функций библиотеки iostream предоставляет пользователю возможность управлять форматным вводом-выводом. Для установки флагов пользователя используется функция setf():

long setf(long flags);

Эта функция возвращает предыдущую установку флагов и изменяет флаги, определенные в flags. Обращение к функции имеет вид

stream.setf(ios::flag);

где stream - поток, на который вы желаете воздействовать; flag - изменяемый флаг. Например, следующая программа изменяет флаги hex и scientific:

```
#include <iostream.h>  main(void)
{
cout.setf(ios::hex);
cout.setf(ios::scientific);  cout << 365 << “ “ << 365.78 << “\n”;
}
```

В результате работы программы получим:

```
16d 3.6578e+02
```

В одном вызове функции setf() можно одновременно устанавливать несколько флагов, объединяя их с помощью поразрядного сложения:

cout.setf(ios::hex | ios::scientific);

Для сброса флагов используется функция unsetf():

long unsetf(long flags);

Функция возвращает предыдущую установку и сбрасывает флаги, определенные в flags. Совместить действия функций setf() и unsetf() можно с помощью функции setf(), принимающей два аргумента:

long setf(long setbits, long field);

- которая вначале сбрасывает флаги, определенные в field, а затем устанавливает флаги, отмеченные в setbits.

Для определения значений флагов без их изменения используется функция flags() без параметров: *long flags(void);*

Функция flags() с аргументом:

long flags(long bits);

- работает точно так же, как и setf(long). Для установки флагов в умалчиваемое значение используется flags(0).

В качестве примера рассмотрим программу вывода умалчиваемых значений форматирующих флагов стандартных потоков:

```
#include <iostream.h>
// функция type_flags выводит значения форматирующих
// флагов f
void type_flags(long f)
{
for (long i = 0x4000; i; i >>= 1) // формирование маски  if (i & f)  cout << “1”; // определение значения
else cout << “0”; // флага  cout << “\n”;
}
main(void)
{  long f;  f = cout.flags(); type_flags(f);  f = cin.flags();  type_flags(f);  f = cerr.flags(); type_flags(f);  f = clog.flags(); type_flags(f);  cout.setf(ios::hex | ios::scientific);
f = cout.flags(); type_flags(f);
}
```

Результатом работы программы будет:

010000000000001
000000000000001
010000000000001
000000000000001
010100001000001

Следующие три функции библиотеки iostream позволяют устанавливать ширину поля, заполняющий символ и число цифр, показываемых после запятой:

int width(int len); char fill(char ch); int precision(int num);

где len - ширина поля,
ch - символ заполнения,
num - число цифр после запятой в отображении числа с плавающей точкой.

Все эти функции возвращают предыдущие значения соответствующих параметров. По умолчанию ширина поля равна нулю, это означает, что будет выведено минимальное число символов, которыми может быть представлено выводимое значение. В избыточных позициях выводится символ заполнителя, заданный функцией fill(), по умолчанию - это пробел. Если указанная ширина не достаточна для представления выводимого значения, то она будет проигнорирована и вывод выполняется как для нулевой ширины. Вызов функций width(), fill() и precision() без аргументов возвращает предыдущее значение соответствующих параметров без их изменения, например:

```
#include <iostream.h>
main(void)
{
cout.setf(ios::hex);
cout.setf(ios::scientific);
    cout << 365 << " " << 365.78 << "\n";
cout.precision(2);
cout.width(10);
    cout << 365 << " " << 365.78 << "\n";
    cout.fill('*');
cout << 365 << " " << 365.78 << "\n";
}
```

Результат работы программы получим

16d 3.6578e+02
16d 3.66e+02
16d 3.66e+02

Форматирование с помощью манипуляторов

Мы уже могли видеть как функции ввода-вывода языка С были заменены операторами ввода-вывода в языке С++. Однако в эту концепцию не вписываются функции управления флагами формата.

Возникает вопрос: нельзя ли их также заменить некоторыми конструкциями, подобными операторам ввода-вывода, с тем, чтобы не ломать общий стиль системы ввода-вывода языка С++ ?

С этой целью в язык С++ введено понятие манипулятора. Манипуляторы - это специальные функции, которые принимают в качестве аргументов ссылку на поток и возвращают ссылку на тот же поток. Поэтому они могут объединяться в цепочку вместе с операторами ввода-вывода. Сами манипуляторы никаких действий по вводу или выводу не выполняют, однако, осуществляют "побочный" эффект, воздействуя на флаги формата и другие параметры ввода-вывода. Например, для вывода переменной x в поле шириной 5, а переменной y в поле шириной 10, используется манипулятор setw() установки ширины поля:

cout << setw(5) << x << setw(10) << y << "\n";

Для возможности работы с манипуляторами С++ необходимо в программу включить файл iomanip.h.

Установленные с помощью манипуляторов режимы ввода-вывода не сохраняются на последующие операторы ввода-вывода, например, в результате выполнения следующего фрагмента :

```
int x = 561;
float f = 3.141529;
cout << setw(5) << x << setw(10) << y << 15 << "AAA" << "\n";
```

получим:

561 3.14152915AAA

Создание собственных манипуляторов

Как вы уже, наверное, догадались, что такая гибкая система, как С++, не может не предоставить средство пользователю для создания манипуляторов по своему усмотрению. Определение собственных манипуляторов пользователя для вывода имеет следующую структуру:

```
ostream & <имя_манипулятора>(ostream & stream)
{
// необходимый код
return stream;
}
```

Хотя здесь в качестве аргумента используется ссылка на поток, на самом деле этот аргумент не используется. Рассмотрим пример заказного манипулятора, определяющего вывод числа с плавающей точкой из предыдущего примера :

```
#include <iostream.h>  #include <iomanip.h>
ostream & my_manip(ostream & stream)
{
stream << setw(10) << setprecision(2) << setfill('*');  retutr stream;
}
main(void)
{
cout << 365.766 << my_manip << 365.766 << endl;
}
```

Заказные манипуляторы являются полезными в двух случаях. Во-первых, когда осуществляется вывод на устройство, которое не выполняет применяемые манипуляторы, например, плоттер. В этом случае создание собственных манипуляторов позволяет выполнять необходимые преобразования во время вывода. Во-вторых, когда часто повторяется последовательность одних и тех же манипуляторов. В этом случае их можно объединить в один манипулятор, как было показано в предыдущем примере.

Определение собственных манипуляторов пользователя для ввода имеет следующую структуру:

```
istream & <имя_манипулятора>(istream & stream)
{
// необходимый код
return stream;
}
```

Например:

```
#include <iostream.h>
#include <iomanip.h>

istream & manip_in_hex(istream & stream)
{
cout << "Введите число, используя шестнадцатеричный формат";  cin >> hex;
retutr stream;
}
main(void)
{
int i;
cin >> manip_in_hex >> i;  cout << i << endl;
}
```

Перегрузка операторов ввода-вывода

Наибольшая эффективность операторов ввода-вывода языка C++ наблюдается при их использовании для работы с объектами классов. С этой

целью выполняется их перегрузка. Перегружаемый оператор ввода-вывода первым аргументом должен иметь ссылку на связанный с ним поток, вторым аргументом - объект, записываемый справа от оператора (обычно это ссылка на объект данного класса). Возвращаемым значением является ссылка на связанный с данным оператором поток.

Поскольку первый аргумент в операторе ввода-вывода не может быть объектом данного класса, а значит, и указателем this, то перегружаемые операции ввода-вывода не могут быть компонентами класса. Поэтому общий формат перегрузки операторов ввода-вывода для некоторого класса X имеет вид

```
class X{
. . . . . .
friend ostream & operator << (ostream &, X &);
friend istream & operator >> (istream &, X &);
}
. . . . . .
ostream & operator << (ostream & stream, X x);
{
// операторы вывода
return stream;
}

istream & operator >> (istream & stream, X &x);
{
// операторы ввода
return stream;
}
```

Отметим, что рациональнее в качестве первого аргумента указывать абстрактный поток stream, а не cout или cin, для того, чтобы операторы могли работать с любыми потоками. В качестве примера рассмотрим перегрузку операторов ввода-вывода для класса complex:

```
#include <iostream.h> class complex{ public:
double re, im; complex (double r=0, double i=0)
{ re = r; im = i; };
friend ostream & operator << (ostream &, complex &); friend istream & operator >> (istream &, complex &);
};
ostream & operator << (ostream & stream, complex &x);
{
stream << "Комплексное число= "; stream << "(" << x.re << "," << x.im << ")" << endl; return stream;
};
istream & operator >> (istream & stream, complex &x);
{
```

cout << "Введите действительную и мнимую части: "; stream >> x.re >> x.im;
return stream;
}; main(void)
{ complex a; cin >> a;
cout << a;
}

Ошибки потоков ввода-вывода

С каждым открытым потоком в C++ связывается перечислимая переменная io_state, определяющая биты состояния потока. Она объявлена в классе ios и может рассматриваться как целая величина: *public:*
enum io_state{ goodbit = 0x00, // если бит не установлен,
// то ошибок нет eofbit = 0x01, // обнаружен конец файла failbit = 0x02, // сбой в последней операции
// ввода-вывода
badbit = 0x04, // попытка недопустимой операции hardfail = 0x80 // в потоке невосстанавливаемая ошибка
};

В случае установки eofbit игнорируются попытки выполнить операции извлечения; бит failbit может быть сброшен и продолжено использование потока; после сброса badbit не всегда можно восстановить работоспособность потока; перед сбросом hardfail требуется установить причину, вызвавшую ошибку.

После того как для некоторого потока возникло состояние ошибки, все попытки ввода-вывода будут игнорироваться до тех пор, пока не будет устранена причина, вызвавшая ошибку, а биты ошибки не сброшены с помощью функции clear(), например:

in.clear(0); // очистка всех бит ошибок
in.clear(ios::eofbit); // очистка бита eofbit

Хорошим стилем программирования считается проверка состояния ошибки в наиболее ответственных точках программы. Это можно выполнить с помощью следующих функций, возвращающих ненулевые значения, если:

good() - не было ошибки; eof() - обнаружен конец файла;

fail() - был установлен один из битов failbit, badbit, hardfail; bad() - был установлен бит badbit или hardfail.

Текущее состояние ошибки можно получить с помощью функции rdstate(), которая возвращает номер бита ошибки, например :

cout << "Состояние потока cin: " << cin.rdstate() << endl; cout << "Состояние потока cout: " << cout.rdstate() << endl; cout << "Состояние

потока cerr: “ << cerr.rdstate() << endl; cout << “Состояние потока clog: “ << clog.rdstate() << endl;

Кроме того, в классе ios имеются перегруженные операции

```
int operator !();
operator void *();
```

Операция void *() определяет преобразование потока в указатель, который будет равен нулю в случае установления бит failbit, badbit или hardfail и ненулевому значению в противном случае. Операция “!” наоборот возвращает ненулевое значение, если установлен один из бит failbit, badbit или hardfail, и возвращает нулевое значение в противном случае. Это позволяет рассматривать в логических выражениях в качестве переменной непосредственно сами потоки ввода-вывода, например:

```
#include <iostream.h>
int main(){  int x;  if (!cout) // ошибка вывода!
return -1;
cout << “Введите целое число\n”;  if (cin >> x)  // все в порядке!  cout << “Введено” << x << endl;  else{  // ошибка вывода!
cout << “Ошибка ввода!\n”;
return -1;}  return 0;
}
```

ФАЙЛОВЫЙ ВВОД-ВЫВОД. КЛАССЫ ФАЙЛОВОГО ВВОДА-ВЫВОДА. ОРГАНИЗАЦИЯ ДОСТУПА К ФАЙЛУ. ОСНОВНЫЕ ФУНКЦИИ

Библиотека fstream и открытие файлов

Файлы в C++ рассматриваются как потоки, связанные с конкретными именами физических файлов, например, на диске. Библиотека fstream языка C++ определяется с помощью заголовочного файла <fstream.h>. Она содержит следующую структуру классов:

class filebuf: public streambuf {...}; class fstreambase: virtual public ios {...}; class ifstream: public fstreambase, public istream {...}; class ofstream: public fstreambase, public ostream {...}; class fstream: public fstreambase, public iostream {...};

Класс filebuf (буфер файла) специализирует низкоуровневый класс streambuf для управления файлами. Ключевую роль при этом играет дескриптор файла - 16-битовое значение, определяемое при открытии файла. Все последующие манипуляции с файлом (чтение, запись, позиционирование, закрытие и т.д.) так или иначе ссылаются на дескриптор файла. Характерными для класса filebuf являются компоненты- функции открытия и закрытия файла (open() и close()), подсоединения буфера к дескриптору открытого файла (attach()), определения дескриптора файла (fd()) и проверки открыт ли файл (is_open()). Кроме того, файл filebuf содержит ряд виртуальных функций (overflow(), seekoff(), setbuf(), sync() и underflow()), которые в случае использования должны переопределяться в производных классах.

Класс fstreambase образуется из класса ios и обеспечивает основные операции ввода-вывода для файловых потоков. Связь класса fstreambase с буфером файла осуществляется при помощи указателя на класс filebuf. Класс fstreambase кроме компонент- функций, наследуемых из класса ios, содержит также функции открытия и закрытия файла (open() и close()), подсоединения дескриптора к открытому файлу (attach()), задания определенного буфера (setbuf()) и определения используемого буфера файла (rdbuf()). Остальные классы библиотеки fstream: ifstream, ofstream и fstream производятся из класса fstreambase и классов istream, ostream и iostream соответственно и новых компонент- функций не содержат.

Для того, чтобы начать работать с файлом, прежде всего необходимо определить файловый поток, который должен быть объектом соответствующего класса, например:

ifstream in; // in - входной поток ofstream out; // out - выходной поток fstream inout; // inout - может быть как входным, так и

// выходным потоком

Уже созданный поток связывается с конкретным именем физического файла с помощию функции open(), которая имеет вид:

void open(char *filename, int mode, int access);

где filename - имя физического файла, которое может включать определенный путь; mode - режим открытия файла; access - задает атрибут доступ к файлу.

Режим открытия файла определяется в классе ios следующим образом:

```
public:
    enum open_mode{  in = 0x01,  // открыть для чтения  out = 0x02,  // открыть для записи  ate = 0x04,  // переместиться в конец файла  // при первоначальном открытии  app = 0x08,  // режим добавления в конец: все  // дополнения выполняются в конец файла  trunc = 0x10, // очистка содержимого: если файл
     // существует, то его содержимое
     // уничтожается
     nocreate = 0x20, // не создавать: если файл не существует,  // то новый не создается и не открывается  noreplace = 0x40, // не замещать: если файл уже существует,  // то новый не создается и не открывается  binary  = 0x80 // создается бинарный файл (по умолчанию
     // создается текстовый файл
    }
```

При задании режима открытия файла эти параметры могут объединяться с помощью операции поразрядного сложения (|). Атрибут доступа к файлу соответствует атрибуту файла в операционной системе (MS-DOS, UNIX и др.) и может принимать следующие значения:

- обычный файл;
- файл доступен только для чтения;
- скрытый файл;

4 - системный файл; 8 - архивный файл.

Примеры открытия файлов:

```
in.open("INFILE",ios::in,0);  // файл для ввода
out.open("OUTFILE",ios::out | ios::app,0);  // файл для вывода
inout.open("IOFILE",ios::in | ios::out,0);  // файл для ввода
// и вывода
```

Отметим, что умалчиваемым значением атрибута доступа access является 0; умалчиваемым значением режима открытия mode для объектов класса ifstream является ios::in, для ofstream - ios::out.

Открытие файла можно выполнять одновременно с объявлением переменной потока, передавая конструктору соответствующие параметры, например:

```
ifstream in("INFILE");  // файл для ввода
ofstream out("OUTFILE",ios::out | ios::app); // файл для вывода
fstream inout("IOFILE",ios::in | ios::out); // файл для ввода
  // и вывода
```

Если файл открыть не удалось, переменная потока будет иметь нулевое значение, поэтому проверить успешное открытие файла можно следующим образом: *ifstream in(“INFILE”);*

```
if (in == 0)
{
cout << “Нельзя откратъ файл INFILE\n”;
. . . . . .
}
```

Закрытие файлов выполняется функцией close(), не имеющей параметров и не возвращающей никакого значения, например: *in.close(); out.close(); inout.close();*

Не смотря на то, что при завершении программы на языке C++ все открытые файлы закрываются автоматически, хорошим стилем программирования считается явное закрытие файлов при выходе их программы. Не следует также в процессе выполнения программы держать длительное время файлы открытыми, если в этом нет необходимости. Рекомендуется также всегда проверять успешное открытие файла перед выполнением каких-либо манипуляций с файловым потоком (для стандартных потоков это не столь актуально).

Чтение и запись текстовых файлов

По умолчанию все файлы в языке C++ открываются в текстовом режиме. Поэтому нет необходимости в специальном указании того, что работа ведется с текстовым файлом. Чтение и запись текстовых файлов осуществляется с помощью операторов форматного ввода- вывода языка C++ >> и <<, определенных в классе ios, только вместо стандартных потоков cin и cout указываются имена файлов. При этом справедливы все механизмы C++, касающиеся форматного ввода-вывода (использование манипуляторов, перегрузка операторов ввода-вывода и т.д.). Например :

```
#include <iostream.h>
#include <fstream.h>
#include <iomanip.h>
int main(){  ofstream out(“MY_FILE”);
if (!out){
cout << “Нельзя открыть файл MY_FILE\n”;  return -1;
}
out << setw(5) << 3.15 << setw(5) << 15  << endl << “Это текстовый файл\n”;
out.close();
int i;
float pi;
char s1[81],s2[81],s3[81];
ifstream in(“MY_FILE”);
```

```
 if (!in){
cout << “Нельзя открыть файл MY_FILE\n”;  return -1;
 }
 in >> pi >> i  // чтение переменных
>> s1 >> s2 >> s3;  // чтение строки
cout << “pi= “ << pi << “i= “ << i  // вывод переменных
 << “s= “ << s1 << s2 << s3 << endl; // вывод строки
in.close();
return 0;
}
```

Напомним, что оператор извлечения >> предусматривает пропуск пробельных символов потока и строки читаются до первого пробельного символа. Поэтому в предыдущем примере строку из трех слов пришлось считывать в три разные переменные. Кроме того, при вводе текстовых файлов последовательность двух управляющих символов возврата каретки и перевода строки “x10x13” автоматически преобразуется в один символ новой строки “\n”. При выводе выполняется обратное преобразование. Если подобные преобразования по какой-либо причине не приемлемы, то следует пользоваться бинарными файлами. **Чтение и запись бинарных файлов**

В C++ бинарные и текстовые файлы определяются не способом их открытия (как может ожидаться), а используемыми функциями (операторами) ввода-вывода. Для простейшего чтения и записи символов бинарного файла используются функции get() и put(). Прочитать единственный символ ch из файла можно с помощью функции get(). В этом случае ее прототип имеет вид:

ifstream & get(char & ch);

Функция get() имеет несколько перегруженных версий , одна из которых объявляется как

ifstream & get(char *str, int n, char term = '\n');

и позволяет читать в строку str до тех пор, пока не встретится терминатор term, либо не будет прочитано n-1 символов. Функция put() записывает в файл единственный символ ch, ее прототип:

ofstream & put(char & ch);

В качестве примера рассмотрим программу копирования двух файлов:

```
 #include <iostream.h>
#include <fstream.h>
void main(int argc, char *argv[]){
if (argc != 3){
 cout << “Использование:\n”
 << “ исходный_файл принимающий_файл\n”;  return -1;
 }
 ifstream in(argv[1]);
if (!in){
```

```
cout << “Нельзя открыть исходный файл\n”;
return -1;
 }
 ofstream out(argv[2]);
 if (!out){
cout << “Нельзя открыть принимающий файл\n”;
return -1;
 }  char ch;  while ( (ch=in.get()) != EOF)
out.put(ch);
return 0;
 }
```

Чтение и запись блоков двоичных данных выполняется с помощью функций read() и write(), которые имеют вид:

*istream & read(char *buf, int n); ostream & write(char *buf, int n);*

Функция read() читает n байт из потока и помещает их в буфер buf, причем buf может быть объявлена как знаковая, так и беззнаковая символьная переменная. Функция write() пишет n байт в поток из буфера, указанного buf. В случае, когда наступает конец файла до прочтения n символов, функция read() останавливается, а буфер будет содержать прочитанные символы. Определить число символов, прочитанных последней функцией read(), можно с помощью функции gcount(), имеющую прототип: int gcount();

Если нас не устраивает посимвольное копирование файлов предыдущего примера, это можно сделать с помощью функций read() и write() :

```
 #include <iostream.h>
#include <fstream.h>
void main(int argc, char *argv[]){  if (argc != 3){
cout << “Использование:\n”
 << “ исходный_файл принимающий_файл\n”;  return -1;
 }
 ifstream in(argv[1]);  if (!in){
cout << “Нельзя открыть исходный файл\n”;  return -1;
 }
 ofstream out(argv[2]);
 if (!out){
cout << “Нельзя открыть принимающий файл\n”;  return -1;
 }
 char s[512];  while (in){  in.read(s,512);  out.write(s,gcout());
 }  return 0;  }
```

Кроме рассмотренных имеется еще ряд полезных функций, облегчающих работу с файлами:

ifstream & getline(char *str, int n, char term = '\n');

работает также, как get(), только терминатор извлекается из потока;

ifstream & ignore(int n = 1, int term = EOF);

пропускает в потоке n символов, останавливается, если встрерится term; ostream & flush();

разгружает выходной поток на физическое устройство.

Произвольный доступ к файлу

При открытии в C++ всякого файла образуются два указателя get - для определения позиции, откуда будет считываться очередная информация, и put - куда будут помещаться данные в следующей операции вывода.

На логическом уровне файл в языке C++ может рассматриваться как бинарный массив, расположенный в оперативной памяти Размер массива может быть достаточно большим, он определяется объемом физического устройства. Будем считать, что файловые указатели put и get могут свободно перемещаться по всей длине файла. В дествительности доступный участок файла расположен в буфере, который периодически загружается/разгружается на физическое устройство. Вопросы буферизации реализуются средствами операционной системы и здесь не рассматриваются. Программист может, при необходимости, только устанавливать размер буфера. Для определения значений указателей get и put служат функции tellg() и tellp(), имеющие прототипы:

streampos tellg(); streampos tellp();

где streampos - тип, определенный в iostream.h, который в состоянии содержать большую величину. Функция tellg() позволяет получить позицию get, а функция tellp() - позицию put. Перемещение файловых указателей get и put выполняется функциями seekg() и seekp():

istream & seekg(streamoff offset, seek_dir orgn); ostream & seekp(streamoff offset, seek_dir orgn);

Типы streamoff и seek_dir также определены в iostream.h. streamoff аналогичен streampos и определяет смещение указателя на offset байт, а seek_dir является перечислением и имеет вид: *public:*

enum seek_dir{

beg = 0, // смещение от начала файла cur = 1, // смещение от текущего положения указателя end = 2 // смещение от конца файла

}

Функция seekg() управляет указателем get, а функция seekp() - указателем put.

Произвольный доступ к файлу продемонстрируем на примере программы, которая позволяет просматривать содержимое любого байта файла:

```
#include <iostream.h>
#include <fstream.h>
void main(int argc, char *argv[]){
if (argc != 2){
cout << "Использование:\n <имя_файла>\n"; return -1;
}
```

```
ifstream in(argv[1]);  if (!in){
cout << “Нельзя открыть файл” << argv[1] << endl;  retutn -1;
} int offset;
cout << “Введите номер просматриваемого байта\n”;  cin >> offset;
in.seekg(offset,ios::beg);  cout << in.get() << endl;
return 0;
}
```

Строковый ввод-вывод

Иногда бывает удобно рассматривать в качестве потока массив символов, расположенный непосредственно в оперативной памяти. Например, при разработке компилятора может возникнуть необходимость неоднократного сканирования одной и той же строки, либо возврата обратно в поток целой строки символов. Для обеспечения такой возможности C++ предоставляет библиотеку strstream для работы со строками символов как с обычными потоками ввода-вывода. Классы библиотеки strstream определены в заголовочном файле <strstream.h> и имеют следующую структуру:

```
class strstreambuf:
public
streambuf {...};
class strstreambase:
virtual public ios {...};
class istrstream:
public strstreambase,
public istream {...};
class ostrstream:
public strstreambase,
public ostream {...};
class strstream:
public strstreambase,
public iostream {...};
```

Связь потока с конкретным символьным массивом осуществляется с помощью конструкторов соответствующих классов. Класс istrstream определяет входной строковый поток и имеет два конструктора: *istrstream(char *buf); istrstream(char *buf, int n);*

Первый конструктор позволяет рассматривать всю строку buf как входной поток, ограниченный нулевым символом '\0' конца строки, а второй делает то же самое, но разрешает использовать только n символов строки buf.

Класс ostrstream определяет выходной строковый поток и также имеет два конструктора: *ostrstream();*

*ostrstream(char *buf, int n, int mode=ios::out);*

Первый конструктор создает динамический выходной строковый поток, память которому выделяется по мере необходимости. Второй конструктор

определяет строку buf как выходной поток в режиме mode, ограниченный n символами. Если mode имеет значение ios::app или ios::ate, то указатель put устанавливается на символ конца строки '\0'.

Класс strstream имеет два конструктора:

*strstream(); strstream(char *buf, int n, int mode);* которые аналогичны конструкторам класса ostrstream.

Приведем примеры определения строковых потоков:

```
char buf[81];
istrstream ins(buf,81); // входной строковый поток
ostrstream out(buf,81,ios::app); // выходной строковый поток
strstream ios(buf,81,ios::in | ios::out); // как входной, так
 // и выходной поток
ostrstream outsd; // динамический выходной поток
```

Для всех строковых потоков справедливо применение рассмотренных выше операторов форматного ввода-вывода << и >>, а также использование манипуляторов. Строковые потоки также наследуют все функции неформатного ввода-вывода. Кроме этого чтение и запись в массив строк единственного символа осуществляется с помощью функций sgetc() и sputc(), имеющих прототипы:

int sgetc(); int sputc(int ch); например: *char ch; ch = is.sgetc(); os.sput(ch);*

Получение из массива и помещение в массив сразу n символов выполняется с помощью функций sgetn() и sputn(); имеющих вид: *int sgetn(char *srt,int n); int sputtn(char *srt,int n);*

Возврат в массив последнего прочитанного символа выполняет функция sputbackc() вида:

int sputbackc(char);

Заполнение выходного буфера можно определить с помощью функции pcount(): *char * pcount();*

которая возвращает количество байт, запомненных в буфере. Можно также “заморозить” буфер, запретив запись в него любых символов с помощью функции str():

*char * str();* причем, если буфер был динамический, то его следует освободить. В качестве примера рассмотрим программу, которая позволяет вводить в произвольном формате строку, содержащую целое, плавающее и строку символов, и выводить ее в отформатированном виде :

```
#include <strstrea.h>
#include <string.h>
void main(){ int i;
float f;
char str[81], buf[81];
```

cout << "Введите строку, содержащую целое, плавающее " << "и строку символов\n";
cin.getline(buf,81);
istrstream ins(buf, strlen(buf)); ins >> i >> f >> ws;
ins.getline(str,81);
cout << i << "\t" << f << "\t" << str << endl;
}

Консольный вывод

Для потокового вывода на экран в текстовом режиме служит библиотека constream. Ее классы определены в заголовочном файле <constream.h> и имеют вид:

class conbuf: public streambuf{...}; class constream: public ostream{...}; class omanip_int_int{...};

Класс constream предоставляет функциональные возможности библиотечных функций языка С, объявленных в <conio.h>. Конструктор класса *constream();*

позволяет создавать консольные выходные потоки, не связанные с экраном. Для установления соответствия консольного потока с прямоугольной областью на экране служит функция window(): *void window(int left, int top, int right, int bottom);*
где left и top определяют координаты левого верхнего угла окна на экране, а right и bottom - координаты правого нижнего угла. После установления связи консольного потока с окном на экране, можно производить вывод на экран, пользуясь всеми возможностями форматного и неформатного вывода языка С++. Кроме этого, библиотека constream предоставляет рад функций, специально предназначенных для вывода на экран. Эти функции фактически повторяют библиотечные функции языка С, поэтому ограничимся их кратким описанием:

- void clrscr() - очищает окно; void clreol() - очищает до конца строки; void setcursortype(int) - устанавливает вид курсора; void textcolor(int) - устанавливает цвет символов; void textbackground(int) - устанавливает цвет фона; void textattr(int) - устанавливает байт атрибута;
- void highvideo() - устанавливает повышенную яркость символов;
- void lowvideo() - устанавливает пониженную яркость символов; void normvideo() - устанавливает нормальную яркость символов; void gotoxy(int, int) - перемещает курсор в заданную позицию; int wherex() - возвращает горизонтальную позицию курсора; int wherey() - возвращает вертикальную позицию курсора; void delline() - удаляет строку; void insline() - вставляет строку;

• static void textmode(int) - устанавливает новый текстовый режим экрана. Например, вывод строки в центре экрана можно осуществить следующим образом:

```
#include <constrea.h>
void main(){  constream win;  win.window(30,12,50,13);
win.clrscr();  win << "Минск - город-герой!";
}
```

Библиотека constream позволяет также управлять консольными потоками с помощью следующих специальных манипуляторов:

Следующий пример демонстрирует использование консольных функций и манипуляторов :

```
#include <constrea.h>
void main(){
constream win;
int left=2, top=2, right=79, bottom=24;
win.window(left,top,right,bottom);
win.clrscr();
textattr((BLUE << 4) | YELLOW);
for (int i=1; i<=(right-left+1)*(bottom-top+1); i++)
win << " ";
win << setxy(30,3) << "СТОЛИЦЫ ГОСУДАРСТВ:";
constream win1;
int left1=5, top1=7, right1=28, bottom1=13;
win1.window(left1,top1,right1,bottom1);  win1.clrscr();
textattr((GREEN << 4) | WHITE);
for (i=1; i<=(right1-left1+1)*(bottom1-top1+1); i++)  win1 << " ";
win1 << setxy(8,2) << "ГОРОДА" << setclr(MAGENTA)
<< setxy(10,4) << "Минск"
<< setxy(10,5) << "Москва"  << setxy(10,6) << "Киев";  constream win2;
int left2=45, top2=7, right2=68, bottom2=13;
win2.window(left2,top2,right2,bottom2);
win2.clrscr();
textattr((GREEN << 4) | WHITE);  for (i=1; i<=(right2-left2+1)*(bottom2-
top2+1); i++)  win2 << " ";
win2 << setxy(8,2) << "ГОСУДАРСТВА" << setclr(MAGENTA)
<< setxy(5,4) << "Республика Беларусь"
<< setxy(5,5) << "Россия"
<< setxy(5,6) << "Украина";
}
```

ТЕХНОЛОГИЯ ПРОГРАММИРОВАНИЯ. ПОНЯТИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ. ЖИЗНЕННЫЙ ЦИКЛ ПРОГРАММЫ. МОДЕЛИ ЖИЗНЕННОГО ЦИКЛА ПО

Понятие программного обеспечения. Понятие «программное обеспечение» вошло в жизнь с развитием компьютерной индустрии. Без него компьютер - это всего лишь электронное устройство, которое не управляется и поэтому не может приносить пользы.

Основной целью программного обеспечения (ПО) является функционирование вычислительной системы. Для понимания сути ПО нужно определить это понятие и его составляющие. ПО - это совокупность взаимосвязанных компьютерных программ, баз данных и документации. **Жизненный цикл программы.** Одним из базовых понятий методологии проектирования программ является понятие жизненного цикла программного обеспечения (ЖЦ ПО). ЖЦ ПО - это непрерывный процесс, который начинается с момента принятия решения о необходимости его создания и заканчивается в момент его полного изъятия из эксплуатации.

Основным нормативным документом, регламентирующим ЖЦ ПО, является международный стандарт ISO/IEC 12207 (ISO - International Organization of Standardization - Международная организация по стандартизации, IEC - International Electrotechnical Commission - Международная комиссия по электротехнике). Он определяет структуру ЖЦ, содержащую процессы, действия и задачи, которые должны быть выполнены во время создания ПО.

Структура ЖЦ ПО по стандарту ISO/IEC 12207 базируется на трех группах процессов:

- основные процессы ЖЦ ПО (приобретение, поставка, разработка, эксплуатация, сопровождение);
- вспомогательные процессы, обеспечивающие выполнение основных процессов (документирование, управление конфигурацией, обеспечение качества, верификация, аттестация, оценка, аудит, решение проблем);
- организационные процессы (управление проектами, создание инфраструктуры проекта, определение, оценка и улучшение самого ЖЦ, обучение).

Разработка включает в себя все работы по созданию ПО и его компонент в соответствии с заданными требованиями, включая оформление проектной и эксплуатационной документации, подготовку материалов, необходимых для проверки работоспособности и соответствующего качества программных продуктов, материалов, необходимых для организации обучения персонала и т.д. Разработка ПО включает в себя, как правило, анализ, проектирование и реализацию (программирование).

Эксплуатация включает в себя работы по внедрению компонентов ПО в эксплуатацию, в том числе конфигурирование баз данных и рабочих мест пользователей, обеспечение эксплуатационной документацией, проведение

обучения персонала и т.д., и непосредственно эксплуатацию, в том числе локализацию проблем и устранение причин их возникновения, модификацию ПО в рамках установленного регламента, подготовку предложений по совершенствованию, развитию и модернизации системы.

Управление проектом связано с вопросами планирования и организации работ, создания коллективов разработчиков и контроля за сроками и качеством выполняемых работ. Техническое и организационное обеспечение проекта включает выбор методов и инструментальных средств для реализации проекта, определение методов описания промежуточных состояний разработки, разработку методов и средств испытаний ПО, обучение персонала и т.п. Обеспечение качества проекта связано с проблемами верификации, проверки и тестирования ПО. Верификация - это процесс определения того, отвечает ли текущее состояние разработки, достигнутое на данном этапе, требованиям этого этапа. Проверка позволяет оценить соответствие параметров разработки с исходными требованиями. Проверка частично совпадает с тестированием, которое связано с идентификацией различий между действительными и ожидаемыми результатами и оценкой соответствия характеристик ПО исходным требованиям. В процессе реализации проекта важное место занимают вопросы идентификации, описания и контроля конфигурации отдельных компонентов и всей системы в целом.

Управление конфигурацией является одним из вспомогательных процессов, поддерживающих основные процессы жизненного цикла ПО, прежде всего процессы разработки и сопровождения ПО. При создании сложных проектов, состоящих из многих компонентов, каждый из которых может иметь разновидности или версии, возникает проблема учета их связей и функций, создания унифицированной структуры и обеспечения развития всей системы. Управление конфигурацией позволяет организовать, систематически учитывать и контролировать внесение изменений в ПО на всех стадиях ЖЦ. Общие принципы и рекомендации конфигурационного учета, планирования и управления конфигурациями ПО отражены в проекте стандарта ISO 12207-2. Каждый процесс характеризуется определенными задачами и методами их решения, исходными данными, полученными на предыдущем этапе, и результатами. Результатами анализа, в частности, являются функциональные модели, информационные модели и соответствующие им диаграммы. ЖЦ ПО носит итерационный характер: результаты очередного этапа часто вызывают изменения в проектных решениях, выработанных на более ранних этапах.

Модели жизненного цикла ПО

Стандарт ISO/IEC 12207 не предлагает конкретную модель ЖЦ и методы разработки ПО (под моделью ЖЦ понимается структура, определяющая последовательность выполнения и взаимосвязи процессов, действий и задач, выполняемых на протяжении ЖЦ. Модель ЖЦ зависит от специфики ИС и специфики условий, в которых последняя создается и функционирует). Его регламенты являются общими для любых моделей ЖЦ, методологий и

технологий разработки. Стандарт ISO/IEC 12207 описывает структуру процессов ЖЦ ПО, но не конкретизирует в деталях, как реализовать или выполнить действия и задачи, включенные в эти процессы.

К настоящему времени наибольшее распространение получили следующие две основные модели ЖЦ: каскадная модель (70-85 г.г.); спиральная модель (86-90 г.г.).

В изначально существовавших однородных программах каждое приложение представляло собой единое целое. Для разработки такого типа приложений применялся каскадный способ. Его основной характеристикой является разбиение всей разработки на этапы, причем переход с одного этапа на следующий происходит только после того, как будет полностью завершена работа на текущем (рис. 10.1). Каждый этап завершается выпуском полного комплекта документации, достаточной для того, чтобы разработка могла быть продолжена другой командой разработчиков.

Положительные стороны применения каскадного подхода заключаются в следующем:

- на каждом этапе формируется законченный набор проектной документации, отвечающий критериям полноты и согласованности;
- выполняемые в логичной последовательности этапы работ позволяют планировать сроки завершения всех работ и соответствующие затраты.
- сроки завершения всех работ и соответствующие затраты.

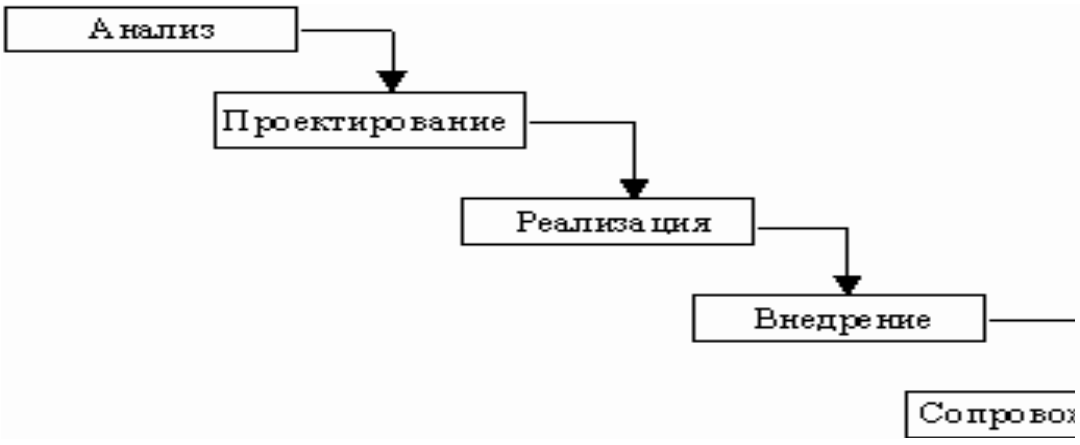


Рисунок 10.1 – Каскадная схема разработки ПО

Каскадный подход хорошо зарекомендовал себя при построении программ, для которых в самом начале разработки можно достаточно точно и полно сформулировать все требования, с тем чтобы предоставить разработчикам свободу реализовать их как можно лучше с технической точки зрения. В эту категорию попадают сложные расчетные системы, системы реального времени и другие подобные задачи. Однако, в процессе использования этого подхода обнаружился ряд его недостатков, вызванных прежде всего тем, что реальный процесс создания ПО никогда полностью не укладывался в такую жесткую схему. В процессе создания ПО постоянно возникала потребность в возврате к

предыдущим этапам и уточнении или пересмотре ранее принятых решений. В результате реальный процесс создания ПО принимал следующий вид (рис. 10.2):

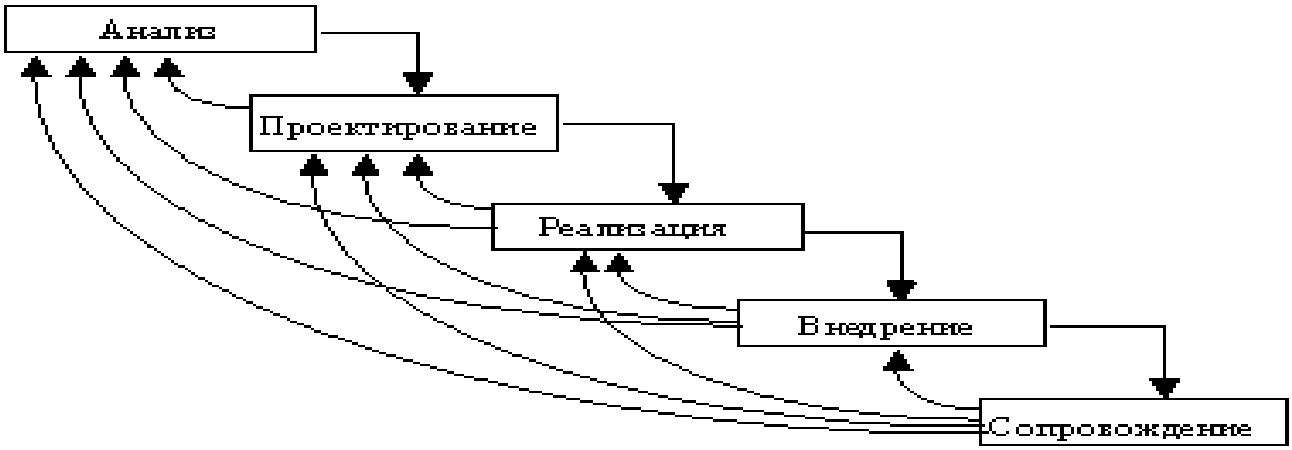


Рисунок 10.2 – Модификация каскадной схемы разработки ПО

Основным недостатком каскадного подхода является существенное запаздывание с получением результатов. Согласование результатов с пользователями производится только в точках, планируемых после завершения каждого этапа работ, требования к программе “заморожены” в виде технического задания на все время ее создания. Таким образом, пользователи могут внести свои замечания только после того, как работа над системой будет полностью завершена. В случае неточного изложения требований или их изменения в течение длительного периода создания ПО, пользователи получают систему, не удовлетворяющую их потребностям. Модели (как функциональные, так и информационные) автоматизируемого объекта могут устареть одновременно с их утверждением.

Для преодоления перечисленных проблем была предложена спиральная модель ЖЦ (рис. 10.3), делающая упор на начальные этапы ЖЦ: анализ и проектирование. На этих этапах реализуемость технических решений проверяется путем создания прототипов. Каждый виток спирали соответствует созданию фрагмента или версии ПО, на нем уточняются цели и характеристики проекта, определяется его качество и планируются работы следующего витка спирали. Таким образом углубляются и последовательно конкретизируются детали проекта и в результате выбирается обоснованный вариант, который доводится до реализации.

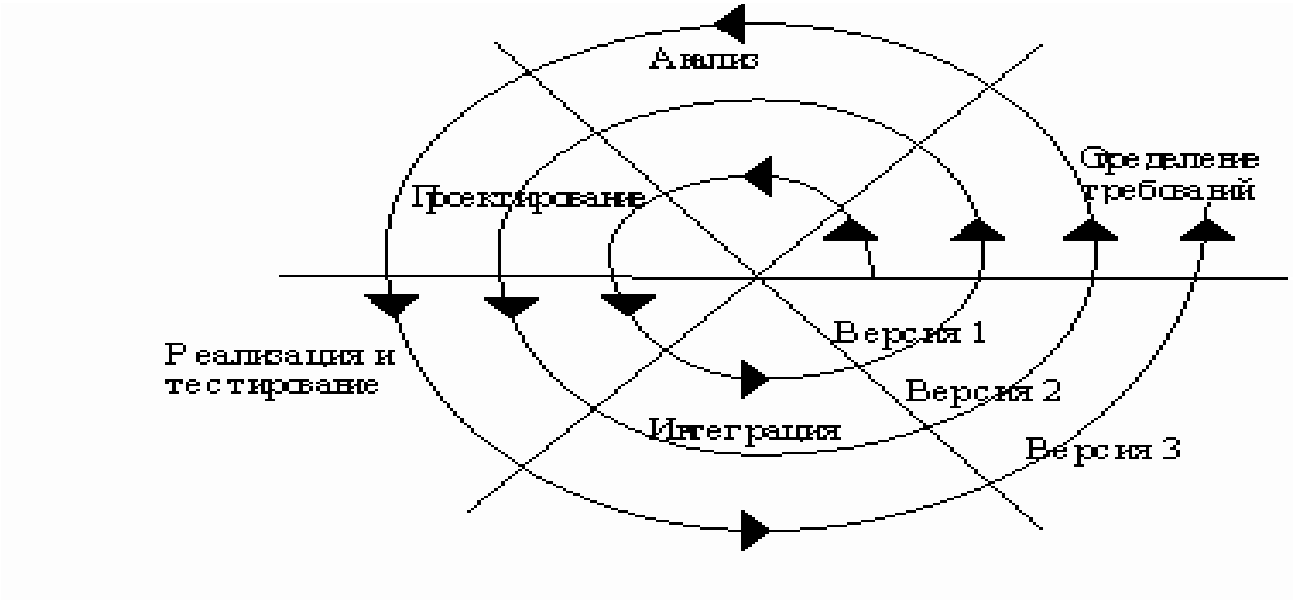


Рисунок 10.3 – Спиральная модель ЖЦ

Неполное завершение работ на каждом этапе позволяет переходить на следующий этап, не дожидаясь полного завершения работы на текущем. При итеративном способе разработки недостающую работу можно будет выполнить на следующей итерации. Главная же задача - как можно быстрее показать пользователям системы работоспособный продукт, тем самым активизируя процесс уточнения и дополнения требований.

Основная проблема спирального цикла - определение момента перехода на следующий этап. Для ее решения необходимо ввести временные ограничения на каждый из этапов жизненного цикла. Переход осуществляется в соответствии с планом, даже если не вся запланированная работа закончена. План составляется на основе статистических данных, полученных в предыдущих проектах, и личного опыта разработчиков.

II. ПРАКТИЧЕСКИЙ РАЗДЕЛ

МАТЕРИАЛЫ ДЛЯ ЛАБОРАТОРНЫХ РАБОТ

Лабораторная работа 1
Объектно-ориентированное программирование

1. Что определяет класс? Чем отличается класс от объекта?
2. Можно ли объявлять массив объектов? А массив классов?
3. Разрешается ли объявлять указатель на объект? А указатель на класс?
4. Можно ли совместить определение класса с объявлением объекта?
5. Объясните разницу между определением класса и объявлением класса.
6. Объясните, чем различаются два объявления указателя:

TClass *p = new TClass;

TClass *p = new TClass();

7. Как называется использование объекта одного класса в качестве поля другого класса?
8. Является ли структура классом? Чем класс отличается от структуры?
9. Какие ключевые слова в C++ обозначают класс?
10. Объясните принцип инкапсуляции.
11. Для чего нужны ключевые слова public и prIvate? Можно ли использовать ключевые слова public и private в структуре?
12. Существуют ли ограничения на использование ключевых слов public и prIvate в классе? А в структуре?
13. Обязательно ли делать поля класса приватными? Как инициализировать приватные поля класса?
14. Что такое «метод»? Как вызывается метод? Может ли метод быть приватным?
15. Как определить метод непосредственно внутри класса? А вне класса?
16. Объясните, что понимается под интерфейсом класса.
17. Что обозначается ключевым словом this? Для чего может использоваться конструкция *this?
18. Что такое композиция?
19. Разрешается ли внутри метода объявлять объекты «своего» класса? Как присваивать таким объектам начальное значение?
20. Сколько места в памяти занимает объект класса? Как это узнать?
21. Каков размер «пустого» объекта? Влияют ли методы на размер объекта?
22. Одинаков ли размер класса и аналогичной структуры?
23. Что такое выравнивание и от чего оно зависит? Влияет ли выравнивание на размер класса?
24. Покажите, как осуществить выравнивание полей класса по границе двух байтов.

25. Разрешается ли параметрам методов присваивать значение по умолчанию?

26. Объясните, почему методы, реализующие бинарные операции (например, сложение), должны иметь один параметр.

27. Объясните назначение директивы #pragma.

28. Какой принцип объектно-ориентированного программирования проявляется в перегрузке методов?

Варианты заданий

Во всех упражнениях, помимо указанных методов, обязательно должны быть реализованы метод инициализации Init(), метод ввода данных с клавиатуры Read (), метод вывода данных на экран Display ().

Для демонстрации работы с объектами нового типа во всех заданиях требуется написать главную функцию.

1. Создать класс Трапеция с методами вычисления площади и периметра. Определить, какие поля необходимы в классе. Площадь трапеции вычисляется по формуле

$$S = (a+b)*h/2.$$

2. Реализовать класс Power с двумя полями: first и second. Поле first — дробное число; поле second — дробное число, показатель степени. Реализовать метод power () — возведение числа first в степень second.

3. Реализовать функцию с именем make_Power(). Функция должна получать в качестве параметров значения для полей класса Power из предыдущего задания, а возвращать объект типа Power. При передаче ошибочных параметров — выводить сообщение и заканчивать работу.

4. Создать класс Double, имитирующий стандартный тип double. Помимо метода инициализации init(), методов ввода read() и вывода display(), реализовать методы сложения и вычитания, а также метод возведения в произвольную степень.

5. Создать класс Fraction для работы с дробными числами. Число должно быть представлено двумя полями: целая часть — целое со знаком, дробная часть — беззнаковое целое. Реализовать арифметические операции сложения, вычитания, умножения и сравнения на равенство.

6. Создать класс Decimal для работы с десятичными целыми числами со знаком произвольной длины. Число должно быть представлено строкой типа string, каждый символ которой — десятичная цифра, знак стоит слева первым и представлен символами + (плюс) или - (минус). Младшая цифра имеет младший индекс. Реализовать арифметические операции сложения и вычитания, а также сравнения на равенство и на больше.

7. Реализовать класс Bill, представляющий собой разовый платеж за телефонный разговор. Класс должен включать в себя поля номера телефона, тарифа за минуту разговора, скидки (в процентах), времени разговора (в минутах) и суммы к оплате. Реализовать метод вычисления суммы к оплате. В

главной программе продемонстрировать создание, инициализацию и обработку массива объектов типа Bill с различными исходными данными для вычисления сумм к оплате. Вычислить общую сумму к оплате.

8. Реализовать класс Money с двумя полями: first и second. Поле first — целое положительное число, номинал купюры; номинал может принимать значения 1, 2, 5, 10, 50, 100, 500. Поле second — целое положительное число, количество купюр данного достоинства. Реализовать метод summa() — вычисление суммы денег. В главной программе продемонстрировать создание, инициализацию и обработку массива объектов типа Honey с различными номиналами купюр. Вычислить общую сумму денег. 9. Создать класс Time с двумя полями, представляющими собой часы и минуты. Разработать метод, вычисляющий время через заданное количество часов и минут, а также метод, вычисляющий количество минут между двумя моментами времени.

10. Реализовать структуру Trio с тремя полями (целыми без знака). В структуре реализовать методы init(), read() и display(). Реализовать класс Date, использовав в качестве поля объект типа Trio. В классе реализовать методы получения и изменения отдельных полей даты, метод инициализации строкой вида «год.месяц.день» (например, «2006.04.07»), метод вычисления даты через заданное количество дней.

11. Создать класс BitString для работы с битовыми строками произвольной длины. Битовая строка должна быть представлена строкой типа string, один символ этой строки представляет собой один бит и принимает значение 0 или 1. Младший бит имеет младший индекс. Реализовать традиционные операции для работы с битами (and, or, xor, not).

12. Реализовать класс Account, представляющий собой банковский счет. В классе должны быть пять полей: фамилия владельца, номер счета, процент начисления, дата открытия счета и сумма в рублях. Для представления даты реализовать и использовать класс Date. Необходимо выполнять следующие операции: сменить владельца счета, снять некоторую сумму денег со счета, положить деньги на счет, начислить проценты, перевести сумму в доллары.

13. Создать класс Вектор, который содержит массив int, число элементов и переменную состояния. Определить индексатор. В переменную состояния устанавливать код ошибки, при определенной ситуации. Определить методы: сложения и умножения, которые производят эти операции с данными класса вектор и числом int. Создать массив объектов. Вывести: a) список векторов, содержащих 0;

b) список векторов с наименьшим модулем.

14. Создать класс Стек вещественных. Определить индексатор, методы проверки стека, добавления и удаления элементов.

Создать массив объектов. Вывести:

a) стеки с наименьшим/наибольшим верхним элементом;

b) список стеков, содержащих отрицательные элементы.

15. Создать класс типа - Множество. Методы: добавляют элемент к множеству, удаляют элемент, выводят текущее количество элементов множества.

Создать массив объектов. Вывести:

a) множества с наименьшей/наибольшей суммой элементов;
b) список множеств, содержащих отрицательные элементы.

Лабораторная работа 2.
Объектно-ориентированное программирование

Контрольные вопросы

1. Дайте определение конструктора. Каково назначение конструктора?
2. Перечислите отличия конструктора от метода.
3. Сколько конструкторов может быть в классе? Допускается ли перегрузка конструкторов?
4. Какие виды конструкторов создаются по умолчанию?
5. Может ли конструктор быть приватным? Какие последствия влечет за собой объявление конструктора приватным?
6. Приведите несколько случаев, когда конструктор вызывается неявно.
7. Как инициализировать динамическую переменную?
8. Как объявить константу в классе? Можно ли объявить дробную константу?
9. Каким образом разрешается инициализировать константные поля в классе?
10. В каком порядке инициализируются поля в классе? Совпадает ли этот порядок с порядком перечисления инициализаторов в списке инициализации конструктора?
11. Какие конструкции C++ разрешается использовать в списке инициализации в качестве инициализирующих выражений?
12. Влияет ли наличие целочисленных констант-полей на размер класса?
13. Объясните, что такое «инициализация нулем».
14. Что такое деструктор? Может ли деструктор иметь параметры? Допускается ли перегрузка деструкторов?
15. Зачем нужны константные методы? Чем отличается определение константного метода от определения обычного?
16. Может ли константный метод вызываться для объектовпеременных? А обычный метод – для объектов-констант?

Варианты заданий

Во всех упражнениях, помимо указанных методов, обязательно должны быть **реализованы набор необходимых конструкторов**, метод ввода данных с клавиатуры Read(), метод вывода данных на экран Display(). Конструкторы должны проверять корректность задаваемых значений параметров. **Подходящие методы должны быть реализованы как константные.**

Для демонстрации работы с объектами нового типа во всех заданиях требуется написать главную функцию.

1. Реализовать класс Power с двумя полями: first и second. Поле first – дробное число; поле second – дробное число, показатель степени. Реализовать конструктор без аргументов, присваивающий полям нулевые значения, и конструктор инициализации, присваивающий второму полю значение 0 по умолчанию. Реализовать метод power() — возведение числа first в степень second.

2. Создать класс Double, имитирующий стандартный тип double, реализовав подходящие конструкторы. Реализовать методы методов ввода read() и вывода display(),методы сложения и вычитания, а также метод возведения в произвольную степень.

3. Создать класс LongInteger с двумя полями: high и low, представляющими собой старшую и младшую части числа. Реализовать конструкторы инициализации целым любого типа со значениями по умолчанию и строкой цифр. Реализовать методы сравнения на больше и на равенство, а также метод сложения. Реализовать функцию преобразования в строку toString().

4. Реализовать класс Date с тремя полями: день, месяц и год. Реализовать три конструктора инициализации: с тремя целыми аргументами (минутам и секундам присвоить значения по умолчанию), строкой вида «год.месяц.день» (например, «2020.04.07»), одним целым числом вида ггггммдд. Для представления месяцев определить в классе перечислимый тип month. В классе реализовать методы получения и изменения отдельных полей даты, метод вычисления даты через заданное количество дней, метод вычисления количества дней между датами.

5. Создать класс Time для работы со временем в формате «час:минута:секунда». Класс должен включать в себя три конструктора инициализации: числами, строкой (например, «23:59:59»), секундами. Конструктор инициализации тремя числами должен присваивать значения по умолчанию минутам и секундам. Реализовать операции вычисления разницы между двумя моментами времени в секундах, сложения времени и заданного количества секунд, вычитания из времени заданного количества секунд, сравнения моментов времени. Реализовать функцию преобразования в строку toStrlng().

6. Реализовать класс Account, реализовать и использовать в нем класс TMoney для представления суммы счета и класс Date для представления даты. Реализовать необходимые конструкторы инициализации.

7. Реализовать класс Fraction, реализовать и использовать в нем структуру Pair. Структура включает в себя два поля подходящего типа: first и second. Обеспечить реализацию конструктора инициализации структуры Pair и конструкторов инициализации класса Fraction с присвоением значений по умолчанию. Класс Fraction предназначен для работы с дробными числами. Число должно быть представлено двумя полями: целая часть — целое со знаком,

дробная часть — беззнаковое целое. Реализовать арифметические операции сложения, вычитания, умножения и сравнения на равенство. Реализовать функцию преобразования в строку toString().

8. Создать класс Decimal для работы с десятичными целыми числами со знаком произвольной длины. Число должно быть представлено строкой типа string, каждый символ которой – десятичная цифра, знак стоит слева первым и представлен символами + (плюс) или - (минус). Младшая цифра имеет младший индекс. Реализовать арифметические операции сложения и вычитания, а также сравнения на равенство и на больше. Реализовать в классе Decimal конструкторы инициализации числами любого целого типа и строкой.

9. Создать класс BitString для работы с битовыми строками длиной 64 бита. Битовая строка должна быть представлена двумя полями типа unsigned long. Конструкторы инициализации должны обеспечивать инициализацию полей целыми числами любого типа и строкой типа string. Реализовать традиционные операции для работы с битами (and, or, xor, not).

10. Рациональная (несократимая) дробь представляется парой целых чисел (a, b), где a – числитель, b – знаменатель. Создать класс Rational для работы с рациональными дробями. Реализовать конструкторы инициализации с параметрами по умолчанию. Один из конструкторов должен принимать параметр-структуру с двумя полями, first и second, целого типа. Обязательно должны быть реализованы операции:

1) сложения add:

$$(a, b) + (c, d) = (ad + bc, bd)$$

2)вычитания sub:

$$(a, b) - (c,d) = (ad - bc, bd)$$

3) умножения mu1:

$$(a, b)\ (c, d) = (ac, bd)$$

4) деления div:

$$(a, b)/(c, d) = (ad, bc);$$

5) сравнения equal (равно), greate (больше), less (меньше).

Должна быть также реализована приватная функция сокращения дроби reduce, которая обязательно вызывается при выполнении арифметических операций.

11. Построить описание класса, содержащего информацию о почтовом адресе организации. Предусмотреть возможность раздельного изменения составных частей адреса, создания и уничтожения объектов этого класса.

Написать программу, демонстрирующую работу с этим классом. Программа должна содержать меню, позволяющее осуществить проверку всех методов класса.

12. Составить описание класса для представления комплексных чисел. Реализовать все виды конструкторов. Обеспечить выполнение операций сложения, вычитания и умножения комплексных чисел.

Написать программу, демонстрирующую работу с этим классом. Программа должна содержать меню, позволяющее осуществить проверку всех методов класса.

13. Составить описание класса для объектов-векторов, задаваемых координатами концов в трехмерном пространстве. Реализовать все виды конструкторов. Обеспечить операции сложения и вычитания векторов с получением нового вектора (суммы или разности), вычисления скалярного произведения двух векторов, длины вектора, косинуса угла между векторами.

Написать программу, демонстрирующую работу с этим классом. Программа должна содержать меню, позволяющее осуществить проверку всех методов класса.

14. Описать класс «домашняя библиотека». Реализовать все виды конструкторов. Предусмотреть возможность работы с произвольным числом книг, поиска книги по какому-либо признаку (например, по автору или по году издания), добавления книг в библиотеку, удаления книг из нее, сортировки книг по разным полям.

Написать программу, демонстрирующую работу с этим классом. Программа должна содержать меню, позволяющее осуществить проверку всех методов класса.

15. Описать класс «записная книжка». Реализовать все виды конструкторов. Предусмотреть возможность работы с произвольным числом записей, поиска записи по какому-либо признаку (например, по фамилии, дате рождения или номеру телефона), добавления и удаления записей, сортировки по разным полям.

Написать программу, демонстрирующую работу с этим классом. Программа должна содержать меню, позволяющее осуществить проверку всех методов класса.

Лабораторная работа 3
Перегрузка операций

1. Цель и задачи работы, требования к результатам ее выполнения

Цель работы состоит в овладении навыками разработки программ на языке Си++, использующих перегрузку стандартных операций. Для достижения цели необходимо выполнить следующие задачи:

- изучить учебные материалы, посвященные перегрузке стандартных операций в языке Си++ [3, 7];
- разработать программу на языке Си++ для решения заданного варианта;
- отладить программы;
- выполнить решение контрольного примера с помощью программы и ручной расчет контрольного примера.

Выполнив работу, нужно подготовить отчет.

2. Краткая характеристика объекта изучения

Перегрузка операций в языке Си++ — это возможность распространения действия стандартных операций на операнды, для которых операции первоначально не предназначались. Такое возможно, если хотя бы один из операндов является объектом класса. Для этого создается специальная так называемая оператор-функция, которая может быть как членом класса, так и функцией, не принадлежащей классу.

Формат определения оператор-функции имеет вид:

<тип_возвращаемого_значения> operator
<знак_операции>
(спецификация_параметров)
{
операт оры_т ела_функции
}

Существует три способа перегрузки. Оператор-функция определяется как функция:

- не принадлежащая классу; • принадлежащая классу;
- дружественная классу.

Особенности перегрузки операций. Можно перегружать только стандартные операции. Например, нельзя перегрузить операцию '**' (возведение в степень в языке Фортран — отсутствует в языке Си++). Не допускают перегрузки операции: '.', '.*', '?:', '::', 'sizeof', '#', '##'. При перегрузке сохраняется арность операций (унарная операция остается унарной, а бинарная – бинарной). Бинарная операция перегружается либо как функция, не принадлежащая классу с двумя параметрами, один из которых обязательно объект (ссылка на объект) класса, либо как функция класса с одним параметром; первым операндом операции выступает объект класса, для которого вызывается функция. Бинарные операции '=', '->', '[]', должны обязательно определяться как компонентные функции класса. Унарная операция перегружается либо как функция, не принадлежащая классу с одним параметром – объектом (ссылкой на объект) класса, либо как функция класса без параметров, операндом операции выступает объект класса, для которого вызывается функция.

3. Задачи и порядок выполнения работы

Студент должен создать в работе класс и необходимые оператор-функции для перегрузки заданных в своем варианте операций. Особое внимание следует обратить на способы перегрузки унарных и бинарных операций и параметры оператор-функций для этих операций, когда операторфункция является членом класса и когда не является. Знать те случаи, когда оператор-функция должна быть обязательно членом класса и случаи, когда оператор-функция обязательно не принадлежит классу. При защите работы необходимо обосновать выбор способа определения оператора-функции – внутри класса или вне его. При необходимости, студент

выполняет расчет вручную, чтобы проверить работу программы для задачи небольшой размерности. Результаты работы программы и этого расчета нужно представить в отчете.

Условие задачи

Дан класс (например, с именем Vector), задающий вектор размерности n. Поля класса: указатель на массив, задающий вектор (тип элемента double), количество элементов (размерность) вектора (тип int). Массив нужно создавать динамически. Класс включает в себя: конструктор без параметров, задающий пустой вектор (количество элементов равно 0); конструктор, создающий объект «вектор» на основе обычного одномерного массива размерности n; деструктор.

Необходимо перегрузить операции и продемонстрировать их работу. Перегрузить операцию «[]» (обращение к элементу вектора по индексу), операцию «=» (копирование вектора или создание копии вектора), операцию «*» (умножение числа на вектора), на выходе – вектор такой же размерности, каждый элемент которого равен произведению соответствующего элемента исходного вектора на число.

Пример выполнения работы

Для решения задачи в среде Microsoft Visual Studio было создано стандартное консольное приложение. Исходные коды основного файла проекта (с расширением .cpp) и файла stdafx.h приведены ниже.

Листинг программы с комментариями

```cpp
#include "stdafx.h"
class Vector                        // Класс "вектор".
{
    double *p;                      // Указатель на массив (вектор) ,
    int n;      /* Размерность вектора (количество элементов) массива */
public:
    Vector(double *p, int n) /* Конструктор на входе массив, задающий вектор */
    {
        this->n = n;                // Задать количество элементов.
        this->p = new double [n];   // Выделение памяти,
        for (int i = 0; i < n; i++) this->p[i] = p[i]; // Копирование одного массива в другой
    }
    void print()                    // Печать вектора (массива).
    {
        std::cout << "\n";
        for (int i = 0; i < n; i++)
            std::cout << p[i] << " ";
    }
    Vector() { p = 0; n = 0; }      /* Конструктор без параметров, задает «пустой» объект */
    double & operator[](int index)  /* Оператор-функция (перегрузка операции обращения к элементу) */
    {
        return p[index];
    }
    Vector & operator = (Vector & v2)   /* Оператор-функция копирования объекта */
    {
        n = v2.n; p = new double[n];
        for (int i = 0; i < n; i++) p[i] = v2.p[i];
        return *this;
    }
    ~Vector()   // Деструктор.
    {
        if (n > 0) delete[] p;      // Освобождение памяти,
    }
    // Дружественная функция, определенная вне класса.
    friend Vector & operator *(double x, Vector & v2);
};
    /* Умножение числа на вектор (первый операнд – не объект класса, функция обязательно определяется вне класса)*/
Vector & operator *(double x, Vector & v2)  /* Оператор функция вне класса */
{
    double *p = new double[v2.n];   // Создание нового массива,
    for (int i = 0; i<v2.n; i++) p[i] = x*v2.p[i]; /* Заполнение массива */
    Vector *pV = new Vector(p, v2.n);   /* Создание нового объекта на основе массива */
    delete[] p;                     // Освобождение массива,
    return *pV;                     // Возвращение ссылки на объект.
}
int main(int argc, char* argv[])
{
    double m1[] = { 1, 2, 3, 4.5, 7 };
    Vector V1(m1, 5);               // Создание объекта.
    V1.print();                     // Печать объекта.
    std::cout << "\n";
    for (int i = 0; i < 5; i++)
        std::cout << V1[i] << " ";  // Пример обращения к операции [] .
    V1 [0]  = 10.6;                 // Пример обращения к операции [] .
    V1.print();                     // Печать объекта.
    Vector V2;                      // Новый объект (вначале «пустой»).
    V2 = 100 * V1;      /* Пример выполнения перегруженной операции */
    // V2=operator *(100, VI);
    V2.print();                     // Печать полученного объекта.
    system ("pause") ;  /* Остановка программы до нажатия любой клавиши */
    return 0;
}
```

Листинг1. Файл реализации с расширением .cpp

```cpp
#pragma once    // препроцессорная директива, разработанная для контроля за тем,
                // чтобы конкретный исходный файл при компиляции подключался строго один раз

#include <stdio.h>
#include <tchar.h>
#include <stdlib.h>
#include <iostream>
using namespace std;
```

Листинг 2. Заголовочный файл stdafx.h Задания

для самоконтроля

1. Назовите способы перегрузки операций в языке Си++.

2. Раскройте особенности перегрузки бинарной операции, когда первый операнд не является объектом класса.

3. Укажите, в каких целях в некоторых случаях оператор-функцию делают дружественной функцией класса.

4. Перечислите операции, для перегрузки которых оператор-функция обязательно должна принадлежать классу.

Варианты заданий для лабораторной работы

Дан класс (например, с именем Vector), задающий вектор размерности n. Поля класса: указатель на массив, задающий вектор (тип элемента int или double, в зависимости от варианта), — массив должен создаваться динамически, число элементов (размерность) вектора (тип int). Класс включает в себя: конструктор без параметров, задающий пустой вектор (число элементов равно 0), конструктор, создающий объект «вектор» на основе обычного одномерного массива размерности n, деструктор.

Необходимо перегрузить операции и продемонстрировать их работу. Перегрузить операцию «[]» (обращение к элементу вектора по индексу) и операцию «=» (копирование вектора или создание копии вектора). Остальные перегружаемые операции выбрать в соответствии со своим вариантом. Варианты заданий представлены в таблице.

Контрольные вопросы

1. Какие операции нельзя перегружать? Как вы думаете, почему?

2. Можно ли перегружать операции для встроенных типов данных?

3. Можно ли при перегрузке изменить приоритет операции?

4. Можно ли определить новую операцию?

5. Можно ли перегрузить операцию «запятая»?

6. Чем отличается функциональная форма вызова бинарной операции от инфиксной формы?

7. Перечислите особенности перегрузки операций как методы класса. Чем отличается перегрузка внешним образом от перегрузки как метод класса?

8. Какой результат должны возвращать операции с присваиванием?

9. Объясните, что такое 1-значение.

10. Как различаются перегруженная префиксная и постфиксная операции инкремента и декремента?

11. Что означает выражение *this? В каких случаях оно используется?

12. Какие операции не рекомендуется перегружать как методы класса? Почему?

13. Объясните, почему при перегрузке операций нельзя задавать параметры по умолчанию?

14. Какие операции разрешается перегружать только как методы класса?

15. Перечислите все возможные способы задания явного преобразования типов.

16. Дайте определение дружественной функции. Как объявляется дружественная функция? А как определяется?

17. Объясните, какую операцию выполняет конструкция static_cast<>?

18. Какой вид конструктора фактически является конструктором преобразования типов?

19. Чем различаются операции reinterpret_cast<> и static_cast<>?

20. Для чего нужны функции преобразования? Как объявить такую функцию в классе?

21. Объясните назначение операции const_cast<>.

22. Как запретить неявное преобразование типа, выполняемое конструктором инициализации?

23. Какие проблемы могут возникнуть при определении функций преобразования?

24. Для чего служит ключевое слово explicit?

25. Как с помощью функции преобразования и конструктора инициализации определить «новые» операции для встроенных типов данных.

Варианты заданий

Номер вариан-та	Операция	Типы операндов и результата для перегруженной операции			Тип элемента вектора	Описание назначения перегруженной операции
		Первый операнд	Второй операнд	Результат		
1	+	Vector&	Vector&	Vector&	double	Сложение векторов одинаковой размерности; на выходе вектор такой же размерности, элемент которого равен сумме соответствующих элементов двух векторов
2	+	Vector&	double *	Vector&	double	
3	+	double *	Vector&	Vector&	double	
4	+	Vector&	Vector&	Vector&	double	Сложение векторов; на выходе вектор, длина которого равна сумме длин векторов; вначале идут элементы первого вектора, затем второго; если один из векторов задан обычным массивом, то считать, что его длина равна длине вектора, заданного объектом класса
5	+	Vector&	double *	Vector&	double	
6	+	double *	Vector&	Vector&	double	

Номер вариан-та	Операция	Типы операндов и результата для перегруженной операции			Тип элемента вектора	Описание назначения перегруженной операции
		Первый операнд	Второй операнд	Результат		
7	*	Vector&	double	Vector&	double	Умножение вектора на число; на выходе вектор такой же размерности, каждый элемент которого равен произведению
8	*	double	Vector&	Vector&	double	соответствующего элемента исходного вектора на число
9	++	Vector&		Vector& (тот же объект)	int	Каждый элемент исходного вектора увеличивается на 1; оператор-функция должна возвращать ссылку на тот же объект
10	*	Vector&	Vector&	double	double	Скалярное произведение векторов (одинаковой размерности); на выходе значение этого произведения
11	*	Vector&	double *	double	double	
12	*	double *	Vector&	double	double	
13	^	Vector&	Vector&	Vector&	int	Побитовая операция с двумя векторами одинаковой размерности, на выходе вектор такой же размерности, элемент которого равен результату побитовой операции ^ соответствующих элементов двух векторов
14	^	Vector&	int*	Vector&	int	
15	^	int*	Vector&	Vector&	int	
16	--	Vector&		Vector& (тот же объект)	int	Каждый элемент исходного вектора уменьшается на 1, оператор-функция должна возвращать ссылку на тот же объект
17	&	Vector&	Vector&	Vector&	int	Побитовая операция с двумя векторами одинаковой размерности; на выходе вектор такой же размерности, элемент которого равен результату побитовой операции & соответствующих элементов двух векторов
18	&	Vector&	int*	Vector&	int	
19	&	int*	Vector&	Vector&	int	
20	|	Vector&	Vector&	Vector&	int	Побитовая операция с двумя векторами одинаковой размерности; на выходе вектор такой же размерности, элемент которого равен результату побитовой операции | соответствующих элементов двух векторов
21	|	Vector&	int*	Vector&	int	
22	|	int*	Vector&	Vector&	int	
23	<<	ostream&	Vector&	ostream&	double	Поразрядный сдвиг влево; реализовать для вставки всех элементов вектора в поток вывода через пробелы; вначале в поток записывается размерность вектора, затем элементы
24	>>	istream&	Vector&	istream &	double	Поразрядный сдвиг вправо; реализовать для чтения всех элементов вектора из потока ввода; вначале из потока читается размерность вектора, затем элементы

Номер вариан-та	Операция	Типы операндов и результата для перегруженной операции			Тип элемента вектора	Описание назначения перегруженной операции
		Первый операнд	Второй операнд	Результат		
25	<<	Vector&	int	Vector &	int	Поразрядный сдвиг влево; реализовать для циклического сдвига всех элементов вектора влево на заданное число позиций (число позиций определяет второй операнд операции)
26	>>	Vector&	int	Vector &	int	Поразрядный сдвиг вправо; реализовать для циклического сдвига всех элементов вектора вправо на заданное число позиций (число позиций определяет второй операнд операции)

Лабораторная работа 4
Изучение возможностей наследования классов

1. Цель и задачи работы, требования к результатам ее выполнения

Цель работы состоит в овладении навыками разработки программ на языке Си++, использующих возможности наследования классов для решения различных задач. Для достижения цели необходимо выполнить следующие задачи:

- изучить учебные материалы, посвященные наследованию классов в языке Си++;
- разработать программу на языке Си++ для решения заданного варианта;
- отладить программы;
- представить результаты работы программы.

Выполнив работу, нужно подготовить отчет.

Общие сведения о наследовании классов

Основная идея, используемая при наследовании классов, заключается в том, что на основе существующего класса (класс-родитель, или базовый класс), создается производный класс (класс-наследник, дочерний класс), который включает в себя поля и функции базового класса (наследует их) и содержит дополнительные поля (обладает новыми свойствами) и функции [3, 7].

Примеры определения производных классов:

```
class S: X, Y, Z
{…};
class B: public A
{…};
class D: public X, protected B
{…};
```

Статусы доступа при наследовании классов

Перед именем базового класса можно указать статус доступа наследования (одно из ключевых слов public, protected, private). Статус доступа наследования определяет статус доступа наследуемых полей и функций из базового класса внутри производного класса.

Производный класс может определяться с ключевым словом struct или class (с ключевым словом union производный класс не определяется). Если производный класс определен с ключевым словом class, то по умолчанию статус доступа наследования private.

Если производный класс определен с ключевым словом struct, то по умолчанию статус доступа наследования public.

Статусы доступа при наследовании классов представлены в таблице.

Статусы доступа при наследовании классов

Тип наследования доступа	Доступность в базовом классе	Доступность компонент базового класса в производном
public	public	public
	protected	protected
	private	недоступны
protected	public	protected
	protected	protected
	private	недоступны
private	public	private
	protected	private
	private	недоступны

Особенности конструкторов при наследовании

Конструктор производного класса в первую очередь должен вызывать конструктор базового класса. Если это действие не выполняется явно, то, по умолчанию, вызывается конструктор без параметров (если он есть, если его нет, будет ошибка). Если класс имеет несколько базовых, то конструкторы базовых классов должны вызываться в порядке перечисления этих классов в списке базовых.

Особенности деструкторов при наследовании

Деструктор производного класса всегда неявно, по умолчанию после выполнения своего тела вызывает деструкторы базовых классов, причем порядок разрушения объекта (вызовов деструкторов) обратен порядку создания (вызова конструкторов).

Переопределение функций. Виртуальные функции

Если в производном классе объявлена функция с именем, типом возвращаемого значения и количеством и типами параметров, такая же, как в базовом классе, то данная функция является переопределенной. (Нельзя путать с перегрузкой функций!) С помощью простых переопределенных функций

реализуется механизм статического полиморфизма. (Полиморфизм — возможность функции в производном классе работать по-другому.)

Суть статического связывания: когда указатель одного типа ссылается на объект другого типа при наследовании классов, выбор переопределенного метода определяется типом указателя, а не типом объекта.

Суть динамического связывания: когда указатель одного типа ссылается на объект другого типа при наследовании классов, выбор переопределенного метода определяется типом объекта, а не типом указателя, для этого переопределенный метод должен быть объявлен виртуальным в базовом классе. С помощью динамического связывания реализуется механизм динамического полиморфизма.

Динамический полиморфизм при переопределении метода в производном классе обеспечивается объявлением заголовка метода с ключевым словом virtual в базовом классе. Встретив у функции модификатор virtual, компилятор создает для класса таблицу виртуальных функций, а в класс добавляет новый скрытый для программиста член — указатель на эту таблицу.

Таблица виртуальных функций хранит в себе адреса всех виртуальных методов класса (по сути, это массив указателей), а также всех виртуальных методов базовых классов этого класса. За счет этих указателей на функции обеспечивается динамическое связывание: при вызове метода через указатель, который имеет тип базового класса, но содержит адрес объекта производного класса (такое преобразование допустимо и выполняется неявно), вызывается именно метод производного класса (вызов метода определяется типом объекта).

Задачи и порядок выполнения работы

В работе первоначально необходимо создать базовый класс и на его основе создать производный класс. При этом необходимо использовать вызов конструктора базового класса внутри конструктора производного класса, а также возможности переопределения методов (функций) класса. Следует разобраться с понятием виртуального метода, а также статическим и динамическим полиморфизмом. Продемонстрировать данные возможности в выполняемом примере.

Условие задачи

Создать базовый класс «точка на плоскости». Элементы класса: поля, задающие координаты точки; конструктор для инициализации полей; функция для печати значений полей. Создать производный класс «точка в трехмерном пространстве». Элементы класса: дополнительное поле, задающее дополнительную координату; конструктор для инициализации полей; переопределенная функция для печати значений полей (внутри переопределенной функции в первую очередь должна вызываться функция из базового класса). Создать по одному объекту каждого из классов. Показать вызов созданных функций. При переопределении функций обеспечить динамический полиморфизм, показать его особенности в программе.

Пример выполнения работы

Для решения задачи в среде Microsoft Visual Studio было создано стандартное консольное приложение. В файле реализации проекта добавлен исходный код, приведенный ниже.

Листинг программы с комментариями

```
#include <stdlib.h>
#include <iostream> using namespace std;
class point // Базовый класс «Точка на плоскости».
{
    double x, y; // Координаты точки,
public:
    point(double x, double y) /* Конструктор для инициализации полей */
    {
        this->x = x; this->y = y;
    }
    virtual void print() /* Метод для печати полей (виртуальный) */
    {
        std::cout << "\nx= " << x << " y= " << y; /* Печать значения полей */
    }
};
class point3d: public point /* Производный класс «Точка в пространстве»*/
{
    double z; // Новое поле – координата z.
public:
    point3d(double x, double y, double z): // Конструктор,
    point (x, y) // Явный вызов конструктора базового класса.
    {
        this->z = z;
    }
    void print( )   // Переопределенный метод print.
    {
    point::print(); /* Вызов метода базового класса в переопределенном методе */
    std::cout << " z= " << z << std::endl; // Допечатывание поля z.
    }
};
int main(int argc, char* argv[])
{
    point p1(1, 2); // Создание объекта с вызовом конструктора,
    point3d p3{ 3, 4, 5 }; /* Создание объекта с вызовом конструктора */
    point *pp; // Указатель типа базового класса.
    pp=&p1; // Настройка указателя на объект базового класса,
    pp->print(); // Вызов метода через указатель.
    pp=&p3; /* Настройка указателя на объект производного класса (преобразование типа допустимо) */
    pp->print (); /* Вызов метода через указатель, вызывается метод класса point3d.
                  Если метод print в классе point был объявлен без virtual,
                  то вызывался бы метод print класс point */
    system("pause"); /* Остановка программы до нажатия любой клавиши */
    return 0;
}
```

Задания и вопросы для самоконтроля

1. Раскройте понятие производного класса (класса-наследника) в объектно-ориентированном программировании.

2. Перечислите статусы доступа наследуемых полей и методов в производных классах. Как они определяются?

3. Раскройте особенности конструкторов в производных классах.

4. Раскройте особенности деструкторов в производных классах.

5. Дайте понятие переопределения функции в производном классе.

6. Раскройте понятие виртуальной функции.

7. Что такое статическое и динамическое связывание в языке Си++?

Варианты заданий

Вариант № 1

Создать базовый класс «Вектор на плоскости». Элементы класса: поля, задающие координаты точки (статус доступа protected), определяющей конец вектора (начало вектора находится в точке с координатами 0; 0); конструктор для инициализации полей; функция для вычисления длины вектора, функция для печати полей и длины вектора. Создать производный класс «Вектор в трехмерном пространстве». Элементы класса: дополнительное поле, задающее дополнительную координату; конструктор для инициализации полей; переопределенная функция для вычисления длины вектора; переопределенная функция для печати полей и длины вектора. Создать по одному объекту каждого из классов. Показать вызов созданных функций. При переопределении функций обеспечить динамический полиморфизм, показать его особенности в программе.

Вариант № 2

Создать базовый класс «Точка на плоскости». Элементы класса: поля, задающие координаты точки (статус доступа protected); конструктор для инициализации полей; функция для печати значений полей. Создать производный класс «Точка в трехмерном пространстве». Элементы класса: дополнительное поле, задающее дополнительную координату; конструктор для инициализации полей; переопределенная функция для печати значений полей (внутри переопределенной функции в первую очередь должна вызываться функция из базового класса). Создать по одному объекту каждого из классов. Показать вызов созданных функций. При переопределении функций обеспечить динамический полиморфизм, показать его особенности в программе.

Вариант № 3

Создать базовый класс «Квадрат». Элементы класса: поле, задающее длину стороны (статус доступа protected); конструктор для инициализации поля; функция для вычисления площади квадрата; функция для печати поля и площади квадрата. Создать производный класс «Куб». Элементы класса: конструктор для инициализации поля; переопределенная функция для вычисления объема (вместо площади) куба (внутри переопределенной функции должна вызываться функция из базового класса). Создать по одному объекту каждого из классов. Показать вызов созданных функций. При переопределении функций обеспечить динамический полиморфизм, показать его особенности в программе.

Вариант № 4

Создать базовый класс «Прямоугольник». Элементы класса: поля, задающие длины сторон (статус доступа protected); конструктор для инициализации полей; функция для вычисления площади прямоугольника; функция для печати полей и значения площади. Создать производный класс «Прямоугольный параллелепипед». Элементы класса: дополнительное поле, задающее высоту; конструктор для инициализации полей; переопределенная функция для вычисления объема (вместо площади) прямоугольного параллелепипеда (внутри переопределенной функции должна вызываться функция из базового класса); переопределенная функция для печати полей и значения объема. Создать по одному объекту каждого из классов. Показать вызов созданных функций. При переопределении функций обеспечить динамический полиморфизм, показать его особенности в программе.

Вариант № 5

Создать базовый класс «Круг». Элементы класса: поле, задающее радиус; конструктор для инициализации поля (статус доступа protected); функция для вычисления площади круга (площадь круга □□); функция для печати полей и площади. Создать производный класс «Шар». Элементы класса: конструктор для инициализации поля; переопределенная функция для вычисления объема (вместо площади круга) шара (площадь шара 4/□□). Создать по одному объекту каждого из классов. Показать вызов созданных функций. При переопределении функций обеспечить динамический полиморфизм, показать его особенности в программе.

Вариант № 6

Создать базовый класс «Автомобиль». Элементы класса: поле, содержащее наименование модели автомобиля; поле, содержащее значение максимальной скорости (статус доступа protected); конструктор для инициализации полей; функция для печати параметров автомобиля. Создать производный класс «Грузовой автомобиль». Элементы класса: дополнительно поле, содержащее грузоподъемность автомобиля в тоннах; конструктор для инициализации полей; переопределенная функция печати параметров автомобиля (внутри переопределенной функции должна вызываться функция из базового класса). Создать по одному объекту каждого из классов. Показать вызов созданных функций. При переопределении функций обеспечить динамический полиморфизм, показать его особенности в программе.

Вариант № 7

Создать базовый класс «Вещественное число». Элементы класса: поле, задающее значение числа (статус доступа protected); конструктор для инициализации поля; функция для вычисления модуля числа; функция для печати поля и модуля числа. Создать производный класс «Комплексное число». Элементы класса: дополнительно поле, задающее значение мнимой части числа; конструктор для инициализации полей; переопределенная функция для вычисления модуля числа (модуль числа — корень квадратный из суммы квадратов вещественной и мнимой частей числа); переопределенная функция

для печати полей и модуля числа. Создать по одному объекту каждого из классов. Показать вызов созданных функций. При переопределении функций обеспечить динамический полиморфизм, показать его особенности в программе.

Вариант № 8

Создать базовый класс «Вектор на плоскости». Элементы класса: поля, задающие координаты точки (статус доступа protected), определяющей конец вектора (начало вектора находится в точке с координатами 0, 0); конструктор для инициализации полей; функция для печати координат вектора. Создать производный класс «Вектор в трехмерном пространстве». Элементы класса: дополнительное поле, задающее дополнительную координату; конструктор для инициализации полей; переопределенная функция для печати координат вектора (внутри переопределенной функции должна вызываться функция из базового класса). Создать по одному объекту каждого из классов. Показать вызов созданных функций. При переопределении функций обеспечить динамический полиморфизм, показать его особенности в программе.

Вариант № 9

Создать базовый класс «Квадрат». Элементы класса: поле, задающее длину стороны (статус доступа protected); конструктор для инициализации поля; функция для вычисления периметра квадрата; функция для печати длины стороны и периметра. Создать производный класс «Прямоугольник». Элементы класса: дополнительное поле, задающее другую сторону; конструктор для инициализации полей; переопределенная функция для вычисления периметра прямоугольника; переопределенная функция для печати длин сторон и периметра. Создать по одному объекту каждого из классов. Показать вызов созданных функций. При переопределении функций обеспечить динамический полиморфизм, показать его особенности в программе.

Вариант № 10

Создать базовый класс «Автомобиль». Элементы класса: поле, содержащее наименование модели автомобиля; поле, содержащее значение максимальной скорости (статус доступа protected); конструктор для инициализации полей; функция для печати параметров автомобиля. Создать производный класс «Автобус». Элементы класса: дополнительно поле, содержащее максимальное число перевозимых пассажиров; конструктор для инициализации полей; переопределенная функция печати параметров автобуса (внутри переопределенной функции должна вызываться функция из базового класса). Создать по одному объекту каждого из классов. Показать вызов созданных функций. При переопределении функций обеспечить динамический полиморфизм, показать его особенности в программе.

Вариант № 11

Создать базовый класс «Школа». Элементы класса: поле, содержащее название школы; поле, содержащее количество обучаемых в школе (статус доступа protected); конструктор для инициализации полей; функция для печати параметров школы. Создать производный класс «Специализированная школа».

Элементы класса: дополнительно поле, содержащее название специализации школы; конструктор для инициализации полей; переопределенная функция печати параметров школы (внутри переопределенной функции должна вызываться функция из базового класса). Создать по одному объекту каждого из классов. Показать вызов созданных функций. При переопределении функций обеспечить динамический полиморфизм, показать его особенности в программе.

Вариант № 12

Создать базовый класс «Круг». Элементы класса: поле, содержащее значение радиуса круга (статус доступа protected); конструктор для инициализации поля; функция для печати радиуса круга. Создать производный класс «Эллипс». Элементы класса: дополнительно поле, содержащее значение второй полуоси эллипса (для задания первой полуоси использовать наследуемое поле радиуса круга); конструктор для инициализации полей; переопределенная функция печати параметров эллипса (внутри переопределенной функции должна вызываться функция из базового класса). Создать по одному объекту каждого из классов. Показать вызов созданных функций. При переопределении функций обеспечить динамический полиморфизм, показать его особенности в программе.

Вариант № 13

Создать базовый класс «Сотрудник предприятия». Компоненты класса — поля: Ф.И.О., оклад, надбавка за стаж (в процентах оклада за 1 год), стаж (в годах), статус доступа полей protected; конструктор для инициализации полей; функция для вычисления зарплаты; функция для печати параметров сотрудника. Создать производный класс «Начальник подразделения». Дополнительные поля: процентная надбавка к окладу за выполнение обязанностей начальника, название подразделения. Переопределить функцию для исчисления зарплаты и функцию для печати параметров начальника. Внутри переопределенных функций вызывать соответствующие функции из базового класса. Создать по одному объекту каждого из классов. Показать вызов созданных функций. При переопределении функций обеспечить динамический полиморфизм, показать его особенности в программе.

Вариант № 14

Создать базовый класс «Счет в банке». Компоненты класса — поля: Ф.И.О. владельца, начальная сумма счета, ставка вклада (проценты в год), время существования вклада (в годах), статус доступа полей protected; конструктор для инициализации полей; функция для вычисления суммы на счете с учетом начисленных процентов за время существования вклада; функция для печати параметров счета. Создать производный класс «Привилегированный счет». Дополнительное поле: процент кредита, предоставляемого по счету (проценты доступной на счете суммы с учетом времени существования вклада). Переопределенная функция для исчисления суммы на счете с учетом доступного кредита. Переопределенная функция для печати параметров счета. Внутри переопределенных функций вызывать соответствующие функции из базового

класса. Создать по одному объекту каждого из классов. Показать вызов созданных функций. При переопределении функций обеспечить динамический полиморфизм, показать его особенности в программе.

Вариант № 15

Создать базовый класс «Вектор на плоскости». Элементы класса: поля, задающие координаты точки (статус доступа protected), определяющей конец вектора (начало вектора находится в точке с координатами 0, 0); конструктор для инициализации полей; функция для вычисления длины вектора; функция для печати полей и длины вектора. Создать производный класс «Вектор в трехмерном пространстве». Элементы класса: дополнительное поле, задающее дополнительную координату; конструктор для инициализации полей; переопределенная функция для вычисления длины вектора; переопределенная функция для печати полей и длины вектора. Создать по одному объекту каждого из классов. Показать вызов созданных функций. При переопределении функций обеспечить статический полиморфизм, показать его особенности в программе.

Вариант № 16

Создать базовый класс «Точка на плоскости». Элементы класса: поля, задающие координаты точки (статус доступа protected); конструктор для инициализации полей; функция для печати значений полей. Создать производный класс «Точка в трехмерном пространстве». Элементы класса: дополнительное поле, задающее дополнительную координату; конструктор для инициализации полей; переопределенная функция для печати значений полей (внутри переопределенной функции в первую очередь должна вызываться функция из базового класса). Создать по одному объекту каждого из классов. Показать вызов созданных функций. При переопределении функций обеспечить статический полиморфизм, показать его особенности в программе. Вариант № 17

Создать базовый класс «Квадрат». Элементы класса: поле, задающее длину стороны (статус доступа protected); конструктор для инициализации поля; функция для вычисления площади квадрата; функция для печати поля и площади квадрата. Создать производный класс «Куб». Элементы класса: конструктор для инициализации поля; переопределенная функция для вычисления объема (вместо площади) куба (внутри переопределенной функции должна вызываться функция из базового класса). Создать по одному объекту каждого из классов. Показать вызов созданных функций. При переопределении функций обеспечить статический полиморфизм, показать его особенности в программе.

Вариант № 18

Создать базовый класс «Прямоугольник». Элементы класса: поля, задающие длины сторон (статус доступа protected); конструктор для инициализации полей; функция для вычисления площади прямоугольника; функция для печати полей и значения площади. Создать производный класс «Прямоугольный параллелепипед». Элементы класса: дополнительное поле,

задающее высоту; конструктор для инициализации полей; переопределенная функция для вычисления объема (вместо площади) прямоугольного параллелепипеда (внутри переопределенной функции должна вызываться функция из базового класса); переопределенная функция для печати полей и значения объема. Создать по одному объекту каждого из классов. Показать вызов созданных функций. При переопределении функций обеспечить статический полиморфизм, показать его особенности в программе.

Вариант № 19

Создать базовый класс «Круг». Элементы класса: поле, задающее радиус; конструктор для инициализации поля (статус доступа protected); функция для вычисления площади круга (площадь круга □□); функция для печати полей и площади. Создать производный класс «Шар». Элементы класса: конструктор для инициализации поля; переопределенная функция для вычисления объема (вместо площади круга) шара (площадь шара 4/ □□). Создать по одному объекту каждого из классов. Показать вызов созданных функций. При переопределении функций обеспечить статический полиморфизм, показать его особенности в программе.

Вариант № 20

Создать базовый класс «Автомобиль». Элементы класса: поле, содержащее наименование модели автомобиля; поле, содержащее значение максимальной скорости (статус доступа protected); конструктор для инициализации полей; функция для печати параметров автомобиля. Создать производный класс «Грузовой автомобиль». Элементы класса: дополнительно поле, содержащее грузоподъемность автомобиля в тоннах; конструктор для инициализации полей; переопределенная функция печати параметров автомобиля (внутри переопределенной функции должна вызываться функция из базового класса). Создать по одному объекту каждого из классов. Показать вызов созданных функций. При переопределении функций обеспечить статический полиморфизм, показать его особенности в программе.

Вариант № 21

Создать базовый класс «Вещественное число». Элементы класса: поле, задающее значение числа (статус доступа protected); конструктор для инициализации поля; функция для вычисления модуля числа; функция для печати поля и модуля числа. Создать производный класс «Комплексное число». Элементы класса: дополнительно поле, задающее значение мнимой части числа; конструктор для инициализации полей; переопределенная функция для вычисления модуля числа (модуль числа — корень квадратный из суммы квадратов вещественной и мнимой частей числа); переопределенная функция для печати полей и модуля числа. Создать по одному объекту каждого из классов. Показать вызов созданных функций. При переопределении функций обеспечить статический полиморфизм, показать его особенности в программе.

Вариант № 22

Создать базовый класс «Вектор на плоскости». Элементы класса: поля, задающие координаты точки (статус доступа protected), определяющей конец вектора (начало вектора находится в точке с координатами 0, 0); конструктор для инициализации полей; функция для печати координат вектора. Создать производный класс «Вектор в трехмерном пространстве». Элементы класса: дополнительное поле, задающее дополнительную координату; конструктор для инициализации полей; переопределенная функция для печати координат вектора (внутри переопределенной функции должна вызываться функция из базовою класса). Создать по одному объекту каждого из классов. Показать вызов созданных функций. При переопределении функций обеспечить статический полиморфизм, показать его особенности в программе.

Вариант № 23

Создать базовый класс «Квадрат». Элементы класса: поле, задающее длину стороны (статус доступа protected); конструктор для инициализации поля; функция для вычисления периметра квадрата; функция для печати длины стороны и периметра. Создать производный класс «Прямоугольник». Элементы класса: дополнительное поле, задающее другую сторону; конструктор для инициализации полей; переопределенная функция для вычисления периметра прямоугольника; переопределенная функция для печати длин сторон и периметра. Создать по одному объекту каждого из классов. Показать вызов созданных функций. При переопределении функций обеспечить статический полиморфизм, показать его особенности в программе.

Вариант № 24

Создать базовый класс «Автомобиль». Элементы класса: поле, содержащее наименование модели автомобиля; поле, содержащее значение максимальной скорости (статус доступа protected); конструктор для инициализации полей; функция для печати параметров автомобиля. Создать производный класс «Автобус». Элементы класса: дополнительно поле, содержащее максимальное количество перевозимых пассажиров; конструктор для инициализации полей; переопределенная функция печати параметров автобуса (внутри переопределенной функции должна вызываться функция из базового класса). Создать по одному объекту каждого из классов. Показать вызов созданных функций. При переопределении функций обеспечить статический полиморфизм, показать его особенности в программе.

Вариант № 25

Создать базовый класс «Школа». Элементы класса: поле, содержащее название школы; поле, содержащее количество обучаемых в школе (статус доступа protected); конструктор для инициализации полей; функция для печати параметров школы. Создать производный класс «Специализированная школа». Элементы класса: дополнительно поле, содержащее название специализации школы; конструктор для инициализации полей; переопределенная функция печати параметров школы (внутри переопределенной функции должна

вызываться функция из базового класса). Создать по одному объекту каждого из классов. Показать вызов созданных функций. При переопределении функций обеспечить статический полиморфизм, показать его особенности в программе.

Вариант № 26

Создать базовый класс «Круг». Элементы класса: поле, содержащее значение радиуса круга (статус доступа protected); конструктор для инициализации поля; функция для печати радиуса круга. Создать производный класс «Эллипс». Элементы класса: дополнительно поле, содержащее значение второй полуоси эллипса (для задания первой полуоси использовать наследуемое поле радиуса круга); конструктор для инициализации полей; переопределенная функция печати параметров эллипса (внутри переопределенной функции должна вызываться функция из базового класса). Создать по одному объекту каждого из классов. Показать вызов созданных функций. При переопределении функций обеспечить статический полиморфизм, показать его особенности в программе.

Вариант № 27

Создать базовый класс «Сотрудник предприятия». Компоненты класса — поля: Ф.И.О., оклад, надбавка за стаж (в процентах оклада за 1 год), стаж (в годах), статус доступа полей protected; конструктор для инициализации полей; функция для исчисления зарплаты; функция для печати параметров работника. Создать производный класс «Начальник подразделения». Дополнительные поля: процентная надбавка к окладу за выполнение обязанностей начальника и название подразделения. Переопределить функцию для исчисления зарплаты и функцию для печати параметров начальника. Внутри переопределенных функций вызывать соответствующие функции из базового класса. Создать по одному объекту каждого из классов. Показать вызов созданных функций. При переопределении функций обеспечить статический полиморфизм, показать его особенности в программе.

Вариант № 28

Создать базовый класс «Счет в банке». Компоненты класса — поля: Ф.И.О, владельца, начальная сумма счета, ставка вклада (проценты в год), время существования вклада (в годах), статус доступа полей protected; конструктор для инициализации полей; функция для вычисления суммы на счете с учетом начисленных процентов за время существования вклада; функция для печати параметров счета. Создать производный класс «Привилегированный счет». Дополнительное поле: процент кредита, предоставляемого по счету (проценты доступной на счете суммы с учетом времени существования вклада). Переопределенная функция для исчисления суммы на счете с учетом доступного кредита. Переопределенная функция для печати параметров счета. Внутри переопределенных функций вызывать соответствующие функции из базового класса. Создать по одному объекту каждого из классов. Показать вызов созданных функций. При переопределении функций обеспечить статический полиморфизм, показать его особенности в программе.

Лабораторная работа 5
Создание иерархии классов на основе абстрактного базового класса «геометрический объект»

Описать абстрактный базовый класс «Геометрический объект»*(Body)*, моделирующий общие свойства и поведение геометрических тел различной формы. Базовый класс содержит закрытые переменные, представляющие:

- название, отражающее форму геометрического тела (шар, пирамида, куб и т.п.) – поле перечисляемого типа *shape*, описание которого имеет вид: *enum shape { shar, prisma, parallelepiped, cub, piramida, conus, cylinder };*
- исходные параметры объектов: *r* – радиус шара, основания цилиндра или конуса; *h (или H)* – высота конуса, призмы, прямоугольного параллелепипеда и цилиндра; *a* – ребро куба; *a, b* – ребра основания прямоугольного параллелепипеда;
- площадь основания объекта *s_osn,* площадь боковой поверхности *s_bok,* площадь полной поверхности объекта *s_full*, объем объекта *v,* вес тела *p;*
- материал, из которого сделан объект – переменная перечисляемого типа *material*: *enum material { metal, wood, plastic, cardboard };*

В описания классов иерархии включить следующие виртуальные методы:

- для вычисления площади основания, боковой и полной поверхностей объекта класса;
- для моделирования сжатия или растяжения тела (увеличение радиуса шара, уменьшение стороны куба).
- для отображения фигуры на экране. В соответствующей функции-члене класса достаточно вывести текст вида «Рисую . . . », где многоточием указано значение поля с названием формы фигуры.
- для моделирования перемещения фигуры на плоскости (сдвиг центра шара или вершины куба);

Алгоритм решения задачи реализуется в глобальной функции, одним из формальных параметров которой является указатель на базовый класс. Все виртуальные методы вызываются через этот соответствующим образом проинициализированный указатель. Выводу результатов должно предшествовать название моделируемого геометрического тела и название параметра, значение которого выводится.

Описания классов иерархии разместить в заголовочном файле, определения методов базового и производных классов – каждого в отдельном файле. Глобальную функцию, реализующую алгоритм решения задачи, и главную функцию программы разместить также в отдельных файлах.

При программировании используйте формулы:

объем тела $v = k * S_{осн} * h$, где $S_{осн}$ – площадь основания объекта, h – его высота, k – коэффициент, зависящий от формы объекта: 1/3 для конусов и пирамид, 1 – для призм, кубов, параллелепипедов и цилиндров;

вес тела $p = \rho * v$. Удельный вес вещества ρ определяется по виду материала.

Контрольные вопросы

1. Как связаны виртуальные функции и принцип подстановки?

2. Приведите классификацию целей наследования.

3. Объясните разницу между наследованием интерфейса и наследованием реализации.

4. Объясните, зачем нужны виртуальные функции.

5. Что такое связывание?

6. Чем раннее связывание отличается от позднего?

7. Какие два вида полиморфизма реализованы в C++?

8. Влияет ли наличие виртуальных функций на размер класса?

9. Дайте определение полиморфного класса.

10. Может ли виртуальная функция быть дружественной функцией класса?

11. Наследуются ли виртуальные функции?

12. Каковы особенности вызова виртуальных функций в конструкторах й деструкторах?

13. Можно ли сделать виртуальной перегруженную операцию, например сложение?

14. Можно ли виртуальную функцию вызвать невиртуально?

15. Может ли конструктор быть виртуальным? А деструктор?

16. В каких случаях вызов виртуальной функции класса-наследника через указатель базового класса требует явного преобразования типа?

17. Как виртуальные функции влияют на размер класса?

18. Как объявляется чистая виртуальная функция?

19. Дайте определение абстрактного класса.

20. Наследуются ли чистые виртуальные функции?

21. Можно ли объявить деструктор чисто виртуальным?

22. Чем отличается чистый виртуальный деструктор от чистой виртуальной функции?

23. Зачем требуется определение чистого виртуального деструктора?

24. Можно ли сделать операцию присваивания виртуальной? А операцию индексирования? А чистой виртуальной?

25. Объясните, как можно реализовать «виртуальность» независимой функции.

26. Приведите классификацию целей наследования.

Варианты заданий

Вариант 1.

Класс «Куб» (Cub*).* Создать объект класса и вычислить площадь грани, полную поверхность объекта, объем и вес железного куба, используя формулы: $S_{осн} = a^2$, $S_{бок} = 4a^2$, $S_{полн} = 2S_{осн} + S_{бок}$.

Вариант 2.

Класс «Прямоугольный параллелепипед» (Rectange_Parallel). Создать объект класса, моделирующий параллелепипед, сделанный из дерева. Основание объекта расположено на плоскости *XY*. Вычислить площадь основания, боковую и полную поверхности объекта, полученного разрезанием исходного объекта пополам плоскостью, параллельной плоскости *XY*.

$S_{осн} = a * b$, $S_{бок} = 2a * H + 2b * H$, $S_{полн} = 2S_{осн} + S_{бок}$

Вариант 3.

Класс «Прямая призма» (Prisma_Inscribed_Rectilineal_Triangular – прямая призма, основание которой – правильный треугольник, вписанный в окружность радиуса *R*). Создать объект класса, моделирующий металлическую призму с высотой *H*. Вычислить площадь основания, боковую и полную поверхности, объем и вес объекта-призмы. $S_{осн} = a^2\sqrt{3/4}$, $S_{бок} = 3a * H$, $S_{полн} = 2S_{осн} + S_{бок}$,

Вариант 4.

Класс «Прямая призма» (Triangle_Prisma – прямая призма, основание которой –треугольник): *a, b* – стороны треугольника-основания, α - угол между этими сторонами, *H* – высота призмы. Вычислить площадь основания, боковую и полную поверхности, объем и вес объекта-призмы, сделанной из пластмассы:

$$S_{осн} = a * b * \sin(\alpha / 2), \quad S_{бок} = a + b + \sqrt{(a^2 + b^2 - 2ab * \cos\alpha)} * H, \quad S_{полн} = 2S_{осн} + S_{бок}$$

Вариант 5.

Класс «Прямая призма» (Trapezium_Prisma – прямая призма, основание которой – равнобедренная трапеция): *a, b* – параллельные стороны, *h* – высота трапеции-основания, *H* – высота призмы. Создать объект класса и вычислить площадь основания, боковую и полную поверхности, а также объем и вес объекта, моделирующего металлическую призму.

$$S_{осн} = (a + b) * h / 2, \quad S_{бок} = (a + b + 2\sqrt{(1/4(a-b)^2 + h^2)}) * H, \quad S_{полн} = 2S_{осн} + S_{бок}$$

Вариант 6.

Класс «Правильный тетраэдр» (Tetrahedron) моделирует треугольную пирамиду, грани которой – равносторонние треугольники. Создать два объекта класса с параметром *a* – ребро правильного тетраэдра. Вычислить площадь основания, боковую и полную поверхности, а также объем и вес объекта – правильного железного тетраэдра и объекта – правильного медного тетраэдра. Вычислить отношение весов, если их объемы окажутся равными. $S_{осн} = (a^2\sqrt{3})/4, \quad S_{бок} = 3(a * 2\sqrt{3}).4, \quad S_{полн} = 2S_{осн} + S_{бок}$

Вариант 7.

Класс «Конус» (Conus – прямой конус, основание которого – круг). Создать объект класса с параметрами: *r* – радиус основания, *H* – высота конуса, причем основание конуса лежит на плоскости *XY*, а его центр – в начале координат. Новый объект класса моделирует конус, полученный как результат рассечения исходного объекта плоскостью, параллельной плоскости *XY*. Высота нового объекта-конуса равна половине высоты исходного объекта. Вычислить следующие отношения параметров двух объектов: площадь основания, боковая и полная поверхности, объем и вес.

$$S_{осн} = \pi r^2, \quad S_{бок} = \pi r\sqrt{r^2 + H^2}, \quad S_{полн} = 2S_{осн} + S_{бок}$$

Вариант 8.

Класс «Цилиндр» (Cylinder – прямой цилиндр). Создать массив объектов класса с параметрами: r – радиус основания, H – высота цилиндра. Все объекты сделаны из одного материала. Упорядочить массив по возрастанию веса элементов-объектов. Вывести полную поверхность, объем и вес каждого объекта-цилиндра.

$$S_{осн} = \pi r^2, \quad S_{бок} = 2\pi rH, \quad S_{полн} = 2S_{осн} + S_{бок}$$

Вариант 9.

Класс «Усеченный конус» (Truncated_Conus) – наследник класса «Конус». Создать массив объектов класса с одинаковыми значениями параметров: r, R – радиусы кругов-оснований, L – образующая усеченного конуса, H – высота усеченного конуса. Все объекты «изготовлены» из различных металлов. Вычислить площади оснований, боковую и полную поверхности объектов, а также объем и вес каждого объекта – элемента массива с указанием названия металла. Найти «самый легкий» усеченный конус.

$$S_{осн1} = \pi r^2, \quad S_{осн2} = \pi \mathrm{R}^2, \quad S_{бок} = \pi \mathrm{L}(\mathrm{R} + \mathrm{r}), \quad S_{полн} = S_{осн1} + S_{осн2} + S_{бок}$$

Вариант 10.

Класс «Прямая призма» (Hexagonal_Prisma – прямая призма, основание которой – правильный шестиугольник). Создать объект класса с параметрами: R – радиус описанной окружности, H – высота призмы. Вычислить площадь основания, боковую и полную поверхности нового объекта-призмы, полученного умножением радиуса основания на некоторую константу.

$$S_{осн} = 3\sqrt{3/2} * \mathrm{R}^2, \quad S_{бок} = 6R * H, \quad S_{полн} = 2S_{осн} + S_{бок}$$

Вариант 11.

Класс «Прямая призма» (Prisma_Inscribed_Triangular – прямая призма, основание которой –треугольник, вписанный в окружность). Создать массив объектов класса с параметрами: a, b, c – стороны треугольника, R – радиус описанной окружности, H – высота призмы. Вычислить площадь основания, боковую и полную поверхности объектов-призм, упорядочив массив по возрастанию площади основания.

$$S_{осн} = abc/4R, \quad S_{бок} = a * H + bH + cH, \quad S_{полн} = 2S_{осн} + S_{бок}$$

Вариант 12.

Класс «Прямая призма» (Prisma_Attached_Triangular – прямая призма, в основание которой – правильный треугольник – вписана окружность). Создать массив объектов класса с параметрами: a – сторона правильного треугольника, r – радиус вписанной окружности и r = a / 2tg 60°, H – высота призмы. Вычислить площадь основания, боковую и полную поверхности, а также объем объектов-призм, упорядочив массив по убыванию объема.

$$S_{осн} = a^2\sqrt{3/4}, \quad S_{бок} = 3a * H, \quad S_{полн} = 2S_{осн} + S_{бок}$$

Вариант 13.

Класс «Прямой параллелепипед» (Rhombus_Parallel – прямой параллелепипед, в основание которого – ромб – вписана окружность). Создать массив объектов класса с параметрами: a – сторона ромба-основания, α - угол между сторонами ромба, H – высота параллелепипеда, материал – металл. Вычислить площадь основания, боковую и полную поверхности, а также объем и вес каждого объекта-элемента массива. Определить параметры самого «легкого» параллелепипеда.

$$S_{осн} = a^2 \sin\alpha, \quad S_{бок} = 3a * H, \quad S_{полн} = 2S_{осн} + S_{бок}$$

Вариант 14.

Класс «Куб» (Cub). Создать массив объектов класса, моделирующий металлические

кубики с ребром *a*. Вычислить объем и вес каждого кубика с указанием материала и длины ребра. Определить материал, из которого сделан самый «тяжелый» кубик.

$$S_{осн} = a^2, \quad S_{бок} = 4a^2, \quad S_{полн} = 2S_{осн} + S_{бок}$$

Вариант 15.

Класс «Прямоугольный параллелепипед» (Rectange_Parallel). Создать объект класса, моделирующий параллелепипед, сделанный из пластмассы, с параметрами: *a, b* – стороны прямоугольника-основания, *H* – высота параллелепипеда. Вычислить полную поверхность и объем исходного объекта и нового объекта, полученного удвоением параметров исходного параллелепипеда.

$$S_{осн} = a * b, \quad S_{бок} = 2a * H + 2b * H, \quad S_{полн} = 2S_{осн} + S_{бок}$$

Вариант 16.

Класс «Прямая призма» (Prisma_Inscribed_Rectilineal_Triangular – прямая призма, в основание которой – правильный треугольник, вписанный в окружность). Создать массив объектов класса с параметрами: *a* – сторона правильного треугольника, *R* – радиус описанной окружности (R = a/2*sin 60°), *H* – высота призмы. Все объекты сделаны из одного и того же материала. Вычислить полную поверхность, объем и вес каждого объекта-элемента массива. Результаты обработки вывести по возрастанию значений веса объектов.

$$S_{осн} = a^2\sqrt{3/4}, \quad S_{бок} = 3a * H, \quad S_{полн} = 2S_{осн} + S_{бок}$$

Вариант 17.

Класс «Прямая призма» (Triangle_Prisma – прямая призма, основание которой – треугольник). Создать массив объектов класса с параметрами: *a, b* – стороны треугольника, α - угол между этими сторонами, *H* – высота призмы, материал – металл. Вычислить площадь основания, боковую и полную поверхности, объем и вес каждого объекта. Упорядочить объекты массива, изготовленные из одного и того же металла, по возрастанию значения их объема.

$$S_{осн} = a * b * \sin(\alpha / 2), \quad S_{бок} = a + b + \sqrt{(a^2 + b^2 - 2ab * \cos\alpha)) * H}, \quad S_{полн} = 2S_{осн} + S_{бок}$$

Вариант 18.

Класс «Прямая призма» (Trapezium_Prisma – прямая призма, основание которой – равнобедренная трапеция*).* Создать массив объектов класса с параметрами: *a, b* – параллельные стороны, *h* – высота трапеции-основания, *H* – высота призмы. Вычислить площадь основания, боковую и полную поверхности каждого объекта. Упорядочить элементы массива по убыванию значений полной поверхности объектов класса.

$$S_{осн} = (a + b) * h / 2, \quad S_{бок} = (a + b + 2\sqrt{(1/4(a-b)^2 + h^2)}) * H, \quad S_{полн} = 2S_{осн} + S_{бок}$$

Вариант 19.

Класс «Правильный тетраэдр» (Tetrahedron) моделирует треугольную пирамиду, грани которой – равносторонние треугольники со стороной *a*. Создать массив объектов класса с параметром *a* – ребро правильного тетраэдра. Вычислить площадь основания, боковую и полную поверхности, а также объем и вес объекта – правильного медного тетраэдра. Вывести параметры самого «легкого» и самого «тяжелого» объекта – правильного медного тетраэдра.

$$S_{осн} = (a^2\sqrt{3}) / 4, \quad S_{бок} = 3(a * 2\sqrt{3}).4, \quad S_{полн} = 2S_{осн} + S_{бок}$$

Вариант 20.

Класс «Конус» (Conus – прямой конус, основание которого – круг). Создать массив объектов класса с параметрами: r – радиус круга-основания, H – высота конуса, причем основание каждого объекта-конуса лежит на плоскости XY. Вычислить площадь основания, боковую и полную поверхности, объем каждого объекта – элемента массива. Упорядочить элементы массива объектов по возрастанию значения объема.

$$S_{осн} = \pi r^2, \quad S_{бок} = \pi r\sqrt{r^2 + H^2}, \quad S_{полн} = 2S_{осн} + S_{бок}$$

Лабораторная работа 6
Изучение абстрактных классов

Цель работы состоит в овладении навыками разработки программ на языке Си++ с использованием абстрактных классов при наследовании. Для достижения цели необходимо выполнить следующие задачи:

- изучить учебные материалы, посвященные абстрактным классам в языке Си++;
- разработать программу на языке Си++ для решения заданного варианта;
- отладить программу;
- запустить программу, показать результаты выполнения программы, объяснить алгоритм работы программы.
- ответить на контрольные вопросы.

Краткая характеристика объекта изучения

Абстрактный класс – это класс, который имеет в своем составе хотя бы одну чистую виртуальную функцию. Такая функция не имеет тела и ничего не делает.

Формат объявления чистой виртуальной функции внутри класса:

```
virtual <тип_возвр_значения>
<имя_функщси:> (<список_форм_парам>) =0;
```

Пример объявления такой функции с именем **MyFun** и двумя параметрами типа **double** (имена параметров в заголовке необязательны):

```
virtual void MyFun(double, double)=0;
```

Нельзя создать объект абстрактного класса (тип указателя на абстрактный класс может быть). Абстрактный класс нужен, чтобы на его основе создавать обычные классы, являющиеся его «наследниками», в которых чистые виртуальные функции переопределяются или заменяются на обычные функции.

Задачи и порядок выполнения работы

В работе требуется создать абстрактный класс «геометрическая фигура на экране», который содержит чистую виртуальную функцию для рисования фигуры (неизвестно заранее, какая именно фигура). На основе этого абстрактного класса нужно создать производные классы, определяющие конкретные геометрические фигуры. При работе с объектами этих классов важно использовать массив указателей, имеющих тип абстрактного класса, что позволит для работы с объектами разных классов использовать один цикл.

Условие задачи

Создать абстрактный класс – «Геометрическая фигура» (на экране). Класс содержит координаты геометрического центра фигуры на экране и следующие поля: поле, задающее размер фигуры (например, расстояние от центра до вершины или радиус окружности, в пикселях); поле, задающее угловое положение фигуры; поле, задающее угловую скорость вращения фигуры; поле, определяющее направление движения (возможны два варианта: движение по вертикали и движение по горизонтали); поле, определяющее скорость движения; поле, определяющее цвет фигуры; поле, содержащее хэндл окна для рисования. При необходимости можно включить другие поля. Класс включает в себя: конструктор для инициализации полей, функцию, изменяющую угловое положение фигуры и положение на экране во время движения за один такт времени, и чистую виртуальную функцию (или функции) для рисования и стирания фигуры на экране.

На основе абстрактного класса «Фигура» следует разработать программу, содержащую описание трех графических классов: «Правильный многоугольник»; «Отрезок» («Линия»); «Половина окружности»; создать несколько объектов каждого из трех классов (не менее трех), для чего использовать один массив указателей типа базового класса «Фигура». Реализуя механизм полиморфизма, привести объекты классов в одновременное вращение вокруг их центров с различивши угловыми скоростями и в движение с отскоком от краев окна в заданном режиме (по горизонтали или по вертикали). При этом использовать обработку сообщений от таймера. Таймер периодически с интервалом несколько миллисекунд генерирует сообщение, а при его обработке стирается старая фигура и рисуется новая на новом месте.

Возможный внешний вид окна приложения с движущимися фигурами представлен на рисунке.

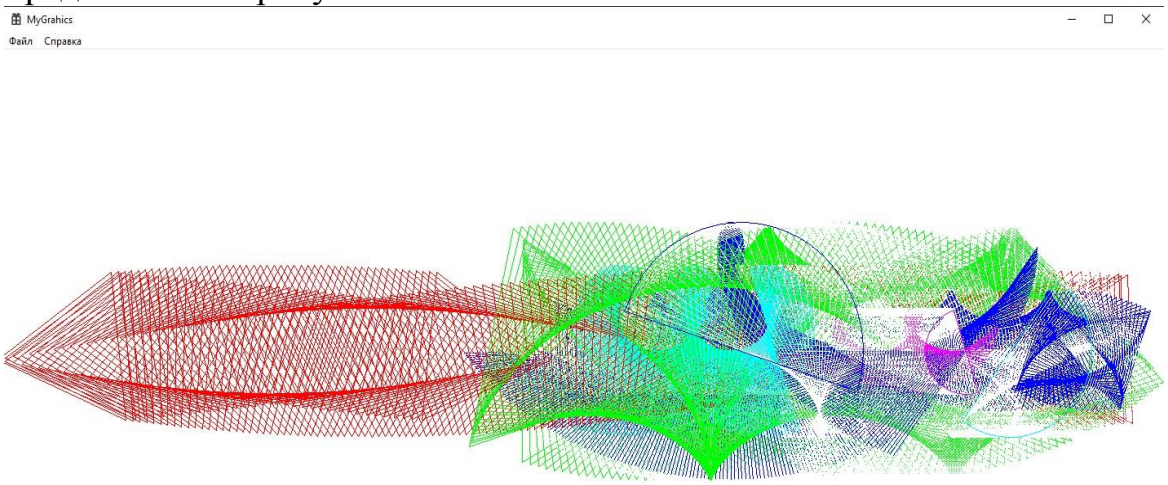
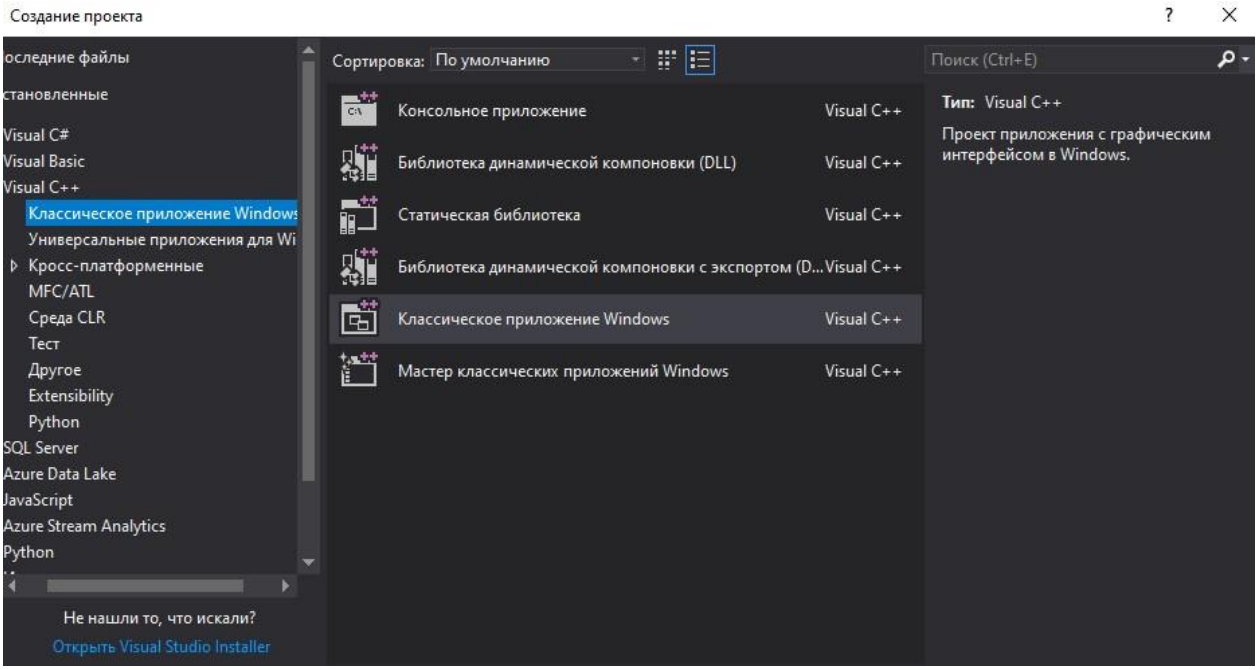


Рисунок 6.1 – Итог программы

Пример выполнения работы

В работе создается приложение Windows с графическим интерфейсом пользователя.



В листинге, представленном ниже, приведен только код файла `MyGrahics.cpp`. В среде Microsoft Visual Studio было создано стандартное оконное приложение с именем `MyGrahics` (имя может быть другим), все остальные файлы – `MyGrahics. h` и др. – стандартные, сгенерированы автоматически при создании проекта. В файл `MyGrahics.cpp` необходимо в конце добавить две строчки:

```
#define _USE_MATH_DEFINES
#include <cmath>
```

Листинг файла MyGraphics.cpp

```
#include "framework.h"
#include "MyGrahics.h"

#define _USE_MATH_DEFINES
#include <cmath>



#define MAX_LOADSTRING 100

// Глобальные переменные:
HINSTANCE hInst;                                // текущий экземпляр
WCHAR szTitle[MAX_LOADSTRING];                  // Текст строки заголовка
WCHAR szWindowClass[MAX_LOADSTRING];            // имя класса главного окна

// Отправить объявления функций, включенных в этот модуль кода:
ATOM                MyRegisterClass(HINSTANCE hInstance);
BOOL                InitInstance(HINSTANCE, int);
LRESULT CALLBACK    WndProc(HWND, UINT, WPARAM, LPARAM);
INT_PTR CALLBACK    About(HWND, UINT, WPARAM, LPARAM);

int APIENTRY wWinMain(_In_ HINSTANCE hInstance,
                      _In_opt_ HINSTANCE hPrevInstance,
                      _In_ LPWSTR    lpCmdLine,
                      _In_ int       nCmdShow)
{
    UNREFERENCED_PARAMETER(hPrevInstance);
    UNREFERENCED_PARAMETER(lpCmdLine);


    // TODO: Разместите код здесь.

    // Инициализация глобальных строк
    LoadStringW(hInstance, IDS_APP_TITLE, szTitle, MAX_LOADSTRING);
    LoadStringW(hInstance, IDC_MYGRAHICS, szWindowClass, MAX_LOADSTRING);
    MyRegisterClass(hInstance);

    // Выполнить инициализацию приложения:
    if (!InitInstance (hInstance, nCmdShow))
    {
        return FALSE;
    }

    HACCEL hAccelTable = LoadAccelerators(hInstance, MAKEINTRESOURCE(IDC_MYGRAHICS));

    MSG msg;

    // Цикл основного сообщения:
    while (GetMessage(&msg, nullptr, 0, 0))
    {
        if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
        {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
    }

    return (int) msg.wParam;
}
```

```
65  //
66  //  ФУНКЦИЯ: MyRegisterClass()
67  //
68  //  ЦЕЛЬ: Регистрирует класс окна.
69  //
70  ATOM MyRegisterClass(HINSTANCE hInstance)
71  {
72      WNDCLASSEXW wcex;
73
74      wcex.cbSize = sizeof(WNDCLASSEX);
75
76      wcex.style          = CS_HREDRAW | CS_VREDRAW;
77      wcex.lpfnWndProc    = WndProc;
78      wcex.cbClsExtra     = 0;
79      wcex.cbWndExtra     = 0;
80      wcex.hInstance      = hInstance;
81      wcex.hIcon          = LoadIcon(hInstance, MAKEINTRESOURCE(IDI_MYGRAHICS));
82      wcex.hCursor        = LoadCursor(nullptr, IDC_ARROW);
83      wcex.hbrBackground  = (HBRUSH)(COLOR_WINDOW+1);
84      wcex.lpszMenuName   = MAKEINTRESOURCEW(IDC_MYGRAHICS);
85      wcex.lpszClassName  = szWindowClass;
86      wcex.hIconSm        = LoadIcon(wcex.hInstance, MAKEINTRESOURCE(IDI_SMALL));
87
88      return RegisterClassExW(&wcex);
89  }
90
91  //
92  //   ФУНКЦИЯ: InitInstance(HINSTANCE, int)
93  //
94  //   ЦЕЛЬ: Сохраняет маркер экземпляра и создает главное окно
95  //
96  //   КОММЕНТАРИИ:
97  //
98  //        В этой функции маркер экземпляра сохраняется в глобальной переменной, а также
99  //        создается и выводится главное окно программы.
100 //
101 BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
102 {
103    hInst = hInstance; // Сохранить маркер экземпляра в глобальной переменной
104
105    HWND hWnd = CreateWindowW(szWindowClass, szTitle, WS_OVERLAPPEDWINDOW,
106       CW_USEDEFAULT, 0, CW_USEDEFAULT, 0, nullptr, nullptr, hInstance, nullptr);
107
108    if (!hWnd)
109    {
110       return FALSE;
111    }
112
113    ShowWindow(hWnd, nCmdShow);
114    UpdateWindow(hWnd);
115
116    return TRUE;
117 }
118 //*****************************
119
120 // Абстрактный класс «Фигура»,
121 class Figure {
122 protected:      /* Поля наследуются; должен быть доступ в производном классе */
123     int x, y;   // Координаты геометрического центра,
124     int R;      // Расстояние от центра до вершины,
125     int Ang;    // Угловое положение в градусах,
126     int VAng;   /* Угловая скорость (в градусах за интервал срабатывания таймера) */
127     int V;      /* Скорость (в пикселях за интервал срабатывания таймера) */
128     int Napr;   /* Направление движения 0 – вертикально, 1 – горизонтально*/
```

```
    COLORREF col;   // Цвет фигуры.
    HWND hWnd;  // Хэндл окна, где нужно рисовать фигуру
    int N_Reg;  /* Направление перемещения (1 – вправо или вниз; -1 – влево или вверх) */
public:
    // Конструктор для инициализации всех параметров.
    Figure(int R, int VAng, int V, int Napr, COLORREF col, HWND hWnd);
    // Метод меняет положение фигуры за один такт таймера,
    virtual void step();
    /* Чистая виртуальная функция для рисования (стирания) фигуры */
    virtual void draw(int Reg) = 0;
};
 // Определение конструктора и функций за пределами класса,
Figure::Figure(int R, int VAng, int V, int Napr, COLORREF col, HWND hWnd)
{
    this->R = R;
    this->VAng = VAng;
    this->V = V;
    this->Napr = Napr;
    this->col = col;
    this->hWnd = hWnd;
    Ang = 0;
    N_Reg = 1;  // Координаты увеличиваются,
    RECT rect;  // Размеры окна.
    GetClientRect(hWnd, &rect);     // Получение размеров окна.
    // Начальное положение фигуры в центре окна,
    x = rect.right / 2;
    y = rect.bottom / 2;
}
 /* Определение метода, изменяющего положение фигуры за один такт таймера */
void Figure::step()
{
    // Изменение углового положения фигуры.
    Ang += VAng;
    if (Ang >= 360) Ang -= 360;
    // Получение размеров окна.
    RECT rect;
    GetClientRect(hWnd, &rect);
    // Изменение положения центра фигуры.
    if (Napr = 1) // Движение горизонтально изменяется x.
    {
        x += V * N_Reg;
        if (N_Reg == 1) // Движение вправо.
        {
            if (x + R >= rect.right) /* Достижение правой границы окна */
                N_Reg = -1; // Изменение направления движения.
        }
        else // Движение влево.
        {
            if (x - R <= 0) // Достижение левой границы,
                N_Reg = 1;      // Изменение направления движения,
        }
    }
    else // Движение вертикально изменяется y.
    {
        y += V * N_Reg;
        if (N_Reg == 1) // Движение вниз.
        {
            if (y + R >= rect.bottom) /* Достижение нижней границы окна */
                N_Reg = -1; // Изменение направления движения.
        }
        else // Движение вверх.
        {
            if (y - R <= 0) // Достижение верхней границы.
                N_Reg = 1; // Изменение направления движения.
```

```
193             }
194         }
195     }
196     // Класс многоугольник,
197     class MyPolygon : public Figure
198     {
199     protected:
200         int N;      // Количество вершин.
201         POINT *p;   // Массив координат вершин,
202     public:
203         // Заголовок конструктора,
204         MyPolygon(int R, int VAng, int V, int Napr, COLORREF col, HWND hWnd, int N);
205         void step(); /* Метод дополнительно считает новые координаты вершин */
206         void draw(int Reg); // Метод рисования фигуры
207     };
208     // Определение конструктора.
209     MyPolygon::MyPolygon(int R, int VAng, int V, int Napr, COLORREF col, HWND hWnd, int N) :
210         // Вызов конструктора базового класса.
211         Figure(R, VAng, V, Napr, col, hWnd)
212     {
213         this->N = N;
214         p = new POINT[N]; // Создание массива координат вершин. / / Расчет координат вершин.
215         double A = Ang * M_PI / 180; /* Угол в градусах следует перевести в радианы */
216         double A1 = 2 * M_PI / N; /* Угол между направлениями на соседние вершины из центра фигуры */
217         for (int i = 0; i < N; i++, A += A1)
218         {
219             p[i].x = x + R * cos(A);
220             p[i].y = y - R * sin(A);
221         }
222     }
223     void MyPolygon::step()  /* Метод дополнительно считает новые координаты вершин */
224     {
```

```
225         Figure::step(); // Вызов метода базового класса.
226         // Расчет координат вершин многоугольника.
227         double A = Ang * M_PI / 180;    /* Угол в градусах следует перевести в радианы*/
228         double A1 = 2 * M_PI / N;       /* Угол между направлениями на соседние вершины из центра фигуры */
229         for (int i = 0; i < N; i++, A += A1)
230         {
231             //A += A1;
232             p[i].x = x + R * cos(A);
233             p[i].y = y - R * sin(A);
234         }
235     }
236     void MyPolygon::draw(int Reg) // Метод рисования фигуры.
237     {
238         HPEN pen;
239         if (Reg = 1)    // Режим рисования фигуры.
240             pen = CreatePen(PS_SOLID, 1, col);
241         else // Режим стирания (белое перо).
242             pen = CreatePen( PS_SOLID, 1, RGB(255, 255, 255));
243         HDC hdc;
244         // Получение контекста устройства,
245         hdc = GetDC(hWnd);
246         SelectObject(hdc, pen); /* Загрузка пера в контекст устройства */
247         MoveToEx(hdc, p[0].x, p[0].y, 0);   /* Графический курсор в первую вершину */
248         for (int i = 1; i < N; i++)
249             LineTo(hdc, p[i].x, p[i].y);
250         LineTo(hdc, p[0].x, p[0].y); /* Соединение первой и последней вершин */
251         ReleaseDC(hWnd, hdc);
252         DeleteObject(pen);  // Удаление пера.
253     }
254     // Класс «Отрезок»,
255     class MyOtrezok : public Figure
256     {
```

```
257 protected:
258     int xl, yl, x2, y2; // Координаты концов отрезка,
259 public:
260     // Заголовок конструктора.
261     MyOtrezok(int R, int VAng, int V, int Napr, COLORREF col, HWND hWnd);
262     void step(); /* Метод дополнительно считает новые координаты концов отрезка */
263     void draw(int Reg); // Метод рисования фигуры.
264 };
265  // Определение конструктора.
266 MyOtrezok::MyOtrezok(int R, int VAng, int V, int Napr, COLORREF col, HWND hWnd) :
267     // Вызов конструктора базового класса.
268     Figure(R, VAng, V, Napr, col, hWnd)
269 {
270     // Расчет координат вершин.
271     double A = Ang * M_PI / 180; /* Угол в градусах следует перевести в радианы */
272     xl = x + R * cos(A);
273     yl = y - R * sin(A);
274     x2 = x - R * cos(A);
275     y2 = y + R * sin(A);
276 }
277 void MyOtrezok::step()  /* Метод дополнительно считает новые координаты вершин */
278 {
279     Figure::step(); // Вызов метода базового класса.
280     // Расчет координат вершин многоугольника.
281     double A = Ang * M_PI / 180; /* Угол в градусах следует перевести в радианы*/
282     xl = x + R * cos(A);
283     yl = y - R * sin(A);
284     x2 = x - R * cos(A);
285     y2 = y + R * sin(A);
286 }
287 void MyOtrezok::draw(int Reg) // Метод рисования фигуры.
288 {
289     HPEN pen;
290     if (Reg = 1)    // Режим рисования фигуры.
291         pen = CreatePen(PS_SOLID, 1, col);
292     else // Режим стирания (белое перо).
293         pen = CreatePen(PS_SOLID, 1, RGB(255, 255, 255));
294     HDC hdc;
295     // Получение контекста устройства.
296     hdc = GetDC(hWnd);
297     SelectObject(hdc, pen);     /* Загрузка пера в контекст устройства */
298     MoveToEx(hdc, xl, yl, 0);   /* Графический курсор в первую вершину */
299     LineTo(hdc, x2, y2);        /* Соединение первой вершины с последней */
300     ReleaseDC(hWnd, hdc);
301     DeleteObject(pen);          // Удаление пера.
302 }
303  // Класс «Половина круга»,
304 class MyPoluKrug : public Figure
305 {
306 public:
307     // Заголовок конструктора.
308     MyPoluKrug(int R, int VAng, int V, int Napr, COLORREF col, HWND hWnd);
309     void draw(int Reg); // Метод рисования фигуры.
310 };
311  // Определение конструктора.
312 MyPoluKrug::MyPoluKrug(int R, int VAng, int V, int Napr, COLORREF col, HWND hWnd):
313     // Вызов конструктора базового класса.
314     Figure(R, VAng, V, Napr, col, hWnd)
315 {
316
317 }
318 void MyPoluKrug::draw(int Reg) // Метод рисования фигуры.
319 {
320     HPEN pen;
```

```
    if (Reg == 1)   // Режим рисования фигуры.
        pen = CreatePen(PS_SOLID, 1, col);
    else // Режим стирания (белое перо) .
        pen = CreatePen(PS_SOLID, 1, RGB(255, 255, 255));
    HDC hdc;
    // Получаем контекст устройства.
    hdc = GetDC(hWnd);
    SelectObject(hdc, pen); /* Загрузка пера в контекст устройства */
    MoveToEx(hdc, x, y, 0); // Помещение курсора в центр круга.
    AngleArc(hdc, x, y, R, Ang, 180);   /* Рисование полуокружности */
    LineTo(hdc, x, y);  /* Соединение полуокружности с центром */
    ReleaseDC(hWnd, hdc);
    DeleteObject(pen);  // Удаление пера.
}
Figure *pF[9]; // Будет создано девять объектов.

//****************************
//
//  ФУНКЦИЯ: WndProc(HWND, UINT, WPARAM, LPARAM)
//
//  ЦЕЛЬ: Обрабатывает сообщения в главном окне.
//
//  WM_COMMAND  - обработать меню приложения
//  WM_PAINT    - Отрисовка главного окна
//  WM_DESTROY  - отправить сообщение о выходе и вернуться
//
//

LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    int wmId, wmEvent;
     PAINTSTRUCT ps;
     HDC hdc;
     switch (message)
     {
     case WM_CREATE: // Создание окна.
        SetTimer(hWnd, 1, 10, 0);   /* Таймер с номером 1 срабатывает через 10 мс */
        // Создание объектов.
        pF[0] = new MyPolygon(100, 1, 10, 0, RGB(255, 0, 0), hWnd, 3);
        pF[1] = new MyPolygon(150, 2, 5, 0, RGB(0, 255, 0), hWnd, 3);
        pF[2] = new MyPolygon(70, 3, 3, 1, RGB(0, 0, 255), hWnd, 3);
        pF[3] = new MyOtrezok(50, 4, 2, 1, RGB(255, 0, 255), hWnd);
        pF[4] = new MyOtrezok(100, 2, 1, 1, RGB(0, 255, 255), hWnd);
        pF[5] = new MyOtrezok(150, 1, 2, 0, RGB(0, 0, 255), hWnd);
        pF[6] = new MyPoluKrug(50, 2, 2, 0, RGB(255, 0, 255), hWnd);
        pF[7] = new MyPoluKrug(100, 1, 3, 0, RGB(0, 255, 255), hWnd);
        pF[8] = new MyPoluKrug(150, 3, 4, 1, RGB(0, 0, 255), hWnd);
        break;
     case WM_TIMER:  // Сообщение от таймера,
        for (int i = 0; i < 9; i++)
        {
            pF[i]->draw(0); // Стирание старой фигуры.
            pF[i]->step(); // Изменение положения фигуры.
            pF[i]->draw(1); // Рисование фигуры на новом месте.
        }
        break;
     case WM_COMMAND:
        wmId = LOWORD(wParam);
        wmEvent = HIWORD(wParam);
        // Разобрать выбор в меню:
        switch (wmId)
        {
            case IDM_ABOUT:
```

```
384                 DialogBox(hInst, MAKEINTRESOURCE(IDD_ABOUTBOX), hWnd, About);
385                 break;
386             case IDM_EXIT:
387                 KillTimer(hWnd, 1); // Удаление таймера.
388                 DestroyWindow(hWnd);
389                 break;
390             default:
391                 return DefWindowProc(hWnd, message, wParam, lParam);
392             }
393             break;
394         case WM_PAINT:
395             hdc = BeginPaint(hWnd, &ps);
396             // TODO: добавьте любой код отрисовки...
397             EndPaint(hWnd, &ps);
398             break;
399         case WM_DESTROY:
400             KillTimer(hWnd, 1); // Удаление таймера.
401             PostQuitMessage(0);
402             break;
403         default:
404             return DefWindowProc(hWnd, message, wParam, lParam);
405         }
406         return 0;
407     }
408
409     /* ... */
448     // Обработчик сообщений для окна "О программе".
449     INT_PTR CALLBACK About(HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam)
450     {
451         UNREFERENCED_PARAMETER(lParam);
452         switch (message)
453         {
```

```
454         case WM_INITDIALOG:
455             return (INT_PTR)TRUE;
456
457         case WM_COMMAND:
458             if (LOWORD(wParam) == IDOK || LOWORD(wParam) == IDCANCEL)
459             {
460                 EndDialog(hDlg, LOWORD(wParam));
461                 return (INT_PTR)TRUE;
462             }
463             break;
464         }
465         return (INT_PTR)FALSE;
466     }
467
```

Задания и вопросы для самоконтроля

1. Раскройте понятие абстрактного класса.
2. Что такое чистая виртуальная функция?
3. Раскройте особенности использования указателей, имеющих тип

«указатель на абстрактный класс».

Варианты заданий для лабораторной работы

Создать абстрактный класс «Геометрическая фигура» (на экране). Класс содержит следующие поля: координаты геометрического центра фигуры на экране; поле, задающее размер фигуры (например, расстояние от центра до вершины или радиус окружности, в пикселях); поле, задающее угловое положение фигуры; поле, задающее угловую скорость вращения фигуры; поле, определяющее направление движения (возможны два варианта: движение по вертикали и движение по горизонтали); поле, определяющее скорость движения; поле, определяющее цвет фигуры; поле, содержащее хэндл окна для рисования; при необходимости можно включить другие поля. Класс включает в себя: конструктор для инициализации полей, функцию, изменяющую угловое положение фигуры и положение на экране во время движения за один такт времени, и чистую виртуальную функцию (или функции) для рисования и стирания фигуры на экране. На основе абстрактного класса «Фигура» разработать три производных класса, задающих геометрические фигуры своего варианта, варианты представлены в табл. 5. Создать несколько объектов каждого из трех классов (не менее трех), для этого использовать один массив указателей типа базового класса «Фигура». Реализуя механизм полиморфизма, привести объекты классов в одновременное вращение вокруг их центров с различными угловыми скоростями и в движение с отскоком от краев окна в заданном режиме (по горизонтали или по вертикали). При этом использовать обработку сообщений от таймера. Таймер периодически с интервалом в несколько миллисекунд генерирует сообщение, при обработке сообщения стирается старая фигура и рисуется новая на новом месте.

Геометрические фигуры:

- отрезок (линия);
- окружность с вырезанной четвертью;
- правильный многоугольник (количество вершин — поле класса);
- ромб (не квадрат);
- прямоугольник (не квадрат);
- равнобедренный прямоугольный треугольник;
- параллелограмм, не являющийся ромбом или прямоугольником.

Варианты заданий

Номер варианта	Номер фигуры	Номер варианта	Номер фигуры
1	1,2,3	15	1,6,7
2	1,2,4	16	2, 3, 4
3	1,2,5	17	2, 3,5
4	1,2,6	18	2, 3, 6
5	1,2,7	19	2, 3, 7

6	1,3,4	20	2, 4, 5
7	1,3,5	21	2, 4, 6
8	1,3,6	22	2, 4, 7
9	1,3,7	23	3,4,5
10	1,4,5	24	3,4,6
11	1,4,6	25	3,4,7
12	1,4,7	26	3,5,6
13	1,5,6	27	3,5,7
14	1,5,7	28	3,6,7

Лабораторная работа 7
Шаблоны классов

Цель работы: изучить приемы создания и использования шаблонов классов. Краткие теоретические сведения

Механизм шаблонов C++ – это средство построения обобщенных определений функций и классов, которые не зависят от используемых типов данных. Этот механизм позволяет сократить трудоемкость создания программ, т.к. повышает лаконичность текста, т.е. использование шаблонов избавляет от необходимости дублировать коды классов и функций для разных типов данных.

Компилятор по заданному типу аргументов на основе описания шаблона автоматически создает соответствующие экземпляры классов и функций, которые называются представители конкретных классов и функций.

Шаблоны класса в отличие от шаблона функции позволяют параметризовать, т.е. использовать в качестве параметров не только типы элементов данных, но и константы разных типов.

Синтаксис определения шаблона класса следующий:

template <список параметров шаблона > обычное описание структуры класса;

При этом список параметров шаблона не может быть пустым. Элементы в списке разделяются запятыми. Внутри пространства класса параметры шаблона должны быть хотя бы один раз упомянуты.

В список параметров могут входить два вида параметров:

1) типизированные параметры, начинающиеся со слов class

Идентификатор ,

компилятор при создании экземпляра класса заменит его на конкретный тип данных;

2) нетипированные параметры шаблона:

Стандартный тип числовых данных идентификатор

Таким образом, типированные параметры – это фиктивные имена типов данных, входящих в класс. Нетипированные параметры – это поименованные

типы числовых констант. При этом им при декларировании можно присваивать умалчиваемые значения.

Каждый параметр является локальным в рамках пространства класса.

В описании класса хотя бы один раз необходимо упомянуть ID типированных и нетипированных параметров, конкретные значения для которых будут переданы в момент создания объекта этого класса через список аргументов, который указывается через запятые в треугольных скобках сразу после ID класса:

ID_ класса <список аргументов> ID_объектов;

ID_объектов – идентификаторы объектов, которые создает данный класс (записанные через запятые, например a,Ь,c). При этом в списке аргументов каждому типированному параметру шаблона должен соответствовать известный конкретный тип данных, а каждому нетипированному параметру – константное выражение соответствуючего типа. Таким образом, между списком параметров шаблона и списком аргументов должно быть абсолютное соответствие по количеству, порядку их следования и типам. Если нетипированные параметры имеют умалчиваемые значения, их располагают в списке последними. Пример с шаблоном класса, конструктор которого порождает объекты с двумя параметризованными полями:

```
template <class T1>
class X {
protected:
    T1 a, b;
public:
    X(T1 i, T1 j) {     // Конструктор
        a = i; b = j;
        cout << "\n Object created, size = " << sizeof(T1) << endl;
    }
    void Print(void) {
        cout << "a = " << a << "b = " << b << endl;
    }
};
void main(void) {
    X < int >  x1(2, 3);      // Целочисленные объекты
    x1.Print();               // На экране: Object created, size = 2
    getch();                  //    a = 2  b = 3
    X < double >  x2(0.5, 1.2);   // Вещественные объекты
    x2.Print();               // На экране: Object created, size = 4
    getch();                  //    a = 0.5  b = 1.2
}
```

Методы шаблона класса можно определять вне его пространства. В этом случае формат их определения будет следующим:

```
template <class T1>
class X {
protected:
    T1 a, b;
public:
    X(T1, T1);
    void Print(void) {
        cout << "a = " << a << "b = " << b << endl;
    }
};
// Определение конструктора вне состава класса
template <class T1> X <T1> ::X(T1 i, T1 j) {
    a = i; b = j;
    cout << "Object created, size = " << sizeof(T1) << endl;
}
void main(void) {
    // Код предыдущего примера
}
```

Задания для самостоятельного выполнения.

Выполнение заданий осуществляется без использования библиотеки STL.

Создать шаблонный класс NameClass<T> и публичную вложенную структуру Node (имя класса указано в задании). Данный класс должен быть объявлен и определен в пространстве имен containers.

Структура Node должна содержать:

- Публичное поле value типа T со значением, хранимым в данном элементе шаблонного класса.
- Приватные поля для осуществления связи элементов в шаблонном классе.
- Приватный конструктор(ы). Ничто кроме шаблонного класса NameClass не должно иметь возможности создания элементов типа Node.

Класс должен содержать:

- конструктор по умолчанию, создающий экземпляр класса нулевого размера;
- Конструктор от std::initializer_list<T>.
- Конструктор копирования и копирующий оператор присваивания.
- Конструктор перемещения и перемещающий оператор присваивания.
- операцию индексирования, возвращающую ссылку на
- соответствующий элемент экземпляра класса;
- метод, добавляющий элемент в произвольную позицию экземпляра
- класса;

- метод, удаляющий элемент из конца экземпляра класса (начала, произвольный элемент (в зависимости от задания)).

Добавить в класс другие методы, необходимые для реализации задания. При выборе методов ориентироваться на методы, которые есть у шаблонных классов библиотеки STL.

В клиенте main() продемонстрировать использование этого класса для векторов, содержащих элементы типов int, double, string.

Для этого написать программу, демонстрирующую работу с этим шаблоном для различных типов параметров шаблона. Программа должна содержать меню, позволяющее осуществить проверку всех методов шаблона.

Вариант 1

Создать шаблон класса «стек» (Stack).

Вариант 2

Создать шаблон класса «однонаправленный линейный список» (List).

Вариант 3

Создать шаблон класса «двунаправленный линейный список» (DList)

Варианте 4

Создать шаблон класса «бинарное дерево» (BinaryTree).

Вариант 5

Создать шаблон класса «односторонняя очередь» (Queue)

Вариант 6

Создать шаблон класса «двусторонняя очередь (допускает вставку и удаление из обоих концов)» (Deque)

Вариант 7

Создать шаблон класса «очередь с приоритетами» (PQueue). При добавлении элемента в такую очередь его номер определяется его приоритетом.

- Каждому элементу очереди с приоритетами сопоставлено некоторое значение, именуемое приоритетом этого элемента. Приоритеты допускают сравнение друг с другом;
- Функция извлечения из очереди с приоритетами возвращает тот элемент, приоритет которого является максимальным.

Вариант 8

Создать шаблон класса «однонаправленный кольцевой список» (RList).

Вариант 9

Создать шаблон класса «двунаправленный кольцевой список» (DRList).

Вариант 10

Разработать шаблонный класс Vect, для представления векторов. Каждый вектор включает в себя собственно имя вектора и динамический одномерный массив для хранения размерности вектора.

Вариант 11

Разработать шаблонный класс Set, для представления множества. Повторяющиеся элементы в множество не заносятся. Элементы в множестве хранятся отсортированными. Необходимо

Вариант 12

Разработать шаблонный класс MSet, для представления множества. Множество может хранить повторяющиеся элементы. Элементы в множестве хранятся отсортированными.

Вариант 13

Разработать шаблонный класс Map, для представления ключей и значений (словаря). Словарь не может хранить повторяющиеся ключи. Элементы в словаре хранятся отсортированными.

Вариант 14

Разработать шаблонный класс MMap, для представления ключей и значений (словаря). Словарь может хранить повторяющиеся ключи. Элементы в словаре хранятся отсортированными.

Вариант 15

Разработать шаблонный класс Tuple, для наборов данных фиксированной длины. Кортеж может содержать элементы разных типов.

Вариант 16

Разработать шаблонный класс для представления разреженных одномерных массивов SparseArr. Размер логического массива передавать через аргумент конструктора.

Класс должен обеспечивать хранение данных любого типа T, для которого предусмотрены конструктор по умолчанию, конструктор копирования и операция присваивания. Класс должен содержать операцию индексирования, возвращающую ссылку на найденный элемент. Если элемент с заданным индексом не найден, операция должна создать новый элемент с этим индексом и поместить его в массив. При необходимости добавить в класс другие методы.

Контрольные вопросы

1. В чем в C++ заключается механизм шаблонов классов и функций?
2. Каков общий формат шаблона класса?
3. Перечислите виды параметров шаблона класса.
4. Как создать конкретный экземпляр класса, используя шаблон класса?
5. Как определить метод вне пространства шаблона класса?

Лабораторная работа 8
Обработка исключительных ситуаций

Цель работы: приобрести навыки по организации контроля за возникновением непредвиденных или аварийных ситуаций во время работы программы.

Краткие теоретические сведения

Исключения возникновение непредвиденных ошибочных условий в момент работы программы. В то же время исключение - это более общее чем ошибка понятие, так как может возникать и тогда, когда в программе нет ошибок (например, сбой выделения памяти при создании объекта класса). В общем, исключение обозначает особую ситуацию, когда требуется безвозвратно переключить выполнение программы на блок обработки такой ситуации. Выявление особой ситуации производится только программным путём при помощи проверки нормального хода выполнения программы.

Средства обработки ошибочных ситуаций позволяют передать обработку исключений из кода, в котором возникло исключение, некоторому другому программному блоку, который выполнит в данном случае некоторые определенные действия. Таким образом, основная идея данного механизма состоит в том, что функция проекта, которая обнаружила непредвиденную ошибочную ситуацию, которую она не знает, как решить, генерирует сообщение об этом (бросок исключения). А система вызывает по этому сообщению программный модуль, который перехватит исключение и отреагирует на возникшее нештатное событие. Такой программный модуль называют «обработчик» или перехватчик исключительных ситуаций. И в случае возникновения исключения в его обработчик передаётся произвольное количество информации с контролем ее типа. Эта информация и является характеристикой возникшей нештатной ситуации.

Обработка исключений в C++ — это обработка с завершением. Это означает, что исключается невозможность возобновления выполнения программы в точке возникновения исключения.

Для обеспечения работы такого механизма были введены следующие ключевые слова:

try — проба испытания;

catch — перехватить (обработать);

throw — бросать.

Кратко рассмотрим их назначение.

Ключевое слово try открывает блок кода, в котором может произойти ошибка; это обычный составной оператор:

```
try {
        //код
}
```

Код содержит набор операций и операторов, который и будет контролироваться на возникновение ошибки. В него могут входить вызовы функции пользователя, которые компилятор также возьмет на контроль.

Среди данного набора операторов и операций обязательно указывают операцию броска исключения throw, которая имеет следующий формат: throw выражение; где выражение опреДеляет тип информации, которая и описывает исключение (например конкретные типы Данных).

Обработчик исключения catch перехватывает информацию:

```
catch ( тип и параметр) {
        //код
}
```

Через параметр обработчику передаются данные определенного типа, описывающие обрабатываемое исключение. Код определяет те действия, которые надо выполнить при возникновении данной конкретной ситуации. В C++ используют несколько форм обработчиков. Рассмотренный обработчик получил название параметризованный специализированный перехватчик.

Перехватчик должен следовать сразу же после блока контроля, т.е. между обработчиком и блоком контроля не должно быть ни одного оператора. При этом в одном блоке контроля можно вызывать исключения разных типов для различных ситуаций, поэтому обработчиков может быть несколько. В этом случае их необходимо расположить сразу же за контролирующим блоком последовательно друг за другом.

Кроме того, запрещены переходы как извне в обработчик, так и между обработчиками.

Можно воспользоваться универсальным или абсолютным обработчиком:

```
catch (. . . ) {
        / / код
}
```

где означают способность данного перехватчика обрабатывать информацию любого типа. Такой обработчик располагают последним в пакете специализированных обработчиков. Тогда, если исключение не будет перехвачено специализированными обработчиками, то будет выполнен последний — универсальный.

Если не возникнет исключение, набор обработчиков будет обойден, т.е. проигнорирован.

Если же исключение было «брошено» при возникновении критической ситуации, то будет вызван конкретный перехватчик при совпадении его параметра с выражением в операторе броска, т.е. управление будет передано найденному обработчику. После выполнения кода вызванного обработчика управление передается оператору, который расположен за последним перехватчиком, или проект корректно завершает работу.

Существенное отличие вызова конкретного обработчика от вызова обычной функции заключается в следующем: при возникновении исключения и

передаче управления определенному обработчику система осуществляет вызов всех деструкторов для всех объектов классов, которые были созданы с момента начала контроля и до возникновения исключительной ситуации, с целью их уничтожения.

Блоки try как составные блоки могут быть вложены друг в друга. В случае возникновения исключения в некотором текущем блоке, поиск обработчика последовательно продолжается в блоках, предшествующих уровней вложенности с продолжением вызова деструкторов.

Пример обработки исключительных ситуаций. Функция Divide() возвращает частное от деления чисел, принимаемых в качестве аргументов (a, Ь), если делитель равен нулю (Ь = 0), то возникает исключительная ситуация.

```
double Divide(double, double);
void main(void) {
    double a, b, result;
    cout << " Input a, b : " << endl;
    cin >> a >> b;
    try {
        result = Divide(a, b);
        cout << " Normal Work " << endl;
        cout << " a =  " << a << ",  b =  " << b << endl;
        cout << " Ansver : " << result << endl;
        getch();
    }
    catch (double z) {
        cout << " Division by zero... " << endl;
        cout << " b = " << z << endl;
        getch();
    }
}
double Divide(double a1, double b1) {
    if (b1 == 0) throw b1;
    return a1 / b1;
}
```

double Divide(double, doub1e);

Оператор throw сигнализирует об исключительном событии (попытку деления на ноль) и генерирует объект исключительной ситуации. В данном случае он находится внутри функции Divide(). Объект перехватывается обработчиком catch. Этот процесс, как уже известно, и называется вызовом исключительной ситуации. В рассмотренном примере исключительная ситуация имеет форму вещественной переменной — делителя.

Задание к лабораторной работе

Общая постановка. Даны два выражения Z1 и П. Написать функции для вычисления этих выражений с организацией обнаружения исключительной ситуации (например, деление на ноль) и ее обработки. Передача аргументов в функции — по ссылкам.

Для генерации исключения реализовать свой класс CustomException, которы будет производным от Exception и будет содержать:

- строку с сообщением об ошибке;

- значение, которое привело к ошибке;
- код ошибки.

При обработке исключения показать использование класса CustomException. В случае успеха значения Z1 и Z2 должны быть приблизительно одинаковыми.

Индивидуальные задания

$z_1 = \frac{\sqrt{2b+2\sqrt{b^2-4}}}{\sqrt{b^2-4}+b+2},\ z_2 = \frac{1}{\sqrt{b+2}}.$

$z_1 = \frac{x^2+2x-3+(x+1)\sqrt{x^2-9}}{x^2-2x-3+(x-1)\sqrt{x^2-9}},\ z_2 = \sqrt{\frac{x+3}{x-3}}.$

$z_1 = \frac{\sqrt{(3m+2)^2-24m}}{3\sqrt{m}-2/\sqrt{m}},\ z_2 = -\sqrt{m}.$

$z_1 = \left(\frac{a+2}{\sqrt{2a}} - \frac{a}{\sqrt{2a}+2} + \frac{2}{a-\sqrt{2a}}\right) \cdot \frac{\sqrt{a}-\sqrt{2}}{a+2},\ z_2 = \frac{1}{\sqrt{a}+\sqrt{2}}.$

$z_1 = \left(\frac{1+a+a^2}{2a+a^2} + 2 - \frac{1-a+a^2}{2a-a^2}\right)^{-1} \cdot (5-2a^2),\ z_2 = \frac{4-x}{2}$

$z_1 = \frac{(m-1)\sqrt{m}-(n-1)\sqrt{n}}{\sqrt{m^3n}+nm+m^2-m},\ z_2 = \frac{\sqrt{m}-\sqrt{n}}{m}.$

$z_1 = \frac{\sqrt{2m+2\sqrt{m^2-4}}}{m+\sqrt{m^2-4}+2},\ z_2 = \frac{1}{\sqrt{m+2}}.$

$z_1 = \frac{(x+1)\sqrt{x^2-9}+x(x+2)-3}{(x-1)\sqrt{x^2-9}+x^2-2x-3},\ z_2 = \sqrt{\frac{x+3}{x-3}}.$

$z_1 = (2 + \frac{1+x+x^2}{2x+x^2} - \frac{1-x+x^2}{2x-x^2})^2(5-2x^2),\ z_2 = \frac{4-x^2}{2}.$

$\left(\frac{2}{x-\sqrt{2x}} + \frac{x+2}{\sqrt{2x}} - \frac{x}{\sqrt{2x}+2}\right) \cdot \frac{\sqrt{x}-\sqrt{2}}{x+2},\ z_2 = \frac{1}{\sqrt{x}+\sqrt{2}}.$

$z_1 = \frac{\sqrt{(3x+2)^2-24x}}{3\sqrt{x}-2/\sqrt{x}},\ z_2 = -\sqrt{x}.$

$z_1 = \frac{(a-1)\sqrt{a}-(b-1)\sqrt{b}}{\sqrt{a^3b}+ba+a^2-a},\ z_2 = \frac{\sqrt{a}-\sqrt{b}}{a}.$

$z_1 = \frac{\sqrt{2\sqrt{x^2-4}+2x}}{x+\sqrt{x^2-4}+2},\ z_2 = \frac{1}{\sqrt{x+2}}.$

$z_1 = \left(\frac{m+2}{\sqrt{2m}} + \frac{2}{m-\sqrt{2m}} - \frac{m}{\sqrt{2m}+2}\right) \cdot \frac{\sqrt{m}-\sqrt{2}}{m+2},\ z_2 = \frac{1}{\sqrt{m}+\sqrt{2}}.$

$z_1 = \frac{(x-1)\sqrt{x}-(y-1)\sqrt{y}}{\sqrt{x^3y}+xy+x^2-x},\ z_2 = \frac{\sqrt{x}-\sqrt{y}}{x}.$

1. Что называют исключением?
2. Что такое «блок с контролем»?
3. Дайте характеристику обработчику исключений. Какие бывают виды обработчиков?
4. Какие правила налагаются на соотношения между блоком контроля и обработчиками?
5. Чем отличается вызов обработчика от вызова обычной функции?

Лабораторная работа 9
Использование параметризованных классов

Тема работы: практические приемы использования шаблонов классов.

Цель работы: применить знания, полученные в предыдущих работах, к реализации smart-указателей.

Smart-указатели. При работе с динамическими объектами, используя указатели на них, в случае если объект становится не нужен, то его необходимо уничтожить. В то же время на один объект могут ссылаться множество указателей и, следовательно, нельзя однозначно сказать, нужен ли еще этот объект или он уже может быть уничтожен. Пример реализации класса, для которого происходит уничтожение объектов, в случае если уничтожается последняя ссылка на него.

```
#pragma once
template <typename T >
class smart_pointer {
    T *object;
public:
    smart_pointer(T *object) : object(object) {}
    ~smart_pointer() {
        delete object;
    }
    T* operator->() {
        return object;
    }
    T& operator*() {
        return *object;
    }
};
```

Контрольные вопросы

1. С какой целью используются smart-указатели?
2. Что представляет собой smart-указатель?
3. Реализации smart-указателей используемых в C++: unique_ptr, auto_ptr, scoped_ptr, weak_ptr, shared_ptr. В чем их различие, какие недостатки?

Порядок выполнения работы

4. Изучить краткие теоретические сведения.
5. Ознакомиться с материалами литературных источников.
6. Ответить на контрольные вопросы.
7. Разработать алгоритм программы.
8. Написать, отладить и выполнить программу.

Варианты заданий

Выполнение заданий осуществляется без использования библиотек memory и boost.

В классе должны быть объявлены необходимые конструкторы.

При реализации параметризованного класса предусмотреть методы, для обнуления указателя, обмена адресами для указателей, возвращения «сырого» указателя и др., предусмотренные функционалом соответствующего библиотечного указателя.

В клиенте main() продемонстрировать использование этого класса для векторов, содержащих элементы типов int, double, string.

Для этого написать программу, демонстрирующую работу с этим шаблоном для различных типов параметров шаблона. Программа должна содержать меню, позволяющее осуществить проверку всех методов шаблона.

1. Создать шаблонный класс smart-указателя UPtr, реализующий функционал unique_ptr.
2. Создать шаблонный класс smart-указателя APtr, реализующий функционал auto_ptr.
3. Создать шаблонный класс smart-указателя ScPtr, реализующий функционал scoped_ptr.
4. Создать шаблонный класс smart-указателя WPtr, реализующий функционал weak_ptr.
5. Создать шаблонный класс smart-указателя ShPtr, реализующий функционал shared_ptr.
6. Создать шаблонный класс smart-указателя IPtr, реализующий функционал intrusive_ptr.
7. Создать шаблонный класс smart-указателя UPtrArr, реализующий функционал unique_ptr, для работы с массивами.
8. Создать шаблонный класс smart-указателя APtrArr, реализующий функционал auto_ptr, для работы с массивами.
9. Создать шаблонный класс smart-указателя ScPtrArr, реализующий функционал scoped_ptr, для работы с массивами.
10. Создать шаблонный класс smart-указателя WPtrArr, реализующий функционал weak_ptr, для работы с массивами.
11. Создать шаблонный класс smart-указателя ShPtrArr, реализующий функционал shared_ptr, для работы с массивами.
12. Создать шаблонный класс smart-указателя IPtrArr, реализующий функционал intrusive_ptr, для работы с массивами.

Лабораторная работа 10
Потоки ввода вывода

Тема работы: организация ввода/вывода, динамическое выделение памяти.

Цель работы: изучить организацию ввода/вывода и работу с динамической памятью при программировании алгоритмов в C++.

Теоретические сведения: В C++ ввод и вывод данных производится потоками байт. Поток (последовательность байт) – это логическое устройство, которое выдает и принимает информацию от пользователя и связано с физическими устройствами ввода/вывода. При операциях ввода байты направляются от устройства в основную память. В операциях вывода – наоборот. Имеется четыре потока (связанных с ними объекта), обеспечивающих ввод и вывод информации и определенных в заголовочном файле iostream.

В файле iostream перегружаются два оператора побитового сдвига

<< // поместить в выходной поток

>> // считать со входного потока и объявляются три стандартных потока:

cout // стандартный поток вывода (экран) cin // стандартный поток ввода (клавиатура) cerr // стандартный поток диагностики (ошибки)

Объект cin. Для ввода информации с клавиатуры используется объект cin.

Формат записи cin имеет следующий вид:

cin >> имя_переменной;

При вводе необходимо, чтобы данные вводились в соответствии с форматом переменных.

Объект cout. Объект cout позволяет выводить информацию на стандартное устройство вывода – экран. Формат записи cout имеет следующий вид: cout << data << data; где data – это переменные, константы, выражения или комбинации всех трех типов.

Пример использования объектов cin и cout.

```
#include<iostream>  using
namespace std;
int main()
{
setlocale (LC_ALL,"Russian");  int i;
double x;
cout << "Введите число с двойной точностью" << endl;;
cin >> x;      // ввод числа с плавающей точкой cout << "Введите
положительное число" << endl;
 cin >> i;                                    // ввод целого числа
cout << "i * x=" << i*x << endl; // вывод результата return 0;
}
```

Идентификатор endl называется манипулятором. Он очищает поток cerr и добавляет новую строку.

Для управления выводом информации в языке C++ используются манипуляторы, флаги и функции. Для их использования необходим заголовочный файл iomanip.

Манипуляторы hex и oct используются для вывода числовой информации в шестнадцатеричном или восьмеричном представлении. Применение их можно видеть на примере следующей простои программы:

```
#include<iostream>  using
namespace std;
void main()
{
setlocale(LC_ALL,"Russian");
 int a=0x11, b=4,              // целые числа: шестнадцатеричное и десятичное
c=051, d=8,  //       восьмеричное и десятичное i,j;
i=a+b; j=c+d; cout << i << ` ` << hex << i << ` ` << oct << i << ` ` << dec << endl;  cout << hex << j << ` ` <<j << ` ` << dec << j << ` ` << oct << j << endl; }
```

Манипуляторы изменяют значение некоторых переменных в объекте cout. Эти переменные называются флагами состояния. Когда объект посылает данные на экран, он проверяет эти флаги.

Пример использования манипуляторов форматирования информации.

```
#include<iostream>
#include <iomanip>  using
namespace std;
void main()
{ int a=0x11;  double d=12.362;
cout << setw(4) << a << endl;
cout << setw(10) << setfill('*') << a << endl;
cout << setw(10 ) << setfill(' ') << setprecision(3) << d << endl; }
```

Манипуляторы setw(), setfill(' ') и setprecision() позволяют изменять флаги состояния объекта cout. Они имеют следующий формат:

setw(количество_позиций_для_вывода_числа)

setfill(символ_для_заполнения_пустых_позиций)

setprecision (точность_при_выводе_дробного_числа)

Наряду с перечисленными выше манипуляторами в C++ используются также манипуляторы **setiosflags()** и **resetiosflags()** для установки определенных глобальных флагов, используемых при вводе и выводе информации. На эти флаги ссылаются как на *переменные состояния.* Манипулятор setiosflags() устанавливает указанные в ней флаги, а resetiosflags() сбрасывает (очищает) их.

Для того чтобы установить или сбросить некоторый флаг, могут быть использованы функции **setf()** или **unsetf().** Флаги формата объявлены в классе **ios.** Рассмотрим пример, демонстрирующий использование манипуляторов.

```
#include<iostream>
#include <iomanip>  using
namespace std;
int main()
{
setlocale(LC_ALL,"Russian");  char
S[]="МИТСО";
cout << setw(30) << setiosflags(ios::right) << s << endl; cout <<
resetiosflags(ios::right);
cout << setw(30) << setiosflags(ios::left) << s << endl; return 0;
}
```

Контрольные вопросы

1. Как осуществляется ввод/вывод в C++?
2. Для чего используются манипуляторы в потоках ввода/вывода?
3. Какие манипуляторы ввода/вывода вы знаете?
4. Для чего используются функции в потоках ввода/вывода?
5. Какие функции, используемые в потоках ввода/вывода вы знаете?
6. Для чего используются флаги в потоках ввода/вывода?
7. Какие флаги, используемые в потоках ввода/вывода вы знаете?

Порядок выполнения работы

1. Изучить краткие теоретические сведения.
2. Ознакомиться с материалами литературных источников.
3. Ответить на контрольные вопросы.
4. Разработать алгоритм программы.
5. Написать, отладить и выполнить программу.

Варианты заданий

1. Написать программу, печатающую все вводимые символы в нижнем регистре. В строку выводится символ, шестнадцатеричный и восьмеричный код. 2. Написать программу, печатающую строчные и прописные буквы русского алфавита. В строку выводится символ, шестнадцатеричный и восьмеричный код.

2. Написать программу, которая получает данные либо по Фаренгейту в виде 59 F и преобразует их в данные по Цельсию 15 °C, либо наоборот. Нуль по Цельсию равен 32 по Фаренгейту. Один градус Цельсия равен 1,8 по Фаренгейту. Установить ширину поля в 10 символов, точность – 4 цифры, заполнить пробелы символом «\» с помощью функций и манипуляторов.

3. Написать программу решения квадратного уравнения. Установить ширину поля в 10 символов, точность – 4 цифры, заполнить пробелы символом «#».

4. Написать программу, печатающую все вводимые символы. В строку выводится символ, шестнадцатеричный и восьмеричный код.

5. Написать программу решения линейного уравнения. Установить ширину поля в 10 символов, точность – 4 цифры, заполнить пробелы символом «%».

6. Написать программу с двумя своими манипуляторами. Один выводит сообщение, другой устанавливает шестнадцатеричный вывод и символ заполнения «$».

7. Написать программу вычисления значение y=cos(t) для t∈[0;3] с шагом 0,5. Установить ширину поля в 10 символов, точность – 3 цифры.

8. Перегрузите операцию << для класса , который будет просить ввести любую строку, а затем должен опять вывести её на экран но без больших букв и цифр.

9. Напишите программу, которая печатает (1) все буквы в нижнем регистре, (2) все буквы, (3) все буквы и цифры, (4) все символы, которые могут встречаться в идентификаторах C++ на вашей системе, (5) все символы пунктуации

10. Создайте свой манипуляторы которые будут выполнять 1) вывод всех символов большими а чисел в OCT формате 2) вывод всех символов маленькими и замену пробелов на # 3)вывод всех чисел в HEX формате и не выводить пробелов

11. Напишите программу которая будет просить ввести строку символов , после этого он печатает её на экране 1) все буквы в нижнем регистре , 2) все буквы в большом регистре , 3) все числа в строке выводить в HEX формате , 4) вывод с заменой всех пробелов на символ #

12. Написать программу которая создаёт массив размером [256] элементов и заполняет его случайным образом числами от 1 до 1000 . После этого вывести все числа на экран в несколько форматированых столбцов , при этом в первом столбце выводить числа в DEC формате , во втором в HEX формате , в третьем в OCT формате , четвёртый опять в DEC и т.д. до конца.

13. Создать класс который при инициализации (в конструкторе) получает имя файла с текстовой строкой, а при уничтожении (в деструкторе) сохраняет строку в другом файле, убрав при этом все пробелы, обрезав у всех дробных чисел количество знаков после запятой до двух и заменив все маленькие буквы на большие.

14. Перегрузите операцию << для класса, который будет просить ввести любую строку, а затем должен опять вывести её на экран но без маленьких букв

15. Перегрузите операцию >> для класса, перегруженная функция должна передать строку из случайных символов, а класс должен вывести всю строку форматировано на экран и без пробелов

16. Перегрузите операцию >> для класса , перегруженная функция должна предать строку из случайных символов, а класс должен вывести всё строку большими буквами ,а все числа в строке вывести в HEX формате

Следующие задания требуется решить с привлечением текстовых файлов. Нужно написать функцию, с помощью которой подготовить входной файл, записав в него 100 случайных целых чисел в диапазоне от -50 до +50 по одному на строке. Сформировать выходной файл, преобразовав числа входного файла.

1. Записать выходной файл, добавить к каждому числу последнее число файла.
2. Записать выходной файл, разделить каждое число на полусумму первого отрицательного и 50-го числа файла.
3. Записать выходной файл, вычесть из каждого числа набольшее число файла.
4. Записать выходной файл, умножить каждое число на минимальное из чисел файла.
5. Записать выходной файл, разделив все нечетные по абсолютной величине числа на среднее арифметическое.
6. Записать выходной файл, вычесть из каждого числа сумму чисел файла.
7. Записать выходной файл, умножить каждое третье число на удвоенную сумму первого и последнего отрицательных чисел.
8. Записать выходной файл, добавить к каждому числу первое нечетное по абсолютной величине число файла.
9. Записать выходной файл, умножить каждое четное число на первое отрицательное число файла.
10. Записать выходной файл, добавить к каждому числу половину последнего отрицательного числа файла.
11. Записать выходной файл, разделить все числа на половину максимального числа.
12. Записать выходной файл, заменив все нули средним арифметическим.
13. Записать выходной файл, заменив все положительные числа квадратом минимума.
14. Записать выходной файл, добавив к каждому числу полусумму всех отрицательных чисел.
15. Записать выходной файл, добавить к каждому числу среднее арифметическое наименьшего по абсолютной величине и наибольшего из чисел файла.
16. Записать выходной файл, разделив число на минимум и добавив максимум.
17. Записать выходной файл, заменив все положительные числа на максимум.

III. РАЗДЕЛ КОНТРОЛЯ ЗНАНИЙ

ПРИМЕРНЫЙ ПЕРЕЧЕНЬ ВОПРОСОВ (ЗАДАНИЙ) ДЛЯ ПОДГОТОВКИ К ТЕКУЩЕЙ АТТЕСТАЦИИ

1. Основные принципы объектно-ориентированного программирования.
2. Базовые абстракции объектно-ориентированного программирования.
3. Понятие класса и объекта.
4. Структурные элементы класса и методы взаимодействия объектов. Конструкторы и деструкторы класса.
5. Разграничение доступа к атрибутам объектов Классы в программных модулях. Атрибуты доступа к элементам объектов. Термин «инкапсуляция».
6. Жизненный цикл объектов.
7. Методы и механизмы разработки объектно-ориентированных программ.
8. Механизм наследования. Производные классы. Свойства базового и производного класса.
9. Понятие виртуального метода. Перекрытие виртуального метода в производном классе. Абстрактный виртуальный метод. Механизм вызова виртуального метода. Методы обработки сообщений. Термин «полиморфизм».
10. Понятие ссылки на метод объекта. Понятие события. Применение ссылок на методы для расширения объектов.
11. Исключительные ситуации. Классы исключительных ситуаций. Создание и обработка исключительных ситуаций. Защита от утечки ресурсов в случае возникновения исключительных ситуаций.
12. Понятие интерфейса. Описание интерфейса. Поддержка интерфейса классом. Совместимость интерфейсов и классов. Механизм вызова метода объекта через интерфейс. Применение интерфейса для доступа к объекту динамически-подключаемой библиотеки.
13. Параметризация объектов. Параметризированные классы и методы, их свойства. Совместное использование параметризации и принципов наследования. Организация внешнего доступа к компонентам параметризированных классов.
14. Контейнерные типы и их применение.
15. Понятие компонента. Понятие визуального программирования. Инструментальные средства визуального компонентного программирования. Современные библиотеки компонентов.
16. Объектно-ориентированный анализ и проектирование программных средств.
17. Модели процесса разработки. Классификация методологий создания программных проектов. Выбор методологии для определенного типа программного проекта.
18. Виды отношений классов (ассоциация, агрегация, обобщение, зависимость, инстанцирование).
19. Назначение и условия применимости паттернов проектирования.

Способ и результат применения. Классификация паттернов по стилю. Структурные и порождающие паттерны, паттерны поведения.

20. Архитектура и архитектурный подход. Архитектурные шаблоны в работе с уровнем данных. Основные шаблоны уровня данных (доступа к данным и базам данных).

21. Паттерны проектирования.

IV. ВСПОМОГАТЕЛЬНЫЙ РАЗДЕЛ

РЕКОМЕНДУЕМЫЙ ПЕРЕЧЕНЬ ЛИТЕРАТУРЫ

ОСНОВНАЯ ЛИТЕРАТУРА

1. Шилдт, Г. C++: полное руководство / Герберт Шилдт. – М. : Вильямс, 2018. – 796 с.

2. Прата, С. Язык программирования С--+.Лекции и упражнения : пер. с англ. / Стивен Прата. – 6-е изд. – М. [и др.] : Вильямс, 2018. – 1244 с.

3. Страуструп, Б. Программирование. Принципы и практика с использованием C++ : пер. с англ. / Б. Страуструп. – 2-е изд. – М. : ООО «И. Д. Вильямс», 2018. – 1328 с.

4. Маклафлин, Б. Объектно-ориентированный анализ и проектирование : пер. с англ. / Б. Маклафлин, Г. Поллайс, Д. Уэст. – СПб. [и др.] : Питер : Питер Пресс, 2018. – 601 с.

5. Эккель, Б. Философия Java : пер. с англ. / Брюс Эккель. – 4-е полное изд. – СПб. [и др.] : Питер : Питер Пресс, 2018. – 1165 с.

ДОПОЛНИТЕЛЬНАЯ ЛИТЕРАТУРА

6. ГОСТ 19.701-90 – Единая система программной документации – Схемы алгоритмов, программ, данных и систем – Условные обозначения и правила выполнения.

7. Шилдт, Г. Полный справочник по C++ / Г. Шилдт ; пер. с англ. – М. : Вильямс, 2016. – 800 с.

8. Страуструп, Б. Программирование. Принципы и практика с использованием C++ : пер. с англ. / Бьярне Страуструп. – 2-е изд. – М. [и др.] : Вильямс, 2018. – 1328 с.