МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ

**«БРАТСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»**

**Кафедра управления в технических системах**

УТВЕРЖДАЮ:
Проректор по учебной работе
_____________ Е.И. Луковникова
«29» мая 2020 г.

# РАБОЧАЯ ПРОГРАММА ДИСЦИПЛИНЫ

# ПРИКЛАДНОЕ ПРОГРАММИРОВАНИЕ

**Б1.В.ДВ.04.01**

## НАПРАВЛЕНИЕ ПОДГОТОВКИ

**27.03.04 Управление в технических системах**

## ПРОФИЛЬ ПОДГОТОВКИ

**Управление и информатика в технических системах**

Программа академического бакалавриата

Квалификация (степень) выпускника: бакалавр

Программа составлена в соответствии с федеральным государственным образовательным стандартом высшего образования по направлению подготовки 27.03.04 Управление в технических системах от 20.10.2015 г № 1171 и учебным планом ФГБОУ ВО «БрГУ» от 03.02.2020 г № 46 для очной формы обучения, заочно - ускоренной формы обучения для набора 2020 года

**СОДЕРЖАНИЕ ПРОГРАММЫ** **Стр.**


**1. ПЕРЕЧЕНЬ ПЛАНИРУЕМЫХ РЕЗУЛЬТАТОВ ОБУЧЕНИЯ ПО ДИСЦИПЛИНЕ, СООТНЕСЕННЫХ С ПЛАНИРУЕМЫМИ РЕЗУЛЬТАТАМИ ОСВОЕНИЯ ОБРАЗОВАТЕЛЬНОЙ ПРОГРАММЫ** ……… 3

**2. МЕСТО ДИСЦИПЛИНЫ В СТРУКТУРЕ ОБРАЗОВАТЕЛЬНОЙ ПРОГРАММЫ** …………………………………………………………….……….. 3

**3. РАСПРЕДЕЛЕНИЕ ОБЪЕМА ДИСЦИПЛИНЫ** ………………………………….. 4

3.1 Распределение объёма дисциплины по формам обучения……………...……… 4

3.2 Распределение объёма дисциплины по видам учебных занятий и трудоемкости ………………………………………………………………..……… 4

**4. СОДЕРЖАНИЕ ДИСЦИПЛИНЫ** ……………………………………..………… 5

4.1 Распределение разделов дисциплины по видам учебных занятий ……………… 5

4.2 Содержание дисциплины, структурированное по разделам и темам …..…………6

4.3 Лабораторные работы……………………………………………………………34

4.4 Семинары / практические занятия…..……………………………….……… 35

4.5. Контрольные мероприятия: курсовой проект (курсовая работа), контрольная работа, РГР, реферат…………………………………………………….……… 35

**5. МАТРИЦА СООТНЕСЕНИЯ РАЗДЕЛОВ УЧЕБНОЙ ДИСЦИПЛИНЫ К ФОРМИРУЕМЫМ В НИХ КОМПЕТЕНЦИЯМ И ОЦЕНКЕ РЕЗУЛЬТАТОВ ОСВОЕНИЯ ДИСЦИПЛИНЫ** …………………………………… 36

**6. ПЕРЕЧЕНЬ УЧЕБНО-МЕТОДИЧЕСКОГО ОБЕСПЕЧЕНИЯ ДЛЯ САМОСТОЯТЕЛЬНОЙ РАБОТЫ ОБУЧАЮЩИХСЯ ПО ДИСЦИПЛИНЕ**37

**7. ПЕРЕЧЕНЬ ОСНОВНОЙ И ДОПОЛНИТЕЛЬНОЙ ЛИТЕРАТУРЫ, НЕОБХОДИМОЙ ДЛЯ ОСВОЕНИЯ ДИСЦИПЛИНЫ**…………………………… 37

**8. ПЕРЕЧЕНЬ РЕСУРСОВ ИНФОРМАЦИОННО – ТЕЛЕКОММУНИКАЦИОННОЙ СЕТИ «ИНТЕРНЕТ» НЕОБХОДИМЫХ ДЛЯ ОСВОЕНИЯ ДИСЦИПЛИНЫ** …………………………………………. 38

**9. МЕТОДИЧЕСКИЕ УКАЗАНИЯ ДЛЯ ОБУЧАЮЩИХСЯ ПО ОСВОЕНИЮ ДИСЦИПЛИНЫ**………………………………………………………………… 38

9.1. Методические указания для обучающихся по выполнению лабораторных работ / семинаров / практических работ ……………………………………….……… 38

**10. ПЕРЕЧЕНЬ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, ИСПОЛЬЗУЕМЫХ ПРИ ОСУЩЕСТВЛЕНИИ ОБРАЗОВАТЕЛЬНОГО ПРОЦЕССА ПО ДИСЦИПЛИНЕ** ………………………………………………………………..…… 44

**11. ОПИСАНИЕ МАТЕРИАЛЬНО-ТЕХНИЧЕСКОЙ БАЗЫ, НЕОБХОДИМОЙ ДЛЯ ОСУЩЕСТВЛЕНИЯ ОБРАЗОВАТЕЛЬНОГО ПРОЦЕССА ПО ДИСЦИПЛИНЕ** ……………………………………………………………...…… 45

**Приложение 1.** Фонд оценочных средств для проведения промежуточной аттестации обучающихся по дисциплине……………………………46

**Приложение 2.** Аннотация рабочей программы дисциплины ………………………. 51

**Приложение 3.** Протокол о дополнениях и изменениях в рабочей программе ………… 52

## 1. ПЕРЕЧЕНЬ ПЛАНИРУЕМЫХ РЕЗУЛЬТАТОВ ОБУЧЕНИЯ ПО ДИСЦИПЛИНЕ, СООТНЕСЕННЫХ С ПЛАНИРУЕМЫМИ РЕЗУЛЬТАТАМИ ОСВОЕНИЯ ОБРАЗОВАТЕЛЬНОЙ ПРОГРАММЫ

**Вид деятельности выпускника**

Дисциплина охватывает круг вопросов, относящихся к проектно-конструкторскому виду профессиональной деятельности выпускника в соответствии с компетенцией и видами деятельности, указанными в учебном плане.

**Цель дисциплины**

Изложение базовых принципов объектно-ориентированного программирования в объёме, необходимом для успешного использования современных интегрированных пакетов визуального программирования при проектировании и разработке графических интерфейсов пользователя.

**Задачи дисциплины**

Подготовить обучающихся к самостоятельной работе по решению практических задач, связанных с разработкой графического интерфейса пользователя в интегрированной среде проектирования (ИСР) для решения технических задач, стоящих перед специалистом.

| Код компетенции | Содержание компетенций | Перечень планируемых результатов обучения по дисциплине |
|---|---|---|
| 1 | 2 | 3 |
| ОПК-9 | способность использовать навыки работы с компьютером, владеть методами информационных технологий, соблюдать основные требования информационной безопасности | **знать:**<br>- основные требования информационной безопасности к программным системам;<br>**уметь:**<br>- использовать навыки работы с компьютером;<br>**владеть:**<br>- методами информационных технологий. |
| ПК-6 | способность производить расчёты и проетирование отдельных блоков и устройств систем автоматизации и управления и выбирать стандартные средства автоматики, измерительной и вычислительной техники для проектирования систем автоматизации и управления | **знать:**<br>- основные принципы и методологию разработки прикладного программного обеспечения.<br>**уметь:**<br>- выбирать стандартные средства автоматики, измерительной и вычислительной техники для проектирования систем автоматизации и управления;<br>**владеть:**<br>- навыками расчётов и проетирования отдельных блоков и устройств систем автоматизации и управления. |

## 2. МЕСТО ДИСЦИПЛИНЫ В СТРУКТУРЕ ОБРАЗОВАТЕЛЬНОЙ ПРОГРАММЫ

Дисциплина Б1.В.ДВ.04.01 «Прикладное программирование» относится к вариативной части, дисциплина по выбору.

Дисциплина «Прикладное программирование» базируется на знаниях, полученных при изучении таких учебных дисциплин, как Б1.В.07 Информатика, Б1.В.15 Структуры и алгоритмы обработки данных, Б1.В.18 Технологии программирования, Б1.Б.14 Программирование и основы алгоритмизации.

Дисциплина «Прикладное программирование» представляет основу для изучения дисциплин: Б1.В.ДВ.11.01 Проектирование автоматизированных систем.

Такое системное междисциплинарное изучение направлено на достижение требуемого ФГОС уровня подготовки по квалификации бакалавр.

## 3. РАСПРЕДЕЛЕНИЕ ОБЪЕМА ДИСЦИПЛИНЫ

### 3.1. Распределение объема дисциплины по формам обучения

| *Форма обучения* | *Курс* | *Семестр* | *Трудоемкость дисциплины в часах* | | | | | | *Курсовая работа (проект), контрольная работа, реферат, РГР* | *Вид промежуточной аттестации* |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | *Всего часов (с экз.)* | *Аудиторных часов* | *Лекции* | *Лабораторные работы* | *Практические занятия* | *Самостоятельная работа* | | |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| **Очная** | 4 | 7 | 144 | 51 | 17 | 34 | - | 93 | - | экзамен |
| **Заочная** | 4 | - | 144 | 9 | 4 | 5 | - | 135 | - | экзамен |
| **Заочная** (ускоренное обучение) | - | - | - | - | - | - | - | - | - | экзамен |
| **Очно-заочная** | - | - | - | - | - | - | - | - | - | - |

### 3.2. Распределение объема дисциплины по видам учебных занятий и трудоемкости

| *Вид учебных занятий* | *Трудоемкость (час.)* | *в т.ч. в интерактивной, активной, инновациионной формах, (час.)* | *Распределение по семестрам, час* |
|---|---|---|---|
| | | | *7* |
| 1 | 2 | 3 | 4 |
| **I. Контактная работа обучающихся с преподавателем (всего)** | 51 | 10 | 51 |
| Лекции (Лк) | 17 | 6 | 17 |
| Лабораторные работы (ЛР) | 34 | 4 | 34 |
| Групповые (индивидуальные) консультации | + | - | + |
| **II.Самостоятельная работа обучающихся (СР)** | 66 | - | 66 |
| Подготовка к лабораторным работам | 55 | - | 55 |
| Подготовка к экзамену в течении семестра | 11 | - | 11 |
| **III. Промежуточная аттестация** экзамен | 27 | - | 27 |
| Общая трудоемкость дисциплины ….... час. | 144 | - | 144 |
| зач. ед. | 4 | - | 4 |

## 4. СОДЕРЖАНИЕ ДИСЦИПЛИНЫ

### 4.1. Распределение разделов дисциплины по видам учебных занятий

**- для очной формы обучения:**

| № раздела и темы | Наименование раздела и тема дисциплины | Трудоемкость, (час.) | Виды учебных занятий, включая самостоятельную работу обучающихся и трудоемкость; (час.) | | |
|---|---|---|---|---|---|
| | | | учебные занятия | | самостоятельная работа обучающихся |
| | | | лекции | лабораторные работы | |
| 1 | 2 | 3 | 4 | 5 | 6 |
| **1.** | **Введение. Объектно-ориентированное программирование** | **15** | **2** | **4** | **9** |
| 1.1. | Основные понятия и определения | 7,5 | 1,5 | - | 6 |
| 1.2. | Основы визуального программирования интерфейса | 7,5 | 0,5 | 4 | 3 |
| **2.** | **Основные направления в программировании** | **14** | **2** | **4** | **8** |
| 2.1. | Процедурное программирование | 3,5 | 0,5 | 1 | 2 |
| 2.2. | Модульное программирование | 3,5 | 0,5 | 1 | 2 |
| 2.3. | Концепция типов | 3,5 | 0,5 | 1 | 2 |
| 2.4. | Объектно-ориентированное программирование | 3,5 | 0,5 | 1 | 2 |
| **3.** | **Классы** | **16** | **2** | **6** | **8** |
| 3.1. | Объявление класса | 9 | 1 | 3 | 5 |
| 3.2. | Функции-элементы, дружественные функции, константные функции | 7 | 1 | 3 | 3 |
| **4.** | **Базовые принципы объекно-ориентированного программирования** | **15** | **2** | **4** | **9** |
| 4.1. | Абстрагирование | 3,25 | 0,25 | 1 | 2 |
| 4.2. | Ограничение доступа | 2,25 | 0,25 | 1 | 1 |
| 4.3. | Модульность | 1,3 | 0,3 | - | 1 |
| 4.4. | Иерархия | 1,25 | 0,25 | - | 1 |
| 4.5. | Типизация | 2,25 | 0,25 | 1 | 1 |
| 4.6. | Параллелизм | 2,2 | 0,2 | 1 | 1 |
| 4.7. | Устойчивость | 1,25 | 0,25 | - | 1 |
| 4.8. | Иерархия классов | 1,25 | 0,25 | - | 1 |
| **5.** | **Базовые конструкции объектно-ориентированных программ** | **14** | **2** | **4** | **8** |
| 5.1. | Объекты и классы | 3,5 | 0,5 | 1 | 2 |
| 5.2. | Интерфейс и реализация объекта | 3,5 | 0,5 | 1 | 2 |
| 5.3. | Механизмы наследования | 3,5 | 0,5 | 1 | 2 |
| 5.4. | Обработка исключений | 3,5 | 0,5 | 1 | 2 |
| **6.** | **Данные и функции класса** | **16** | **3** | **4** | **9** |
| 6.1. | Данные-элементы, статические данные, константные данные | 7 | 1 | 2 | 4 |
| 6.2. | Конструкторы и деструкторы | 9 | 2 | 2 | 5 |
| **7.** | **Наследование и полиморфизм,** | **13** | **2** | **4** | **7** |

| | **виртуальные функции, абстрактные классы** | | | | |
|---|---|---|---|---|---|
| 7.1. | Наследование и полиморфизм | 6,5 | 1 | 2 | 3,5 |
| 7.2. | Виртуальные функции, абстрактные классы | 6,5 | 1 | 2 | 3,5 |
| **8.** | **Особенности классов, наследующих классам библиотеки компонентов C++ Builder** | **14** | **2** | **4** | **8** |
| 8.1. | Свойства | 8 | 1 | 2 | 5 |
| 8.2. | События | 6 | 1 | 2 | 3 |
| | **ИТОГО** | **117** | **17** | **34** | **66** |

**- для заочной формы обучения:**

| *№ раздела и темы* | *Наименование раздела и тема дисциплины* | *Трудоемкость, (час.)* | *Виды учебных занятий, включая самостоятельную работу обучающихся и трудоемкость; (час.)* | | |
|---|---|---|---|---|---|
| | | | *учебные занятия* | | *самостоятельная работа обучающихся* |
| | | | *лекции* | *лабораторные работы* | |
| 1 | 2 | 3 | 4 | 5 | 6 |
| **1.** | **Введение. Объектно-ориентированное программирование** | **17** | **0,5** | **0,5** | **16** |
| **2.** | **Основные направления в программировании** | **16** | **0,5** | **0,5** | **15** |
| **3.** | **Классы** | **17,5** | **0,5** | **1** | **16** |
| **4.** | **Базовые принципы объекно-ориентированного программирования** | **17** | **0,5** | **0,5** | **16** |
| **5.** | **Базовые конструкции объектно-ориентированных программ** | **17** | **0,5** | **0,5** | **16** |
| **6.** | **Данные и функции класса** | **17,5** | **0,5** | **1** | **16** |
| **7.** | **Наследование и полиморфизм, виртуальные функции, абстрактные классы** | **17** | **0,5** | **0,5** | **16** |
| **8.** | **Особенности классов, наследующих классам библиотеки компонентов C++ Builder** | **16** | **0,5** | **0,5** | **15** |
| | **ИТОГО** | **135** | **4** | **5** | **126** |

### 4.2. Содержание дисциплины, структурированное по разделам и темам

### Раздел 1:
### ВВЕДЕНИЕ. ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

1.1. Основные понятия и определения

Объектно-ориентированный подход уже давно стал естественным для любых прикладных программ Windows. Когда вы начинаете выполнять любую программу Windows, вы видите обычно диалоговое окно с меню, кнопками, индикаторами, выпадающими списками и т.п. Все это объекты. Сами по себе они не производят никаких действий, пока не получат какое-то сообщение от пользователя. После получения сообщения происходят какие-то дей-

ствия (проводятся вычисления, выполняется какая-то программа, появляется новое диалоговое окно и т.п.). После этого опять всякая активность затухает, пока тот или иной объект не получит нового сообщения от пользователя или от другого объекта. Таким образом, объектно-ориентированная программа не имеет жесткого алгоритма работы. Она представляет собой систему объектов, каждый из которых может выполнять какие-то функции в ответ на полученное сообщение, в частности, может сам генерировать сообщения, на которые будут реагировать другие объекты. Взаимодействие пользователя с компьютерной программой – это тоже взаимодействие двух объектов – программы и человека, которые обмениваются друг с другом определенными сообщениями.

Попробуем разобраться с основным понятием объектно-ориентированного программирования – объектом. Для начала можно определить объект как некую *совокупность данных и способов работы с ними.* Данные можно рассматривать характеристики объекта. Пользователь и объекты программы должны, конечно, иметь возможность читать эти данные объекта, как-то их обрабатывать и записывать в объект новые значения.

Здесь важнейшее значение имеют принципы инкапсуляции и скрытия данных. Принцип скрытия данных заключается в том, что внешним объектам и пользователю прямой доступ к данным, как правило, запрещен. Делается это из двух соображений.

Во-первых, для надежного функционирования объекта надо поддерживать целостность и непротиворечивость его данных. Если не позаботиться об этом, то внешний объект или пользователь могут занести в объект такие неверные данные, что он начнет функционировать с ошибками.

Во-вторых, необходимо изолировать внешние объекты от особенностей внутренней реализации данных. Для внешних потребителей данных должен быть доступен только *пользовательский интерфейс* – описание того, какие имеются данные и функции и как их использовать. А внутренняя реализация – это дело разработчика объекта. При таком подходе разработчик может в любой момент модернизировать объект, изменить структуру хранения и форму представления данных, но, если при этом не затронут интерфейс, внешний потребитель этого даже не заметит. И, значит, во внешней программе и в поведении пользователя ничего не придется менять.

Для того, чтобы эффективно решать сложные задачи, необходимо их разбивать на некоторые подзадачи. Каждая отдельная подзадача должна быть относительно независимой и представлять собой некоторый законченный модуль программы. Каждый модуль характеризуется двумя очень важными свойствами:

- он имеет интерфейс или средства взаимодействия с внешней средой;
- он является самостоятельной программной единицей, выполняющей определённые функции. При этом мы говорим о внутренней реализации, т.е. о совокупности алгоритмов и данных, описанных на языке программирования.

Интерфейс и внутренняя реализация являются определяющими свойствами объектов окружающего нас мира. Интерфейс – это средство общения или взаимодействия с объектом. Он интересен нам тогда, когда мы используем объект. Реализация – это внутреннее свойство объекта. Обычно нас не интересуют детали реализации, но интересует её эффективность.

Чтобы выдержать принцип скрытия данных, в объекте обычно определяются функции, обеспечивающие все необходимые операции с данными: их чтение, преобразование, запись. Эти функции называются *методами* и через них происходит общение с данными объекта (рис. 1.1).

Рис. 1.1. Схема обращения к данным через методы объекта

Совокупность данных и методов их чтения и записи называется *свойством.* Свойства можно устанавливать в процессе проектирования, их можно изменять программно во время выполнения вашей прикладной программы. Все общение с данными происходит через методы их чтения и записи. Это происходит и в процессе проектирования, когда среда проектирования C++Builder запускает в нужный момент эти методы, и в процессе выполнения приложения, поскольку компилятор C++Builder незримо для разработчика вставляет в нужных местах программы вызовы этих методов. Например, если в объекте имеется некоторое числовое свойство A,

то вы можете написать оператор вида:

```
A = A + 1;
```

который прибавляет к значению A единицу. Но в действительности реализация этого оператора во многих случаях выглядит сложнее. Компилятор преобразует этот простой код в вызов метода чтения свойства A, к возвращенному этим методом значению прибавляется единица, а затем вызывается метод записи свойства, который заносит полученный результат в данные объекта.

Средой взаимодействия объектов (как бы силовым полем, в котором существуют объекты) являются *сообщения,* генерируемые в результате различных *событий.* События наступают, прежде всего, вследствие действий пользователя – перемещения курсора мыши, нажатия кнопок мыши или клавиш клавиатуры. Но события могут наступать и в результате работы самих объектов. В каждом объекте определено множество событий, на которые он может реагировать. В конкретных экземплярах объекта могут быть определены *обработчики* каких-то из этих событий, которые и определяют реакцию данного экземпляра объекта. В обработчиках анализируется сгенерированное событие, и предпринимаются те или иные действия: изменение свойств объектов, выполнение каких-то методов, генерация новых событий. К написанию этих обработчиков, часто весьма простых, и сводится, как будет видно далее, основное программирование при разработке графического интерфейса пользователя с помощью C++Builder.

Теперь можно окончательно определить объект как совокупность свойств и методов, а также событий, на которые он может реагировать. Условно это изображено на рис. 1.2. Внешнее управление объектом осуществляется через обработчики событий. Эти обработчики обращаются к методам и свойствам объекта. Начальные значения данных объекта могут задаваться также в процессе проектирования установкой различных свойств. В результате выполнения методов объекта могут генерироваться новые события, воспринимаемые другими объектами программы или пользователем.

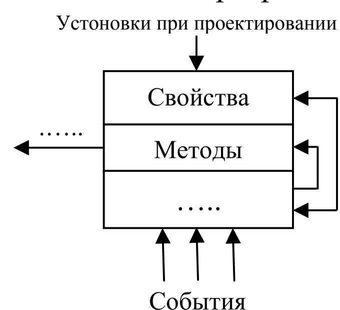


Рис. 1.2. Схема организации объекта

Описанная ранее структура программы в виде некоторой фиксированной совокупности объектов не является полной. Чаще всего сложная программа – это не просто какая-то предопределенная совокупность объектов. В процессе работы объекты могут создаваться и уничтожаться. Таким образом, структура программы является динамическим образованием, меняющимся в процессе выполнения. Основная цель создания и уничтожения объектов – экономия ресурсов компьютера и, прежде всего, памяти. Несмотря на бурное развитие вычислительной техники, память, наверное, всегда будет лимитировать возможности сложных приложений. Это связано с тем, что сложность программных проектов растет теми же, если не более быстрыми, темпами, что и техническое обеспечение. Поэтому от объектов, которые не нужны на данной стадии выполнения программы, нужно освобождаться. При этом освобождаются и выделенные им области памяти, которые могут использоваться вновь создаваемыми объектами. Простой пример этого – окно-заставка с логотипом, появляющееся при запуске многих приложений. После начала реального выполнения приложения эта заставка исчезает с экрана и никогда больше не появляется в данном сеансе работы. Было бы варварством не уничтожить этот объект и не освободить занимаемую им память для более продуктивного использования.

С целью организации динамического распределения памяти во все объекты заложены методы их создания – конструкторы и уничтожения – деструкторы. Конструкторы объектов, которые изначально должны присутствовать в приложении (прикладной программе), срабатывают при запуске программы. Деструкторы всех объектов, имеющихся в данный момент в приложении, срабатывают при завершении его работы. Но нередко и в процессе выполнения различные новые объекты (например, новые окна документов) динамически создаются и уничтожаются с помощью их конструкторов и деструкторов.

Включать объекты в свою программу можно двумя способами:

- вручную – включать в нее соответствующие операторы (это приходится делать не очень часто);

- путем визуального программирования, используя заготовки-компоненты.

## 1.2. Основы визуального программирования интерфейса

Сколько существует программирование, столько существуют в нем и тупики, в которые оно постоянно попадает и из которых, в конце концов, доблестно выходит. Один из таких тупиков или кризисов не так давно был связан с разработкой графического интерфейса пользователя. Программирование вручную всяких привычных пользователю окон, кнопок, меню, обработка событий мыши и клавиатуры, включение в программы изображений и звука требовало все больше и больше времени программиста. В ряде случаев весь этот сервис начинал занимать до 80-90% объема программных кодов. Причем весь этот труд нередко пропадал почти впустую, поскольку через год – другой менялся общепринятый стиль графического интерфейса, и все приходилось начинать заново.

Выход из этой ситуации обозначился благодаря двум подходам. Первый из них – стандартизация многих функций интерфейса, благодаря чему появилась возможность использовать библиотеки, имеющиеся, например, в Windows. В итоге при смене стиля графического интерфейса (например, при переходе от Windows 3.x к Windows 95) приложения смогли автоматически приспосабливаться к новой системе без какого-либо перепрограммирования. На этом пути создались прекрасные условия для решения одной из важнейших задач совершенствования техники программирования – *повторного использования кодов.* Однажды разработанные вами формы, компоненты, функции могли быть впоследствии неоднократно использованы вами или другими программистами для решения их задач. Каждый программист получил доступ к наработкам других программистов и к огромным библиотекам, созданным различными фирмами. Причем была обеспечена совместимость программного обеспечения, разработанного на разных алгоритмических языках.

Вторым революционным шагом, кардинально облегчившим жизнь программистов, явилось появление визуального программирования, возникшего в Visual Basic и нашедшего блестящее воплощение в системах Delphi и C++Builder фирмы Borland. Это явилось решающим шагом в развитии так называемой CASE-технологии (Computer Aided Software Engineering – автоматизированное проектирование программного обеспечения).

Визуальное программирование позволило свести проектирование пользовательского интерфейса к простым и наглядным процедурам, которые дают возможность за минуты или часы сделать то, на что ранее уходили месяцы работы. В современном виде в C++Builder это выглядит так.

Вы работаете в Интегрированной Среде Разработки (ИСР или Integrated development environment – IDE) C++Builder. Среда предоставляет вам *формы* (в приложении их может быть несколько), на которых размещаются компоненты. Обычно это оконная форма, хотя могут быть и невидимые формы. На форму с помощью мыши переносятся и размещаются пиктограммы компонентов, имеющихся в библиотеках C++Builder. С помощью простых манипуляций вы можете изменять размеры и расположение этих компонентов. При этом вы все время в процессе проектирования видите результат – изображение формы и расположенных на ней компонентов. Вам не надо мучиться, многократно запуская приложение и выбирая наиболее удачные размеры окна и компонентов. Результаты проектирования вы видите, даже не компилируя программу, немедленно после выполнения какой-то операции с помощью мыши.

Но достоинства визуального программирования не сводятся к этому. Самое главное заключается в том, что во время проектирования формы и размещения на ней компонентов C++Builder автоматически формирует коды программы, включая в нее соответствующие фрагменты, описывающие данный компонент. А затем в соответствующих диалоговых окнах пользователь может изменить заданные по умолчанию значения каких-то свойств этих компонентов и, при необходимости, написать обработчики каких-то событий. То есть проектирование сводится, фактически, к размещению компонентов на форме, заданию некоторых их свойств и написанию, при необходимости, обработчиков событий.

Компоненты могут быть *визуальные,* видимые при работе приложения, и *невизуальные,* выполняющие те или иные служебные функции. Визуальные компоненты сразу видны на экране в процессе проектирования в таком же виде, в каком их увидит пользователь во время выполнения приложения. Это позволяет очень легко выбрать место их расположения и их дизайн – форму, размер, оформление, текст, цвет и т.д. Невизуальные компоненты видны на форме в процессе проектирования в виде пиктограмм, но пользователю во время выполнения они не видны, хотя и выполняют для него за кадром весьма полезную работу.

В библиотеки визуальных компонентов C++Builder включено множество типов компонентов, и их номенклатура очень быстро расширяется от версии к версии. Имеющегося уже сейчас вполне достаточно, чтобы построить практически любое самое замысловатое приложение, не прибегая к созданию новых компонентов. При этом даже неопытный программист, делающий свои первые шаги на этом поприще, может создавать приложения, которые выглядят совершенно профессионально.

## Раздел 2:
## ОСНОВНЫЕ НАПРАВЛЕНИЯ В ПРОГРАММИРОВАНИИ

Лекция проводится в интерактивной форме с текущим контролем (2 час.)

К настоящему времени в программировании сформировалось несколько направлений:
- процедурное программирование;
- модульное программирование;
- объектно-ориентированное программирование (ООП).

### 2.1. Процедурное программирование

В процедурном программировании основное внимание уделяется алгоритму, т.е. некоторой заданной последовательности действий, выполнение которых приводит к получению результата вычислений. Языки программирования, которые поддерживают эту модель, называются процедурными. Главное внимание в них уделяется построению процедур (подпрограмм) и, как следствие, решению следующих вопросов:
- передача аргументов в процедуры;
- получение вычислительных значений из процедур;
- внутренняя организация процедур.

Первым процедурным языком был Фортран. Далее появилось целое поколение языков указанного типа: Алгол 60, Алгол 68, Паскаль, Си и др.

### 2.2. Модульное программирование

В модульном программировании основные акценты переносятся на построение модулей. При этом необходимо определить модули, которые будут использоваться, и разделить программу на модули так, чтобы её данные были скрыты в этих модулях. В действительности, указанная модель переносит основные акценты на организацию данных (а не на алгоритм, по которому обрабатываются эти данные).

***Опр. 1.*** Модулем называется множество взаимосвязанных процедур (подпрограмм) вместе с данными, которые эти процедуры обрабатывают.

Основной целью данного направления является скрытие данных в модулях. Предположим, надо разработать очень большую программу. Такой разработкой будет заниматься коллектив программистов, в котором каждый программист отвечает за определённую часть общей программы. Если коллектив использует процедурное направление в программировании, то надо решить некоторые проблемы:

- договориться об используемых именах в программе для глобальных переменных, поскольку использование одного имени для разных переменных приводит к ошибке;
- договориться об организации общих данных и способах доступа к этим данным.

Для большой программы указанные проблемы могут оказаться очень сложными. Гораздо проще поручить конкретному программисту некоторую самостоятельную часть программы. В этом случае он будет отвечать за конструирование всех необходимых процедур и данных для этих процедур. Если запретить доступ к данным из пределов модуля, то будет предотвращено их случайное изменение, а значит, и нарушение работы программ. Теперь вместо решения перечисленных выше проблем надо только продумать интерфейс (взаимодействие) сконструированных модулей в разрабатываемой общей программе. Доступ к модулю будет осуществляться только через интерфейс, что исключит случайное изменение данных, и как следствие, ошибки в программе.

Непосредственная поддержка модульного программирования воплощена в языке Модула 2.

## 2.3. Концепция типов

Предположим, мы определили абстрактный тип данных, например, “графический объект”. Реальным графическим объектом могут быть такие фигуры на экране дисплея, как окружность, квадрат и т.п. Рассмотрим следующее представление нашей абстракции:

```
struct Точка {координата x, координата y};
struct Графический объект
     {
          скрытые_(локальные)_компоненты:
          Точка center;
          Вид_графического_объекта z;
          Не_скрытые_(глобальные)_компоненты:
          процедура_Нарисовать;
          процедура_Переместить;
          процедура_Повернуть;
          другие процедуры
     };
```

Здесь мы определили новый тип с именем Точка и новый тип с именем Графический_объект. Тип Точка фактически включает горизонтальную и вертикальную координаты воображаемой на экране дисплея точки. Тип Графический_объект описывает как данные, так и процедуры, манипулирующие этими данными. Данными являются:

```
переменная center типа Точка;
переменная z типа Вид_графического_объекта.
```

Предположим, что вид графического объекта может быть либо окружность, либо квадрат. В результате z может принимать одно из этих Вид_графического_объекта двух значений. Если z= окружность, то определённые нами процедуры позволяют рисовать на экране дисплея, перемещать и вращать окружность (будем считать, что радиус окружности является константой и задан заранее). Если z= квадрат, то подобные действия можно выполнять с квадратом. А что делать, если мы хотим определить новый Вид_графического_объекта, например, треугольник? При процедурном подходе надо менять все три процедуры, процедура_Нарисовать станет такой:

```
процедура_Нарисовать
     {
        если Вид_графического_объекта это окружность,
        то нарисовать окружность,
        в противном случае,
        если Вид_графического_объекта это квадрат
        то нарисовать квадрат,
        в противном случае,
```

```
        если Вид_графического_объекта это треугольник,
        то нарисовать треугольник;
    }
```

**Недостатки:**

- процедуры должны “знать” о всех графических объектах, которые программист захочет нарисовать. На практике это невозможно;
- если вы хотите рассмотреть новый графический объект, нужно менять многие ранее разработанные процедуры, что ведёт к новым ошибкам в программах;
- новый тип данных становится фактически новым типом данных конкретного программиста и теряет универсальность (развиваемость) его использования.

ОО направление в программировании разрешает эту проблему.

## 2.4. Объектно-ориентированное программирование

В ООП используются следующие базовые правила:

- определение классов, которые будут использоваться;
- определение всех необходимых операций для каждого класса;
- обеспечение расширяемости (открытости) классов с использованием принципа наследования.

Мы можем определить базовые (неизменяемые) свойства любого графического объекта. Такой объект имеет цвет и координаты, он может быть нарисован и перемещён и т.п. С другой стороны, каждый конкретный графический объект обладает и присущими только ему специфическими свойствами. Например, окружность имеет радиус и может быть нарисована только с помощью процедуры, которая рисует окружность.

***Опр. 2.*** Язык программирования, который позволяет выделить базовые свойства и потом дополнить специфические, является языком ООП.

Это реализуется через специальный механизм, называемый наследованием. Основная идея такого механизма заключается в следующем. Сначала определяются базовые свойства нового типа данных, например:

```
struct Графический_объект
     {
          скрытые_(локальные)_компоненты:
          Точка center;
          не_скрытые_(глобальные)_компоненты:
          модифицируемая_процедура_Нарисовать;
          процедура_Переместить;
          модифицируемая_процедура_Повернуть;
          другие процедуры
     };
```

***Опр. 3.*** Модифицируемая процедура – это такая процедура, которая в будущем может быть изменена (в языке C++ для этих целей используется ключевое слово **virtual**). В действительности **virtual** говорит о том, что соответствующая процедура может быть изменена в будущем (а может быть и нет). После этого мы можем определить базовый код для модифицируемой процедуры, например:

```
Модифицируемая_процедура_Нарисовать
     (массив_с_координатами_точек, размер_массива)
     {
         //рисует на экране все точки, заданные в массиве
          массив_с_координатами_точек
     }
```

Для того, чтобы определить специфический графический объект, мы должны описать этот специфический объект и добавить в него только новые процедуры, решающие новые, специфические для этого объекта, задачи. Например:

```
struct Окружность наследуется из класса Графический_объект
```

```
{
     скрытые_(локальные)_компоненты:
     Целое radius
     не_скрытые_(глобальные)_компоненты:
     модифицируемая_процедура_Нарисовать;
     модифицируемая_процедура_Повернуть;
     другие модифицируемые_процедуры
};
```

Новый класс Окружность наследуется из старого класса Графический_объект. В соответствии с правилами, принятыми в ООП, класс окружность может использовать данные и процедуры из базового класса (в нашем примере из класса Графический_объект), добавлять новые данные и процедуры (см., например, данное radius), а также (и это очень важно) изменять процедуры, имеющие тип virtual (модифицируемые процедуры).

## Раздел 3:
## КЛАССЫ

### 3.1. Объявление класса

Класс – это тип данных, определяемый пользователем. То, что в C++Builder имеется множество предопределенных классов, не противоречит этому определению – ведь разработчики C++Builder тоже пользователи C++. Понятия класса, структуры и объединения в C++ довольно близки друг к другу. Поэтому почти все, что будет далее говорится о классах, применимо также к структурам и объединениям.

Класс должен быть объявлен до того, как будет объявлена хотя бы одна переменная этого класса. Т.е. класс не может объявляться внутри объявления переменной.

Синтаксис объявления класса следующий:

```
class <имя класса>  :  <список классов – родителей>
{
     public:    //доступно всем
     <данные, методы, свойства, события>
     __published
     <данные, свойства>
     protected:    //доступно только потомкам
     <данные, методы, свойства, события>
     private    //доступно только в классе
     <данные, методы, свойства, события>
} <список переменных>
```

Например:

```
class MyClass  :  public Class1, Class2
{
     public:
          MyClass(int=ё 0);
          void SetA(int);
     private:
          int FA;
          double B, C;
     protected:
          int F(int);
};
```

Имя класса может быть любым допустимым идентификатором. Идентификаторы классов, наследующих классам библиотеки компонентов C++Builder, принято начинать с символа «Т».

Класс может наследовать поля (они называются данные-элементы), методы (они называются функции-элементы), свойства, события от других классов – своих предков, может отменять какие-то из этих элементов класса или вводить новые. Если предусматриваются такие классы-предки, то в объявлении класса после его имени ставится двоеточие, и затем дается список родителей. В приведенном выше примере предусмотрено множественное наследование классам **Class1** и **Class2.** Если среди классов-предков встречаются классы библиотеки компонентов C++Builder или классы, наследующие им, то множественное наследование запрещено.

Если объявляемый класс не имеет предшественников, то список классов-родителей вместе с предшествующим двоеточием опускается. Например:

```
class MyClass1
{
.....
};
```

Доступ к объявляемым элементам класса определяется тем, в каком разделе они объявлены. Раздел **public** (открытый) предназначен для объявлений, которые доступны для внешнего использования. Это открытый интерфейс класса. Раздел **published** (публикуемый) содержит открытые свойства, которые появляются в процессе проектирования на странице свойств Инспектора Объектов и которые, следовательно, пользователь может устанавливать в процессе проектирования. Раздел **private** (закрытый) содержит объявления полей и функций, используемых только внутри данного класса. Раздел **protected** (защищенный) содержит объявления, доступные только для потомков объявляемого класса. Как и в случае закрытых элементов, можно скрыть детали реализации защищенных элементов от конечного пользователя. Однако, в отличие от закрытых, защищенные элементы остаются доступны для программистов, которые захотят производить от этого класса производные классы, причем не требуется, чтобы производные классы объявлялись в этом же модуле.

В приведенном выше примере через объект данного класса можно получить доступ только к функциям **MyC1ass, SetA** и **GetA.** Поля **FA**, **B**, **C** и функция **F** – закрытые элементы. Это вспомогательные данные и функция, которые используют в своей работе открытые функции. Открытая функция **MyClass** с именем, совпадающим с именем класса, это так называемый *конструктор класса,* который должен инициализировать данные в момент создания объекта класса. Присутствие конструктора в объявлении класса не обязательно. При отсутствии конструктора пользователь должен сам позаботиться о задании начальных значений данным – элементам класса.

Перед именами классов-родителей в объявлении класса также может указываться спецификатор доступа (в примере **public**). Смысл этого спецификатора тот же, что и для элементов класса: при наследовании **public** (открытом наследовании) можно обращаться через объект данного класса к методам и свойствам классов-предков, при наследовании **private** подобное обращение невозможно.

По умолчанию в классах (в отличие от структур) предполагается спецификатор **private.** Поэтому можно включать в объявление класса данные и функции, не указывая спецификатора доступа. Все, что включено в описание до первого спецификатора доступа, считается защищенным. Аналогично, если не указан спецификатор перед списком классов-родителей, предполагается защищенное наследование.

Объявления данных-элементов (полей) выглядят так же, как объявления переменных или объявления полей в структурах:

```
<тип>  <имена  полей>;
```

В объявлении класса поля запрещается инициализировать. Для инициализации данных служат конструкторы, о которых уже упоминалось.

Объявления функций-элементов в простейшем случае не отличаются от обычных объявлений функций.

После того, как объявлен класс, можно создавать объекты этого класса. Если ваш класс не наследует классам библиотеки компонентов C++Builder, то объект класса создается как любая переменная другого типа простым объявлением. Например, оператор

```
MyC1ass  MC,  MC10[10],  *Pmc;
```

создает объект **MC** объявленного выше класса **MyC1ass,** массив **MC10** из десяти объектов данного класса и указатель **Pmc** на объект этого класса.

В момент создания объекта класса, имеющего конструктор, можно инициализировать его данные, перечисляя в скобках после имени объекта значения данных. Например, оператор

```
MyClass  MC(3);
```

не только создает объект **MC,** но и задает его полю **FA** значение 3. Если этого не сделать, то в момент создания объекта поле получит значение по умолчанию, указанное в содержащемся в объявлении класса прототипе конструктора.

Создание переменных, использующих класс, можно совместить с объявлением самого класса, размещая их список между закрывающей класс фигурной скобкой и завершающей точкой с запятой. Например:

```
class MyClass  :  public  Class1,  Class2
{
.....
}  MC,  MC10[10],  *Pmc;
```

Если создается динамически размещаемый объект класса, то это делается операцией **new**. Например:

```
MyClass  *PMC  =  new MyClass;
```

или

```
MyClass  *PMC1  =  new MyClass(3);
```

Эти операторы создают где-то в динамически распределяемой области памяти сами объекты и создают указатели на них – переменные **PMC** и **PMC1.**

Создание объектов класса простым объявлением переменных возможно только в случае, если среди предков вашего класса нет классов библиотеки компонентов C++Builder. Если же такие предки есть, то создание указателя на объект этого класса возможно только операцией **new**. Например, если класс объявлен так:

```
class  MyC1ass2  :  public  TObject
{
...........
};
```

то создание указателя на объект этого класса может осуществляться оператором

```
MyC1ass2  *P2  =  new MyC1ass2;
```

### 3.2. Фукции-элементы, дружественные функции, константные функции

Поля данных, исходя из принципа инкапсуляции, всегда должны быть защищены от несанкционированного доступа. Доступ к ним, как правило, должен осуществляться только через функции, включающие методы чтения и записи полей. В этих функциях должна осуществляться проверка данных, чтобы не записать случайно в поля неверные данные или чтобы не допустить их неверной трактовки.

Поэтому данные всегда целесообразно объявлять в разделе **private** – закрытом разделе класса. В редких случаях их можно помещать в **protected** – защищенном разделе класса, чтобы возможные потомки данного класса имели к ним доступ.

**Хороший стиль программирования**

Как правило, делайте данные-элементы класса защищенными, снабжая их при необходимости открытыми функциями чтения и записи Функции записи позволят вам проверять записываемые данные и обеспечивать тем самым непротиворечивость данных. А функции чтения позволят вам не переписывать программу, даже если вы решили изменить что-то в типе, способах хранения и размещения данных в классе

Приведем пример. Пусть класс имеет следующее объявление:

```
class  MyClass
{
     public:
          void SetA(int);    // функция записи
```

```
        int GetA(void);     // функция чтения
    private:
        int FA;
        double B, C;
};
```

Реализация функций записи и чтения может иметь вид:

```
void  MyClass  ::  SetA(int Value)
{
    if (...)    // проверка корректности данных
        FA = Value;
}
int MyClass  ::  GetA(void)  {return  FA;}
```

В данном случае функция чтения просто возвращает значение поля, но в более сложных классах может потребоваться какая-то предварительная обработка данных. Обратите внимание, что все описания функций-элементов содержат ссылку на класс с помощью операции разрешения области действия (**::**).

В приведенном примере объявление класса содержит только прототипы функций, а их реализация вынесена из описания класса. Для простых функций реализация может быть размещена непосредственно в объявлении класса. Например:

```
class  MyClass
{
    public:
        MyClass(int = 0);
        void SetA(int Value); {FA = Value;}; // функция
                                               записи
        int GetA(void); {return  FA;}; // функция
                                          чтения
    private:
        int FA;
        double B, C;
};
```

Функции, описание которых содержится непосредственно в объявлении класса, в действительности являются встраиваемыми функциями **inline**.

Введение описания функций в объявление класса – это плохой стиль программирования: следует избегать смешения открытого интерфейса класса, содержащегося в его объявлении, и реализации класса. Если уж вы хотите реализовать встраиваемые функции, то лучше поместить в объявлении класса их прототип со спецификатором inline**:**

```
inline  void  SetA (int);    // функция записи
```

и отдельно дать реализацию функции. При этом в реализации спецификатор **inline** не указывается.

Объявления классов следует размещать в заголовочном файле модуля, а реализацию функций-элементов в отдельном файле реализации. При этом в объявлении класса должны содержаться только прототипы функций. Это следует из принципа скрытия информации – одного из основных в объектно-ориентированном программировании. Такая организация программы обеспечивает независимость всех модулей, использующих заголовочный файл с объявлением класса, от каких-то изменений в реализации функций-элементов класса.

Функции-элементы класса имеют доступ к любым другим функциям-элементам и к любым данным-элементам, как открытым, так и закрытым. Клиенты класса (какие-то внешние функции, работающие с объектами данного класса) имеют доступ только к открытым функциям-элементам и данным-элементам. Но в некоторых случаях желательно обеспечить доступ к закрытым элементам для функций, не являющихся элементами данного класса. Это можно сделать, объявив соответствующую функцию как *друга класса с* помощью спецификации **friend**. Например, если в объявление класса включить оператор

```
friend  void  IncFA(MyClass  *);
```

то функция **IncFA**, не являясь элементом данного класса, получает доступ к его закрытым элементам. Например, функция **IncFA** может быть описана где-то в программе следующим образом:

```
void IncFA(MyClass  *p)  {P->FA++;}
```

Дружественными могут быть не только функции, но и целые классы. Например, вы можете поместить в объявление своего класса оператор

```
friend  Class1;
```

и все функции-элементы класса **Class1** получат доступ к закрытым элементам вашего класса.

Иногда программист может захотеть создать объект вашего класса как константный с помощью спецификатора **const**. Например:

```
const  Class1  MC1 (3);
```

Если при этом ваш класс содержит не только функции чтения, но и записи данных, то реакция на такой оператор, введенный пользователем, зависит от версии и настройки компилятора. Компилятор может выдать сообщение об ошибке и отказаться от компиляции, а может просто выдать предупреждение и проигнорировать спецификатор пользователя **const**. Если же ваш класс содержит только функции чтения, то все должно бы быть нормально. Но компилятор подойдет к этому чисто формально и все равно выдаст предупреждение, а может и отказаться компилировать программу.

Чтобы избежать этого, можно объявить функции чтения как константные. Для этого и в прототипе, и в реализации после закрывающей список параметров круглой скобки надо написать спецификатор **const**. Например, вы можете включить в объявление класса оператор

```
int GetA(void)  const;
```

а реализацию этой функции оформить как:

```
int MyClass  ::  CetA(viod) const  {return  FA;}
```

Тогда неприятные замечания компилятора о константных объектах исчезнут.

**Хороший стиль программирования**

Если предполагается, что объект вашего класса может быть объявлен константным, снабжайте все функции-элементы класса, предназначенные для чтения данных, спецификаторами **const**.

# Раздел 4:<br>БАЗОВЫЕ ПРИНЦИПЫ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ

В основу ООП положены следующие принципы: абстрагирование, ограничение доступа, модульность, иерархичность, типизация, параллелизм, устойчивость.

## 4.1. Абстрагирование

***<u>Опр. 1.</u>*** Абстрагирование – процесс выделения абстракций в предметной области задачи.

***<u>Опр. 2.</u>*** Абстракция – совокупность существенных характеристик объекта, которые отличают его от всех других видов объектов и, таким образом, чётко определяют особенности данного объекта с точки зрения дальнейшего рассмотрения и анализа.

В соответствии с определением применяемая абстракция реального предмета существенно зависит от решаемой задачи: в одном случае нас будет интересовать форма предмета, в другом – вес, в третьем – материалы. Современный уровень абстракции предполагает объединение всех свойств абстракции (как касающихся состояния анализируемого объекта, так и определяющих его поведение) в единую программную единицу – некий абстрактный тип (класс).

4.2. Ограничение доступа

***Опр. 3.*** Ограничение доступа – сокрытие отдельных элементов реализации абстракции, незатрагивающих существенных характеристик её целого.

Необходимость ограничения доступа предполагает разграничение двух частей в описании абстракции:

- интерфейс – совокупность доступных извне элементов реализации абстракции (основные характеристики состояния и поведения);
- реализация – совокупность недоступных извне элементов абстракции (внутренняя организация абстракции и механизмы реализации её поведения).

Ограничение доступа в ООП позволяет разработчику:

- выполнять конструирование системы поэтапно, не отвлекаясь на особенности реализации используемых абстракций;
- легко модифицировать реализацию отдельных объектов, что в правильно организованной системе не потребует изменения других объектов.

***Опр. 4.*** Инкапсуляция – это сочетание объединения всех свойств предмета (составляющих его состояния и поведения) в единую абстракцию и ограничения доступа к реализации этих свойств.

4.3. Модульность

***Опр. 5.*** Модульность – принцип разработки программной системы, предполагающий реализацию её в виде отдельных частей (модулей).

При выполнении декомпозиции системы на модули желательно объединять логически связанные части, по возможности обеспечивая сокращение количества внешних связей между модулями. Принцип унаследован от модульного программирования, следование ему упрощает проектирование и отладку программы.

4.4. Иерархия

***Опр. 6.*** Иерархия – ранжированная или упорядоченная система абстракций.

В ООП используются два вида иерархии:

- иерархия “*целое/часть*” – показывает, что некоторые абстракции включены в рассматриваемую абстракцию как её части, например, лампа состоит из цоколя, нити накаливания и колбы. Этот вариант иерархии используется в процессе разбиения системы на разных этапах проектирования (на логическом уровне – при декомпозиции предметной области на объекты, на физическом уровне – при декомпозиции системы на модули и выделении отдельных процессов в мультипроцессной системе);
- Иерархия “общее/частное” – показывает, что некоторая абстракция является частным случаем другой абстракции, например, “обеденный стол – конкретный вид стола”, а “столы – конкретный вид мебели”. Используется при разработке структуры классов, когда сложные классы строятся на базе более простых путём добавления к ним новых характеристик.

4.5. Типизация

***Опр. 7.*** Типизация – ограничение, накладываемое на свойства объектов и препятствующее взаимозаменяемости абстракций различных типов (или сильно сужающее возможность такой замены).

В языках с жёсткой типизацией для каждого программного объекта (переменной, подпрограммы) объявляется тип, который определяет множество операций над соответствую-

щим программным объектом. Языки программирования на основе С имеют среднюю степень типизации.

Использование принципа типизации обеспечивает:

- ранее обнаружение ошибок, связанных с недопустимыми операциями над программными объектами (ошибки обнаруживаются на этапе компиляции программы при проверке допустимости выполнения данной операции над программным объектом);
- упрощение документирования;
- возможность генерации более эффективного кода.

### 4.6. Параллелизм

***Опр. 8*** Параллелизм – свойство нескольких абстракций одновременно находиться в активном состоянии, т.е. выполнять некоторые операции.

Существует целый ряд задач, решение которых требует одновременного выполнения некоторых последовательностей действий. К таким задачам, например, относятся задачи автоматического управления несколькими процессами.

### 4.7. Устойчивость

***Опр. 9.*** Устойчивость – свойство абстракции существовать во времени независимо от процесса, породившего данный программный объект, и/или в пространстве, перемещаясь из адресного пространства, в котором он был создан.

Различают:

- временные объекты, хранящие промежуточные результаты некоторых действий, например, вычислений;
- локальные объекты, существующие внутри подпрограмм, время жизни которых исчисляется от вызова подпрограммы до её завершения;
- глобальные объекты, существующие пока программа загружена в память;
- сохраняемые объекты, данные которых хранятся в файлах внешней памяти между сеансами работы программы.

### 4.8. Иерархия классов

При построении объектно-ориентированной программы рассматривается некоторая иерархия классов. Часто, например, в ObjectWindows, вы можете рассматривать некоторую базовую иерархию классов или библиотеку. Библиотеки создаются опытными программистами. Предположим, имеется некоторая иерархия библиотечных классов. В этом случае можно:

- определить объект для заданного класса;
- построить новый класс, наследуя его из существующего класса;
- изменить поведение нового класса (изменить его функции или добавить новые функции).

Когда вы строите новый класс, наследуя его из существующего класса, можно:

- добавить в новый класс новые компоненты-данные;
- добавить в новый класс новые компоненты-функции;
- заменить в новом классе наследуемые из старого класса компоненты-функции.

При использовании принципов множественного наследования применяется следующая модель: ребёнок (наследуемый или новый класс) имеет много родителей (много старых классов, из которых образуется новый класс). В этом случае новый класс будет наследовать поведение многих старых классов и далее можно определить объект, имеющий комбинированное (смешанное) поведение.

Определим более точно, что понимается под ООП.

***Опр. 10.*** ООП – это метод построения программ в виде множества взаимодействующих объектов, структура и поведение которых описаны соответствующими классами, и все эти классы являются компонентами иерархии классов, выражающей отношения наследования.

Это определение можно разделить на несколько частей. Базовым абстрактным типом объектно-ориентированной программы является класс. Совокупность классов можно описать в виде такой иерархической структуры, что:

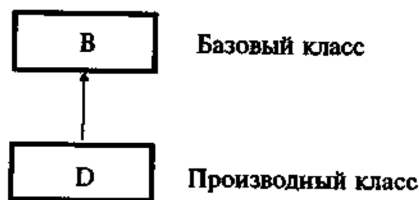

Рис. 4.1. Отношение наследования

- если класс **D** наследует структуру и поведение класса **B** (рис. 4.1), то класс **B** называется **базовым**, а класс **D** – **производным**. В иерархической структуре такое отношение наследования изображается стрелкой, идущей от класса D к классу B. В литературе можно встретить разные названия классов B и D. Так, класс B называют базовым (base), суперклассом (superclass), родителем (parent), предшественником (predecessor). Класс D называют: производным (derive), подклассом (subclass), ребёнком (child), потомком (descendant);

- она выражает иерархичное упорядочение, которое сегодня является одним из наиболее эффективных средств построения сложных программ.

Объектно-ориентированная программа состоит из объектов, которые:

- описываются в виде $CLASS_i$ $OBJECT_j$, где $CLASS_i$ соответствует одной из вершин иерархической структуры (рис. 4.2), $OBJECT_j$ – это действительный (не абстрактный) компонент программы, реально существующий во времени и в пространстве;

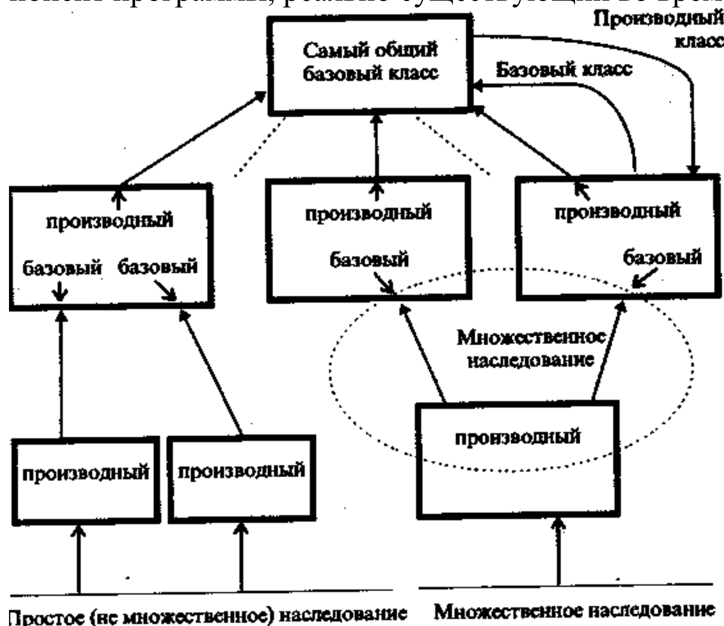

Рис. 4.2. Пример иерархической структуры

- взаимодействуют друг с другом через сообщения.

При разработке объектно-ориентированных программ часто используются библиотеки классов. Библиотека может рассматриваться как заданная базовая иерархическая структура (граф). Для конкретной программы, использующей библиотеку, мы выбираем некоторую подструктуру (подграф) этой библиотеки и можем расширять эту подструктуру, используя принципы наследования. Окончательно полученная подструктура и является иерархией классов для нашей программы.

***Опр. 11.*** Язык программирования является объектно-ориентированным, если:

- он поддерживает абстрактные типы данных, которыми являются объекты с определёнными интерфейсами (средствами взаимодействия с внешней средой) и скрытым внутренним состоянием;

- объекты имеют связанные с ними типы (т.е. классы);

- поддерживаются механизмы наследования классов.

Язык не является объектно-ориентированным, если он не поддерживает наследование и полиморфизм;

При разработке объектно-ориентированной программы различают этапы анализа и проектирования.

***Опр. 12.*** Объектно-ориентированный анализ – это метод проверки требований, предъявляемых к программе с позиций классов и объектов.

***Опр. 13.*** Объектно-ориентированное проектирование – это метод, основанный на объектно-ориентированной декомпозиции, отражающей различные уровни (модели) представления программы.

Таками уровнями (моделями) являются:

- логический (структуры классов и объектов);
- физический, выражающий архитектуры модулей и процессов;
- статический;
- динамический.

## Раздел 5:
## БАЗОВЫЕ КОНСТРУКЦИИ ОБЪЕКТНО-ОРИЕНТИРОВАННЫХ ПРОГРАММ

Лекция проводится в интерактивной форме с разбором конкретных ситуаций (2 час.)

5.1. Объекты и классы

Базовыми строительными блоками объектно-ориентированной программы являются объекты и классы. На рис. 5.1 показаны взаимоотношения между объектами и классом. Когда мы объявляем класс, например,

```
class my_class {           };
```

мы вводим новый АТД. Когда мы описываем объекты, например:

```
my_class ob1,…, obn;
```

хотим выделить память, структура которой соответствует описанию класса, и, возможно, выполнить инициализацию, т.е. присвоить определённые значения компонентам-данным объекта. На рис. 5.1 показано, что компоненты-данные класса имеют неопределённые значения (это отмечено знаком “?”). Когда описывается объект, то можно присвоить значения компонентам-данным. Компонент-данное может быть как переменной, так и указателем. Во втором случае в процессе инициализации можно динамически выделить память и назначить адрес её начала соответствующему указателю (см. объект_n на рис. 5.1). Значения некоторых компонентов-данных объекта могут остаться неопределёнными (см. объект_n на рис. 5.1).

Следует различать **объявление класса** (class declaration) и **описание объектов** класса (object definition). При объявлении класса мы описываем его компоненты-данные и объявляем и описываем его компоненты-функции.

**Объявление** компонента-функции содержит описание имени функции, типов её параметров и типа возвращаемого значения. Описание функции задаёт имена параметров и тело (код) функции (т.е. совокупность и последовательность выполнения её инструкций).

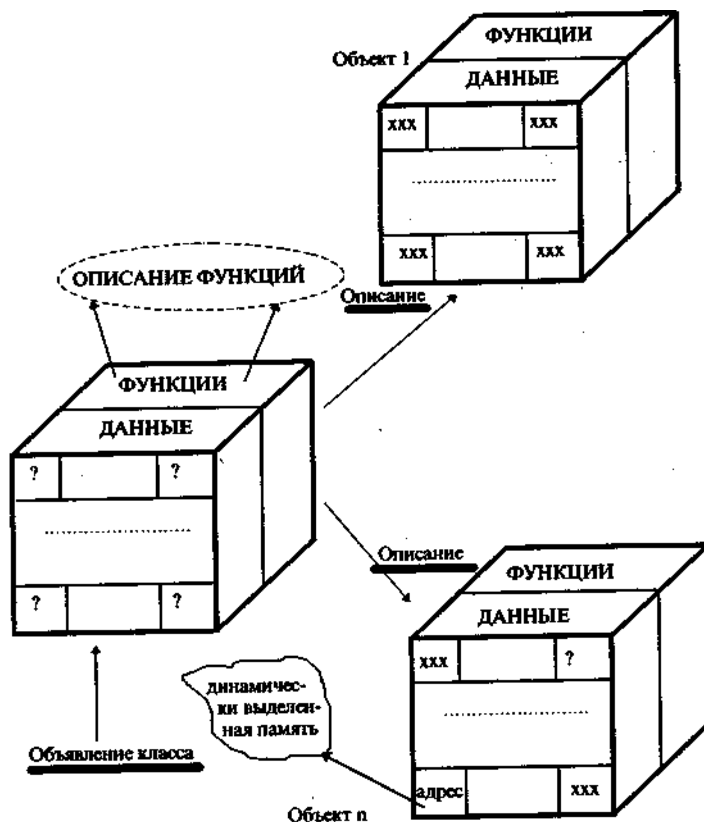

Рис. 5.1. Взаимоотношения между объектами и классами

Описание объекта задаёт его тип (т.е. имя класса) и, возможно, некоторые параметры, которые необходимы для инициализации.

При описании объекта мы даём указание на его построение на основании заданного типа (класса). При построении объекта вызывается специальная функция класса, называемая конструктором (constructor). Конструктор управляет выделением памяти для объекта, строит объект в памяти и, возможно, инициализирует его компоненты-данные. Наряду с конструктором существует другая функция класса, называемая деструктором (destructor). Деструктор разрушает объект, т.е. удаляет его из памяти.

## 5.2. Интерфейс и реализация объекта

Говоря об объекте, можно выделить две его характеристики: интерфейс и реализацию (рис. 5.2). Интерфейс показывает, как объект общается с внешней средой. Его рассматривают как окно в некотором чёрном ящике, через которое можно заглянуть внутрь и получить доступ к функциям и данным объекта.

Рис. 5.2. Пример объекта

Все данные делят на глобальные и локальные. Локальные данные являются недоступными (их не видно в окне). Изменение локальных данных и получение их значений можно осуществить только в функциях-комнонентах этого же объекта. С другой стороны, глобальные данные можно менять и получать их значение через окно (т.е. извне). Сообщение проходит через окно и активизирует некоторую глобальную функцию.

Основная идея класса как абстрактного типа заключается в разделении интерфейса и реализации. Говоря о разделении, подразумевается следующее.

Интерфейс показывает, как мы можем использовать класс. Мы хотим, чтобы это использование было эффективным. Однако нас совершенно не интересует, каким образом соответствующие функции реализованы внутри класса.

Реализация – это внутренняя особенность класса. Когда разрабатывается код внутренней функции, учитываются уже другие критерии. Одним из основных является критерий изоляции кода функции от воздействия на него извне. Здесь можно использовать локальные компоненты-данные класса, через которые осуществляется взаимодействие между различными функциями этого класса, и локальные компоненты-функции класса.

В любом случае интерфейс и реализация должны быть максимально независимы друг от друга. В первую очередь это понимается так. Изменение кода внутренних функций, возможное в процессе развития программных продуктов, не должно изменять соответствующий интерфейс.

## 5.3. Механизмы наследования

Принцип наследования позволяет:

- выразить общность интерфейса базовых и производных классов;
- осуществить развитие и совершенствование программ без изменения того, что уже сделано раньше.

Из рис. 5.3 видно, что в общем случае производный класс “больше”, чем его базовый, в связи с тем, что он может:

- использовать некоторые или все компоненты-данные базового класса. Почему иногда только некоторые? Потому, что здесь вводятся новые механизмы защиты данных базового класса: private – недоступен за пределами класса; protected – доступен в производном классе; public – доступен везде;

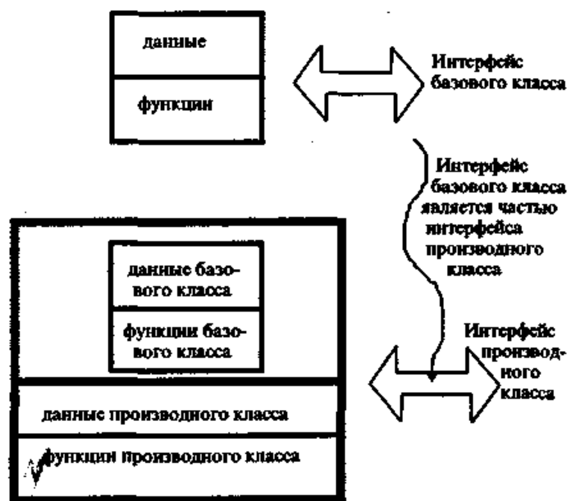

Рис. 5.3. Принцип наследования обеспечивает общность интерфейса базовых и производных классов

- использовать без изменений некоторые или все компоненты-функции базового класса;

- заменить старые версии функций базового класса новыми версиями тех же функций, определённых в производном классе. Обратим внимание на то, что здесь меняется реализация, но сохраняется интерфейс, а это очень важно;

- ввести новые данные в производный класс;

- ввести новые функции в производный класс.

Интерфейс базового класса является частью интерфейса производного класса (рис. 5.3), т.е. через окно производного класса мы видим всё, что доступно для наблюдения как в производном, так и в базовом классе.

Различают простое наследование и множественное наследование. В первом случае производный класс имеет один базовый класс (см. рис. 5.3), а во втором он может иметь несколько базовых классов (рис. 5.4). Перечисленные выше принципы распространяются и на множественное наследование. Поэтому производный класс на рис. 5.4 можно рассматривать как класс, приобретающий все полезное из его n базовых классов. Соответственно все интерфейсы n базовых классов являются частями интерфейса производного класса.

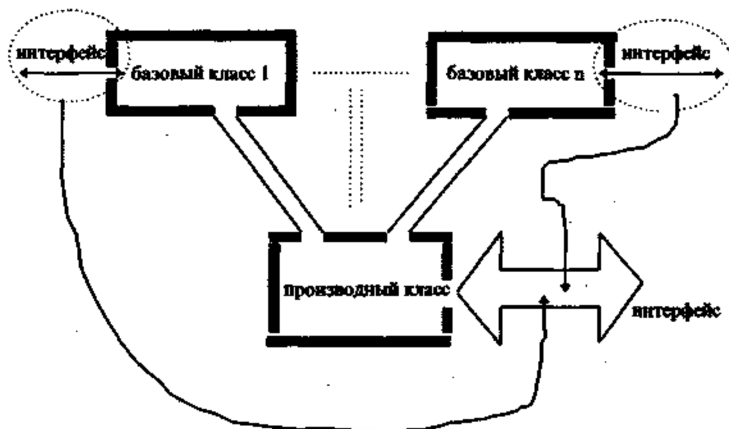

Рис. 5.4. Иллюстрация множественного наследования

5.4. Обработка исключений

При написании программ очень важным является выявление и устранение ошибок. Для этих целей в объектно-ориентированных программах используется специальный метод, называемый обработкой исключений, частным случаем которого является обработка ошибок. Основная идея этого метода поясняется на рис. 5.5. Объектно-ориентированную программу можно рассматривать как совокупность объектов, взаимодействующих через сообщения. Типами объектов являются библиотечные и определённые пользователем классы. Каждый класс имеет внешнее представление (интерфейс), и внутреннее представление, или реализацию. Когда объекту посылается сообщение, начинается обработка этого сообщения внутри объекта. Допустим, что сообщение передаваемое в объект является ошибочным, например, мы обращаемся к элементу внутреннего массива с недопустимо большим значением индекса. Возникают вопросы: кто может обнаружить и кто может исправить ошибку? Очевидно, что проверить индекс и определить ошибку в общем случае можно только там, где соответствующий массив построен, т.е. в классе. Однако исправить ошибку может только функция, посылающая сообщение объекту класса. Эти соображения и легли в основу метода обработки исключений, в соответствии с которым выполняются определённые действия (рис. 5.5):

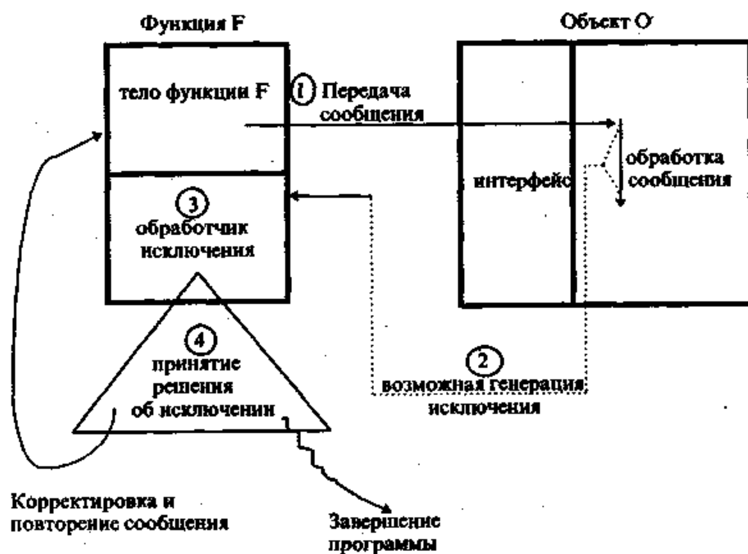

Рис. 5.5. Пояснение метода обработки исключений

- пусть функция F посылает сообщение объекту O;
- если объект O обнаруживает в сообщении ошибку, неоднозначность или то, что невозможно выполнить, он прерывает свои функции и генерирует исключение;
- функция F представляется в виде двух частей: собственно функции и обработчика исключения. Следствием генерации исключения в объекте является активизация обработчика исключения;
- обработчик исключения делает вывод о возможности продолжения программы (изменяет ошибочное сообщение) или о необходимости её завершения по ошибке.

Такой метод имеет ряд преимуществ:

- поиск ошибок осуществляется внутри класса, а их обработка – за пределами класса. Это поддерживает независимость интерфейса и реализации;
- осуществляется явное разделение (а не смешивание) кода собственно функции и кода для обработки ошибок.

# Раздел 6: ДАННЫЕ И ФУНКЦИИ КЛАССА

## 6.1. Данные-элементы, статические данные, константные данные

Теперь рассмотрим несколько подробнее данные-элементы. Обычно каждый объект класса имеет свою собственную копию всех данных-элементов класса. Но в определенных случаях во всех объектах класса должна фигурировать только одна копия некоторых данных. Например, это может быть счетчик числа созданных объектов класса.

Единственную копию данных полезно иметь и во многих иных случаях. Например, если в классе имеются некоторые константы, одинаковые для всех объектов класса, то нерационально хранить в каждом объекте собственные копии этих констант. Рациональнее иметь единственные экземпляры этих констант для всех объектов.

Для введения в класс подобных данных используются статические данные, которые содержат информацию “для всего класса”. Объявление статических элементов в классе начинается с ключевого слова **static**. Например:

```
statc  int  D;
```

Данные, общие для всех объектов класса, надо объявлять как статические данные-элементы. Это сократит затраты памяти и гарантирует единство данных во всех объектах.

Статические элементы могут быть открытыми, закрытыми или защищенными (**protected**). Доступ к открытым статическим элементам класса возможен посредством любого объекта класса или посредством имени класса с помощью бинарной операции разрешения области действия. Например:

```
MyClass::D  =  10;
```

Закрытые и защищенные статические элементы класса должны быть доступны открытым функциям-элементам этого класса или друзьям класса.

Статические элементы класса существуют даже тогда, когда не существует никаких объектов этого класса. В этом случае доступ к открытому статическому элементу обеспечивается так же, как указано выше: с помощью имени класса и бинарной операции разреше-

ния области действия. Для обеспечения доступа в отсутствие объектов к закрытому или защищенному элементу класса должна быть предусмотрена открытая статическая функция-элемент, которая должна вызываться с добавлением перед ее именем имени класса и бинарной операции разрешения области действия.

Начальные значения статических элементов (как открытых, так и закрытых) должны задаваться вне объявления класса. Для этого достаточно разместить где-то в файле, например, после объявления класса или среди реализаций функций-элементов (но не внутри их) оператор вида

```
int MyClass::D =  0;
```

**Хороший стиль программирования**

Статическим данным-элементам можно задать начальные значения один и только один раз в области действия файла. Если вы нигде не инициализировали статический элемент данных, будет выдано сообщение компилятора о неразрешенной внешней ссылке и программа не будет скомпилирована. Если вы дважды инициализируете статический элемент, будет выдано сообщение о дублировании инициализации и программа также не будет скомпилирована.

Приведем пример, демонстрирующий все сказанное относительно статических данных-элементов:

```
class MyClass
{
     public:
          static int D;
          static int GetD1(void);

     private:
          static int D1;

};
…..
int MyClass::GetD1(void) {return D1;}
int MyClass::D  =  0;
int MyClass::D1  =  1;
```

В этом примере имеются два статических элемента данных: открытый **D** и закрытый **D1.** Если пользователь должен иметь возможность получать значение закрытой статической переменной **D1,** то должна быть предусмотрена функция ее чтения, названная в примере **GetD1.** Она должна быть объявлена открытой (**public**) и статической (со спецификатором **static**). Статической может быть объявлена любая функция, работающая только со статическими данными.

После объявления класса в примере расположена реализация функции **GetD1.** В реализации не требуется указывать спецификатор **static**. Далее приведены предложения, инициирующие открытые и закрытые статические данные. На этом все, связанное с объявление и инициализацией статических данных завершается. В дальнейшем вы можете из любой внешней функции обращаться к ним с помощью операции разрешения области действия. Например:

```
i  =  MyClass: :D;
j  =  MyClass::GetD1();
```

Среди данных- элементов могут быть объявлены именованные константы. Например:

```
static  const  int MaxA =  10;
const  int  MinA;
```

Значения статических именованных констант могут задаваться в момент их объявления в классе, как показано в предыдущем примере. Инициализация нестатических констант – вопрос более сложный, связанный с построением конструкторов. Он рассматривается в следующем разделе.

6.2. Конструкторы и деструкторы

Остановимся теперь на конструкторах класса. Прежде всего отметим, что наличие конструктора в классе не обязательно. Но если конструктор отсутствует, то клиенты класса (внешние функции, использующие класс) должны сами заботиться об инициализации данных, т.е. о задании им некоторых начальных значений. Это не всегда возможно. Например, если класс имеет закрытые данные, предназначенные только для чтения, то для этих данных не предусматриваются открытые функции записи, И клиент не в состоянии присвоить данным какие-то начальные значения.

Конструктором класса называется открытая функция-элемент, которая вызывается в момент создания объекта класса и должна инициализировать данные указанными в вызове значениями или значениями по умолчанию. Конструктор имеет то же имя, что и сам класс.

Пример объявления и реализации конструктора:

```
class  MyClass
{
     public:
          MyClass(void);    // конструктор класса
…..
     private:
          int A;
…..
};
…..
MyClass::MyClass(void)  {  A  =  0;}
```

В этом примере объявлен конструктор **MyClass** без параметров, который при создании объекта задает начальное значение поля **A** равным 0. Обратите внимание на то, что в отличие от других функций в объявлении конструктора не указывается тип возвращаемого значения.

Простое задание в конструкторе значений данных в общем случае не гарантирует их целостность. Обычно нужна еще проверка допустимости данных. Например, если в классе есть функция записи **SetA**, осуществляющая такие проверки, то лучше обратиться к ней и при задании начального значения. В этом случае реализация конструктора может иметь вид:

```
MyC1ass::MyC1ass  (void)  {  SetA(0);  }
```

Создание объекта описанного класса **MyClass** в программе должно осуществляться или объявлением соответствующей переменной:

```
MyClass  MC;
```

или динамическим размещением переменной в памяти:

```
MyC1ass  *PMC  =  new  MyC1ass;
```

В момент выполнения каждого из этих операторов неявным образом вызывается конструктор, устанавливающий начальные значения данных.

Недостатком конструкторов показанного типа является то, что все начальные значения данных задаются в них конструктором. Вызывающая функция никак не может вмешаться в этот процесс и задать какое-то другое значение.

Другой крайностью являются конструкторы, в которых все начальные значения задаются как параметры. Например, прототип конструктора может иметь вид

```
MyC1ass (int);
```

а его реализация:

```
MyC1ass::MyC1ass(int  a}  (  SetA(a);  }
```

В этом случае поле **FA** инициализируется параметром, передаваемым в конструктор. Создание объекта подобного класса должно выполняться операторами

```
MyC1ass  MC (1)
```

или

```
MyC1ass  *PMC  =  new MyC1ass (1)
```

в которых подразумевается, что начальное значение поля **FA** должно быть равно 1.

Такой конструктор обычно тоже неудобен, поскольку в классе может быть много параметров и задавать значения их всех при создании объекта очень громоздко и чревато ошибками.

Чаще всего используются конструкторы с параметрами по умолчанию. В этом случае объявление конструктора может иметь вид:

```
MyClass (int = 0);
```

а его реализация:

```
MyClass: :MyClass (int a) { SetA(a); }
```

Объект такого класса можно создавать любым из приведенных ранее операторов создания объекта. Если при создании указывается аргумент, то его значение присваивается полю. Если аргумент не указывается, то присваивается значение по умолчанию (в нашем примере 0). Этот вариант конструктора наиболее гибкий. Поэтому он чаще всего используется при построении классов.

Как правило, в классе надо предусматривать конструктор с параметрами по умолчанию.

В объявлении класса могут быть определены не только поля переменных, но и некоторые именованные константы. Например:

```
const int MaxA;
```

Подобная константа может служить, в частности, предельно допустимым значением поля **FA**;

Если такая константа объявлена как статическая (см. предшествующий раздел), то в ее объявление в классе можно непосредственно включить инициализацию:

```
static const int MaxA = 10;
```

Но тогда это значение клиент при желании не сможет изменить. А задать значение такой константы в конструкторе невозможно, поскольку компилятор не разрешает присваивать значения константам. Выходом из положения является специальный синтаксис конструктора с *инициализатором элементов.* Инициализатор элементов записывается после заголовка конструктора в его реализации, предваряется двоеточием и содержит имена константных данных, после которых в скобках указываются их значения. Например, если в объявлении вашего класса **MyC1ass** имеются строки

```
const int MaxA;
const int MinA;
```

вводящие две константы – максимальное и минимальное значения переменной **A**, то реализацию конструктора такого класса с описанным ранее прототипом

```
MyC1ass (int = 0);
```

надо дополнить инициализатором элементов:

```
MyC1ass::MyC1ass (int a) : MaxA(10), MinA(1) { SetA (a); };
```

В данном случае инициализатор задает константе **MaxA** начальное значение 10, а константе **MinA** – значение 1.

Можно предоставить пользователю возможность изменять значения констант в момент создания объекта. В этом случае в конструкторе с умолчанием надо предусмотреть для констант соответствующие значения по умолчанию:

```
MyClass (int A = 0, int MaxA = 10, int MinA = 1};
```

или

```
MyC1ass (int = 0, int = 10, int = 1 );
```

(второй вариант менее удобен, так как не позволяет по прототипу функции понять, в какой последовательности должны задаваться параметры). Тогда реализацию конструктора можно оформить так:

```
MyC1ass::MyC1ass (int a, int i, int j ) : MaxA(i), MinA(j) {
SetA(a); }
```

Создание объектов такого типа может осуществляться, например, такими операторами:

```
MyClass MC;          //умолчание: A=0, MaxA=10, MinA=1;
MyClass MC(20);      //задано:    A=20, MaxA=10, MinA=1;
MyClass MC(20,15 );  //задано:    A=20, MaxA=15, MinA=1;
MyClass MC(20,15,2); //задано:    A=20, MaxA=15, MinA=2;
```

Теперь остановимся на деструкторах. Это специальные функции-элементы, срабатывающие при уничтожении динамически размещенного объекта класса и освобождающие занимаемую им память. Имя деструктора совпадает с именем класса, но перед ним записывается символ тильда (~). Как и для конструктора, в деструкторе не указывается возвращаемый тип. Например:

```
class  MyClass
{
     public:
          ~MyClass();    // деструктор класса
…..
};
```

Деструкторы необходимы, если конструктор или какие-то функции-элементы класса динамически распределяют память, создавая в ней какие-то объекты. Тогда деструктор должен эти объекты удалять. В остальных случаях можно обычно обойтись без деструктора.

Если деструктор явным образом в классе не объявлен, компилятор сам генерирует необходимые коды освобождения памяти.

## Раздел 7: НАСЛЕДОВАНИЕ И ПОЛИМОРФИЗМ. ВИРТУАЛЬНЫЕ ФУНКЦИИ, АБСТРАКТНЫЕ КЛАССЫ

### 7.1. Наследование и полиморфизм

При описании нового класса, производного от какого-то одного или нескольких базовых классов, можно добавлять новые функции-элементы и данные-элементы, сохраняя при этом все элементы родителей, а можно родительские элементы переопределить или перегрузить. В производном классе доступны открытые и защищенные элементы базового класса (прямого или косвенного предшественника). Закрытые элементы базового класса в производном классе недоступны.

Производный класс может наследоваться от базового класса как **public**, **protected** или **private**. Защищенное и закрытое наследования встречаются редко и каждое из них нужно использовать с большой осторожностью.

При порождении класса как **public** открытые элементы базового класса становятся открытыми элементами производного класса, а защищенные элементы базового класса становятся защищенными элементами производного класса. Закрытые элементы базового класса никогда не бывают доступны для производного класса.

При защищенном наследовании открытые и защищенные элементы базового класса становятся защищенными элементами производного класса. При закрытом наследовании открытые и защищенные элементы базового класса становятся закрытыми элементами производного класса. При закрытом и защищенном наследованиях не справедливо отношение, что объект производного класса является объектом базового класса.

В целом доступ к элементам базового класса из производного класса можно представить следующей таблицей.

Табл. 7.1

***Таблица доступа к элементам базового класса из производного***

| | **Тип наследования** | | |
|---|---|---|---|
| Спецификатор доступа к элементам в базовом классе | **public** открытое наследование | **protected** защищенное наследование | **private** закрытое наследование |
| **public** | **public** в производном классе Может быть доступен непосредственно любым нестатиче- | **protected** в производном классе Может быть доступен непосредственно любым нестати- | **private** в производном классе Может быть доступен непосредственно любым нестатиче- |

| | ским функциям-элементам, дружественным функциям и функциям, не являющимся элементами. | ческим функциям-элементам и дружественным функциям. | ским функциям-элементам и дружественным функциям. |
|---|---|---|---|
| **protected** | **protected** в производном классе<br>Может быть доступен непосредственно любым нестатическим функциям-элементам и дружественным функциям. | **protected** в производном классе<br>Может быть доступен непосредственно любым нестатическим функциям-элементам и дружественным функциям. | **protected** в производном классе<br>Может быть доступен непосредственно любым нестатическим функциям-элементам и дружественным функциям. |
| **private** | невидим в производном классе<br>Может быть доступен нестатическим функциям-элементам и дружественным функциям через открытые или защищённые функции-элементы базового класса. | невидим в производном классе<br>Может быть доступен нестатическим функциям-элементам и дружественным функциям через открытые или защищённые функции-элементы базового класса. | невидим в производном классе<br>Может быть доступен нестатическим функциям-элементам и дружественным функциям через открытые или защищённые функции-элементы базового класса. |

Если в классе-наследнике переопределить функцию-элемент (ввести новую функцию с тем же именем), то для объектов этого класса новая функция отменит родительскую. Если обращаться к объекту этого класса, то вызываться будет новая функция. Если все-таки нужно вызвать именно функцию базового класса, надо использовать операцию разрешения области действия.

Пусть, например, вы создали класс форм **Shape**:

```
class  Shape
{
     public:
          void Draw(void);
…..
};
```

и наследующий ему класс кругов **Circl**:

```
class  Circl  :  public  Shape
{
     public:
          void  Draw  (void);
};
```

в каждом из классов объявили метод рисования **Draw**, а затем в программе выполнили операторы

```
Shape *PQ1  =  new  Shape;
PQ1->Draw();     // вызов Draw класса  Shape
Circl *PQ2  =  new  Circl;
PQ2->Draw();     // вызов Draw класса  Circl
PQ2->Shape::Draw();     // вызов Draw класса  Shape
((Shape  *)PQ2)->Draw();     // вызов Draw класса  Shape
```

В комментариях к коду указано, функции **Draw** каких классов вызывают эти операторы. Как видно, если мы обращаемся через указатель к самому объекту типа **Circl**, то вызывается переопределенная в нем функция. Но если мы обращаемся к нему как к объекту ба-

зового класса (последний из приведенных операторов) или соответствующим образом используем операцию разрешения области, то вызывается функция базового класса.

Если бы в классе-наследнике **Circl** отсутствовала функция **Draw,** то все приведенные операторы вызывали бы функцию базового класса.

Таким образом, механизм наследования позволят использовать функции базового класса или переопределять их.

## 7.2. Виртуальные функции, абстрактные классы

Теперь рассмотрим другую задачу. Пусть мы имеем несколько классов, наследующих **Shape**: **Circl** (круг), **Rectangle** (прямоугольник), **Square** (квадрат) и т.п. Каждый из этих классов имеет свою функцию Draw, которая умеет рисовать соответствующую форму. Мы хотим работать с объектами этих фигур, как с объектами базового класса **Shape**, не разбираясь в истинной природе каждого объекта. И при этом хотим, чтобы программа сама понимала, что это за объект и как его рисовать. Например, мы хотим создать массив указателей на объекты различных форм:

```
Shape  *ShapeArray[10];
```

загрузить его указателями на объекты разных фигур:

```
ShapeArray[0]  =  new Circl;
ShapeArray[1]  =  new Rectang;
ShapeArray[2]  =  new Square;
```

и затем в цикле выполнять рисование этих фигур:

```
for(int  i  =  0;  i  <  3;  i++)
     ShapeArray[i]->Draw();
```

Рассмотренный ранее механизм наследования такую задачу решить не может. Поскольку ко всем объектам мы обращаемся через тип их базового класса **Shape**, то только функция этого класса и будет вызываться.

Поставленную задачу *полиморфизма* позволяют решить виртуальные функции. Они не связаны с другими функциями с тем же именем в классах – наследниках. Если в классах-наследниках эти функция переопределены, то при обращении к такой функции во время выполнения будет вызываться та из виртуальных функций с одинаковыми именами, которая соответствует классу объекта, указанного при вызове. Поэтому, если в базовом классе **Shape** объявить функцию **Draw** как виртуальную, то задача будет решена, и каждая фигура будет рисоваться своей функцией.

Синтаксически это оформляется следующим образом. В базовом классе **Shape** функция объявляется следующим образом:

```
virtual  void  Draw  (void);
```

И это все! Если функция была однажды объявлена виртуальной, она остается виртуальной и во всех классах наследниках. Таким образом для решения задачи полиморфизма хватило одного спецификатора **virtual**. Правда, обычно предпочитают для большей ясности программы в классах-наследниках тоже вводить спецификатор **virtual**, чтобы была ясна суть этих функций для тех, кто будет строить наследников данного класса. Но с точки зрения языка С++ это не обязательно.

Иногда в базовом классе определяют *чистую виртуальную функцию* (абстрактную функцию). Это функция, для которой не указана реализация. Для того, чтобы определить такую функцию, достаточно указать, что ее тело равно нулю:

```
virtual  void Draw  (void)  =  0;
```

В нашем примере именно так целесообразно объявить функцию **Draw** в базовом классе **Shape**, поскольку непонятно, как можно нарисовать просто абстрактную фигуру. Реализация для чистой полиморфной функции не пишется.

Класс, в котором имеется хоть одна чистая виртуальная функция, называется *абстрактным.* Для абстрактного класса невозможно создать объект. Такие классы предназначены только для построения на их основе конкретных классов-наследников.

**Раздел 8:**
# ОСОБЕННОСТИ КЛАССОВ, НАСЛЕДУЮЩИХ КЛАССАМ БИБЛИОТЕКИ КОМПОНЕНТОВ C++BUILDER

Лекция проводится в интерактивной форме как проблемная (2 час.)

## 8.1. Свойства

О некоторых особенностях построения классов, наследующих классам библиотеки компонентов C++Builder, уже говорилось ранее. К этим особенностям относится невозможность для таких классов множественного наследование и необходимость создавать объекты только с помощью операции new. Теперь остановимся на других особенностях, связанных с понятиями *свойства* и *события*.

Понятие свойства (**property**), объединяет поле данных и функции (методы) его записи и чтения. В рассматриваемых классах сами поля объявляются как обычно, но, как правило, в разделе **private**. Традиционно идентификаторы полей совпадают с именами соответствующих свойств, но с добавлением в качестве префикса символа “**F**”.

Свойство объявляется оператором вида:

```
__property<тип><имя>={read=< имя поля или метода чтения>
                      write=< имя поля или метода записи>
                     < директивы запоминания
                       и значения по умолчанию >
```

Если в разделах **read** или **write** этого объявления записано имя поля, значит предполагается прямое чтение или запись данных.

Если в разделе **read** записано имя метода чтения, то чтение будет осуществляться только функцией с этим именем. Функция чтения – это функция без параметра, возвращающее значение того типа, который объявлен для свойства. Имя функции чтения принято начинать с префикса **Get**, после которого следует имя свойства.

Если в разделе **write** записано имя метода записи, то запись будет осуществляться только процедурой с этим именем. Процедура записи – это процедура с одним параметром того типа, который объявлен для свойства. Имя процедуры записи принято начинать с префикса **Set**, после которого следует имя свойства.

Если раздел **write** отсутствует в объявлении свойства, значит это свойство только для чтения и пользователь не может задавать его значение.

Директивы запоминания определяют, как надо сохранять значения свойств при сохранении пользователем файла формы **.dfm**. Чаще всего используется директива

```
default <значение по умолчанию>
```

Она не задает начальное значение. Это дело конструктора. Директива просто говорит, что если пользователь в процессе проектирования не изменил значение свойства по умолчанию, то сохранять значение свойства не надо.

Приведём пример. Пусть требуется объявить класс с именем **MyClass**, наследующий непосредственно **TObject** и имеющий свойство целого типа с именем **A**. Тогда объявление этого класса может иметь вид:

```
class MyClass1 : public TObject
{
     private:
          int FA;
     protected:
          void __fastcall SetA(int);    // функция записи
     __published:
          __property int A = {read = FA, write =
                                     SetA, default = true};
};
```

Здесь вводится закрытое поле **FA**, объявляется защищенная функция **SetA**, используемая для записи значения в это поле, и вводится опубликованное свойство **A**, оперирующее этим полем. В объявлении свойства после ключевого слова **read** записано просто имя поля. Это означает, что функция чтения отсутствует и пользователь может читать непосредственно значение поля. После ключевого слова **write** следует ссылка на функцию записи **SetA**, с помощью которой будут записываться в поле **A** новые значения. В этой функции можно предусмотреть какие-то проверки допустимости вводимого значения **A**. Описание этой функции может иметь вид:

```
void  __fastcall  MyClass1::SetA(int Value)
{
     if(…)  FA  =  Value;
}
```

В приведенном примере описание свойства **A** помещено в раздел **published.** Следовательно, если этот класс описывает создаваемый вами новый компонент, то после его установки в систему свойство **A** будет появляться в окне Инспектора Объектов при использовании этого компонента. Если перенести объявление свойства в раздел **public,** то свойством по-прежнему можно будет пользоваться, но только во время выполнения приложения, поскольку в окне Инспектора Объектов оно появляться не будет. Если удалить из определения свойства слово **write** с последующей ссылкой на функцию записи, то свойство станет свойством только для чтения, т.к. изменить его непосредственно будет невозможно.

Для свойств типа массивов приведенный ранее оператор **property** изменяется следующим образом:

```
__property<тип><имя><список размерностей>=
                    {read=< имя поля или метода чтения>
                     write=< имя поля или метода записи>
                    < директивы запоминания
                      и значения по умолчанию >
```

Список размерностей представляет собой последовательность квадратных скобок, в которых записывается тип размерности и может записываться идентификатор. Приведем в качестве примера возможный вариант описания класса матриц действительных чисел размером **N**×**M**:

```
// Класс матриц действительных чисел
class  Matrix
{
     float  *data;
     int N;     // число строк
     int M;    // число столбцов
     public:
          Matrix(int,  int);
          ~Matrix(  )  {delete[  ]  data;  }
          __property  float  Items  [int  i][int  j]  =
                      { read= GetItems, write=SetItems };
     private:
          float  __fastcall  GetItems(int  i,  int j);
          void  __fastcall  SetItems(int  i,  int  j,
                                      float  value);
};
Matrix::Matrix(int  n,  int  m)    // конструктор
{
     data=new float[n*m];   // создание экземпляра класса
     for(int  i  =  0;  i<  n*m;  i++) // инициализация
          data[i]  =  0.;
          N  =  n;
          M  =  m;
}
```

```
void __fastcall Matrix::SetItems(int i, int j, float
                                              value)
{
// запись значения value в элемент (i,j)
     if((i<1) || (i>N) || (j<1) || (j>M))
          ShowMessage("Недопустимые индексы (" +
          IntToStr(i) + "," + IntToStr(j) + ")");
               else data[(i - 1) * M + j - 1] = value;
}
float  __fastcall  Matrix::GetItems(int  i,  int j)
{
// чтение значения элемента  (i,j)
     if if((i<1) || (i>N) || (j<1) || (j>M))
          ShowMessage("Недопустимые индексы (" +
          IntToStr(i) + "," + IntToStr(j) + ")");
               else return data[(i - 1) * M + j -1];
}
```

В приведенном коде создается класс матриц **Matrix.** Класс имеет открытое свойство **Items,** к которому можно обращаться как к двумерному массиву. Об этом говорит его определение в операторе **__property**: **float Items** [**int i**][**int j**]. Указание в списке размерностей идентификаторов **i** и **j** не является обязательным. Список мог бы иметь вид: [**int**] [**int**].

Задание размерностей изменяет вид функций чтения и записи. В функцию чтения **GetItems** передаются два целых параметра, определяющих индексы читаемого элемента матрицы. В приведенном примере индексы матриц отсчитываются от 1, а не от нуля, что, вероятно, более удобно пользователю. В функцию записи **SetItems** помимо записываемого значения **value** также передаются индексы того элемента, в который должно быть записано это значение.

Создание экземпляра матрицы в программе может, осуществляться, например, оператором:

```
Matrix  x(4,5);
```

Этот оператор создает матрицу x размерностью 4x5. Запись и чтение элементов матрицы в программе осуществляется через свойство **Items**. Например:

```
x.Items[2][3]  =  1.5;  float  y  =  x.Items[2][3];
```

Первый из этих операторов заносит значение 1.5 в 3-ий элемент 2-ой строки, а второй оператор читает это значение.

## 8.2. События

Событие – это специальное свойство, являющееся указателем функции. В C++Builder тип обобщенного указателя на функцию, которой передается один параметр типа **TObject** (обычно **this**), – **TNotifyEvent**. Подобный тип используется в C++Builder для событий типа **OnClick** и многих других, которые передают в обработчик только один параметр – **TObject *Sender**. Если требуется ввести в класс подобное событие, достаточно определить в объявлении класса соответствующее поле и метод работы с ним. Например:

```
private:
…..
     TNotifyEvent  FMyEvent;
…..
     __published:
          __property  TNotifyEvent  MyEvent  =  {  read
                         =  FMyEvent,  write  =  FMyEvent};
```

Остается только вызвать в нужный момент обработчик событий пользователя, если пользователь его предусмотрел. Проверка, имеется ли обработчик пользователя, осуществляется проверкой соответствующего события как булевой величины, возвращающей **true**,

если пользователь предусмотрел свой обработчик. Значит, при возникновении события надо проверять, имеется ли обработчик пользователя, и, если имеется, то вызывать его. Для этого можно использовать оператор вида:

```
if (FMyEvent)  OnMyEvent (this);
```

Функция **OnMyEvent**, которая вызывается этим оператором, это и есть обработчик пользователя. Его имя совпадает с именем свойства, перед которым добавляется префикс "On".

Место, куда надо включать подобный оператор, зависит от вида события. Если событие вызывается каким-то из ваших методов, то вызов обработчика пользователя надо осуществлять из этого метода. Если событие связано с какими-то сообщениями, поступающими от других приложений или от Windows, то надо предусмотреть обработчик соответствующего сообщения и из него вызывать обработчик пользователя.

Если в обработчик события надо передать какие-то параметры помимо **this**, то тип функции **TNotifyEvent** уже не подходит и надо объявить свой собственный тип. Это объявление делается с помощью ключевого слова **__closure**. Например:

```
typedef  void  __fastcall  (__closure  *TMyEvent)
(System::TObject  *Sender,  bool&  MyParam);
class  T  :  public  TObject
{
     private
          TMyEvent  FMyEvent;
     published
          __property  TMyEvent  FMyEvent  =  {  read  =
                              FMyEvent,  write  =  FMyEvent};
…..
}
```

Выше было рассмотрено введение в класс какого-то нового события. Если же вам надо переопределить одно из традиционных событий, связанных с клавиатурой, мышью и т.п., то это можно сделать, переопределив соответствующий стандартный обработчик родительского класса.

### 4.3. Лабораторные работы

| *№ п/п* | *Номер раздела дисциплины* | *Наименование лабораторной работы* | *Объем (час.)* | *Вид занятия в интерактивной, активной, инновационной формах, (час.)* |
|---|---|---|---|---|
| 1 | **2, 5.** | Разработка простейшего приложения с использованием элементов ИСР C++ Builder | 6 | - |
| 2 | **3, 4.** | Разработка приложения с использованием компонентов ввода и отображения однострочного текста | 7 | - |
| 3 | **1, 8.** | Разработка приложения с использованием компонетов ввода и отображения многострочного текста | 7 | - |
| 4 | **2, 7.** | Разработка приложения с использованием управляющих компонентов | 7 | Решение проблем в группах смешанного состава (4 час.) |
| 5 | **4, 6.** | Разработка приложения с использованием графического компонента | 7 | - |
| | | **ИТОГО** | **34** | **4** |

### 4.4. Практические занятия

Учебным планом не предусмотрены.

### 4.5. Контрольные мероприятия: курсовой проект (курсовая работа), контрольная работа, РГР, реферат

Учебным планом не предусмотрены.

## 5. МАТРИЦА СООТНЕСЕНИЯ РАЗДЕЛОВ УЧЕБНОЙ ДИСЦИПЛИНЫ К ФОРМИРУЕМЫМ В НИХ КОМПЕТЕНЦИЯМ И ОЦЕНКЕ РЕЗУЛЬТАТОВ ОСВОЕНИЯ ДИСЦИПЛИНЫ

| ***Компетенции*** / ***№, наименование разделов дисциплины*** | ***Кол-во часов*** | ***Компетенции*** | | ***Σ комп.*** | $t_{ср}$, ***час*** | ***Вид учебных занятий*** | ***Оценка результатов*** |
|---|---|---|---|---|---|---|---|
| | | ***ОПК*** | ***ПК*** | | | | |
| | | **9** | **6** | | | | |
| **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** |
| **1.** Введение. Объектно-ориентированное программирование | 15 | + | + | 2 | 7,5 | Лк, ЛР, СРС | Экзамен |
| **2.** Основные направления в программировании | 14 | + | + | 2 | 7 | Лк, ЛР, СРС | Экзамен |
| **3.** Классы | 16 | + | + | 2 | 8 | Лк, ЛР, СРС | Экзамен |
| **4.** Базовые принципы объектно-ориентированного программирования | 15 | + | + | 2 | 7,5 | Лк, ЛР, СРС | Экзамен |
| **5.** Базовые конструкции объектно-ориентированных программ | 14 | + | + | 2 | 7 | Лк, ЛР, СРС | Экзамен |
| **6.** Данные и функции класса | 16 | + | + | 2 | 8 | Лк, ЛР, СРС | Экзамен |
| **7.** Наследование и полиморфизм. Виртуальные функции, абстрактные классы | 13 | + | + | 2 | 6,5 | Лк, ЛР, СРС | Экзамен |
| **8.** Особенности классов, наследующих классам библиотеки компонентов C++ Builder | 14 | + | + | 2 | 7 | Лк, ЛР, СРС | Экзамен |
| ***всего часов*** | **117** | **58,5** | **58,5** | **2** | **58,5** | | |

## 6. ПЕРЕЧЕНЬ УЧЕБНО-МЕТОДИЧЕСКОГО ОБЕСПЕЧЕНИЯ ДЛЯ САМОСТОЯТЕЛЬНОЙ РАБОТЫ ОБУЧАЮЩИХСЯ ПО ДИСЦИПЛИНЕ

**1.** Глушаков, С.В. Программирование в среде Borland C++ Builder 6 / Глушаков С.В., Зорянский В.Н., Хоменко С.Н. - Харьков: Фолио, 2003. - 508с. Стр-цы для СР: 38-41, 114-118, 311-357.
**2.** Калверт, Ч. Borland C++ Bulder: Энциклопедия программиста / Калверт Ч., Рейсдорф К. - М.: Диасофт, 2005. - 1008с. Стр-цы для СР: 25-37, 54-75, 105-113, 157-205.
**3.** Павловская, Т.А. С++. Объектно-ориентированное программирование. Практикум: Учеб. пособие для вузов. - СПб.: Питер, 2004. - 265с. Стр-цы для СР: 10-18, 43-89, 101-114, 123-138.

## 7. ПЕРЕЧЕНЬ ОСНОВНОЙ И ДОПОЛНИТЕЛЬНОЙ ЛИТЕРАТУРЫ, НЕОБХОДИМОЙ ДЛЯ ОСВОЕНИЯ ДИСЦИПЛИНЫ

| *№* | ***Наименование издания*** *(автор, заглавие, выходные данные)* | ***Вид занятия*** | ***Количество экземпляров в библиотеке,*** **шт.** | ***Обеспеченность,*** **(экз./ чел.)** |
|---|---|---|---|---|
| **1** | **2** | **3** | **4** | **5** |
| **Основная литература** | | | | |
| 1. | Агафонов, Е.Д. Прикладное программирование : учебное пособие / Е.Д. Агафонов, Г.В. Ващенко; Министерстов образования и науки Российской Федерации, Сибирский Федеральный университет. – Красноярск : Сибирский федеральный университет, 2015. – 112 с. : табл., граф., ил. - Библиогр. в кн. – ISBN 978-5-7638-3165-8; То же [Электронный ресурс]. – URL://biblioclub.ru/index.php?page=book&id=435640 | Лк | ЭР | 1,0 |
| 2. | Ашарина, И. В. Объектно-ориентированное программирование в С++: лекции и упражнения : учеб. пособие для вузов / И. В. Ашарина. - Москва : Горячая линия-Телеком, 2008. - 320 с. | Лк, ПЗ | 20 | 1,0 |
| 3. | Иванова, Г. С. Объектно-ориентированное программирование : учебник для вузов / Г. С. Иванова, Т. Н. Ничушкина, Е. К. Пугачев. - 2-е изд., перераб. и доп. - М. : МГТУ, 2003. - 368 с. - (Информатика в техническом университете). | Лк | 10 | 0,7 |
| 4. | Пахомов, Б. И. C/C++ и Borland C++ Builder для начинающих : учебное пособие / Б. И. Пахомов. - Санкт-Петербург : БХВ- Петербург, 2007. - 640 с. | ЛР, кр | 10 | 0,7 |
| **Дополнительная литература** | | | | |
| 5. | Архангельский, А. Я. Приемы программирования в C++Builder 6 и 2006 : учебное пособие / А. Я. Архангельский. - Москва : БИНОМ, 2006. - 992 с. | Лк | 5 | 0,3 |
| 6. | Ашарина, И. В. Язык С ++ и объектно- ориентированное программирование в С ++. Лабораторный практикум : учебное пособие / И. В. Ашарина, Ж. Ф. Крупская . - Москва : Горячая линия- Телеком, 2015. - 232 с. | ЛР | 5 | 0,3 |
| 7. | Крумин, О.К. Синтез графических образов простыми средствами: Лабораторный практикум / О.К. Крумин. – Братск: ФГБОУ ВПО «БрГУ», 2012. – 91 с. | ЛР | 40 | 1,0 |
| 8. | Орлов, С. А. Теория и практика языков программирования : учебник для бакалавров и магистров / С. А. Орлов. - Санкт-Петербург : Питер, 2013. - 688 с. - (Учебное пособие. Стандарт третьего поколения). | Лк, ПЗ | 5 | 0,3 |
| 9. | Павловская, Т.А. С/С++. Структурное программирование: Практикум / Павловская Т.А., Щупак Ю.А. - СПб.: Питер, 2007. - 239 с. | Лк | 24 | 1,0 |
| 10. | Шамис, В. А. C++ Builder Borland Developer Studio 2006 | кр | 5 | 0,3 |

|  | : учебное пособие / В. А. Шамис. - Санкт-Петербург : Питер, 2007. - 781 с. - (Для профессионалов). |  |  |  |
|---|---|---|---|---|

## 8. ПЕРЕЧЕНЬ РЕСУРСОВ ИНФОРМАЦИОННО - ТЕЛЕКОММУНИКАЦИОННОЙ СЕТИ «ИНТЕРНЕТ» НЕОБХОДИМЫХ ДЛЯ ОСВОЕНИЯ ДИСЦИПЛИНЫ

1.Электронный каталог библиотеки БрГУ http://irbis.brstu.ru/CGI/irbis64r_15/cgiirbis_64.exe?LNG=&C21COM=F&I21DBN=BOOK&P21DBN=BOOK&S21CNR=&Z21ID=.

2. Электронная библиотека БрГУ http://ecat.brstu.ru/catalog .

3. Электронно-библиотечная система «Университетская библиотека online» http://biblioclub.ru .

4. Электронно-библиотечная система «Издательство «Лань» http://e.lanbook.com .

5. Информационная система "Единое окно доступа к образовательным ресурсам" http://window.edu.ru .

6. Научная электронная библиотека eLIBRARY.RU http://elibrary.ru .

7. Университетская информационная система РОССИЯ (УИС РОССИЯ) https://uisrussia.msu.ru/ .

8. Национальная электронная библиотека НЭБ http://xn--90ax2c.xn--p1ai/how-to-search /.

## 9. МЕТОДИЧЕСКИЕ УКАЗАНИЯ ДЛЯ ОБУЧАЮЩИХСЯ ПО ОСВОЕНИЮ ДИСЦИПЛИНЫ

### 9.1. Методические указания для обучающихся по выполнению лабораторных работ

**Лабораторная работа №1**

**Разработка простейшего приложения с использованием элементов ИСР C++ Builder**

Цель работы:

1. Выработать практические навыки работы с элементами ИСР C++ Builder, научиться создавать, компилировать, выполнять и исправлять простейшие приложения в системе визуального программирования C++ Builder;

2. на основе индивидуального задания разработать и отладить приложение обработки алгоритмов линейной структуры.

Задание (один из возможных вариантов):

Треугольник задан величинами своих углов и радиусом вписанной окружности. Найти стороны треугольника.

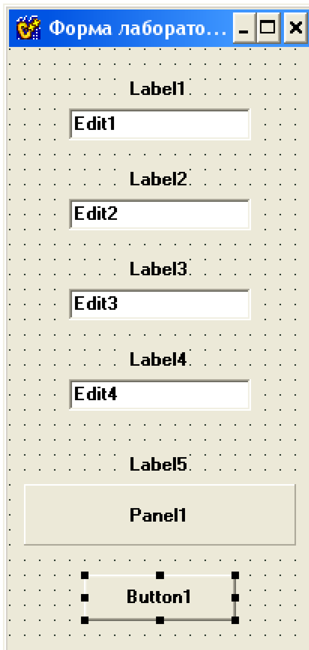

Рис. 1. Окно формы приложения

Порядок выполнения:
Ознакомиться с основными теоретическими положениями к работе. При разработке приложения, как один из вариантов, можно использовать следующую комбинацию визуальных компонентов C++ Builder палитры Standard: метки Label, окна редактирования Edit, панель Panel и кнопки Button. В свойстве Caption меток Label должно быть записано название параметров, необходимых для выполнения задания. В окна редактирования Edit пользователь будет вводить некоторые значения этих параметров. Срабатывание кнопки Button1 предполагает выполнение некоторого действия в соответствии с темой ВИЗ. Панель Panel должна отображать результат работы программы. Как вариант, в качестве компонента отображения работы приложения можно предусмотреть метку Label. Окно формы может выглядеть следующим образом (рис. 1). Для симметричного расположения компонентов, для одинакового их размера по горизонтали и вертикали можно воспользоваться командами Edit|Align – выравнивание размещения, Edit|Size – выравнивание размеров и Edit|Scale – масштабирование.

Форма отчетности:
Отчёт сдаётся в печатном виде. В отчёте должны присутствовать:
1. Номер варианта индивидуального задания (ВИЗ);
2. Цель работы;
3. Задание на лабораторную работу;
4. Окно формы приложения с результатами работы. В заголовке формы указать номер лабораторной работы, номер индивидуального задания, а также фамилию и.о. обучающегося;
5. Листинг файла реализации .cpp модуля формы.

Задания для самостоятельной работы:
Предусмотрены ВИЗ обучающегося.

Рекомендации по выполнению заданий и подготовке к лабораторной работе
Ознакомиться с теоретическим материалом, представленным во втором и пятом разделах данной дисциплины.

Основная литература

1. Пахомов, Б. И. C/C++ и Borland C++ Builder для начинающих : учебное пособие / Б. И. Пахомов. - Санкт-Петербург : БХВ- Петербург, 2007. - 640 с.

Дополнительная литература

2. Крумин, О.К. Синтез графических образов простыми средствами: Лабораторный практикум / О.К. Крумин. – Братск: ФГБОУ ВПО «БрГУ», 2012. – 91 с.

Контрольные вопросы для самопроверки
1. Назовите элементы ИСР C++ Builder и их назначение.
2. Каким образом загружается заголовочный файл в Редакторе Кода?
3. Опишите приёмы работы в Инспекторе Объектов.

## Лабораторная работа №2

**Разработка приложения с использованием компонентов ввода и отображения однострочного текста**

Цель работы:
1. Приобрести практические навыки использования компонентов отображения и ввода текстовой информации библиотеки C++ Builder;

2. На основе индивидуального задания разработать и отладить приложение обработки однострочного текста.

Задание (один из возможных вариантов):
Введена строка, содержащая текст. Разработать приложение, определяющее количество слов в предложении с чётным количеством согласных букв.

Порядок выполнения:
Ознакомиться с основными теоретическими положениями к работе. При разработке приложения, как один из вариантов, можно использовать следующую комбинацию визуальных компонентов C++ Builder палитры Standard: метки Label, окно редактирования Edit и кнопки Button. В свойстве Caption метки Label1 должно быть кратко записано индивидуальное задание. В окно редактирования Edit1 пользователь будет вводить некоторое предложение. Срабатывание кнопки Button1 предполагает выполнение некоторого действия над введённым текстом в соответствии с темой ВИЗ. Метка Label2 может отображать результат работы программы. Как вариант, в качестве компонента отображения работы приложения можно предусмотреть окно редактирования Edit2. При срабатывании кнопки Button2 приложение должно закрываться. Окно формы может выглядеть следующим образом (рис. 2)

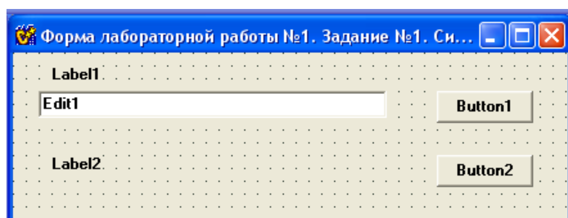

Рис. 2. Окно формы приложения

Форма отчетности:
Отчёт сдаётся в печатном виде. В отчёте должны присутствовать:
1. Номер варианта индивидуального задания (ВИЗ);
2. Цель работы;
3. Задание на лабораторную работу;
4. Окно формы приложения с результатами работы. В заголовке формы указать номер лабораторной работы, номер индивидуального задания, а также фамилию и.о. обучающегося;
5. Листинг файла реализации .cpp модуля формы.

Задания для самостоятельной работы:
Предусмотрены ВИЗ обучающегося.

Рекомендации по выполнению заданий и подготовке к лабораторной работе
Ознакомиться с теоретическим материалом, представленным в третьем и четвёртом разделах данной дисциплины.

Основная литература


1. Пахомов, Б. И. С/С++ и Borland C++ Builder для начинающих : учебное пособие / Б. И. Пахомов. - Санкт-Петербург : БХВ- Петербург, 2007. - 640 с.


Дополнительная литература


2. Крумин, О.К. Синтез графических образов простыми средствами: Лабораторный практикум / О.К. Крумин. – Братск: ФГБОУ ВПО «БрГУ», 2012. – 91 с.
3. Ашарина, И. В. Язык С ++ и объектно- ориентированное программирование в С ++. Лабораторный практикум : учебное пособие / И. В. Ашарина, Ж. Ф. Крупская . - Москва : Горячая линия- Телеком, 2015. - 232 с.


Контрольные вопросы для самопроверки
1. Какое свойство компонентов Label, StaticText, Panel определяет текст, отображаемый в них?
2. Назовите функции, переводящие числовую информацию в строки и обратно.
3. Опишите функции или методы, которые использованы в приложении для обработки строки.

**Лабораторная работа №3**

**Разработка приложения с использованием компонентов ввода и отображения многострочного текста**

Цель работы:
1. приобрести практические навыки использования компонентов ввода и отображения текстовой информации библиотеки C++ Builder;
2. на основе индивидуального задания разработать и отладить приложение обработки многострочного текста.

Задание (один из возможных вариантов):
Дана целочисленная матрица размера MxN. Определить сумму положительных элементов над побочной диагональю.

Порядок выполнения:
Ознакомиться с основными теоретическими положениями. При разработке этого приложения, как один из вариантов, можно использовать следующую комбинацию визуальных компонентов C++ Builder: окно многострочного редактирования, например, StringGrid, метки Label, окна редактирования Edit и кнопки Button. Окно многострочного редактирования должно отображать элементы матрицы. Метки Label1 и Label2 будут пояснять значение окон редактирования Edit1 и Edit2. Те, в свою очередь, должны служить местом, в котором пользователь указывает количество строк и столбцов, соответственно. Срабатывание кнопки Button1 предполагает генерацию исходной матрицы. При нажатии на кнопку Button2 должно выполняться некоторое действие над элементами матрицы в соответствии с темой ВИЗ. Метка Label3 может отображать результат работы программы. При срабатывании кнопки Button3 приложение должно закрываться. Окно формы может выглядеть следующим образом (рис. 3)

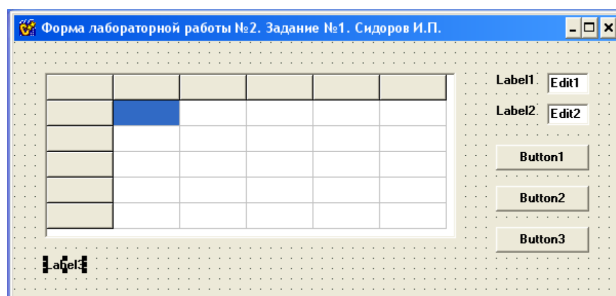

Рис. 3. Окно формы приложения

В программе так же следует предусмотреть обработчик окон редактирования Edit (EditChange), выполняющий действия при изменении текста окон.

Форма отчетности:
Отчёт сдаётся в печатном виде. В отчёте должны присутствовать:
1. Номер варианта индивидуального задания (ВИЗ);
2. Цель работы;
3. Задание на лабораторную работу;
4. Окно формы приложения с результатами работы. В заголовке формы указать номер лабораторной работы, номер индивидуального задания, а также фамилию и.о. обучающегося;
5. Листинг файла реализации .cpp модуля формы.

Задания для самостоятельной работы:
Предусмотрены ВИЗ обучающегося.

Рекомендации по выполнению заданий и подготовке к лабораторной работе
Ознакомиться с теоретическим материалом, представленным в первом и восьмом разделах данной дисциплины.

Основная литература

1. Пахомов, Б. И. C/C++ и Borland C++ Builder для начинающих : учебное пособие / Б. И. Пахомов. - Санкт-Петербург : БХВ- Петербург, 2007. - 640 с.

Дополнительная литература

2. Крумин, О.К. Синтез графических образов простыми средствами: Лабораторный практикум / О.К. Крумин. – Братск: ФГБОУ ВПО «БрГУ», 2012. – 91 с.

Контрольные вопросы для самопроверки

1. Каким образом устанавливается начальное значение текста компонента Memo?
2. Укажите различия между компонентами ListBox и ComboBox.
3. Какое свойство позволяет редактировать содержимое компонента TStringGrid?

## Лабораторная работа №4

## Разработка приложения с использованием управляющих компонентов

Занятие в интерактивной форме с решением проблем в группах смешанного состава (4 час.)

Цель работы:

1. Приобрести пракические навыки использования управляющих компонентов библиотеки C++ Builder;
2. Приобрести практические навыки использования компонента построения графиков и диаграмм Chart;
3. На основе индивидуального задания разработать и отладить приложение построения динамических характеристик звеньев.

Задание:

Путём моделирования заданного типового звена получить графики:

- переходной h(t) и весовой w(t) функций;
- реальной частотной Re(ω) и мнимой частотной Im(ω) характеристик;
- амплитудно-частотной Am(ω) и фазовочастотной F(ω) характеристик;
- логарифмической амплитудно-частотной 20lgAm(lgω) и логарифмической фазовочастотной F(lg ω) характеристик.

Порядок выполнения:

Ознакомиться с основными теоретическими положениями. При разработке этого приложения, как один из вариантов, можно использовать следующую комбинацию визуальных компонентов C++ Builder: компонент RadioGroup, Chart, окна редактирования Edit**,** метки Label и кнопки BitBtn. В панели группы радиокнопок RadioGroup пользователь может выбирать тип графика. Компоненты Chart1, Chart2 будут строить графики. В окна редактирования Edit1-Edit4 пользователь будет заносить значения коэффициентов передаточной функции в соответствии с индивидуальным заданием. При срабатывании кнопки BitBtn1 должны строиться графики в компонентах Chart. При нажатии кнопки BitBtn2 приложение будет закрываться. Для кнопки BitBtn1 предусмотреть изображение, соответствующее назначению кнопки. В свойстве Kind кнопки BitBtn2 задать тип bkClose. Так же для обеих кнопок следует предусмотреть использование клавиш ускоренного доступа. Окно формы может выглядеть следующим образом (рис. 4). На форме так же можно разместить компоненты для задания параметров времени ($t_{нач.}$, $t_{кон.}$, $dt$) и частоты ($\omega_{нач.}$, $\omega_{кон.}$, $d\omega$).

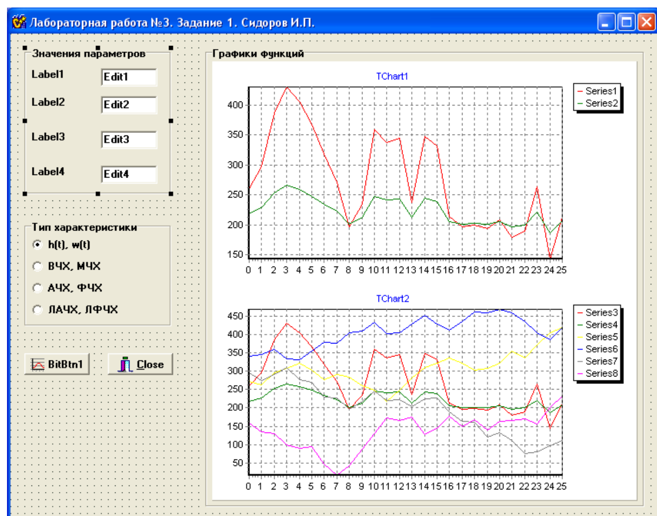


Рис. 4. Окно формы приложения

Форма отчетности:

Отчёт сдаётся в печатном виде. В отчёте должны присутствовать:

1. Номер варианта индивидуального задания (ВИЗ);
2. Цель работы;
3. Задание на лабораторную работу;
4. Окно формы приложения с результатами работы. В заголовке формы указать номер лабораторной работы, номер индивидуального задания, а также фамилию и.о. обучающегося;
5. Листинг файла реализации .cpp модуля формы.

Задания для самостоятельной работы:

Предусмотрены ВИЗ обучающегося.

Рекомендации по выполнению заданий и подготовке к лабораторной работе

Ознакомиться с теоретическим материалом, представленным во втором и седьмом разделах данной дисциплины.

Основная литература

1. Пахомов, Б. И. C/C++ и Borland C++ Builder для начинающих : учебное пособие / Б. И. Пахомов. - Санкт-Петербург : БХВ- Петербург, 2007. - 640 с.

Дополнительная литература

2. Крумин, О.К. Синтез графических образов простыми средствами: Лабораторный практикум / О.К. Крумин. – Братск: ФГБОУ ВПО «БрГУ», 2012. – 91 с.

Контрольные вопросы для самопроверки

1. Укажите основные свойства кнопок Button и BitBtn.
2. Объясните, в чём разница между компонентами RadioGroup и RadioButton.
3. Перечислите методы серий Series. Каково назначение параметров, входящих в эти методы?

## Лабораторная работа №5
## Разработка приложения с использованием графического компонента

Цель работы:

1. Приобрести практические навыки использования графических компонентов C++ Builder;
2. На основе индивидуального задания разработать и отладить приложение построения графических изображений.

Задание:

Разработать пользовательское приложение, которое:

1. рисует на канве заданное изображение, с прорисовкой координатных осей и текстовых обозначений;
2. определяет принадлежность некоторой точки с координатами (X,Y) закрашенной области.

Порядок выполнения:

Ознакомиться с основными теоретическими положениями. При разработке этого приложения, как один из вариантов, можно использовать следующую комбинацию визуальных компонентов C++ Builder: компонент Image, окна редактирования Edit**,** метки Label и кнопки Button. В компоненте Image должно располагаться графическое изображение, в соответствии

с вариантом индивидуального задания. В окна редактирования Edit1 и Edit2 пользователь будет заносить координаты точки *X*,*Y*. При срабатывании кнопки Button1 должна проверяться принадлежность точки с координатами *X*,*Y* закрашенной области. Результат этой проверки должен выводиться в метку Label3. При срабатывании кнопки Button2 приложение должно закрываться. Окно формы может выглядеть следующим образом (рис. 5).

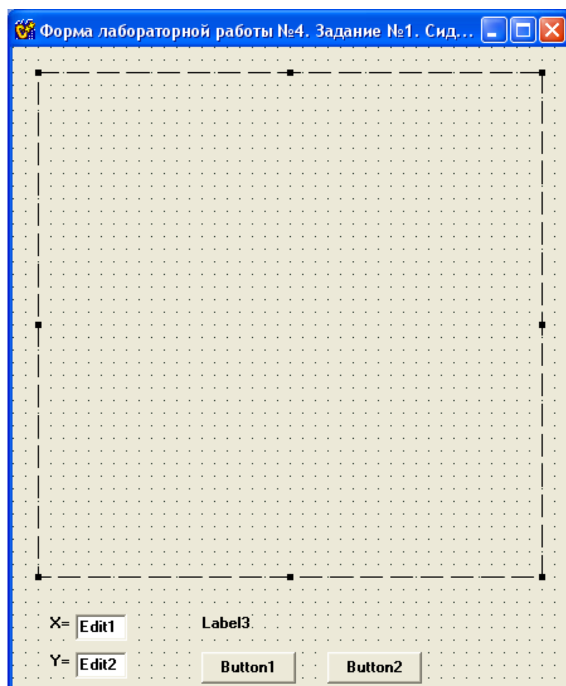

Рис. 5. Окно формы приложения

Форма отчетности:

Отчёт сдаётся в печатном виде. В отчёте должны присутствовать:

1. Номер варианта индивидуального задания (ВИЗ);
2. Цель работы;
3. Задание на лабораторную работу;
4. Окно формы приложения с результатами работы. В заголовке формы указать номер лабораторной работы, номер индивидуального задания, а также фамилию и.о. обучающегося;
5. Листинг модуля формы .cpp файла реализации.

Задания для самостоятельной работы:

Предусмотрены ВИЗ обучающегося.

Рекомендации по выполнению заданий и подготовке к лабораторной работе

Ознакомиться с теоретическим материалом, представленным в четвёртом и шестом разделах данной дисциплины.


Основная литература

1. Пахомов, Б. И. C/C++ и Borland C++ Builder для начинающих : учебное пособие / Б. И. Пахомов. - Санкт-Петербург : БХВ- Петербург, 2007. - 640 с.

Дополнительная литература

2. Крумин, О.К. Синтез графических образов простыми средствами: Лабораторный практикум / О.К. Крумин. – Братск: ФГБОУ ВПО «БрГУ», 2012. – 91 с.


Контрольные вопросы для самопроверки

1. Укажите способы рисования по канве.
2. Опишите методы, которые использованы в приложении для рисования заданной фигуры.
3. Укажите различия между параметрами fsSurface и fsBorder метода FloodFill.

## 10. ПЕРЕЧЕНЬ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, ИСПОЛЬЗУЕМЫХ ПРИ ОСУЩЕСТВЛЕНИИ ОБРАЗОВАТЕЛЬНОГО ПРОЦЕССА ПО ДИСЦИПЛИНЕ

1. ОС Windows 7 Professional.
2. Microsoft Office 2007 Russian Academic OPEN No Level.
3. Антивирусное программное обеспечение Kaspersky Security.

## 11. ОПИСАНИЕ МАТЕРИАЛЬНО-ТЕХНИЧЕСКОЙ БАЗЫ, НЕОБХОДИМОЙ ДЛЯ ОСУЩЕСТВЛЕНИЯ ОБРАЗОВАТЕЛЬНОГО ПРОЦЕССА ПО ДИСЦИПЛИНЕ

| ***Вид занятия*** | ***Наименование аудитории*** | ***Перечень основного оборудования*** | ***№ ПЗ и ЛР*** |
|---|---|---|---|
| **1** | **3** | **4** | **5** |
| ЛР | Дисплейный класс | AMD Athlon 64 (5GHz/250Gb/2Gb/DD-RW), 2 ядра | ЛР 1-5 |
| СР | ЧЗ №3 | Оборудование 15 - CPU 5000/RAM 2Gb/HDD (Монитор TFT 19 LG 1953S-SF);принтер HP LaserJet P3005 | - |

**Приложение 1**

## ФОНД ОЦЕНОЧНЫХ СРЕДСТВ ДЛЯ ПРОВЕДЕНИЯ ПРОМЕЖУТОЧНОЙ АТТЕСТАЦИИ ОБУЧАЮЩИХСЯ ПО ДИСЦИПЛИНЕ

### 1. Описание фонда оценочных средств (паспорт)

| № компетенции | Элемент компетенции | Раздел | Тема | ФОС |
|---|---|---|---|---|
| ОПК-9 | Способность использовать навыки работы с компьютером, владеть методами информационных технологий, соблюдать основные требования информационной безопасности | **1.** Введение. Объектно-ориентированное программирование | 1.1. Основные понятия и определения | Экзаменационный вопрос 1.1 |
| | | **2.** Основные направления в программировании | 2.1. Процедурное программирование | Экзаменационный вопрос 2.1 |
| | | | 2.3. Концепция типов | Экзаменационный вопрос 2.3 |
| | | **3.** Классы | 3.1. Объявление класса | Экзаменационный вопросы 3.1-3.3 |
| | | **4.** Базовые принципы объектно-ориентированного программирования | 4.1. Абстрагирование | Экзаменационный вопрос 4.1 |
| | | | 4.2. Ограничение доступа | Экзаменационный вопрос 4.2 |
| | | | 4.3. Модульность | Экзаменационный вопрос 4.3 |
| | | | 4.4. Иерархия | Экзаменационный вопрос 4.3 |
| | | **5.** Базовые конструкции объектно-ориентированных программ | 5.1. Объекты и классы | Экзаменационный вопрос 5.1 |
| | | | 5.2. Интерфейс и реализация объекта | Экзаменационный вопрос 5.2 |
| | | **6.** Данные и функции класса | 6.1. Данные-элементы, статические данные, константные данные | Экзаменационный вопрос 6.1 |
| | | **7.** Наследование и полиморфизм, виртуальные функции, абстрактные классы | 7.2. Виртуальные функции, абстрактные классы | Экзаменационный вопрос 7.1 |
| | | **8.** Особенности классов, наследующих классам библиотеки компонентов C++ Builder | 8.2. События | Экзаменационный вопрос 8.2 |
| ПК-6 | способность производить расчёты и проетирование отдельных блоков и устройств систем автоматизации и управления и выбирать стандартные средства автоматики, измерительной и вычислительной техники для проектирования систем автоматизации и | **1.** Введение. Объектно-ориентированное программирование | 1.2. Основы визуального программирования интерфейса | Экзаменационный вопрос 1.2 |
| | | **2.** Основные направления в программировании | 2.2. Модульное программирование | Экзаменационный вопрос 2.2 |
| | | | 2.4. Объектно-ориентированное программирование | Экзаменационный вопрос 2.4 |
| | | **3.** Классы | 3.2. Функции-элементы, дружественные функции, константные функции | Экзаменационный вопросы 3.4, 3.5 |

| | | | | |
|---|---|---|---|---|
| | управления | **4.** Базовые принципы объектно-ориентированного программирования | 4.5. Типизация | Экзаменационный вопрос 4.4 |
| | | | 4.6. Параллелизм | Экзаменационный вопрос 4.4 |
| | | | 4.7. Устойчивость | Экзаменационный вопрос 4.5 |
| | | | 4.8. Иерархия классов | Экзаменационный вопрос 4.6 |
| | | **5.** Базовые конструкции объектно-ориентированных программ | 5.3. Механизмы наследования | Экзаменационный вопрос 5.3 |
| | | | 5.4. Обработка исключений | Экзаменационный вопрос 5.4 |
| | | **6.** Данные и функции класса | 6.2. Конструкторы и деструкторы | Экзаменационные вопросы 6.2-6.6 |
| | | **7.** Наследование и полиморфизм, виртуальные функции, абстрактные классы | 7.1. Наследование и полиморфизм | Экзаменационный вопрос 7.1 |
| | | **8.** Особенности классов, наследующих классам библиотеки компонентов C++ Builder | 8.1. Свойства | Экзаменационный вопрос 8.1 |

**2. Вопросы к экзамену**

| № п/п | Компетенции | | ВОПРОСЫ К ЭКЗАМЕНУ | № и наименование раздела |
|---|---|---|---|---|
| | Код | Определение | | |
| 1 | 2 | 3 | 4 | 5 |
| **1.** | ОПК-9 | способность использовать навыки работы с компьютером, владеть методами информационных технологий, соблюдать требования информационной безопасности | **1.1.** Основные понятия и определения объектно-ориентированного программирования (ООП) | **1.** Введение. Объектно-ориентированное программирование |
| | | | **2.1.** Процедурное программирование | **2.** Основные направления в программировании |
| | | | **2.3.** Концепция типов данных | |
| | | | **3.1.** Объявление класса | **3.** Классы |
| | | | **3.2.** Спецификаторы доступа к элементам класса | |
| | | | **3.3.** Создание объектов класса | |
| | | | **4.1.** Абстрагирование | **4.** Базовые принципы объектно-ориентированного программирования |
| | | | **4.2.** Ограничение доступа | |
| | | | **4.3.** Модульность. Иерархия | |
| | | | **5.1.** Классы и объекты | **5.** Базовые конструкции объектно-ориентированных программ |
| | | | **5.2.** Интерфейс и реализация | |
| | | | **6.1.** Статические данные, константные данные | **6.** Данные и функции класса |
| | | | **7.1.** Полиморфизм. Виртуальные функции | **7.** Наследование и полиморфизм. Виртуальные функции, абстрактные классы |

| | | | | |
|---|---|---|---|---|
| | | | **8.1.** Свойства | **8.** Особенности классов, наследующих классам библиотеки компонентов C++ Builder |
| **2.** | ПК-6 | способность производить расчёты и проетирование отдельных блоков и устройств систем автоматизации и управления и выбирать стандартные средства автоматики, измерительной и вычислительной техники для проектирования систем автоматизации и управления | **1.2.** Основы визуального программирования интерфейса | **1.** Введение. Объектно-ориентированное программирование |
| | | | **2.2.** Модульное программирование | **2.** Основные направления в программировании |
| | | | **2.4.** Объектно-ориентированное програмирование | |
| | | | **3.4.** Функции-элементы | **3.** Классы |
| | | | **3.5.** Дружественные функции, константные функции | |
| | | | **4.4.** Типизация. Параллелизм | **4.** Базовые принципы объектно-ориентированного программирования |
| | | | **4.5.** Устойчивость | |
| | | | **4.6.** Иерархия классов | |
| | | | **5.3.** Механизм наследования | **5.** Базовые конструкции объектно-ориентированных программ |
| | | | **5.4.** Обработка исключений | |
| | | | **6.2.** Конструктор без параметров | **6.** Данные и функции класса |
| | | | **6.3.** Конструктор с параметрами | |
| | | | **6.4.** Конструктор с параметрами по умолчанию | |
| | | | **6.5.** Конструктор с инициализатором элементов | |
| | | | **6.6.** Деструктор | |
| | | | **7.2.** Наследование классов | **7.** Наследование и полиморфизм. Виртуальные функции, абстрактные классы |
| | | | **8.2.** События | **8.** Особенности классов, наследующих классам библиотеки компонентов C++ Builder |

### 3. Описание показателей и критериев оценивания компетенций

| Показатели | Оценка | Критерии |
|---|---|---|
| **Знать**<br>(ОПК-9):<br>- основные требования информационной безопасности;<br>(ПК-6):<br>**-** основные принципы и методологию разработки прикладного программного обеспечения.<br>**Уметь**<br>(ОПК-9):<br>- использовать навыки работы с компьютером; | **отлично** | Обучающийся должен во время ответа показать знания: основных закономерностей визуального программирования интерфейса, основных типов программирования, объявления классов, основных терминов, используемых в научно-технической литературе по объектно-ориентированному программированию. Обучающийся должен иметь навыки владения: методами информационных технологий, понимания материала и способности высказывания мыслей на научно-техническом языке. Обучающийся во время ответа должен продемонстриро- |

| | | |
|---|---|---|
| (ПК-6):<br>- выбирать стандартные средства автоматики, измерительной и вычислительной техники для проектирования систем автоматизации и управления.<br>**Владеть**<br>(ОПК-9):<br>- методами информационных технологий;<br>(ПК-6):<br>- навыками расчётов и проетирования отдельных блоков и устройств систем автоматизации и управления. | | вать умения: использовать навыки работы с компьютером, разработки и проектирования многофункционального графического интерфейса пользователя, использовать основные визуальные компоненты интегрированной среды разработки C++ Builder. |
| | **хорошо** | Ответ содержит неточности. Дополнительные вопросы требуются, но обучающийся с ними справляется отлично. |
| | **удовлетворительно** | Ответил только на один вопрос, либо слабо ответил на оба вопроса. На дополнительные вопросы отвечает неуверенно. |
| | **неудовлетворительно** | На оба вопроса обучающийся отвечает неубедительно. На дополнительные вопросы преподавателя также не может ответить. |

## 4. Методические материалы, определяющие процедуры оценивания знаний, умений, навыков и опыта деятельности

Дисциплина «Прикладное программирование» направлена на формирование у обучающихся знаний и навыков использования современных технологий и методов разработки программных систем для решения практических задач с использованием современных инструментальных средств, необходимых в дальнейшем, при проектировании и эксплуатации систем управления и автоматизации.

Изучение дисциплины предусматривает:

- лекции;
- лабораторные работы;
- самостоятельную работу;
- экзамен.

В ходе освоения раздела 1 «Введение. Объектно-ориентированное программирование» обучающиеся должны уяснить: основные понятия и определения ООП (методы, свойства, объект), два подхода к визуальному программированию интерфейса.

В ходе освоения раздела 2 «Основные направления в программировании» обучающиеся должны знать: три направления в программировании, концепцию типов данных.

В ходе освоения раздела 3 «Классы» обучающиеся должны уяснить: синтаксис объявления класса, создание объектов класса, функции-элементы.

В ходе освоения раздела 4 «Базовые принципы объектно-ориентированного программирования» обучающиеся должны знать: семь принципов ООП, иерархическую структуру объектно-ориентированной программы.

В ходе освоения раздела 5 «Базовые конструкции объектно-ориентированных программ» обучающиеся должны уяснить: взаимоотношения между объектами и классами, интерфейс и реализацию объекта, принцип наследования, метод обработки исключений.

В ходе освоения раздела 6 «Данные и функции класса» обучающиеся должны знать: назначение и описание статических данных-элементов, три типа конструкторов, деструктор

В ходе освоения раздела 7 «Наследование и полиморфизм. Виртуальные функции, абстрактные классы» обучающиеся должны уяснить: правила доступа к элементам базового класса из производного класса, назначение виртуальных и абстрактных функций.

В ходе освоения раздела 8 «Особенности классов, наследующих классам библиотеки компонентов C++ Builder» обучающиеся должны знать: синтаксис объявления свойства и события.

В процессе проведения лабораторных работ происходит закрепление знаний, формирование умений и навыков использования ИСР C++ Builder для разработки объектно-ориентированного обеспечения.

При подготовке к экзамену рекомендуется особое внимание уделить следующим вопросам: синтаксис объявления класса, объекта, свойства, события, метод обработки исключений.

Работа с литературой является важнейшим элементом в получении знаний по дисциплине. Прежде всего, необходимо воспользоваться списком рекомендуемой литературы. Дополнительные сведения по изучаемым темам можно найти в периодической печати и Интернете.

Предусмотрено проведение аудиторных занятий в интерактивной форме (лекции с текущим контролем, лабораторные работы с разбором конкретных ситуаций) в сочетании с внеаудиторной работой.

**Приложение 2**

# АННОТАЦИЯ
## рабочей программы дисциплины
### Прикладное программирование

**1. Цель и задачи дисциплины**

Целью дисциплины является изложение базовых принципов объектно-ориентированного программирования в объёме, необходимом для успешного использования современных интегрированных пакетов визуального программирования при проектировании и разработке графических интерфейсов пользователя.

Задачей дисциплины является подготовка обучающихся к самостоятельной работе по решению практических задач, связанных с разработкой графического интерфейса пользователя в интегрированной среде проектирования для решения технических задач, стоящих перед специалистом.

**2. Структура дисциплины**

2.1 Распределение трудоемкости по отдельным видам учебных занятий, включая самостоятельную работу: Лк – 17 час.; ЛР – 34 час.; СР – 66час.

Общая трудоёмкость дисциплины составляет 144 часа, 4 зачётных единицы.

2.2 Основные разделы дисциплины:

1. Введение. Объектно-ориентированное программирование;
2. Основные направления в программировании;
3. Классы;
4. Базовые принципы объектно-ориентированного программирования;
5. Базовые конструкции объектно-ориентированных программ;
6. Данные и функции класса;
7. Наследование и полиморфизм. Виртуальные функции, абстрактные классы;
8. Особенности классов, наследующих классам библиотеки компонентов C++ Builder.

**3. Планируемые результаты обучения (перечень компетенций)**

Процесс изучения дисциплины направлен на формирование следующей компетенции:
ОПК-9 - способность использовать навыки работы с компьютером, владеть методами информационных технологий, соблюдать основные требования информационной безопасности;
ПК-6 - способность производить расчёты и проетирование отдельных блоков и устройств систем автоматизации и управления и выбирать стандартные средства автоматики, измерительной и вычислительной техники для проектирования систем автоматизации и управления.

**4. Вид промежуточной аттестации**: экзамен.

**Приложение 3**

***Протокол о дополнениях и изменениях в рабочей программе***
***на 201___ - 201___ учебный год***

**1.** В рабочую программу по дисциплине вносятся следующие дополнения:
______________________________________________________________________________
______________________________________________________________________________

**2.** В рабочую программу по дисциплине вносятся следующие изменения:
______________________________________________________________________________
______________________________________________________________________________
______________________________________________________________________________

Протокол заседания кафедры №____ от «___» __________ 201 ____ г.,
*(разработчик)*

Заведующий кафедрой ______________________ ______________________
*(подпись)* *(Ф.И.О.)*