**Тема 1. Основные понятия и определения**

**Технология программирования** — совокупность методов и средств, применяемых в процессе разработки программного обеспечения.

**Программа (program, routine)** — упорядоченная последовательность команд (инструкций) компьютера для решения задачи.

**Программное обеспечение (software)** — совокупность программ обработки данных и необходимых для их эксплуатации документов.

**Задача (problem, task)** — проблема, подлежащая решению.

**Приложение (application)** — программная реализация на компьютере решения задачи-

Термин «задача» в программировании означает единицу работы вычислительной системы, требующую выделения вычислительных ресурсов (процессорного времени, памяти).

Процесс создания программ можно представить как последовательность следующих действий:

1) постановка задачи;

2) алгоритмизация решения задачи;

3) программирование.

**Постановка задачи (problem definition)** — это точная формулировка решения задачи на компьютере с описанием входной и выходной информации.

**Алгоритм** — система точно сформулированных правил, определяющая процесс преобразования допустимых исходных данных (выходной информации) в желаемый результат (выходную информацию) за конечное число шагов.

**Программирование (programming)** — теоретическая и практическая деятельность, связанная с созданием программ.

По отношению к ПО компьютерные пользователи делятся на следующие группы:

1) системные программисты. Занимаются разработкой, эксплуатацией и сопровождением системного программного обеспечения;

2) прикладные программисты. Осуществляют разработку и отладку программ для решения различных прикладных задач;

3) конечные пользователи. Имеют элементарные навыки работы с компьютером и используемыми ими прикладными программами;

4) администраторы сети. Отвечают за работу вычислительных сетей;

5) администраторы баз данных. Обеспечивают организационную поддержку базы данных.

**Сопровождение программы** — поддержка работоспособности программы, переход на ее новые версии, внесения изменений, исправление ошибок и т. д.

**Основные характеристики программ:**

1) алгоритмическая сложность;

2) состав функций обработки информации;

3) объем файлов, используемых программой;

4) требования к операционной системе (ОС) и техническим средствам обработки, в том числе объем дисковой памяти, размер оперативной памяти для запуска программы, тип процессора, версия ОС, наличие вычислительной сети и т. д.

**Показатели качества программы:**

1) мобильность (многоплатформенность) — независимость от технического комплекса системы обработки данных, ОС, сетевых возможностей, специфики предметной области задачи и т. д.;

2) надежность — устойчивость, точность выполнения предписанных функций обработки, возможность диагностики возникающих ошибок в работе программы;

3) эффективность как с точки зрения требований пользователя, так и расхода вычислительных ресурсов;

4) учет человеческого фактора — дружественный интерфейс, контекстно-зависимая подсказка, хорошая документация;

5) модифицируемость — способность к внесению изменений, например, расширение функций обработки, переход на другую техническую базу обработки и т. п.

6) коммуникативность — максимально возможная интеграция с другими программами, обеспечение обмена данными между программами.

Все программы по характеру использования и категориям пользователей можно разделить на два класса — утилитарные программы и программные продукты.

**Утилитарные программы** («программы для себя») предназначены для удовлетворения нужд их разработчиков. Чаще всего такие программы выполняют роль отладочных приложений, являются программами решения задач, не предназначенных для широкого распространения.

**Программные продукты (изделия)** используются для удовлетворения потребностей пользователей, широкого распространения и продажи.

В настоящее время существуют и другие варианты легального распространения программных продуктов, которые появились с использованием глобальных телекоммуникаций:

* freeware — бесплатные программы, свободно распространяемые, поддерживаются самим пользователем, который правомочен вносить в них необходимые изменения;
* shareware некоммерческие (условно-бесплатные) программы, которые могут использоваться, как правило, бесплатно. Ряд производителей использует OEM-программы (Original Equipment Manufacturer), т. е. встроенные программы, устанавливаемые на компьютеры или поставляемые вместе с компьютерами.

Программный продукт должен быть соответствующим образом подготовлен к эксплуатации (отлажен), иметь необходимую техническую документацию, предоставлять сервис и гарантию надежной работы программы, иметь товарный знак изготовителя, а также наличие кода государственной регистрации.

**Тема 2. Классификация программного обеспечения**

Можно выделить три класса ПО:

1) системное;

2) пакеты прикладных программ (прикладное ПО);

3) инструментарий технологии программирования (инструментальные средства для разработки ПО).

**Системное ПО** направлено:

• на создание операционной среды функционирования других программ;

• обеспечение надежной и эффективной работы самого компьютера и вычислительной сети;

• проведение диагностики и профилактики аппаратуры компьютера и вычислительных сетей;

• выполнение вспомогательных технологических процессов (копирование, архивация, восстановление файлов программ и БД и т. п.).

**Системное ПО** (System Software) — совокупность программ и программных комплексов для обеспечения работы компьютера и вычислительных сетей.

**Прикладное ПО** служит программным инструментарием решения функциональных задач и является самым многочисленным классом ПО. В данный класс входят программные продукты, выполняющие обработку информации различных предметных областей. Таким образом, прикладное ПО — комплекс взаимосвязанных программ для решения задач определенного класса предметной области.

**Инструментарий технологии программирования** обеспечивает процесс разработки программ и включает специализированное ПО, которое является инструментальным средством разработки. ПО данного класса поддерживает все технологические этапы процесса проектирования, программирования, отладки и тестирования создаваемых программ. Пользователями данного ПО являются системные и прикладные программисты.

**Прикладное программное обеспечение**

Прикладные программы предназначены для того, чтобы обеспечить применение вычислительной техники в различных сферах человеческой деятельности. Одним из вариантов классификации прикладного ПО может быть следующий.

|  |  |  |
| --- | --- | --- |
| **Прикладное программное обеспечение** | | |
| **общего назначения** | **специального назначения** | **профессионального уровня** |
| Текстовые редакторы  Издательские системы  Графические системы  Табличные процессоры  СУБД  Интегрированные системы | Авторские системы  Экспертные системы  Гипертекстовые системы  Системы мультимедиа | АСНИ (автоматизированные системы научных исследований)  САПР (системы автоматизированного проектирования)  АСУ (автоматизированные систему управления)  АРМ (автоматизированное рабочее место)  Системы телекоммуникаций |

**Инструментальные программные средства общего назначения**

Несмотря на широкие возможности использования компьютеров для обработки разной информации, самыми популярными являются программы, предназначенные для работы с текстами - **текстовые редакторы** и **издательские системы**. Текстовыми редакторами называют программы для ввода, обработки, хранения и. печати текстовой информации в удобном для пользователя виде. Эксперты оценивают использование компьютера в качестве печатающей машинки в 80%.

Профессиональные системы для обработки текстов, используемые в издательском деле, называются **издательскими системами**. Настольные компьютерные издательские системы приобрели широкую популярность в различных сферах производства, бизнеса, науки, культуры и образования. Издательское дело становится актуальным практически для любой организации. Выпуск информационных бюллетеней, рекламных проспектов, собственных малотиражных газет и даже книг теперь становится необходимым атрибутом информационного обеспечения современных учреждений. Пожалуй из всех новых информационных технологий, компьютерная подготовка издания является наиболее массовой и практически легко внедряемой.

Большую популярность приобрели программы обработки графической информации. Компьютерная графика в настоящее время является одной из самых динамично развивающихся областей программного обеспечения. Она включает в себя ввод, обработку и вывод графической информации - чертежей, рисунков, картин, текстов и т.д. средствами компьютерной техники. Различные типы **графических систем** позволяют быстро строить изображения, вводить иллюстрации с помощью сканера или видеокамеры, создавать анимационные ролики.

Графические редакторы позволяют пользоваться различным инструментарием художника, стандартными библиотеками изображений, наборами стандартных шрифтов, редактированием изображений, копированием и перемещением фрагментов по страницам экрана и др.

Для выполнения расчетов и дальнейшей обработки числовой информации существуют специальные программы - табличные процессоры. В процессе деятельности любого специалиста часто требуется, представить результаты работы в виде таблиц, где одна часть полей занята исходными данными, а другая - результатами вычислений и графического анализа. Характерными для них является большой объем перерабатываемой информации, необходимость многократных расчетов при изменении исходных данных. Автоматизацией подобной рутинной работы и занимаются табличные процессоры.

Одним из наиболее перспективных направлений развития вычислительной техники является создание специальных аппаратных средств для хранения гигантских массивов информационных данных и последующей нечисловой обработки их - поиска и сортировки. Для компьютерной обработки подобных **баз данных** используют **системы управления базами данных**. **СУБД** - это набор средств программного обеспечения, необходимых для создания, обработки и вывода записей баз данных.

Желание объединить функции различных прикладных программ в единую систему привело к созданию интегрированных систем. Универсальные **интегрированные системы** разрабатывались по принципу единой системы, содержащей в качестве элементов текстовые и графические редакторы, электронные таблицы и систему управления базами данных. Примеры: Framework, Works, Мастер. Современная концепции интеграции программных средств - кооперация отдельных прикладных программных систем по типу широко известного пакета MS Office. Сами системы, входящие в пакет, являются независимыми, более того, они сами представляют локально интегрированный пакет, поскольку помимо основной своей задачи поддерживают функции других систем. Например, текстовый редактор Word обладает возможностью манипулировать с электронными таблицами и базами данных, а в электронной таблице Excel встроен мощный текстовый редактор. Для сопряжения, информационных данных из различных программных систем в них предусматривают импорт - экспортную систему обмена с перекодировкой форматов представления данных.

**Характеристика интегрированного пакета:**

1. Полнота информационных технологий для конечных пользователей;
2. Однотипный интерфейс конечного пользователя для всех программ, входящих в состав интегрированного пакета;
3. Общий сервис для программ интегрированного пакета (словарь, средства орфографии, построение диаграмм, конвертирование данных);
4. Легкость обмена и ссылок на объекты, создание программами интегрированного пакета.

**Инструментальные программные средства специального назначения**

Разработчики создают специальные программные системы целевого назначения для специалистов в некоторой предметной области. Такие программы называют авторскими инструментальными системами. **Авторская система** представляет интегрированную среду с заданной интерфейсной оболочкой, которую пользователь может наполнить информационным содержанием своей предметной области.

**Экспертная система** - это программа, которая ведет себя подобно эксперту в некоторой узкой прикладной области. Экспертные системы призваны решать задачи с неопределенностью и неполными исходными данными, требующие для своего решения экспертных знаний.

Кроме того, эти системы должны уметь объяснять свое поведение и свое решение.

Принципиальным отличием экспертных систем от других программ является их адаптивность, т.е. изменчивость в процессе самообучения.

Принято выделять в экспертных системах три основных модуля: базы знаний; логического вывода; интерфейс с пользователем.

Экспертные системы, являющиеся основой искусственного интеллекта, получили широкое распространение: в науке (классификация животных и растений по видам, химический анализ), в медицине (постановка диагноза, анализ электрокардиограмм, определение методов лечения), в технике (поиск неисправностей в технических устройствах, слежение за полетом космических кораблей и спутников), в политологии и социологии, криминалистике, лингвистике и т.д.

В последнее время широкую популярность получили, **программы обработки гипертекстовой информации**. Гипертекст — это форма организации текстового материала не в линейной последовательности, а в форме указаний возможных переходов (ссылок), связей между отдельными его фрагментами. В обычном тексте используется обычный линейный принцип размещения информации и доступ к нему осуществляется последовательно. В **гипертекстовых системах** информация напоминается текст энциклопедии, и доступ к любому выделенному фрагменту текста осуществляется произвольно по ссылке. Организация информации в гипертекстовой форме используется при создании справочных пособий, словарей, контекстной помощи (Help) в прикладных программах.

Расширение концепции гипертекста на графическую и звуковую информацию приводит к понятию гипермедиа. Идеи **гипермедиа** получили распространение в сетевых технологиях, в частности в Интернет - технологиях. Технология WWW (World Wide Web) позволила структурировать громадные мировые информационные ресурсы посредством гипертекстовых ссылок. Появились программные средства, позволяющие создавать подобные веб-странички. Стали развиваться механизмы поиска нужной информации в лабиринте информационных потоков. Популярными поисковыми средствами в Интернете являются: Yahoo, AltaVista, Magellan, Rambler и др.

**Мультимедиа** (multimedia) - это взаимодействие визуальных и аудиоэффектов под управлением интерактивного программного обеспечения. Появление и широкое распространение компакт-дисков (CD-ROM) сделало эффективным использование систем мультимедиа в рекламной и информационной службе, сетевых телекоммуникационных технологиях, обучении.

Мультимедийные игровые и обучающие системы начинают вытеснять традиционные «бумажные библиотеки». Сегодня в библиотеках CD-ROM можно «гулять» по музеям, Московскому Кремлю и т.д. с помощью «электронного путеводителя».

**Программные средства профессионального уровня**

Каждая прикладная программа этой группы ориентируется на достаточно узкую предметную область, но проникает в нее максимально глубоко. Так функционируют **АСНИ** – автоматизированные системы научных исследований, каждая из которых «привязана» к определенной области науки, **САПР** – системы автоматизированного проектирования, **АСУ** — автоматизированные системы управления (которых в 1960-1970-х гг. были разработаны тысячи) **АСУ ТП** - АСУ технологическими процессами. **АРМ** – автоматизированное рабочее место.

Существуют педагогические комплексы для обеспечения процесса обучения и другие ПС специального назначения.

Наконец, еще раз подчеркнем не только условность предложенной выше классификации, но и наличие пересечений. Так, каждую конкретную экспертную систему вполне можно отнести к ППО профессионального уровня; принцип гипертекста реализован в ряде авторских систем и т.д.

**Пакет прикладных программ** (application program package) – комплекс взаимосвязанных программ для решения задач определенного класса конкретной предметной области.

Проблемно-ориентированные пакеты;

Методо-ориентированные пакеты;

Общего назначения;

Автоматизированного проектирования;

Офисные ПП;

Программные средства мультимедиа;

Настольные издательские системы;

Интеллектуальные системы;

**Проблемно-ориентированные (ПОС):**

Самый представительный. Внутри проводят классификацию по разным признакам (По типам предметных областей, информационным системам, функциям и комплексам задач, реализации программным способом.). Пакеты прикладных программ управляют материальными запасами, управляют производством. Для некоторых предметных областей возможна типизация функций управления, структуры данных и алгоритмов обработки. Это вызвало разработку значительного количества ППП одинакового функционального назначения:

* ППП автоматизированного бухгалтерского учета;
* ППП финансовой деятельности;
* ППП управления персоналом;
* ППП управления производством;
* банковские информационные системы и т. п.

Основные тенденции развития:

* создание программных комплексов в виде автоматизированных рабочих мест (АРМ) управленческого персонала;
* создание интегрированных систем управления предметной областью на базе вычислительных сетей, объединяющих АРМы;
* организация данных больших информационных систем в виде распределенной БД на сети ЭВМ;
* наличие простых языков средств конечного пользователя и др.

**Методо-ориентированный пакет прикладных программ**

Обеспечивает независимо от предметной области и функции информационных систем, математические, статистические и другие методы решения задач;

Наиболее распространены методы математического программирования, решения дифференцированных уравнений, имитационное моделирование, исследовательские операции;

Методы статистической обработки анализа данных (раздел описательной статистики, регрессивный анализ, прогнозирование технико-экономических данных и т.д.) имеют все возрастающее применение.

**Офисные пакеты прикладных программ (ППП):** обеспечивают организационное управление деятельностью офиса (органайзеры). Наиболее часто подобное ПО разрабатывается для ноутбуков;

ПО для планирование рабочего времени, составление протоколов встреч, расписаний, ведения записной и телефонной книжки; калькулятор;

Программы-переводчики, средства проверки орфографии;

Коммуникационные программные продукты (предназначены для организации взаимодействия пользователей с удаленными абонентами или информационными ресурсами сети.

**Системы Искусственно Интеллекта (ИИ)**

Данный класс ПП реализует отдельные функции интеллекта человека.

Основными компонентами системы ИИ являются:

База знаний;

Интеллектуальный интерфейс с пользователем;

Программа формирования логических выводов.

Разработка интеллектуальных систем ведется по следующим направлениям:

Программные оболочки для создания экспертных систем путем наполнения баз знаний и правил логического вывода;

Готовые экспертные системы для принятия решений в рамках определения предметных областей;

Системы управления базами знаний для поддержания семантических моделей;

Системы анализа и распознания речи.

**Тема 3. Жизненный цикл программы**

В современном мире всеобщей компьютеризации и информатизации требования, предъявляемые к программному обеспечению (ПО) вообще и к программным продуктам (ПП), программным средствам (ПС) и программам в частности, весьма высоки. В связи с этим обеспечение удовлетворяющих пользователя потребительских качеств программы, таких как надежность, быстродействие, соответствие заявленным возможностям, полнота документации, возможности расширения, развития и т. д., без строгого соблюдения определенной технологии практически невозможно.

В основе разработки и дальнейшего применения программного обеспечения пользователем лежит понятие жизненного цикла, который, в сущности, является моделью его создания и использования, отражающей различные состояния, начиная с момента осознания необходимости появления данного ПО и заканчивая моментом его полного выхода из употребления [5].

Структуру жизненного цикла ПП, состав процессов, действия и задачи, которые должны быть выполнены во время создания ПП, определяет и регламентирует международный стандарт ISO/IEC 12207 Standard for Information Technoiogy — Software Life Cycle Processes (Стандарт информационных технологий – процессы жизненного цикла программ).

Этот международный стандарт определяет действия, которые могут быть выполнены на протяжении жизненного цикла программного обеспечения.

В России, начиная с 1970-х годов, создание ПП регламентировалось стандартами ЕСПД (Единая система программной документации, серия ГОСТ 19.ХХХ), которые были ориентированы на класс относительно простых программ, небольшого объема, создаваемых отдельными программистами. В настоящее время указанные стандарты устарели концептуально и по форме, их сроки действия закончились и дальнейшее использование нецелесообразно. Поэтому в отечественных разработках следует использовать современные международные стандарты.

В соответствии с ISO/IEC 12207 выделяют 5 основных процессов жизненного цикла программных продуктов, 8 вспомогательных процессов и 4 организационных процесса.

**Основные процессы жизненного цикла**

**1).** **Процесс приобретения.** Определяет действия предприятия-покупателя, которое приобретает систему, программный продукт или сервис программного обеспечения. Этот процесс состоит из следующих действий:

1. инициирование: определение заказчиком своих потребностей, анализ требований к системе, принятие решения о приобретении, разработке или усовершенствовании существующего ПП, проверка наличия необходимой документации, подготовка и утверждение плана приобретения, включающего требования к системе, тип договора, ответственность сторон и т.д.;
2. заявка на подготовку предложения, содержащая требования к разрабатываемой или покупаемой системе, перечень необходимых ПП, технические ограничения;
3. подготовка контракта и модернизация: выбор поставщика, заключение договора;
4. текущий контроль (мониторинг) поставщика;
5. принятие и завершение.

**2). Процесс поставки.** Определяет действия предприятия-поставщика, которое снабжает покупателя системой, программным продуктом или сервисом программного обеспечения. Этот процесс состоит из следующих действий:

1. Инициирование (начало): рассмотрение, дополнение и утверждение поставщиком заявочных предложений;
2. Подготовка ответа;
3. Заключение контракта;
4. Планирование;
5. Выполнение и контроль;
6. Оценка и проверка;
7. Поставка и завершение.

**3). Процесс разработки**. Определяет действия предприятия-разработчика, которое разрабатывает принцип построения программного изделия и программный продукт. Этот процесс состоит из действий:

1. Реализация процесса.
2. Анализ системных требований.
3. Проектирование архитектуры системы.
4. Анализ требований программного обеспечения.
5. Архитектура программного обеспечения.
6. Детальное конструирование программного обеспечения.
7. Кодирование и тестирование программного обеспечения.
8. Интеграция программного обеспечения.
9. Квалификационные испытания программного обеспечения.
10. Интеграция системы.
11. Квалификационные испытания системы.
12. Установка программного обеспечения.
13. Поддержка принятия программного обеспечения.

**4). Процесс эксплуатации**. Определяет действия персонала эксплуатации, который обеспечивает обслуживание вычислительной системы в процессе ее функционирования в интересах пользователей. Этот процесс содержит следующие действия:

1. Подготовительная работа: планирование работ, выполняемых в процессе эксплуатации, установка эксплуатационных стандартов, определение процедур локализации и решения проблем, возникающих в процессе эксплуатации.
2. Операционное (эксплуатационное) тестирование.
3. Функционирование системы.
4. Поддержка пользователя.

**5). Процесс сопровождения.** Представляет собой управление модификациями программно продукта, поддержку его текущего состояния и функциональную пригодность и включает в себя инсталляцию и удаление программного изделия на вычислительной системе. Этот процесс активизируется, когда программный продукт подвергается модификациям для кодирования и связывания (совмещения документации), вызванными проблемой или потребностью улучшения и адаптации. Цель - изменить существующий программный продукт, сохраняя его целостность. Этот процесс включает инсталляцию и удаление программного продукта. Процесс заканчивается удалением программного продукта.

**Вспомогательные процессы жизненного цикла**

Вспомогательный процесс поддерживает реализацию другого процесса, будучи неотъемлемой частью всего жизненного цикла программного изделия, с определенной целью и обеспечивает должное качество проекта программного обеспечения. Вспомогательный процесс используется и выполняется по мере необходимости и инициируется другим процессом.

**1). Процесс документирования.** Это процесс для записи информации, произведенной процессом жизненного цикла или деятельностью. Процесс содержит набор действий, которые планируют, проектируют, разрабатывают, производят, редактируют, распределяют и сопровождают те документы, в которых нуждаются все заинтересованные лица, такие как менеджеры, инженеры и пользователи системы или программного продукта. Включает следующие действия: подготовительная работа, проектирование и разработка документации, выпуск документации, сопровождение (обновление) документации.

**2). Процесс управления конфигурацией.** Конфигурация – совокупность функциональных и физических характеристик, установленных в технической документации и реализованных в ПП. Процесс управления конфигурацией - это процесс применения административных и технических процедур на всем протяжении жизненного цикла программного обеспечения для:

* идентификации, определения и базирования единиц программного обеспечения в системе;
* управления модификацией и выпуском изделий;
* записи и сообщения о состоянии изделия;
* управление и контролирования хранения, обращения (ухода) и поставок изделий.

**3). Процесс обеспечения качества**. Определяет действия для объективной гарантии, что программные продукты и процессы соответствуют определенным требованиям к ним и придерживаются установленным замыслам. Чтобы быть беспристрастной, гарантия качества должна иметь организационную свободу и власть (полномочия) от людей, непосредственно ответственных за разработку программного продукта или выполнение процесса в проекте. Гарантия качества может иметь внутреннюю или внешнюю зависимость от того, демонстрируются ли доказательства качества продукта или процесса к управлению поставщика или покупателя.

Гарантия качества может использовать результаты других обеспечивающих процессов, таких как Верификация, Аттестация, Совместный Обзор, Проверки и Разрешение проблем.

**4). Процесс верификации**. Это процесс для определения, выполняют ли программные продукты действия требования или условия, наложенные на них в предыдущих действиях. Для эффективности затрат и выполнения, верификация должна быть интегрирована как можно раньше с процессом (таким как поставка, разработка, финансирование или сопровождение), который использует ее. Этот процесс может включать анализ, обзор и испытание. Этот процесс может быть выполнен с различными степенями независимости. Степень независимости может колебаться от одного и тогоже лица или другого лица в одной и той же организации до лица в другой организации с различными степенями разделения. В случае когда процесс выполнен администрацией, независящей от поставщика, оператора, разработчика или персонала сопровождения, он называется Независимый процесс верификации.

**5). Процесс аттестации**. Представляет собой процесс определения соответствия требований и конечной технической системы или программного продукта их функциональному использованию. Аттестация проводится на ранних стадиях. Этот процесс может проходить как часть Обеспечения Принятия Программного Обеспечения. Этот процесс может выполняться с различной степенью независимости. В случае, когда процесс выполняется организацией, независимой от поставщика, разработчика, персонала эксплуатации или персонала сопровождения, он называется Независимым Процессом Аттестации.

**6). Процесс совместной оценки.** Определяет действия для оценки состояния и результатов какого либо действия. Совместные оценки применяются в управлении проектом , в технических уровнях и проводятся в срок действительности контракта. Этот процесс может быть использован любыми двумя сторонами, где одна сторона (проверяющая, рецензирующая) проверяет (рецензирует) другую сторону (проверяемую) на совместном форуме.

**7). Процесс проверки (аудита).** Определяет деятельность для определения соответствия с требованиями, замыслами и контрактом. Этот процесс может быть использован любыми двумя сторонами, где одна сторона (проверяющая) проверяет программные продукты или деятельность другой стороны (проверяемой). Аудит служит для установления соответствия реальных работ и отчетов требованиям, планам и контракту. Аудиторы (ревизоры) не должны иметь прямой зависимости от разработчиков ПП. Они оценивают состояние работ, использование ресурсов, соответствие документации спецификациям и стандартам, корректность проводимого тестирования.

**8). Процесс решения проблем.** Определяет процесс анализа и устранения проблем (включая несоответствия), какова бы ни была их природа или источник, которые были обнаружены на протяжении разработки, эксплуатации, сопровождения или других процессов.

**Организационные процессы жизненного цикла**

Организационные процессы жизненного цикла выполняются какой-либо организацией с целью создания и обеспечения деятельности какой-либо нижестоящей структуры, включающей в себя связанные процессы жизненного цикла и персонал и совершенствования структуры и процессов. Они, как правило, инвариантны относительно конкретных проектов и контрактов, однако, уроки, извлеченные из таких проектов и контрактов, способствуют совершенствованию организации. Организационные процессы включают в себя:

1). Процесс управления. Определяет основную деятельность управления, включая проектный менеджмент: управление выпуском продукта, проектом и задачами соответствующих процессов, таких как приобретение, поставка, разработка, эксплуатация, сопровождение и др.

2). Процесс создания инфраструктуры. Охватывает выбор и поддержку технологии, стандартов и инструментальных средств, выбор и установку программных и аппаратных средств, используемых для разработки, эксплуатации, сопровождения ПП. Инфраструктура должна модифицироваться и сопровождаться в соответствии с изменениями требований к соответствующим процессам. Инфраструктура в свою очередь, является одним из объектов управления конфигурацией.

3). Процесс усовершенствования. Определяет основные действия, которые организация (покупатель, поставщик, разработчик, оператор или менеджер другого процесса) выполняет для создания, оценки, управления и совершенствования их процесса жизненного цикла. Усовершенствование направлено на повышение производительности труда за счет совершенствования используемой технологии, методов управления, выбора инструментальных средств и обучения персонала. Усовершенствование основано на анализе достоинств и недостатков каждого процесса. Такому анализу в большой степени способствует накопление в организации исторической, технической и иной информации по реализованным проектам.

4). Процесс обучения (пункт 7.4). Охватывает первоначальное обучение и последующее повышение квалификации персонала. Содержание процесса обучения определяется требованиями к проекту. Разработка учебных материалов является неотъемлемой частью процесса обучения, так как позволяет существенно повысить его эффективность и качество.

**Взаимосвязь между процессами жизненного цикла**

Процессы ЖЦ ПО, регламентируемые стандартом ISO/IEC 12207, могут использоваться различными организациями в конкретных проектах самым различным образом. Тем не менее, стандарт предлагает некоторый базовый набор взаимосвязей между процессами с различных точек зрения (или в различных аспектах).

  
Рисунок Связи между процессами жизненного цикла

В *договорном аспекте* заказчик и поставщик вступают в договорные отношения и реализуют соответственно процессы приобретения и поставки.

В *аспекте управления* заказчик, поставщик, разработчик, оператор, служба сопровождения и другие участвующие в ЖЦ ПО стороны управляют выполнением своих процессов.

В *аспекте эксплуатации* оператор, эксплуатирующий систему, предоставляет необходимые услуги пользователям. В *инженерном аспекте* разработчик или служба сопровождения решают соответствующие технические задачи, разрабатывая или модифицируя программные продукты. В *аспекте поддержки* службы, реализующие вспомогательные процессы, предоставляют необходимые услуги всем остальным участникам работ. В рамках аспекта поддержки можно выделить аспект управления качеством ПО, включающий пять процессов: обеспечение качества, верификация, аттестация, совместная оценка и аудит. Организационные процессы, показанные в нижней части рис. 1.2, выполняются на корпоративном уровне или на уровне всей организации в целом, создавая базу для реализации и постоянного совершенствования остальных процессов ЖЦ ПО.

Процессы и реализующие их организации (или стороны) связаны между собой чисто функционально, при этом внутренняя структура и статус организаций никак не регламентируются. Одна и та же организация может выполнять различные роли: поставщика, разработчика и другие, и наоборот, одна и та же роль может выполняться несколькими организациями.

Взаимосвязи между процессами, описанные в стандарте, носят статический характер. Более важные динамические связи между процессами и реализующими их сторонами устанавливаются в реальных проектах.

Значение данного стандарта трудно переоценить, поскольку он формирует подход к выбору и оценке всех современных технологий и процессов создания и сопровождения ПО. Безусловно, на выбор конкретной технологии в проекте влияет целый ряд факторов, но принципы реализации и состав процессов ЖЦ ПО остаются стабильными. Большинство технологий, поставляемых ведущими производителями (IBM, Oracle, Microsoft и др.), соответствуют требованиям этого стандарта. Анализ различных технологий показывает, что общие принципы описания процессов ЖЦ ПО в стандарте ISO 12207 прошли практическую апробацию и стали общепризнанными.

**Модульное программирование**

Для обеспечения технологичности разрабатываемого программного обеспечения применяется модульное программирование [37].

Приступая к разработке программы, следует иметь в виду, что она, как правило, является большой системой, поэтому необходимо принять меры для ее упрощения. Для этого программу разрабатывают по частям, которые называются программными модулями. Такой метод создания программ называют модульным программированием.

Модульное программирование основано на понятии модуля — программы или функционально завершенного фрагмента программы.

Модуль характеризуют:

• один вход и один выход. На входе программный модуль получает определенный набор исходных данных, выполняет их обработку и возвращает один набор выходных данных;

• функциональная завершенность. Модуль выполняет набор определенных операций для реализации каждой отдельной функции, достаточных для завершения начатой обработки данных;

• логическая независимость. Результат работы данного фрагмента программы не зависит от работы других модулей;

• слабые информационные связи с другими программными модулями. Обмен информацией между отдельными модулями должен быть минимален;

• размер и сложность программного элемента в разумных рамках.

Таким образом, модули содержат описание исходных данных, операции обработки данных и структуры взаимосвязи с другими модулями.

Программный модуль является самостоятельным программным продуктом. Это означает, что каждый программный модуль разрабатывается, компилируется и отлаживается отдельно от других модулей программы. Более того, каждый разработанный программный модуль может включаться в состав разных программных систем при условии выполнения требований, предъявляемых к его использованию в документации к этому модулю. Таким образом, программный модуль может рассматриваться и как средство упрощения сложных программ, и как средство накопления и многократного использования программистских знаний.

**Основные характеристики программного модуля**

1. **Размер модуля** измеряется числом содержащихся в нем операторов. Модуль не должен быть слишком маленьким или слишком большим. Большие модули, как правило, сложны для понимания и неудобны для внесения изменений, они могут существенно увеличить суммарное время повторных трансляций программы при отладке. Маленькие модули усложняют общую структурную схему программы и могут не окупать накладных расходов, связанных с их оформлением. Обычно рекомендуются программные модули размером от нескольких десятков до нескольких сотен операторов.
2. **Прочность модуля** — это мера его внутренних связей. Чем выше прочность модуля, тем больше связей скрыто от внешней по отношению к нему части программы и, следовательно, тем проще сама программа. Самой слабой степенью прочности обладает **модуль, прочный по совпадению**. В данном случае в программный модуль оформляется повторяющаяся в нескольких местах программы последовательность операторов. Если вдруг возникнет необходимость изменения этой последовательности в одном из контекстов, придется изменять сам модуль, что может сделать его использование в других контекстах ошибочным. Такой класс программных модулей не рекомендуется для использования. **Функционально прочный модуль** — это модуль, реализующий одну какую-либо определенную функцию. При этом он может использовать и другие модули. Такой вид прочности модулей рекомендуется для использования.

Высшей степенью прочности обладает **информационно прочный модуль** — это модуль, выполняющий несколько операций над одной и той же структурой данных, которая неизвестна вне этого модуля. Для каждой из этих операций в таком модуле имеется свой вход со своей формой обращения к нему. Информационно прочный модуль может реализовывать, например, абстрактный тип данных.

1. **Сцепление** **модуля** — это мера его зависимости по способу передачи данных от других модулей. Чем слабее сцепление модуля с другими модулями, тем сильнее его независимость от других модулей. Для оценки степени сцепления существует шесть видов сцепления модулей по:

• данным;

• образцу;

• управлению;

• внешним ссылкам;

• общей области данных;

• содержимому.

Худшим видом сцепления модулей является сцепление по содержимому. Таким является сцепление двух модулей, когда один из них имеет прямые ссылки на содержимое другого модуля (например, на константу, содержащуюся в другом модуле). Такое сцепление модулей недопустимо.

Не рекомендуется использовать также сцепление по общей области — это такое сцепление модулей, когда несколько модулей используют одну и ту же область памяти.

Сцепление по образцу предполагает, что модули обмениваются данными, объединенными в структуры. Этот тип обеспечивает неплохие характеристики по сравнению с предыдущими. Недостаток заключается в том, что конкретные передаваемые данные «спрятаны» в структуры, и потому уменьшается «прозрачность» связи между модулями. Кроме того, при изменении структуры передаваемых данных необходимо модифицировать все использующие ее модули.

При сцеплении по управлению один модуль посылает другому некоторый информационный объект (флаг), предназначенный для управления внутренней логикой модуля. Таким способом часто выполняют настройку режимов работы программного обеспечения. Подобные настройки также снижают наглядность взаимодействия модулей и потому обеспечивают не лучшие характеристики технологичности разрабатываемого программного обеспечения.

Сцепление по внешним ссылкам предполагает, что модули ссылаются на один и тот же глобальный элемент данных [1].

Единственным видом сцепления модулей, который рекомендуется для использования современной технологией программирования, является сцепление по данным (параметрическое сцепление) — это случай, когда данные передаются модулю либо при обращении к нему как значения его параметров, либо как результат его обращения к другому модулю для вычисления некоторой функции. Такой вид сцепления модулей реализуется на языках программирования при использовании обращений к процедурам (функциям).

1. **Рутинность модуля** — это его независимость от предыстории обращений к нему. Модуль будем называть рутинным, если результат обращения к нему зависит только от значений его параметров и не зависит от результатов предыдущих обращений к нему. Модуль будем называть зависящим от предыстории, если результат обращения к нему зависит от внутреннего состояния этого модуля, хранящего следы предыдущих обращений к нему.

Считается приемлемой следующая рекомендация:

• всегда следует использовать рутинный модуль, если это не приводит к плохим сцеплениям модулей;

• зависящие от предыстории модули следует использовать только в случае, когда это необходимо для обеспечения параметрического сцепления;

• в спецификации зависящего от предыстории модуля должна быть четко сформулирована эта зависимость таким образом, чтобы было возможно прогнозировать поведение данного модуля при разных последующих обращениях к нему.

1. **Связность модулей** — мера прочности соединения функциональных и информационных объектов внутри одного модуля.

Размещение сильно связанных элементов в одном модуле уменьшает межмодульные связи, в то время как помещение сильно связанных элементов в разные модули не только усиливает межмодульные связи, но и усложняет понимание их взаимодействия. Объединение слабо связанных элементов также уменьшает технологичность модулей, делая их сложнее для понимания.

Различают следующие виды связности (в порядке убывания уровня) [1]:

• функциональную;

• последовательную;

• информационную (коммуникативную);

• процедурную;

• временную;

• логическую;

• случайную.

При функциональной связности модуль предназначен для выполнения одной функции. Его исходные данные и операции предназначены для решения одной конкретной задачи. Такой модуль имеет максимальную связность и, как следствие, хорошую технологичность (простота компиляции, тестирования, сопровождения).

При последовательной связности модуля результат обработки данных одной функцией служит исходными данными для другой функции. Такой модуль реализует одну подпрограмму, выполняющую две функции. Модуль с последовательной связностью функций можно разбить на два модуля или более, как с последовательной, так и с функциональной связностью. При этом данные, используемые последовательными функциями, также связаны последовательно. Такой модуль выполняет несколько функций, и, следовательно, его технологичность хуже с точки зрения понимания и тестирования.

Информационно связанными считают функции, обрабатывающие одни и те же данные. Информационно связанный модуль имеет неплохие показатели технологичности, так как все функции, работающие с одними и теми же данными, собраны в один модуль, что позволяет при изменении формата данных корректировать только его. Данные, которые обрабатываются одной функцией, также считают информационно связанными.

Процедурно связаны функции или данные, которые являются частями одного процесса. При процедурной связности отдельные элементы модуля связаны крайне слабо, так как реализуемые ими операции связаны лишь общим процессом, следовательно, технологичность такого модуля хуже, чем у предыдущих.

Временная связность функций подразумевает, что эти функции выполняются параллельно или в течение некоторого периода времени. Временная связность данных означает, что они используются в некотором временном интервале. Отличительной особенностью временной связности является то, что действия, реализуемые такими функциями, обычно могут выполняться в любом порядке. Например, временную связность имеют функции, выполняемые при инициализации некоторого процесса.

Большая вероятность модификации функции еще больше уменьшает показатели технологичности модулей данного вида по сравнению с предыдущими, кроме того, содержание модуля с временной связностью функций может изменяться: в него могут включаться новые действия и/или исключаться старые.

Логическая связь строится на основе объединения данных или функций в одну логическую группу, например, логически связаны компоненты модуля, содержащего функции обработки текстовой информации или данные одного и того же типа. При выполнении модуля с логически связанными компонентами всегда будет вызываться одна какая-либо его часть, при этом вызывающий и вызываемый модули будут связаны по управлению.

Показатели технологичности таких модулей ниже предыдущих, так как сложно понять логику их работы.

Модуль, элементы которого имеют случайную связность, имеет самые низкие показатели технологичности, так как его элементы вообще не связаны.

При проектировании программных модулей лучше всего использовать функциональную, последовательную и информационную связности.

**Модульная структура программных продуктов**

Модульная структура программы представляет собой древовидную структуру, в узлах которой размещаются программные модули, а направленные дуги показывают статическую подчиненность модулей. Если в тексте модуля имеется ссылка на другой модуль, то их на структурной схеме соединяет дуга, которая исходит из первого и входит во второй модуль. Другими словами, каждый модуль может обращаться к подчиненным ему модулям. При этом модульная структура программной системы, кроме структурной схемы, должна включать в себя еще и совокупность спецификаций модулей, образующих эту систему [37].

Функция верхнего уровня обеспечивается главным модулем; он управляет выполнением нижестоящих функций, которым соответствуют подчиненные модули.

При определении набора модулей, реализующих функции конкретного алгоритма, необходимо учитывать следующее:

1) модуль вызывается на выполнение вышестоящим по иерархии модулем и, закончив работу, возвращает ему управление;

2) принятие основных решений в алгоритме выносится на максимально высокий по иерархии уровень;

3) если в разных местах алгоритма используется одна и та же функция, то она оформляется в отдельный модуль, который будет вызываться по мере необходимости.

Состав, назначение и характер использования программных модулей в значительной степени определяются инструментальными средствами.

Например, при разработке СУБД используются следующие программные модули:

1) экранные формы ввода и/или редактирования информации базы данных;

2) отчеты;

3) макросы;

4) стандартные средства для обработки информации;

5) меню для выбора функции обработки и др.

**Методы разработки при модульном программировании**

Существуют разные методы разработки модульной структуры.

Обычно в литературе обсуждаются два метода [42, 46]: метод восходящей разработки и метод нисходящей разработки.

**Метод восходящей разработки**

Сначала строится древовидная модульная структура программы. Затем поочередно проектируются и разрабатываются модули программы, начиная с модулей самого нижнего уровня, затем предыдущего уровня и т. д. То есть модули реализуются в таком порядке, чтобы для каждого программируемого модуля были уже запрограммированы все модули, к которым он может обращаться. После того как все модули программы запрограммированы, производится их поочередное тестирование и отладка в таком же восходящем порядке. Достоинство метода заключается в том, что каждый модуль при программировании выражается через уже запрограммированные непосредственно подчиненные модули, а при тестировании использует уже отлаженные модули.

Недостатки метода восходящей разработки заключаются в следующем:

• на нижних уровнях модульной структуры спецификации могут быть еще определены не полностью, что может привести к полной переработке этих модулей после уточнения спецификаций на верхнем уровне;

• для восходящего тестирования всех модулей, кроме головного, который является модулем самого верхнего уровня, приходится создавать вызывающие программы, что приводит к созданию большого количества отладочного материала, но не гарантирует, что результаты тестирования верны;

• головной модуль проектируется и реализуется в последнюю очередь, что не дает продемонстрировать его заказчику для уточнения спецификаций.

**Метод нисходящей разработки**

Как и в предыдущем методе, сначала строится модульная структура программы в виде дерева. Затем проектируются и реализуются модули программы, начиная с модуля самого верхнего уровня — головного, далее разрабатываются модули уровнем ниже и т. д. При этом переход к программированию какого-либо модуля осуществляется только в том случае, если уже запрограммирован модуль, который к нему обращается. Затем производится их поочередное тестирование и отладка в таком же нисходящем порядке. При таком порядке разработки программы вся необходимая глобальная информация формируется своевременно, т. е. ликвидируется весьма неприятный источник просчетов при программировании модулей. Существенно облегчается и тестирование модулей, производимое при нисходящем тестировании программы. Первым тестируется головной модуль программы, который представляет всю тестируемую программу, при этом все модули, к которым может обращаться головной, заменяются их имитаторами (так называемыми «заглушками» [45]). Каждый имитатор модуля является простым программным фрагментом, реализующим сам факт обращения к данному модулю с необходимой для правильной работы программы обработкой значений его входных параметров и с выдачей, если это необходимо, подходящего результата. Далее производится тестирование следующих по уровню модулей. Для этого имитатор выбранного для тестирования модуля заменяется самим модулем, и добавляются имитаторы модулей, к которым может обращаться тестируемый модуль. При таком подходе каждый модуль будет тестироваться в «естественных» состояниях информационной среды, возникающих к моменту обращения к этому модулю при выполнении тестируемой программы. Таким образом, большой объем «отладочного» программирования заменяется программированием достаточно простых имитаторов используемых в программе модулей.

Недостатком нисходящего подхода к программированию является необходимость абстрагироваться от реальных возможностей выбранного языка программирования и придумывать абстрактные операции, которые позже будут реализованы с помощью модулей. Однако способность к таким абстракциям является необходимым условием разработки больших программных средств.

Рассмотренные выше методы (нисходящей и восходящей разработок), являющиеся классическими, требуют, чтобы модульная древовидная структура была готова до начала программирования модулей. Как правило, точно и содержательно разработать структуру программы до начала программирования невозможно. При конструктивном и архитектурном подходах к разработке модульная структура формируется в процессе реализации модулей.

**Конструктивный подход**

Конструктивный подход к разработке программы представляет собой модификацию нисходящей разработки, при которой модульная древовидная структура программы формируется в процессе программирования модуля. Сначала программируется головной модуль, исходя из спецификации программы в целом (спецификация программы является одновременно спецификацией головного модуля). В процессе программирования головного модуля в случае, если эта программа достаточно большая, выделяются подзадачи (некоторые функции) и для них создаются спецификации реализующих эти подзадачи фрагментов программы. В дальнейшем каждый из этих фрагментов будет представлен поддеревом модулей (спецификация выделенной функции является одновременно спецификацией головного модуля этого поддерева).

Таким образом, на первом шаге разработки программы (при программировании ее головного модуля) формируется верхняя часть дерева, например, как на рис. 3.12.

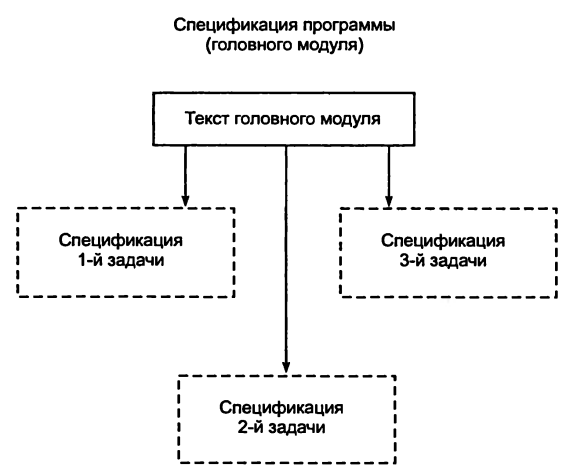


Рис. 3.12. Первый шаг формирования модульной структуры программы при конструктивном подходе

По тому же принципу производится программирование следующих по уровню специфицированных, но еще не запрограммированных модулей в соответствии со сформированным деревом. В результате к дереву добавляются очередные уровни, как показано на рис. 3.13.

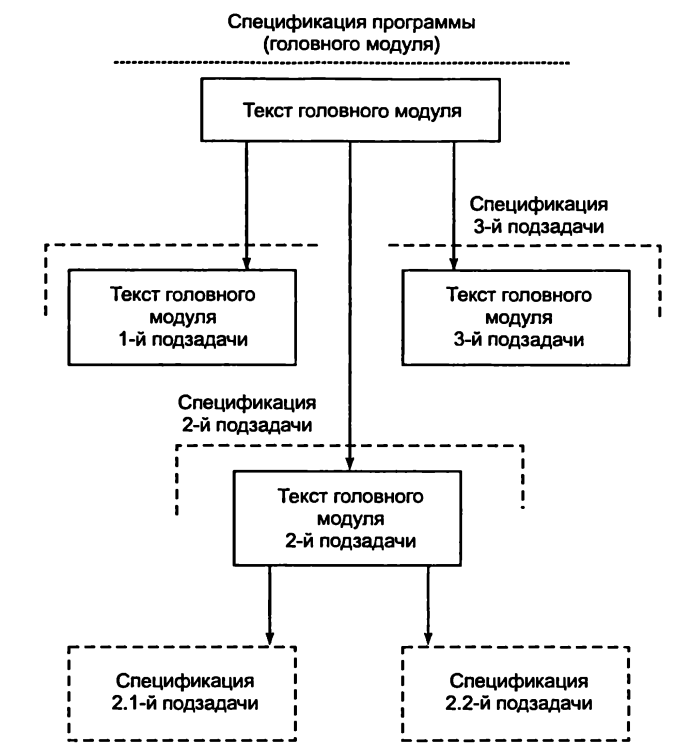


Рис. 3.13. Второй шаг формирования модульной структуры программы при конструктивном подходе

**Архитектурный подход**

Архитектурный подход к разработке программы представляет собой модификацию восходящей разработки, при которой модульная структура программы формируется в процессе программирования модуля. Целью разработки в данном методе является повышение уровня языка программирования, а не разработка конкретной программы. Это означает, что для заданной предметной области выделяются типичные функции, специфицируются, а затем и программируются отдельные программные модули, выполняющие эти функции. Сначала в виде модулей реализуются более простые функции, а затем создаются модули, использующие уже имеющиеся функции, и т. д. Это позволяет существенно сократить трудозатраты на разработку конкретной программы путем подключения к ней уже имеющихся и проверенных на практике модульных структур нижнего уровня, что также позволяет бороться с дублированием в программировании.

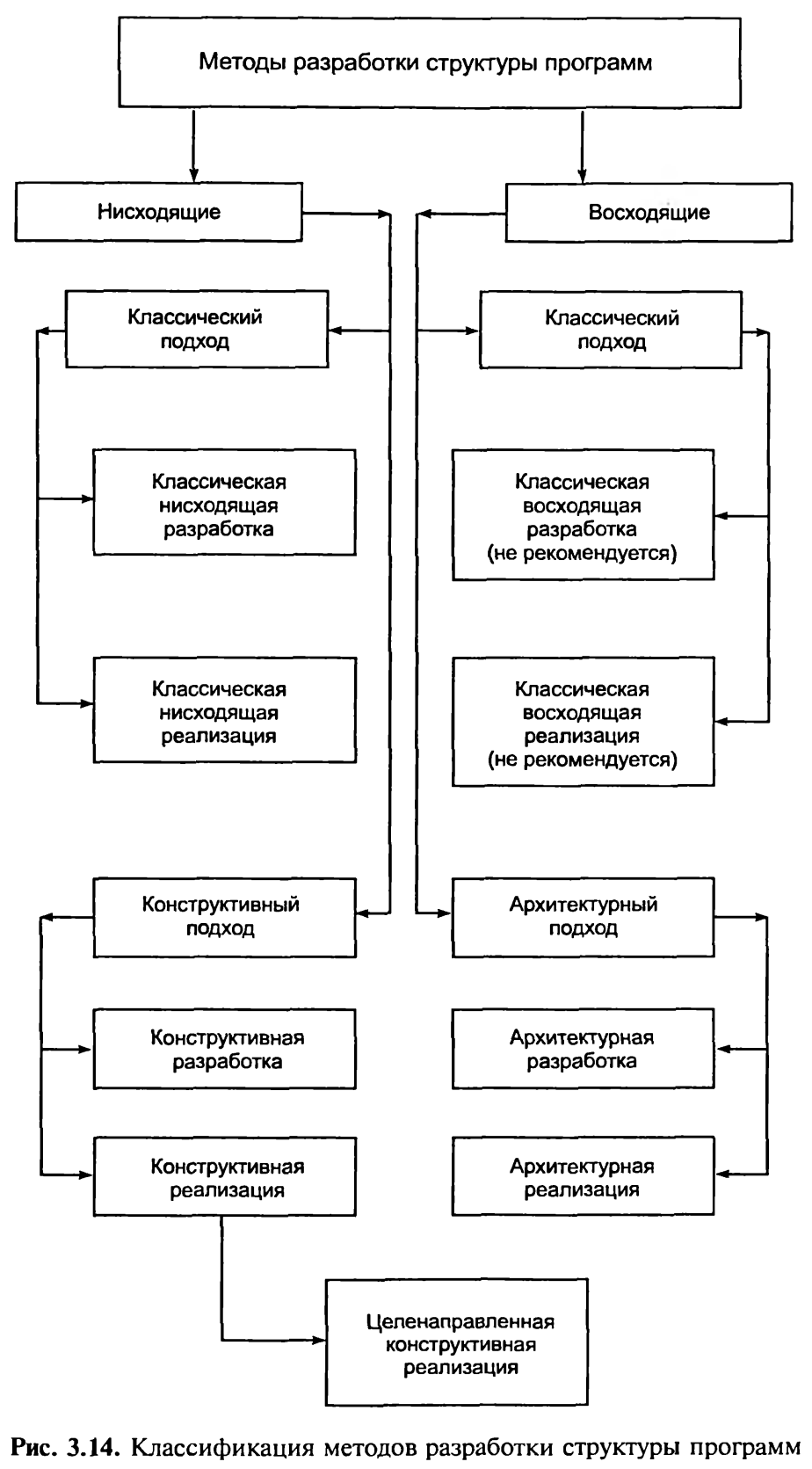
В связи с этим программные модули, создаваемые в рамках архитектурного подхода, обычно параметризуются для того, чтобы облегчить их применение настройкой параметров.

**Нисходящая реализация**

В классическом методе нисходящей разработки сначала все модули разрабатываемой программы программируются, а затем тестируются в нисходящем порядке. При этом тестирование и отладка модулей могут привести к изменению спецификации подчиненных модулей и даже к изменению самой модульной структуры программы. В результате может оказаться, что часть модулей вообще не нужна в данной структуре, а часть модулей придется переписывать. Более рационально каждый запрограммированный модуль тестировать сразу же до перехода к программированию другого модуля. Такой метод в литературе получил название метода нисходящей реализации.

**Целенаправленная конструктивная реализация**

В зависимости от того, в какой последовательности в процессе разработки программы обходятся узлы дерева, существуют разновидности описанных выше методов. Например, метод целенаправленной конструктивной реализации, при котором в рамках конструктивного подхода реализуются сначала модули, необходимые для самого простого рабочего варианта программы, остальные модули заменяются их имитаторами. Затем к имеющемуся варианту добавляются другие модули, обеспечивающие работу программы для других наборов данных, и так далее до полной реализации программы. Достоинством этого метода является то, что уже на достаточно ранней стадии создается работающий вариант разрабатываемой программы. Психологически это играет роль допинга, резко повышающего эффективность разработчика.



**C++ Builder**

Интерес к программированию постоянно растет. Это связано с развитием и внедрением в повседневную жизнь информационных технологий. Если человек имеет дело с компьютером, то рано или поздно у него возникает желание, а иногда и необходимость, научиться программировать. Среди пользователей персональных компьютеров в настоящее время наиболее популярна операционная система Windows, и естественно, что тот, кто хочет программировать, хочет и писать программы, которые будут работать в Windows.

Бурное развитие вычислительной техники, потребность в эффективных средствах разработки программного обеспечения привели к появлению на рынке целого ряда систем программирования, ориентированных на так называемую "быструю разработку", среди которых особо следует отметить Microsoft Visual Basic и Borland Delphi. В основе систем быстрой разработки (RAD-систем, Rapid Application Development — среда быстрой разработки приложений) лежит технология визуального проектирования и событийного программирования, суть которой заключается в том, что среда разработки берет на себя большую часть работы по генерации кода программы, оставляя программисту работу по конструированию диалоговых окон и написанию функций обработки событий. Успех и популярность Delphi вызвал желание фирмы Borland распространить методы быстрой разработки на область профессионального программирования, что и привело к появлению Borland C++ Builder.

C++ Builder — это среда быстрой разработки, в которой в качестве языка программирования используется язык C++ Builder (C++ Builder Language). Язык C++ Builder — это расширенный C++.

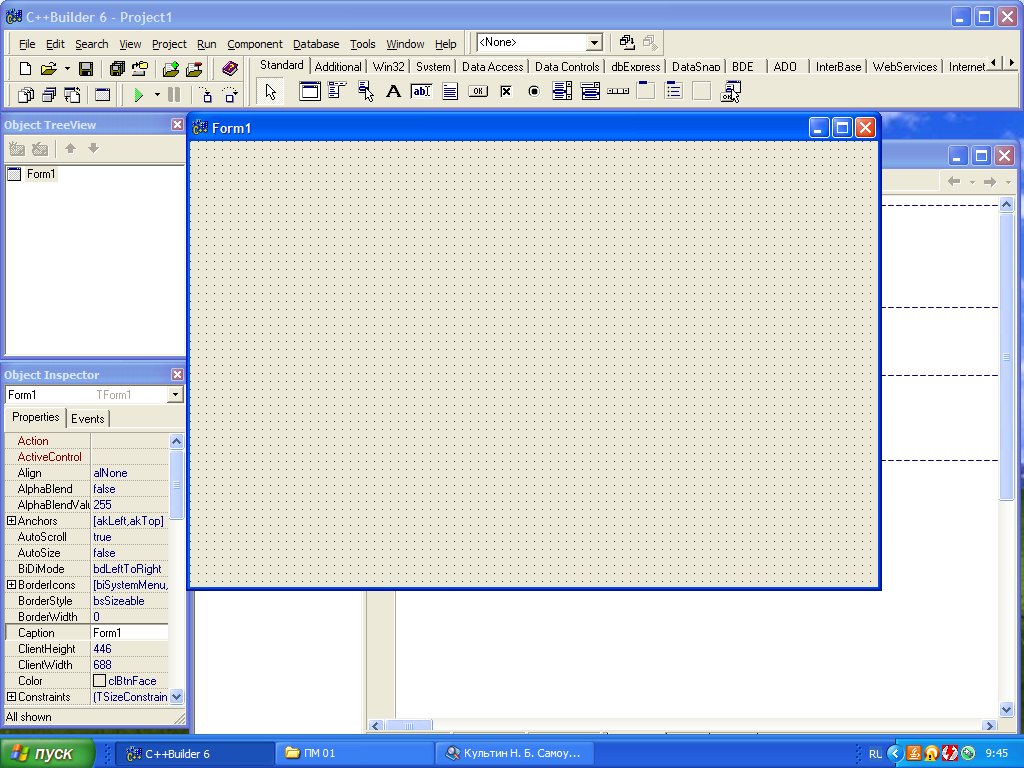
Borland C++ Builder 6 позволяет создавать различные программы: от простейших однооконных приложений до программ управления распределенными базами.

**Среда разработки C++ Builder**

Запускается C++ Builder обычным образом, т. е. выбором из меню Borland C++Builder 6 команды C++Builder 6

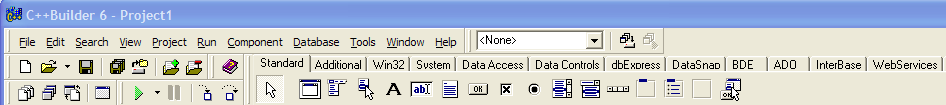
На экране после запуска C++ Builder появляются на экране пять окно:

* главное окно — C++Builder 6;
* окно стартовой формы — Forml;
* окно редактора свойств объектов — Object Inspector;
* окно просмотра списка объектов — Object TreeView;
* окно редактора кода — Unitl.cpp.

Окно редактора кода почти полностью закрыто окном стартовой формы.при работе никогда не закрывайте эти окна, только сворачивайте.

В главном окне находится меню команд, панели инструментов и палитра компонентов.

**Главное окно** осуществляет основные функции управления проекто создаваемой программы. Закрывая это окно, вы закрываете все окна.

Панель инструментов Панель компонентов

**Палитра компонентов** состоит из четырнадцати панелей, на которых компоненты сгруппированы по конкретным областям применения.

• Standard — стандартные элементы управления Windows;

• Additional — дополнительные, нестандартные элементы управления Windows;

• Win32 — элементы управления Windows 9x;

• System — системные объекты (таймер, музыкальный проигрыватель и другие);

• Internet — все для приложений, работающих с Интернетом;

• Data Access — компоненты для организации связи с базами данных;

• Data Controls — управляющие элементы для работ с базами данных;

• ADO — компоненты для доступа к данным на основе одной из самых передовых на сегодняшний день Windows-технологии Microsoft ActiveX Data Objects (ADO);

• InterBase — компоненты для работы с СУБД InterBase производства корпорации Inprise;

• Midas — компоненты для создания приложений, способных работать на нескольких компьютерах;

• InternetExpress и Internet — средства быстрого создания приложений для Интернета;

• FastNet — компоненты, поддерживающие основные сетевые протоколы, ориентированные на Интернет;

• Decision Cube — компоненты системы анализа данных;

• GReport — компоненты создания различных отчетов;

• Dialogs — стандартные диалоговые окна Windows;

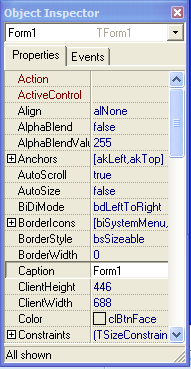
• Win 3.1 — элементы управления Windows 3.1;

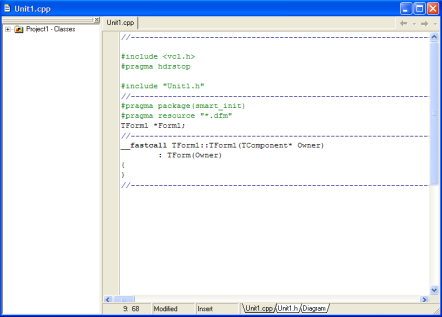
• Samples — примеры компонентов, входящие в поставку системы;

• ActiveX — ActiveX-компоненты (ActiveX — формат активных компонентов, разработанный фирмой Microsoft. Borland C++Builder 4 также поддерживает этот формат);

• Servers — набор компонентов, с помощью которых можно управлять работой офисных программ Word, Excel, PowerPoint, Outlook и др.

**Окно стартовой формы** (Forml) представляет собой заготовку главного окна разрабатываемой программы (приложения). Когда создаваемая программа будет откомпилирована и запущена, форма превратится в обычное окно Windows и станет выполнять те действия, которые для нее определены. Таких окон в программе может быть сколько угодно.Получить список всех созданных форм можно с помощью клавиш SHIFT+F12.

**Окно Object Inspector** — окно редактора свойств объектов предназначено для редактирования значений свойств объектов. В терминологии визуального проектирования объекты — это диалоговые окна и элементы управления (поля ввода и вывода, командные кнопки, переключатели и др.). Свойства объекта — это характеристики, определяющие вид, положение и поведение объекта. Например, свойства width и Height задают размер (ширину и высоту) формы, свойства тор и Left — положение формы на экране, свойство Caption — текст заголовка. В верхней части окна указан объект (имя объекта), значения свойств которого отражены в окне Object Inspector. Инспектор объектов позволяет быстро и удобно менять любые свойства текущего (выделенного на форме) объекта. При этом вносимые изменения немедленно сказываются на внешнем виде этого объекта. Для быстрого вызова Инспектора объектов можно воспользоваться клавишей F11.

Инспектор объектов состоит из нескольких частей. В раскрывающемся списке указывается, какой объект на форме выбран (является текущим) в данный момент. Сделать объект текущим можно, либо щелкнув на нем мышью, либо выбрав в этом списке. В окне Инспектора объектов имеются две вкладки: вкладка свойств выбранного объекта (Properties) и вкладка событий (Events), на которые это объект может реагировать, Каждая из вкладок содержит панель, состоящую из двух колонок. В первой указываются названия свойств (их менять нельзя), во второй — текущие значения соответствующих свойств.

Некоторые свойства могут иметь вложенную структуру. Это обозначается значком «+» слева от названия.

Существует группа свойств, значения которых неудобно менять в Инспекторе объектов. К ним, в частности, относится свойство Font (Шрифт), определяющее параметры шрифта, которым выполняются надписи в текущей форме. Если раскрыть это свойство, то можно увидеть, что оно состоит из непонятных характеристик и значений. для таких свойств, сложных в настройке, созданы специальные Построители (это аналоги Мастеров), которые позволяют задавать нужные значения удобным визуальным способом. Наличие такого Построителя для конкретного свойства легко обнаруживается по специальной кнопке на правом краю поля редактирования. При щелчке на этой кнопке вызывается соответствующий Мастер-построитель.

**В окне редактора кода**, которое можно увидеть, отодвинув в сторону окно формы, следует набирать текст программы. В начале работы над новым проектом окно редактора кода содержит сформированный C++ Builder шаблон программы.

Между текущей формой и редактором можно переключаться с помощью клавиши F12. Окно редактора, как уже говорилось, состоит из двух панелей: панели Просмотрщика классов и панели редактора исходного текста программы на Си++. Просмотрщик классов визуально отображает структуру связей между различными объектами нашей программы и позволяет быстро перемещаться по ее тексту.

**Никогда не меняйте исходные тексты Си++,** которые были созданы автоматически. Это может привести к полной неработоспособности вашей программы.

**Основные компоненты**

Компонент — это строительный кирпичик вашей программы. Компоненты могут быть визуальными и невизуальными.

Визуальный компонент — это элемент управления (например кнопка или список), С которым пользователь нашей программы будет взаимодействовать во время ее работы. Визуальные компоненты, в свою очередь, делятся на объекты Windows (обычные элементы управления) и графические

объекты-контейнеры, с помощью которых объекты Windows можно объединять в группы (всевозможные панели и вкладки). Такое объединение обычно приводит к экономии ресурсов операционной системы.С невизуалъными компонентами во время работы программы пользователь взаимодействовать не может. Они предназначены для программирования вещей, не относящихся к видимому на экране интерфейсу, например для управления встроенными часами программы (обычно называемыми таймером), для подключения к Интернету и т. д. Кроме того, невизуальные компоненты иногда служат для создания сложных элементов управления, например меню, структуру которого удобнее готовить с помощью специального Мастера, содержащегося в компоненте Меню.

|  |  |
| --- | --- |
| **Форма** | **Form** является основой программы |
| ris1_5 | [**Label**](http://it.kgsu.ru/DelStdrd/delc0005.html#2) ***(Метка, Поле надписи)*** используется для отображения текста в форме, который нельзя изменять непосредственно через графический интерфейс пользователя, хотя в программе с помощью надписи можно выводить любой текст, что часто и делается. |
| ris1_6 | В [**Edit**](http://it.kgsu.ru/DelStdrd/delc0006.html#2) ***(Поле ввода)*** хранится текст, который можно помещать в данный компонент, как во время проектирования, так и во время выполнения программы. |
| ris1_7 | [**Memo**](http://it.kgsu.ru/DelStdrd/delc0007.html#2) ***(Область просмотра)*** предназначен для вывода на экран нескольких строк текста. |
| ris1_8 | Обычно с помощью компонента [**Button**](http://it.kgsu.ru/DelStdrd/delc0008.html#2) ***(Кнопка)*** пользователь инициирует выполнение какого-либо фрагмента кода или целой программы. Другими словами, если щелкнуть на элемент управления **TButton**, то программа должна выполнить определенное действие. При этом кнопка принимает такой вид, будто она нажата. |
| ris1_9 | [**CheckBox**](http://it.kgsu.ru/DelStdrd/delc0009.html#2) ***(Флажок)*** используется для того, чтобы пользователь мог указать свое решение типа да/нет или да/нет/не уверен. |
| ris1_10 | [**RadioButton**](http://it.kgsu.ru/DelStdrd/delc0010.html#2) ***(Переключатель)*** предназначен для выбора одного значения из ряда возможных. Переключатели всегда используются группами. |
| ris1_11 | [**ListBox**](http://it.kgsu.ru/DelStdrd/delc0011.html#2) ***(Список)*** очень часто применяется в программах для Windows. Он позволяет выбрать одну или несколько строк в списке. Строки могут содержать как текст, так и рисунки. |
| ris1_12 | [**ComboBox**](http://it.kgsu.ru/DelStdrd/delc0012.html#2) ***(Комбинированный список)*** представляет собой вариант списка с присоединенным дополнительным полем, в котором отображается выбранный элемент списка. |
| ris1_13 | [**ScrollBar**](http://it.kgsu.ru/DelStdrd/delc0013.html#2) ***(Полоса прокрутки)*** обычно используется как вспомогательный инструмент в других элементах управления: списках, текстовых областях и т.д. |
| ris1_15 | [**RadioGroup**](http://it.kgsu.ru/DelStdrd/delc0015.html#2) ***(Группа переключателей)*** используется аналогично **TGroupBox** для группировки объектов **TRadioButton**. |
| ris20_3 | [**BitBtn**](http://it.kgsu.ru/DelAdd/deladd02.html) - ***кнопка с изображением***. Представляет собой популярную разновидность стандартной кнопки **TButton**.Используется для создания кнопок, на которых располагается битовая графика (например, кнопка ОК с галочкой). Ее отличительная особенность - свойство **Glyph**, с помощью которого определяется растровое изображение, рисуемое на поверхности кнопки. |
| ris20_8 | [**Image**](http://it.kgsu.ru/DelAdd/deladd07.html) - ***изображение***. Этот компонент служит для размещения на форме одного из трех поддерживаемых **Delphi** типов изображений: растровой картинки, пиктограммы или метафайла. |
| ris20_9 | [**Shape**](http://it.kgsu.ru/DelAdd/deladd08.html) - ***стандартная фигура***. Компонент рисует одну из простейших геометрических фигур (квадрат, круг и т.п.). |
| ris01_3 | [**Timer**](http://it.kgsu.ru/DelSys/delsys02.html) - ***таймер***. Используется для запуска процедур, функций и событий в указанные интервалы времени. Компонент невизуальный. |

**Основные свойства компонентов**

Любой компонент обладает конкретными свойствами, значения которых можно изменять. Значение свойств изменяется при помощи окна свойств. Либо такое изменение совершается внутри программы. Для присваивания свойству конкретного значения используется знак операции присваивания.

Синтаксис установки значения **свойства компонента:**

**Компонент –> Свойство = Значение Свойства;**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Свойство** | **Описание** | | **Объект** | |
| Name | Имя объекта. В программе имя объекта используется для управления объектом | | Все объекты | |
| Caption | Текст заголовка | | Form, Button, Label, CheckBox, RadioButton, | |
| Text | Текст, находящийся в поле | | Edit, Memo | |
| Top | Расстояние от верхней границы объекта до верхней границы экрана | | Все объекты | |
| Left | Расстояние от левой границы объекта до левой границы экрана | | Все объекты | |
| Width | Ширина объекта | | Все объекты | |
| Height | Высота объекта | | Все объекты | |
| Color | Цвет фона. Цвет можно задать, указав название цвета или привязку к текущей цветовой схеме операционной системы. Во втором случае цвет определяется текущей цветовой схемой, выбранным компонентом привязки и меняется при изменении цветовой схемы операционной системы | | Все объекты | |
| Font | Шрифт. Шрифт, используемый "по умолчанию" компонентами, находящимися на поверхности объекта. Изменение свойства Font объекта приводит к автоматическому изменению свойства Font компонента, располагающегося на поверхности объекта. То есть компоненты наследуют свойство Font от объекта (имеется возможность запретить наследование) | | Все объекты | |
| Canvas | Поверхность, на которую можно вывести графику | | Form, Image | |
| Visible | Позволяет скрыть компонент (False) или сделать его видимым (True) | |  | |
| ParentFont | Признак наследования компонентом характеристик шрифта формы, на которой находится компонент. Если значение свойства равно True, то при изменении свойства Font формы автоматически меняется значение свойства Font компонента | Label, Edit, Memo, RadioButton | |
| Enabled | Доступность (недоступность) объекта | Edit, Button | |
| WordWrap | Признак того, что слова, которые не помещаются в текущей строке, автоматически переносятся на следующую строку (значение свойства AutoSize должно быть False) | | Label | |
| Alignment | Задает способ выравнивания текста внутри поля. Текст может быть выравнен по левому краю (taLeftJustify), по центру (taCenter) или по правому краю (taRightJustify) | | Label | |

**Событие и функция обработки события**

Событие (Event) — это то, что происходит во время работы программы. В C++ Builder каждому событию присвоено имя. Например, щелчок кнопкой мыши -- это событие Onclick, двойной щелчок мышью — событие OnDblClick.

**Основные события Windows**

|  |  |
| --- | --- |
| **Событие** | **Происходит** |
| Onclick | При щелчке кнопкой мыши |
| OnDblClick | При двойном щелчке кнопкой мыши |
| OnMouseDown | При нажатии кнопки мыши |
| OnMouseUp | При отпускании кнопки мыши |
| OnMouseMove | При перемещении мыши |
| OnKeyPress | При нажатии клавиши клавиатуры |
| OnKeyDown | При нажатии клавиши клавиатуры. События OnKeyDown и OnKeyPress — это чередующиеся, повторяющиеся события, кото рые происходят до тех пор, пока не будет отпущена удерживаемая клавиша (в этот момент происходит событие onKeyUp) |
| OnKeyUp | При отпускании нажатой клавиши клавиатуры |
| OnCreate | При создании объекта (формы, элемента управления). Процедура обработки этого события обычно используется для инициализации переменных, выполнения подготовительных действий |
| OnPaint | При появлении окна на экране в начале работы программы; во время работы программы после появления окна (или его части), которое было закрыто другим окном или свернуто |
| OnEnter | При получении элементом управления фокуса |
| OnExit | При потере элементом управления фокуса |

Реакцией на событие должно быть какое-либо действие. В C++ Builder реакция на событие реализуется как функция обработки события. Таким образом, для того чтобы программа выполняла некоторую работу в ответ на действия пользователя, программист должен написать функцию обработки соответствующего события. Следует обратить внимание на то, что значительную часть обработки событий берет на себя компонент. Поэтому программист должен разрабатывать функцию обработки события только в том случае, если реакция на событие отличается от стандартной или не определена. Например, если по условию задачи ограничений на символы, вводимые в поле Edit, нет, то процедуру обработки события onKeypress писать не надо, т. к. во время работы программы будет использована стандартная (скрытая от программиста) процедура обработки этого события.

**Шаблоны кода**

В процессе набора текста удобно использовать шаблоны кода (Code Templates). Шаблон кода — это инструкция программы, записанная в общем виде. Например, шаблон для инструкции if выглядит так:

if ()

{

}

else

{

}

Редактор кода предоставляет программисту большой набор шаблонов: объявления КЛАССОВ, ФУНКЦИЙ, ИНСТРУКЦИЙ Выбора (if, switch), ЦИКЛОВ (for, while). Для некоторых инструкций, например для if и while, есть несколько вариантов шаблонов.

Для того чтобы в процессе набора текста программы воспользоваться шаблоном кода и вставить его в текст программы, нужно нажать комбинацию клавиш <Ctrl>+<J> и из появившегося списка выбрать нужный шаблон. Выбрать шаблон можно обычным образом, прокручивая список, или вводом первых букв имени шаблона (имена шаблонов в списке выделены полужирным). Выбрав в списке шаблон, нужно нажать клавишу <Enter>, шаблон будет вставлен в текст программы.

Программист может создать свой собственный шаблон кода и использовать его точно так же, как и стандартный. Для того чтобы создать шаблон кода, нужно в меню Tools выбрать команду Editor Options и в окне Code Insight щелкнуть на кнопке Add. В появившемся окне Add Code Template надо задать имя шаблона (Shortcut Name) и его краткое описание (Description). Затем, после щелчка на кнопке ОК, в поле Code надо ввести шаблон.

**Сохранение проекта**

Проект — это набор файлов, используя которые компилятор создает выполняемый файл программы (ехе-файл). В простейшем случае проект составляют: файл описания проекта (bpr-файл), файл главного модуля (срр-файл), файл ресурсов (res-файл), файл описания формы (dfm-файл), заголовочный файл формы (h-файл) и файл описания функций формы (срр-файл).

Чтобы сохранить проект, нужно в меню File выбрать команду Save Project As. Если проект еще ни разу не был сохранен, то C++ Builder сначала предлагает сохранить модуль (содержимое окна редактора кода) и поэтому на экране появляется окно Save Unitl As. В этом окне надо выбрать папку, предназначенную для проектов, создать в ней папку для сохраняемого проекта, открыть ее и ввести имя модуля. В результате щелчка на кнопке OK в указанной папке будут созданы три файла: срр, h и dfm, и на экране появится диалоговое окно Save Projectl As, в которое надо ввести имя проекта.

Обратите внимание, что имена файла модуля (срр) и файла проекта (bpr) должны быть разными, т. к. C++ Builder в момент сохранения файла проекта создает одноименный срр-файл (файл главного модуля). Кроме того, надо учесть, что имя генерируемого компилятором выполняемого файла совпадает с именем проекта. Поэтому файлу проекта следует присвоить такое имя, которое, по вашему мнению, должен иметь выполняемый файл программы, а файлу модуля — какое-либо другое имя, например, полученное путем добавления к имени проекта порядкового номера модуля.

**Компиляция**

Процесс преобразования исходной программы в выполняемую состоит из двух этапов: непосредственно компиляции и компоновки. На этапе компиляции выполняется перевод исходной программы в некоторое внутреннее представление. На этапе компоновки выполняется сборка (построение) программы.

После ввода текста функции обработки события и сохранения проекта можно, выбрав в меню Project команду Compile, выполнить компиляцию. Процесс и результат компиляции отражается в диалоговом окне Compiling. Если ,в программе нет синтаксических ошибок, то окно будет содержать сообщение: Done: Compile Unit, в противном случае будет выведено сообщение Done: There are errors.

В случае если компилятор обнаружит в программе ошибки и неточности, диалоговое окно Compiling будет содержать информацию о количестве синтаксических (Errors) и семантических (Warnings) ошибок, а также о числе подсказок (Hints). Сами сообщения об ошибках, предупреждения и подсказки находятся в нижней части окна редактора кода.

Чтобы перейти к фрагменту кода, который, по мнению компилятора, содержит ошибку, надо выбрать сообщение об ошибке (щелкнуть в строке сообщения левой кнопкой мыши) и из контекстного меню выбрать команду Edit Source.

Процесс компиляции можно активизировать, выбрав в меню Run команду Run, которая запускает разрабатываемое приложение. Если будет обнаружено, что с момента последней компиляции в программу были внесены изменения или программа еще ни разу не компилировалась, то будет выполнена компиляция, затем — компоновка, и после этого программа будет запущена (естественно, только в том случае, если в программе нет ошибок).

**Ошибки**

Компилятор переходит ко второму этапу генерации выполняемой программы только в том случае, если исходный текст не содержит синтаксических ошибок.

**Типичные ошибки**

|  |  |
| --- | --- |
| Undefined symbol  (неизвестный символ) | Используется необъявленная переменная  Имя переменной, функции или параметра записано неверно. Например, в программе объявлена переменная Summ, а в инструкциях используется sum |
| Statement missing ;  (отсутствует точка с запятой) | После инструкции не поставлена точка с запятой |
| Unterminated string or  character constant  (незаконченная строковая  или символьная константа) | В конце строковой константы, например, текста сообщения, нет двойных кавычек |
| ) expected  (ожидается закрывающая скобка) | При записи арифметического выражения, содержащего скобки, нарушен баланс открывающих и закрывающих скобок |
| if statement missing ((в инструкции if нет открывающей скобки) | В инструкции if условие не заключено в скобки |
| Compound statement missing } | Нарушен баланс открывающих и закрывающих фигурных скобок. Вероятно, не поставлена закрывающая фигурная скобка отмечающая конец функции или группы инструкций, например, после условия или слова else в инструкции if |
| Extra parameter in call to  (лишний параметр при вызове  функции) | Неверно записана инструкция вызова функции, указан лишний параметр |

Если компилятор обнаружил достаточно много ошибок, то просмотрите все сообщения и устраните сначала наиболее очевидные ошибки и выполните повторную компиляцию. Вполне вероятно, что после этого количество ошибок значительно уменьшится. Это объясняется особенностями синтаксиса языка, когда одна незначительная ошибка может "тащить" за собой довольно большое количество других.

**Компоновка**

Если в программе нет ошибок, то можно выполнить компоновку. Для этого надо в меню Compile выбрать команду Make или Build. Разница между командами Make и Build заключается в следующем. Команда Make обеспечивает компоновку файлов проекта, а команда Build – принудительную перекомпиляцию, а затем — компоновку.

На этапе компоновки также могут возникнуть ошибки. Чаще всего причина ошибок во время компоновки состоит в недоступности файлов библиотек или других ранее откомпилированных модулей. Устраняются эти ошибки путем настройки среды разработки и включением в проект недостающих модулей. В простых проектах ошибки времени компиляции, как правило, не возникают.

**Запуск программы**

Пробный запуск программы можно выполнить непосредственно из среды разработки, не завершая работу с C++ Builder. Для этого нужно в меню Run выбрать команду Run или щелкнуть на командной кнопке Run.

**Функции преобразования**

|  |  |
| --- | --- |
| Функция | Значение функции |
| IntToStr(k) | Строка, являющаяся изображением целого k |
| FloatToStr(n) | Строка, являющаяся изображением вещественного n |
| FloatToStrF(n, f, k,m) | Строка, являющаяся изображением вещественного n. При вызове функции указывают: f — формат; k — точность; m — количество цифр после десятичной точки. Формат определяет способ изображения числа: ffGeneral— универсальный; ffExponent— научный; f f Fixed — с фиксированной точкой; f fNumber — с разделителями групп разрядов; ffCurrency— финансовый. Точность — нужное общее количество цифр: 7 или меньше для значения типа single, 15 или меньше для значения типа Double и 18 или меньше для значения типа Extended |
| StrToInt(s) | Целое число, изображением которого является строка s |
| StrToFloat(s) | Дробное число, изображением которого является строка s |

**Структура простого проекта**

Проект представляет собой набор программных единиц — модулей. Один из модулей, называемый главным, содержит инструкции, с которых начинается выполнение программы. Чтобы увидеть главный модуль, нужно в меню Project выбрать команду View Source.

**Главный модуль Unit1.cpp**

#include <vcl.h>

#pragma hdrstop

USEFORM("Unit1.cpp", Form1);

WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)

{

try

{

Application->Initialize();

Application->CreateForm(\_\_classid(TForm1), &Form1);

Application->Title = "Первая программа на С++";

Application->Run();

}

catch (Exception &exception)

{

Application->ShowException(&exception);

}

catch (...)

{

try

{

throw Exception("");

}

catch (Exception &exception)

{

Application->ShowException(&exception);

}

}

return 0;

}

Начинается главный модуль директивами компилятору (точнее, препроцессору). Директива #include <vcl.h> информирует компилятор, что перед тем как приступить непосредственно к компиляции, в текст главного модуля нужно включить заголовочный файл библиотеки визуальных компонентов — vci.h. Строка usEFORM("Amper\_i.cpp", Formi) указывает, что в проект нужно включить файл модуля формы Unitl.cpp, который содержит функции обработки событий для формы Forml. Далее следует описание главной функции программы — winMain. Функция winMain инициализирует внутренние структуры программы, создает форму Forml и запускает программу, что приводит к появлению на экране стартовой формы. Так как в проекте только одна форма, то на экране именно она и появляется. Инструкция обработки исключений catch выполняется, если в программе возникает ошибка. Таким образом, главный модуль обеспечивает вывод стартовой формы программы, дальнейшее поведение которой определяют функции обработки событий стартовой формы.

Название приложения задается на вкладке Application окна Project Options. Если название приложения не задано, то в заголовке будет имя исполняемого файла.

Помимо главного модуля в состав проекта входят модули формы. Для каждой формы C++ Builder создает отдельный модуль, который состоит из двух файлов: заголовочного файла и файла кода (содержимое этих файлов отражается в окне редактора кода). Заголовочный файл содержит описание формы (Unit1.h), файл кода (модуль формы) - - описание (текст) функций, в том числе и обработки событий (Unit1.cpp).

**Значок приложения**

Чтобы назначить приложению значок, отличный от стандартного, нужно в меню Project выбрать команду Options и в открывшемся окне на вкладке Application щелкнуть на кнопке Load Icon. В результате этих действий откроется стандартное окно, используя которое можно просмотреть каталоги и найти подходящий значок (значки хранятся в файлах с расширением ico). В состав C++ Builder входит утилита Image Editor (Редактор изображений), при помощи которой программист может создать для своего приложения уникальный значок. Запустить Image Editor: выбрав в меню Tools команду Image Editor, или из Windows — командой Пуск | Программы | Borland C++ Builder | Image Editor.

Для начала работы нужно в меню File выбрать команду New | Icon File.

После выбора типа создаваемого файла открывается окно Icon Properties , в котором необходимо выбрать характеристики создаваемого значка:

* Size (Размер) -- 32x32 (стандартный размер значков Windows).
* Colors (Палитра) — 16 цветов.

После нажатия кнопки ОК открывается окно Iconl.ico, в котором можно, используя стандартные инструменты и палитру, нарисовать нужный значок.

Процесс рисования в Image Editor практически ничем не отличается от процесса создания картинки в графическом редакторе Microsoft Paint. Однако есть одна тонкость. Первоначально поле изображения закрашено "прозрачным" цветом. Если значок нарисовать на этом фоне, то при отображении значка части поля изображения, закрашенные "прозрачным" цветом, примут цвет фона, на котором будет находиться значок.

В процессе создания картинки можно удалить (стереть) ошибочно нарисованные элементы, закрасив их прозрачным цветом, которому на палитре соответствует левый квадрат в нижнем ряду.



В палитре есть "инверсный" цвет. Нарисованные этим цветом части рисунка при выводе на экран будут окрашены инверсным относительно цвета фона цветом.

Чтобы сохранить нарисованный значок, надо в меню File выбрать команду Save, в открывшемся диалоговом окне раскрыть папку проекта (приложения, для которого создан значок) и задать имя файла значка, которое обычно совпадает с именем проекта (выполняемого файла приложения).

**Перенос приложения на другой компьютер**

Небольшую программу, которая представляет собой один ехе-файл, можно перенести на другой компьютер вручную. Программа, созданная в C++ Builder, использует DLL версию библиотеки времени выполнения (RTL — Runtime Library) и специальные динамические библиотеки — пакеты. Чтобы программа могла работать на другом компьютере, помимо ехе-файла на этот компьютер надо перенести RTL-библиотеку и используемые программой пакеты или включить их в ехе-файл (что увеличит размер ехе-файла). Для этого в меню Project выбрать команду Options и во вкладках Linker и Packages сбросить соответственно флажки Use dynamic RTL и Build with runtime packages. После этого нужно выполнить перекомпоновку программы.

























**Обзор языка С++**

**Синтаксис языка**

**Используемые символы**

Множество символов, используемых в языке C++, можно разделить на следующие группы:

* Символы для образования ключевых слов и идентификаторов. В эту группу входят прописные и строчные буквы английского алфавита, а также символ подчеркивания.
* Арабские цифры.
* Прописные и строчные буквы русского алфавита.
* Точка, запятая и т. д.
* Управляющие и разделительные символы. К этой группе относятся пробел, символы табуляции, перевода строки, возврата каретки и новая строка. Эти символы отделяют друг от друга объекты, определяемые пользователем, к которым относятся константы и идентификаторы. Последовательность разделительных символов рассматривается компилятором как один символ (например последовательность пробелов).
* Управляющие последовательности, то есть специальные символьные комбинации, используемые в функциях ввода и вывода информации. Управляющая последовательность строится на основе использования обратной дробной черты (\) (обязательный первый символ) и комбинации латинских букв и цифр.

**Переменные**

Для определения переменной нужно указать ее тип и имя. При объявлении переменной для нее резервируется память, размер которой зависит от указанного типа. Можно сразу определить переменную, присвоив ей начальное значение.

Именем переменной может быть последовательность цифр и букв, а также специальных символов, при условии, что первой стоит буква или специальный символ. Для образования имени могут быть использованы строчные или прописные буквы латинского алфавита. В качестве специального символа может использовать символ подчеркивания «\_». Следует отметить, что одинаковые прописные и строчные буквы считаются различными символами, так как имеют различные коды. Например, abc, ABC, A128B, al28b — имена различных переменных.

**Константы**

В качестве констант могут использоваться целые и вещественные числа, символы, строки символов и логические константы.

Выделяют следующие виды целых констант.

* Десятичная константа состоит из одной или нескольких десятичных цифр причем первая цифра не должна быть нулем (в противном случае число будет воспринято как восьмеричное). Например, 16,127, 240.
* Восьмеричная константа состоит из обязательного нуля и одной или нескольких восьмеричных цифр (цифры от 0 до 7). Например, 020, 0177,0360.
* Шестнадцатеричная константа начинается с обязательной последовательности Ох или 0Х и содержит одну или несколько цифр шестнадцатеричной системы счисления: 0, 1, 2, 3,4, 5, 6, 7, 8, 9, А, В, С, D, E, F. Например, 0x10, Ох 0xF0.
* Вещественные константы записываются с использованием десятичной точки. Например, 2.5 или -3.17. Можно применять экспоненциальную запись вещественного числа с использованием символа е или Е. Число 0.25Е-5 означает число 0,25-10-5.
* Символьная константа — любой символ, заключенный в апострофы. Например, ‘а’, ‘h’.
* Строковая константа — любая последовательность символов, заключенная в двойные кавычки. Например, "HELLO".
* Логические константы представляют собой слово false (ложь или 0) и true (истина, причем истинным считается любое значение, отличное от нуля).

**Комментарии**

Комментарий — это набор символов, игнорируемых компилятором. На комментарий накладываются следующие ограничения. Внутри не может быть специальных символов, определяющих начало и конец комментария соответственно (/\* и \*/). Комментарии могут занимать как одну строку, так и несколько. Следует быть внимательными при написании многострочных комментариев, чтобы в них не осталось программного кода, который будет проигнорирован.

Примеры комментариев:

//однострочный комментарий к программе

/\* начало многострочного комментария, который завершается на следующей

строке \*/

**Простые типы данных**

1. **Данные целого типа**

Для определения данных целого типа используются различные ключевые слова, которые устанавливают диапазон значений и размер области памяти, выделяемой под переменные.

Типы целых переменных

|  |  |  |
| --- | --- | --- |
| Тип | Диапазон | Размер (в байтах) |
| char | -128...127 | 1 |
| short int | -32768... 32767 | 2 |
| int | -2147483648...2147483647 | 41 |
| long | -2147483648...2147483647 | 4 |

Эти типы могут употребляться вместе с ключевым словом signed (знаковый). Если требуется определить переменные, которые будут принимать только целые положительные значения, то используют ключевое слово unsigned (беззнаковый) и получают следующие типы:

Типы целых переменных, принимающих лишь положительные значения

|  |  |  |
| --- | --- | --- |
| Тип | Диапазон | Размер (в байтах) |
| unsigned char | 0...255 | 1 |
| unsigned short | 0...65535 | 2 |
| unsigned int | 0...4294967295 | 4 |
| unsigned long | 0...4294967295 | 4 |

1. **Перечисления**

Перечисления используются для задания некоего множества значений. Имена, заданные в перечислении, представляют собой целые константы. Если значения задаются явно, то предполагается, что элементам перечисления присваиваются цифры 0,1 и далее с шагом 1. В следующем примере имеется множество цифр от 0 до 9.

enum digits{zero, one, two, three, four, five, six, seven, eight, nine};

int a=one; //переменная примет значение 1

Можно явно присвоить значения отдельным элементам. На остальные будет действовать правило увеличения значения на единицу.

enum digits{ one=1, two, five=5, six, seven, eight, nine};

В данном примере two=2, six=6 и т. д.

1. **Данные вещественного типа**

При работе с дробными числами используются данные вещественного типа. Диапазон их значений гораздо шире, чем у целых типов, но может происходить пoтеря значащих цифр.

Вещественные типы данных

|  |  |  |  |
| --- | --- | --- | --- |
| Тип | Диапазон | Количество значащих цифр | Размер (в байтах) |
| float | 1,18 Е -38...3,4Е38 | 7 | 4 |
| double | 2,237 Е-308... 1,79Е308 | 15 | 8 |
| long double | 3,377Е -4932...1,18Е4932 | 18 | 10 |

1. **Символьные данные**

Для представления символьных данных можно использовать типы char, AnsiChar и WideChar. Типы char и AnsiChar идентичны и занимают в памяти 1 байт. Символьное значение можно рассматривать как число в диапазоне от 0 до 255 или символ в таблице ANSI с указанным кодом. Например, английская буква «а» имеет код 97.

Для работы с набором символов Unicode используют тип WideChar, который занимает в памяти 2 байта. Следует отметить, что символы с кодами от 0 до 255 в кодировке Unicode совпадают с соответствующими символами в кодировке ANSI.

1. **Логический (булевый) тип данных**

К логическим типам данных относятся типы ByteBool, bool, boolean. Ilepeменные этого типа занимают в памяти 1 байт и могут принимать только два значения true (истина) или false (ложь). Они определены следующим образом: true равно 1, false равно 0, поэтому можно сравнивать значения данного типа, причем true>false.

Родным для C++Builder является тип bool. Для совместимости с Windows введены логические типы WordBool, занимающий 2 байта, и LongBool, занимающий 4 байта.

**Структурированные типы данных**

1. **Массивы**

Массив — один из структурированных типов данных. Характеризуется тем, все его элементы имеют один и тот же тип. При определении массива следует задать его имя и количество элементов. В некоторых случаях допустимо описание без указания количества элементов.

Определение массива предполагает обязательное указание его размеров.

int a[10]; //определение массива из десяти элементов

Компилятор определит количество элементов по количеству начальных значений в списке инициализации.

int b[]={1,2,3,4}; //определение массива с инициализацией его элементов

int c[4]={1,2}; //определение массива с частичной инициализацией

Допускается определение массива с частичной его инициализацией. В данном примере с[0]=1 и с[1]=2. Значения остальных элементов массива не определены.

Если массив статический или внешний, то при его определении может выполняться инициализация по умолчанию.

static int k[5]; //все элементы массива к примут значение 0

int n[5]; //значения массива n будут неопределенными

Для внешнего массива при его описании можно не задавать количество элементов. В месте определения такого массива для него будет выделена память и выполнена инициализация.

extern int M[]; //описание внешнего массива

Объекту типа массив невозможно присвоить никакого другого значения, да если оно является массивом аналогичного типа и размерности.

char ch1[4];

char ch2['a','b’,’c’,’d'};

Если вы попытаетесь выполнить присвоение вида chl=ch2, то появится сообщение Lvalue required. Это означает, что выражение chl не является леводопустимым. Имя массива является константным указателем, поэтому невозможно копирование массивов с помощью простого оператора присваивания.

Существует понятие многомерного массива. Многомерные массивы в C++ рассматриваются как массивы, элементами которых также являются массивы. Определение многомерного массива должно содержать информацию о типе, размерности и количестве элементов каждой размерности.

int A1[10]; //одномерный массив размерности 10

int A2[20][10]; //20 одномерных массивов размерности 10

int A3[30][20][10]; //30 двумерных массивов

Подобно одномерному массиву, многомерный массив при определении может быть проинициализирован.

int MyArray[2][3]={1,2,3,5,6};

Следующие элементы получат начальные значения:

МуАггау[1][0]=1

МуАггау[0][1]=2;

МуАггау[0][2]=3;

МуАггау[1][0]=4;

МуАггау[1][1]=5;

МуАггау[1][2]=6;

Дополнительные фигурные скобки при инициализации позволяют выделить отдельные фрагменты многомерного массива. Каждая пара фигурных скобок специфицирует значения, относящиеся к одной определенной размерности. Предыдущее присвоение можно было записать в следующем виде:

intMyArray[2][3]={{1,2,3},{4,5,6}};

Нахождение количества элементов в массиве:

double A1[3];

int A2[2][3];

int n=ARRAYSIZE(A1); //первый способ

int m=sizeof(A2)/sizeof(A2[0][0]>; //второй способ

Первый способ заключается в применении заранее написанного макроса, который выполняет ту же операцию, что используется во втором способе. При нахождении количества элементов массива размер массива делится на размер одного элемента массива.

1. **Структуры**

Структура — это набор взаимосвязанных данных, объединенных в одно целое. Элементы структуры могут иметь любой тип. При работе со структурами следует вначале объявить структуру, а затем создать ее экземпляр. Можно создавать: элементы структуры непосредственно во время ее объявления. Для доступа к элементам используется оператор, представляющий собой точку между именем структуры и именем ее элемента. При желании можно инициализировать сразу элементы структуры. Следует обратить внимание на наличие обязательного знака «;» перед закрывающей скобкой в описании элементов структуры.

Пример 1. В первой строке создается структура, состоящая из двух элементов, во второй создается экземпляр структуры, в третьей и четвертой идет присвоение значений элементам структуры:

struct comp {float im; float re;};

comp a;

a.im=2;

a.re=4;

Пример 2. Все действия можно оформить в одной строке:

struct comp{int im; int re;} a={2,4};

Пример З. Можно создавать массив структур:

struct comp{float im; float re;};

comp b[4]={{1,2},{2,3},{3,4},{5,6}}

1. **Строковые данные**

Можно рассматривать строку как массив символов типа char.

char string[]="HELLO"; char str1 [12];

В данном примере описан массив, состоящий из восьми элементов, так как последним значением в этой строке должен быть 0. Иногда такой массив называют ноль-терминированным массивом (NULL-terminated). Для работы со строками есть много функций:

* int a=strlen(string); — переменная а содержит значение длины строки
* strcpy(strl, string); — копирование строки string в строку strl.
* strсat (strl, str2, kol); — приписывает kol символов строки str2 к строке str1.
* int a=strcmp(strl, str2); — идет сравнение строк strl и str2. Функция возвращает отрицательное значение, если первая строка меньше второй; положительное значение, если первая строка больше второй; и значение 0, ее строки равны.

Для работы с компонентами VCL (Visual Component Library) чаще используют строковые типы.

|  |  |  |
| --- | --- | --- |
| **Тип** | **Максимальная длина** | **Занимаемая память** |
| ShortString | До 255 символов | От 2 до 256 байт |
| AnsiString | До231 символов | От 4 байт до 2 Гбайт |
| WideString | До 230 символов | От 4 байт до 2 Гбайт |

Тип AnsiString используется почти везде, где требуется работа со строками. Он имеет множество методов. Вот некоторые из них:

* c\_str () — возвращает указатель на ноль-терминированный массив, содержащий ту же информацию, что и исходная строка;
* Length () — возвращает длину строки;
* Delete () — удаляет символы из строки;
* Substring () — копирует подстроку;
* ToInt (), ToDouble () — преобразует строку соответственно в целое и вещественное значение.

**Тип Variant**

Иногда приходиться иметь дело с данными, тип которых неизвестен на этапе компиляции. Такой тип может содержать целое или вещественное число, логическое выражение, строку, массив произвольной размерности и длины, указатель на OLE-объект. В таком случае первые 2 байта занимаемой памяти держат информацию, какого типа данные находятся в переменной. Тип данных Variant — нестандартный тип для языка C++. Этот тип данных реализован разботчиками Borland с использованием класса C++. Например:

Variant A;

А=1;

А=2.5;

А="ПРИВЕТ";

A=true;

При использовании любого другого типа компилятор не дал бы создать исполняемый файл, а в данном примере никаких проблем не возникает и переменная может принимать любое значение. Таким же образом вы можете создавать м сивы.

Variant A[4];

А[0]=1;

А[1]=2.5;

А[2]="ПРИВЕТ";

A[3]=true;

## Математические функции

| **Функция** | **Значение** |
| --- | --- |
| **abs (n)** | Абсолютное значение n |
| **sqrt (n)** | Квадратный корень из n |
| **exp(n)** | Экспонента n |
| **random (n)** | Случайное целое число в диапазоне от 0 до n-1 (перед первым обращением к функции необходимо вызвать функцию **randomize ()**, которая выполнит инициализацию программного генератора случайных чисел) |
| **sin (a)** | Синус выраженного в радианах угла а |
| **cos (a)** | Косинус выраженного в радианах угла а |
| **tan (a)** | Тангенс выраженного в радианах угла а |
| **asin(n)** | Угол (в радианах), синус которого равен n |
| **acos (n)** | Угол (в радианах), косинус которого равен n |
| **atan(n)** | Угол (в радианах), тангенс которого равен n |

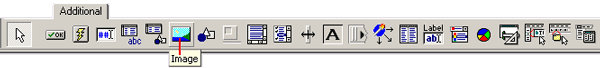
Для того чтобы в программе были доступны приведенные функции, в ее текст надо включить директиву **#include <math.n>**.

Величина угла тригонометрических функций должна быть выражена в радианах. Для преобразования величины угла из градусов в радианы используется формула **(а \* 3.1415256) / 180**, где а– величина угла в градусах; 3.1415926– число "пи".

Вместо константы 3.1415926 можно использовать стандартную именованную константу **M\_PI**, которя определена в файле **math**. **h**.

**Image**

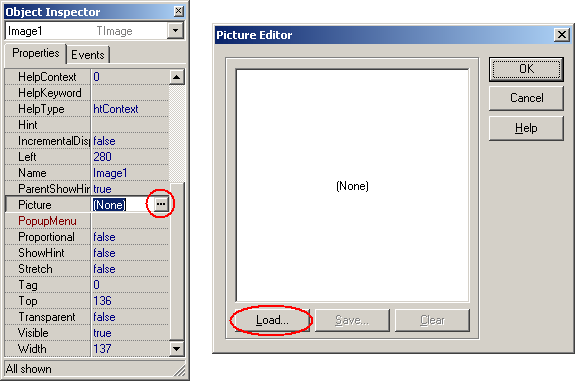
Компонент Image обеспечивает вывод на поверхность формы иллюстраций, представленных в формате .bmp, .jpg или .ico (чтобы компонент можно было использовать для отображения иллюстраций в формате JPG, надо подключить модуль JPEG – включить в текст программы директиву #include <jpeg.hpp>).

  
Свойства компонента Image.

|  |  |
| --- | --- |
| **Свойство** | **Описание** |
| **Picture** | Иллюстрация, которая отображается в поле компонента |
| **Width, Height** | Размер компонента. Если размер компонента меньше размера иллюстрации, а значение свойств **AutoSize**, **strech** и **Proportional** равно **false**, то отображается часть иллюстрации |
| **Proportional** | Признак автоматического масштабирования картинки без искажения. Чтобы масштабирование было выполнено, значение свойства **AutoSize** должно быть **false** |
| **Strech** | Признак автоматического масштабирования (сжатия или растяжения) иллюстрации в соответствии с реальным размером компонента. Если размер компонента не пропорционален размеру иллюстрации, то иллюстрация будет искажена |
| **AutoSize** | Признак автоматического изменения размера компонента в соответствии с реальным размером иллюстрации |
| **Center** | Признак определяет расположение картинки в поле компонента по горизонтали, если ширина картинки меньше ширины поля компонента. Если значение свойства равно **false**, то картинка прижата к правой границе компонента, если **true** – то картинка располагается по центру |
| **Visible** | Отображается ли компонент, и, соответственно, иллюстрация, на поверхности формы |
| **Canvas** | Поверхность, на которую можно вывести графику (см. табл. П1. 25) |

Иллюстрацию, которая будет выведена в поле компонента **image**, можно задать как во время разработки формы приложения, так и во время работы программы.

Во время разработки формы иллюстрация задается установкой значения свойства **Picture** путем выбора файла иллюстрации в стандартном диалоговом окне, которое становится доступным в результате щелчка на командной кнопке **Load** окна **Picture Editor**, которое, в свою очередь, появляется в результате щелчка на кнопке с тремя точками в строке свойства **Picture.**



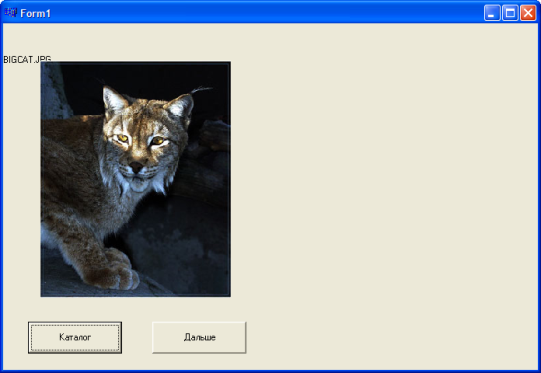
Чтобы вывести иллюстрацию в поле компонента Image во время работы программы, нужно применить метод LoadFromFile к свойству Picture, указав в качестве параметра метода файл иллюстрации. Например, инструкция:

Imagel -› Picture -› LoadFromFile("e:\\temp\\bart.bmp")

Загружает иллюстрацию из файла bart.bmp и выводит ее в поле компонента вывода иллюстрации (Image 1).

По умолчанию компонент Image можно использовать для отображения иллюстраций форматов BMP, ICO и WMF. Чтобы использовать компонент для отображения иллюстраций в формате JPEG (файлы с расширением jpg), надо подключить соответствующую библиотеку (директиву #include <jpeg.hpp>). Если указанной директивы в тексте программы не будет, то компилятор не выведет сообщения об ошибке. Но во время работы программы при попытке загрузить jpg-файл при помощи метода LoadFromFile возникнет ошибка – исключение EInvalidGraphic.

Создадим проект, использующий компонент Image для отображения JPG-иллюстраций.



Кнопка Каталог, в результате щелчка на которой появляется стандартное диалоговое окно Выбор папки, позволяет пользователю выбрать каталог, в котором находятся иллюстрации. Кнопка Дальше обеспечивает отображение следующей иллюстрации.

#include <vcl.h>

#pragma hdrstop

#include "Unit1.h"

#pragma package(smart\_init)

#pragma resource "\*.dfm"

#include <jpeg.hpp> // обеспечивает работу с JPEG-иллюстрациями

#include <FileCTRL.hpp> // для доступа к функции SelectDirectory

TForm1 \*Form1;

\_\_fastcall TForm1::TForm1(TComponent\* Owner)

: TForm(Owner)

{ }

AnsiString aPath; // каталог, в котором находится иллюстрация

TSearchRec aSearchRec; // результат поиска файла

// найти и вывести первую картинку

void TForm1::FirstPicture()

{

Image1-> Visible = false; // скрыть компонент Imagel

Button2->Enabled = false; // кнопка Дальше недоступна

Label1->Caption = "";

if (FindFirst(aPath+ "\*.jpg", faAnyFile, aSearchRec) == 0)

{

Image1->Picture->LoadFromFile(aPath+aSearchRec.Name);

Image1->Visible = true;

Label1->Caption = aSearchRec.Name;

if (FindNext(aSearchRec) == 0) // найти след, иллюстрацию

{

// иллюстрация есть

Button2->Enabled = true; // теперь кнопка Дальше доступна

} }

}

void \_\_fastcall TForm1::FormCreate(TObject \*Sender)

{

aPath = ""; // текущий каталог – каталог, из которого

// запущена программа

Image1->AutoSize = false;

Image1 -> Proportional = true;

Button2 -> Enabled = false;

FirstPicture(); // показать картинку, которая

// есть в каталоге программы

}

// щелчок на кнопке Каталог

void \_\_fastcall TForm1::Button1Click(TObject \*Sender)

{

if (SelectDirectory(

"Выберите каталог, в котором находятся иллюстрации", "",aPath)!= 0)

{

// пользователь выбрал каталог и щелкнул на кнопке ОК

aPath = aPath + "\\";

FirstPicture(); // вывести иллюстрацию

}

}

// щелчок на кнопке Дальше

void \_\_fastcall TForm1::Button2Click(TObject \*Sender)

{

Image1->Picture->LoadFromFile(aPath+aSearchRec.Name);

Label1->Caption = aSearchRec.Name;

if (FindNext(aSearchRec)!= 0) // найти след, иллюстрацию

{

// иллюстраций больше нет

Button2->Enabled = false; // теперь кнопка Дальше недоступна

}

}

В Unit1.h прописать: private:

void FirstPicture();

Загрузку и вывод первой и остальных иллюстраций выполняют соответственно функции FirstPicture И NextPicture. Функция FirstPicture вызывает функцию FindFirst для того, чтобы получить имя файла первой иллюстрации. В качестве параметров функции FindFirst передаются:

* имя каталога, в котором должны находиться иллюстрации;
* структура aSearchRec, поле Name которой, в случае успеха, будет содержать имя файла, удовлетворяющего критерию поиска;
* маска файла иллюстрации.

Если в указанном при вызове функции FindFirst каталоге есть хотя бы один файл с указанным расширением, то значение функции будет равно нулю. В этом случае метод LoadFromFiie загружает файл иллюстрации. После загрузки первой иллюстрации функция FirstPicture вызывает функцию FindNext для поиска следующего файла иллюстрации. Если файл будет найден, то кнопка Дальше будет сделана доступной.

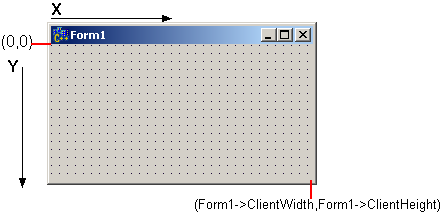
Функция обработки события Onclick на кнопке Дальше загружает следующую иллюстрацию, имя файла которой было найдено функцией FindNext в процессе обработки предыдущего щелчка на кнопке Дальше, и снова вызывает функцию FindNext для поиска следующей иллюстрации. Если файл иллюстрации не будет найден, то кнопка Дальше станет недоступной.

**Работа с графикой**

**Canvas** – поверхность формы или компонента **Image**, на которой соответствующие методы могут вычерчивать графические примитивы. Вид графических элементов определяют свойства поверхности, на которой эти элементы вычерчиваются.

Событие **OnPaint** возникает и в момент запуска программы, когда окно появляется на экране в первый раз. Таким образом, инструкции, обеспечивающие вывод графики на поверхность формы, надо поместить в функцию обработки события **onPaint**.

Каждая точка канвы имеет координаты X и Y. Система координат канвы, как и везде в C++Builder, имеет началом левый верхний угол канвы. Координата X возрастает при перемещении слева направо, а координата Y — при перемещении сверху вниз.



**Методы объекта Canvas**

|  |  |
| --- | --- |
| **Метод** | **Описание** |
| **TextOut (х,у, s)** | Выводит строку s от точки с координатами (х, у). Шрифт определяет свойство **Font** поверхности (**canvas**), на которую выводится тест, цвет закраски области вывода текста – свойство **Brush** этой же поверхности |
| **Draw(x,y,b)** | Выводит от точки с координатами (х, у) битовый образ b. Если значение свойства **Transparent** поверхности, на которую выполняется вывод, равно **true**, то точки, цвет которых совпадает с цветом левой нижней точки битового образа, не отображаются |
| **LineTo (x,y)** | Вычерчивает линию из текущей точки в точку с указанными координатами. Вид линии определяет свойство **Реn** |
| **MoveTo(x,y)** | Перемещает указатель текущей точки в точку с указанными координатами |
| **PolyLine (p1)** | Вычерчивает ломаную линию. Координаты точек перегиба задает параметр р1 – массив структур типа TPoint. Если первый и последний элементы массива одинаковы, то будет. вычерчен замкнутый контур. Вид линии определяет свойство **Реn** |
| **Polygon (p1)** | Вычерчивает и закрашивает многоугольник. Координаты углов задает параметр p1 – массив структур типа **TPoint**. Первый и последний элементы массива должны быть одинаковы. Вид границы определяет свойство Реп, цвет и стиль закраски внутренней области – свойство **Brush** |
| **Ellipse (x1,y1, x2,y2)** | Вычерчивает эллипс, окружность или круг. Параметры x1, y1, х2 и у2 задают размер прямоугольника, в который вписывается эллипс. Вид линии определяет свойство **Реn**  Иллюстрированный самоучитель по C++ Builder › Приложение. C++ Builder – краткий справочник. › Графика. Canvas. |
| **Arc(x1, y1,x2, y2, х3,у3,х4,у4)** | Вычерчивает дугу. Параметры x1, y1, x2, у2 определяют эллипс, из которого вырезается дуга, параметры х2, у2, х3, и у4 – координаты концов дуги. Дуга вычерчивается против часовой стрелки от точки (х3, у3) к точке (х4, у4). Вид линии (границы) определяет свойство **Реn**, цвет и способ закраски внутренней области – свойство **Brush**  Иллюстрированный самоучитель по C++ Builder › Приложение. C++ Builder – краткий справочник. › Графика. Canvas. |
| **Rectangle (x1, y1, х2,у2)** | Вычерчивает прямоугольник. Параметры x1, y1, x2 и у2 задают координаты левого верхнего и правого нижнего углов. Вид линии определяет свойство Реп, цвет и способ закраски внутренней области – свойство **Brush** |
| **RoundRec (x1, y1, х2,у2, х3,у3)** | Вычерчивает прямоугольник со скругленными углами. Параметры x1, y1, х2 и у2 задают координаты левого верхнего и правого нижнего углов, х3 и у3 – радиус скругления. Вид линии определяет свойство **Реn**, цвет и способ закраски внутренней области – свойство **Brush**  Иллюстрированный самоучитель по C++ Builder › Приложение. C++ Builder – краткий справочник. › Графика. Canvas. |

**Свойства объекта Canvas**

|  |  |
| --- | --- |
| **Свойство** | **Описание** |
| **Transparent** | Признак использования "прозрачного" цвета при выводе битового образа методом **Draw**. Если значение свойства равно **true**, то точки, цвет которых совпадают с цветом левой нижней точки битового образа, не отображаются |
| **Pen** | Свойство **Реn** представляет собой объект, свойства которого определяют цвет, толщину и стиль линий, вычерчиваемых методами вывода графических примитивов |
| **Brush** | Свойство **Brush** представляет собой объект, свойства которого определяют цвет и стиль закраски областей, вычерчиваемых методами вывода графических примитивов |
| **Font** | Свойство **Font** представляет собой объект, уточняющие свойства которого определяют шрифт (название, размер, цвет, способ оформления), используемый для вывода на поверхность холста текста |

Свойства объекта Реn определяют цвет, стиль и толщину линий, вычерчиваемых методами вывода графических примитивов.

**Свойства объекта Реn**

|  |  |
| --- | --- |
| **Свойство** | **Описание** |
| **Color** | Цвет линии (**clBlack** – черный; **clMaroon** – каштановый; **clGreen** – зеленый; **clOlive** – оливковый; **clNavy** – темно-синий; **clPurple** – розовый; **clTeal** – зелено-голубой; **clGray** – серый; **clSilver** – серебристый; **clRed** – красный; **clLime** – салатный; **clBlue** – синий; **clFuchsia** – ярко-розовый; **clAqua** – бирюзовый; **clWhite** – белый) |
| **Style** | Стиль (вид) линии. Линия может быть: **psSolid** – сплошная; **psDash** – пунктирная (длинные штрихи); **psDot** – пунктирная (короткие штрихи); **psDashDot** – пунктирная (чередование длинного и короткого штрихов); **psDashDotDot** – пунктирная (чередование одного длинного и двух коротких штрихов); **psClear** – не отображается (используется, если не надо изображать границу, например, прямоугольника) |
| **Width** | Толщина линии задается в пикселах. Толщина пунктирной линии не может быть больше 1 |

Свойства объекта **Brush** пределяют цвет, стиль закраски внутренних областей контуров, вычерчиваемых методами вывода графических примитивов.

**Свойства объекта Brush**

|  |  |
| --- | --- |
| **Свойство** | **Определяет** |
| **Color** | Цвет закрашивания замкнутой области |
| **Style** | Стиль (тип) заполнения области (**bsSolid** – сплошная заливка; **bsClear** – область не закрашивается; **bsHorizontal** – горизонтальная штриховка; **bsVertical** – вертикальная штриховка; **bsFDiagonal** – диагональная штриховка с наклоном линий вперед; **bsBDiagonal** – диагональная штриховка с наклоном линий назад; **bsCross** – горизонтально-вертикальная штриховка, в клетку; **bsDiagCross** – диагональная штриховка, в клетку) |

## Многоугольник

Метод **Polygon** вычерчивает многоугольник. Инструкция вызова метода в общем виде так:

Canvas -> Polygon(p,n)

Где **р** – массив записей типа **TPoint**, который содержит координаты вершин многоугольника; **n** – количество вершин.

Метод **Polygon** чертит многоугольник, соединяя прямыми линиями точки, координаты которых находятся в массиве: первую со второй, вторую с третьей, третью с четвертой и т. д.

Пример1. Нарисуем закрашенный треугольник и ромб.

void \_\_fastcall TForm1::FormPaint(TObject \*Sender)

{

int x,y; TPoint p[4];

x=30; y=250;

Canvas->Pen->Color=clRed;

Canvas->Brush->Color=clYellow;

p[0].x=x;p[0].y=y;

p[1].x=x+100; p[1].y=y-100;

p[2].x=x+200; p[2].y=y;

p[3].x=x;p[3].y=y;

Canvas->Polygon(p,3);

// ромб

p[0].x = 50; p[0].y = 100;

p[1].x = 150; p[1].y = 75;

p[2].x = 250; p[2].y = 100;

p[3].x = 150; p[3].y = 125;

Canvas->Pen->Color=clBlue;

Canvas ->Brush ->Color = clGreen;

Canvas -> Polygon(p,3);

}

**Ломаная линия**

Метод **Polyline** вычерчивает ломаную линию. В качестве параметров методу передается массив типа **TPoint**, содержащий координаты узловых точек линии, и количество звеньев линии. Метод **Polyline** вычерчивает ломаную линию, последовательно соединяя точки, координаты которых находятся в массиве: первую со второй, вторую с третьей, третью с четвертой и т. д.

Например, приведенный ниже фрагмент кода рисует ломаную линию, состоящую из трех звеньев.

void \_\_fastcall TForm1::FormPaint(TObject \*Sender)

{

TPoint p[4];

p[0].x = 100; p[0].y = 100;

p[1].x = 100; p[1].y = 150;

p[2].x = 150; p[2].y = 150;

p[3].x = 150; p[3].y = 100;

Canvas -> Polyline(p,3);

}

**Прямоугольник**

Вместо четырех параметров – координат двух диагональных углов прямоугольника – методу **Rectangle** можно передать один параметр – структуру типа **TRect**, поля которой определяют положение диагональных углов прямоугольной области. Например,

TRect ret; // прямоугольная область

ret.Top = 10;

ret.Left = 10;

ret.Bottom = 50;

ret.Right = 50;

Canvas -> Rectangle(ret); // нарисовать прямоугольник

Метод **FillRect** вычерчивает закрашенный прямоугольник, используя в качестве инструмента только кисть (**Brush**), а метод **FrameRect** – только контур и использует только карандаш (**Реn**). У этих методов только один параметр – структура типа **TRect**. Поля структуры **TRect** содержат координаты прямоугольной области. Значения полей структуры **TRect** можно задать при помощи функции **Rect**.

Например:

TRect ret; // область, которую надо закрасить

ret = Rect(10,10,30,50); // координаты области

Canvas -> Brush-> Color = clRed; // цвет закраски

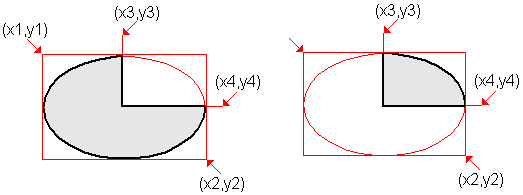
Canvas -> FillRect(ret);

**Сектор**

Метод **Pie** вычерчивает сектор эллипса или круга. Инструкция вызова метода в общем виде выглядит следующим образом:

Canvas -> Pie(x1,y1,х2,у2,х3,у3,х4,у4)

Параметры x1, y1, x2, y2 определяют эллипс (круг), частью которого является сектор; х3, у3, х4 и у4 – прямые – границы сектора. Начальная точка границ совпадает с центром эллипса. Сектор вырезается против часовой стрелки от прямой, заданной точкой с координатами (х3, у3), к прямой, заданной точкой с координатами (х4, у4) (рис. 3.5).

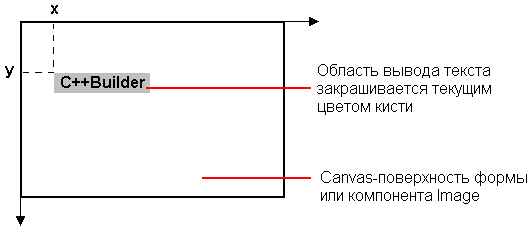


## Текст

Вывод текста (строк типа **Ansistring**) на поверхность графического объекта обеспечивает метод **TextOutA**. Инструкция вызова метода **TextOutA** в общем виде:

Canvas -› TextOutA(x,y,Teкст)

Параметр Текст задает выводимый текст. Параметры х и у определяют координаты точки графической поверхности, от которой выполняется вывод текста.

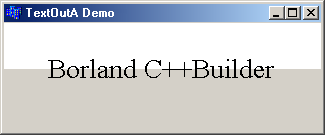


Шрифт, который используется для вывода текста, определяется значением свойства **Font** соответствующего объекта **canvas**. Свойство **Font** представляет собой объект типа **TFont**.

Свойства объекта TFont.

|  |  |
| --- | --- |
| **Свойство** | **Определяет** |
| **Name** | Используемый шрифт. В качестве значения следует использовать название шрифта (например, **Arial**) |
| **Size** | Размер шрифта в пунктах (**points**). Пункт– это единица измерения размера шрифта, используемая в полиграфии. Один пункт равен 1/72 дюйма |
| **Style** | Стиль начертания символов. Может быть: нормальным, полужирным, курсивным, подчеркнутым, перечеркнутым. Стиль задается при помощи следующих констант:  **fsBold** (полужирный),  **fsltalic** (курсив),  **fsUnderline** (подчеркнутый),  **fsStrikeOut** (перечеркнутый).  Свойство style является множеством, что позволяет комбинировать необходимые стили. Например, инструкция, которая устанавливает стиль "полужирный курсив", выглядит так: **Canvas › Font › Style = TFontStyles () <<fsBold<<fsUnderline** |
| **Color** | Цвет символов. В качестве значения можно использовать константу типа **TColor** |

При выводе текста весьма полезны методы **Textwidth** и **TextHeight**, значениями которых являются соответственно ширина и высота области вывода текста, которые, очевидно, зависят от характеристик используемого шрифта. Обоим этим методам в качестве параметра передается строка, которую предполагается вывести на поверхность методом **TextOutA**.

**Пример.** Закрасьте верхнюю половину окна белым, нижнюю – голубым цветом, затем в центре окна, по границе закрашенных областей, выведите текст “Borland C++ Builder”.

void \_fastcall TForml::FormPaint(TObject \*Sender)

{

AnsiString ms = "Borland C++Builder";

TRect aRect;

int x,y; // точка, от которой будет выведен текст

// верхнюю половину окна красим белым

aRect = Rect(0,0,ClientWidth,ClientHeight/2);

Canvas-> Brush -> Color = clWhite;

Canvas -› FillRect(aRect);

// нижнюю половину окна красим голубым

aRect = Rect(0,ClientHeight/2,ClientWidth,ClientHeight);

Canvas -› Brush -› Color = clSkyBlue;

Canvas -› FillRect(aRect);

Canvas -› Font -› Name = "Times New Roman";

Canvas -› Font -› Size = 24;

// Canvas -› Font -› Style = TFontStyles ()<<fsBold<< fsItalic;

// текст разместим в центре окна

х = (ClientWidth – Canvas->TextWidth(ms)) /2;

у = ClientHeight/2 – Canvas->TextHeight(ms) /2;

Canvas -› Brush -› Style = bsClear; // область вывода текста

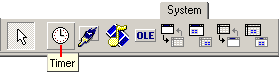
// не закрашивать

Canvas -› Font -› Color = clBlack; Canvas › TextOutA(x,y,ms);

// вывести текст

}

**Компонент Timer**

Компонент **Timer** обеспечивает генерацию последовательности событий **onTimer**.

Значок компонента **Timer** находится на вкладке **System.** Следует обратить внимание, что компонент **Timer** является невизуальным. Это значит, что во время работы программы компонент в диалоговом окне не отображается. Поэтому компонент **Timer** можно поместить в любую точку формы.

Свойства компонента Timer.

|  |  |
| --- | --- |
| **Свойство** | **Определяет** |
| **Name** | Имя компонента. Используется для доступа к свойствам компонента |
| **Interval** | Период возникновения события **OnTimer**, Задается в миллисекундах |
| **Enabled** | Разрешение работы. Разрешает (значение **true**) или запрещает (значение **false**) возникновение события **OnTimer** |

Компонент Timer генерирует событие OnTimer. Период возникновения события OnTimer измеряется в миллисекундах и определяется значением свойства interval. Свойство Enabled дает возможность программе "запустить" или "остановить" таймер. Если значение свойства Enabled равно false, то событие OnTimer не возникает.

**Мультипликация. Метод базовой точки**

Под мультипликацией обычно понимается движущийся и меняющийся рисунок. В простейшем случае рисунок может только двигаться или только меняться.

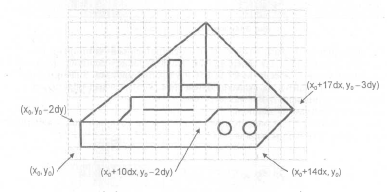
Обеспечить перемещение рисунка довольно просто: надо сначала вывести рисунок на экран, затем через некоторое время стереть его и снова вывести этот же рисунок, но уже на некотором расстоянии от его первоначального положения. Подбором времени между выводом и удалением рисунка, а также расстояния между старым и новым положением рисунка (шага перемещения), можно добиться того, что у наблюдателя будет складываться впечатление, что рисунок равномерно движется по экрану.

**Метод базовой точки**

Задача. Нарисуйте с помощью графических методов движущийся кораблик.

На поверхности формы находится один-единственный компонент **Timer**, который используется для генерации последовательности событий, функция обработки которых обеспечивает вывод и удаление рисунка.

В рассматриваемой программе вывод изображения выполняет функция **Ship**, которая рисует на поверхности формы кораблик. В качестве параметров функция **Ship** получает координаты базовой точки. Базовая точка (х, у) определяет положение графического объекта в целом; от нее отсчитываются координаты графических примитивов, образующих объект. Координаты графических примитивов можно отсчитывать от базовой точки не в пикселах, а в относительных единицах. Такой подход позволяет легко выполнить масштабирование изображения.



**Листинг программы:**

#include <vcl.h>

#pragma hdrstop

#include "Unit1.h"

#pragma package(smart\_init)

#pragma resource "\*.dfm"

TForm1 \*Form1;

\_\_fastcall TForm1::TForm1(TComponent\* Owner)

: TForm(Owner)

{

}

int x = -68, y = 50; // начальное положение базовой точки

// рисует на поверхности формы кораблик

void TForm1::Ship(int x, int y)

{

int dx=4,dy=4; // шаг сетки

// корпус и надстройку будем рисовать

// при помощи метода Polygon

TPoint p1[7]; // координаты точек корпуса

TPoint p2[8]; // координаты точек надстройки

TColor pc,bc; // текущий цвет карандаша и кисти

// сохраним текущий цвет карандаша и кисти

pc=Form1->Canvas->Pen->Color;

bc = Form1->Canvas->Brush->Color;

// установим нужный цвет карандаша и кисти

Canvas->Pen->Color = clBlack;

Form1->Canvas->Brush->Color = clWhite;

// рисуем . . .

// корпус

p1[0].x = x; p1[0].y = y;

p1[1].x = x; p1[1].y = y-2\*dy;

p1[2].x = x+10\*dx; p1[2].y = y-2\*dy;

p1[3].x = x+11\*dx; p1[3].y = y-3\*dy;

p1[4].x = x+17\*dx; p1[4].y =y-3\*dy;

p1[5].x = x+14\*dx; p1[5].y =y;

p1[6].x = x; p1[6].y =y;

Form1->Canvas->Polygon(p1,6);

// надстройка

p2[0].x = x+3\*dx; p2[0].y = y-2\*dy;

p2[1].x = x+4\*dx; p2[1].y = y-3\*dy;p2[2].x = x+4\*dx; p2[2].y = y-4\*dy;

p2[3].x = x+13\*dx; p2[3].y = y-4\*dy;

p2[4].x = x+13\*dx; p2[4].y = y-3\*dy;

p2[5].x = x+11\*dx; p2[5].y = y-3\*dy;

p2[6].x = x+10\*dx; p2[6].y = y-2\*dy;

p2[7].x = x+3\*dx; p2[7].y = y-2\*dy;

Form1->Canvas->Polygon (p2 , 7 ) ;

Form1->Canvas->MoveTo(x+5\*dx,y-3\*dy) ;

Form1->Canvas->LineTo (x+9\*dx,y-3\*dy) ;

// капитанский мостик

Form1->Canvas->Rectangle (x+8\*dx, y-4\*dy,x+11\*dx, y-5\*dy) ;

//труба

Form1->Canvas->Rectangle (x+7\*dx, y-4\*dy,x+8\*dx, y-7\*dy) ;

// иллюминаторы

Form1->Canvas->Ellipse(x+11\*dx,y-2\*dy,x+12\*dx,y-1\*dy) ;

Form1->Canvas->Ellipse(x+13\*dx,y-2\*dy,x+14\*dx,y-1\*dy) ;

// мачта

Form1->Canvas->MoveTo(x+10\*dx,y-5\*dy) ;

Form1->Canvas->LineTo (x+10\*dx, y-10\*dy) ;

// оснастка

Form1->Canvas->Pen->Color = clWhite;

Form1->Canvas->MoveTo(x+17\*dx,y-3\*dy) ;

Form1->Canvas->LineTo (x+10\*dx, y-10\*dy) ;

Form1->Canvas ->LineTo (x, y-2\*dy) ;

// восстановим цвет карандаша и кисти

Form1->Canvas->Pen->Color = pc;

Form1->Canvas->Brush->Color = bc;

}

// обработка события On Timer

void \_\_fastcall TForm1::FormCreate(TObject \*Sender)

{

/\* Таймер можно настроить во время разработки программы

(в процессе создания формы) или so время работы программы. \*/

// настройка и запуск таймера

Form1->Timer1->Interval = 100; // период события OnTimer -0.1 сек.

Form1->Timer1->Enabled = true; // пуск таймера

}

void \_\_fastcall TForm1::Timer1Timer(TObject \*Sender)

{

// стереть кораблик — закрасить цветом, совпадающим

// с цветом фона (формы)

Form1->Canvas->Brush->Color = Form1->Color;

Form1->Canvas->FillRect(Rect(x-1,y+1,x+68,y-40) ) ;

// вычислить координаты базовой точки

x+=3;if (x > ClientWidth) {

// кораблик "уплыл" за правую границу формы

x= -70; // чтобы кораблик "выплывал" из-за левой границы формы

y=random(Form1->ClientHeight) ;

}

// нарисовать кораблик на новом месте

Ship(x, y);

}

Прописать в Unit1.h в разделе private:

void Ship(int x,int y);

**Функции манипулирования датами и временем**

Большинству функций манипулирования датами в качестве параметра передается переменная типа **TDateTime**, которая хранит информацию о дате и времени.

Для того что бы в программе были доступны функции **DayOf**, **WeekOf**, **MonthOf** и др., в ее текст надо включть директиву:

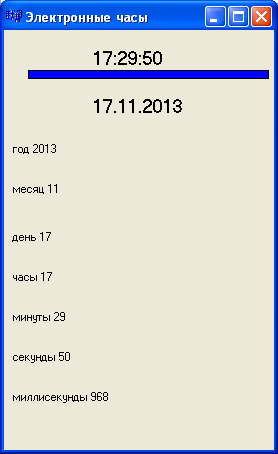
# include <DateUtils.hpp>

Функции манипулирования датами и временем.

|  |  |
| --- | --- |
| **Функция** | **Значение** |
| **Now()** | Системная дата и время – значение типа TDateTime |
| **DateToStr(dt)** | Строка символов, изображающая дату в формате dd.mm.yyyy |
| **TimeToStr(dt)** | Строка символов, изображающая время в формате hh:mm:ss |
| **DayOf (dt)** | День (номер дня в месяце), соответствующий дате, указанной в качестве параметра функции |
| **MonthOf (dt)** | Номер месяца, соответствующий дате, указанной в качестве параметра функции |
| **WeekOf (dt)** | Номер недели, соответствующий дате, указанной в качестве параметра функции |
| **YearOf (dt)** | Год, соответствующий указанной дате |
| **DayOfWeek(dt)** | Номер дня недели, соответствующий указанной дате: 1 – воскресенье, 2 – понедельник, 3 – вторник и т. д. |
| **StartOfWeek(w)** | Дата первого дня указанной недели |
| **HourOf (dt)** | Количество часов |
| **MinuteOf (dt)** | Количество минут |
| **SecondOf (dt)** | Количество секунд |
| **DecodeDate (dt,y,m, d)** | Возвращает год, месяц и день, представленные отдельными числами |
| **DecodeTime (dt,h,m, s,ms)** | Возвращает время (часы, минуты, секунды и милисекунды), представленное отдельными числами |
| **FormatDateTime (s, dt)** | Строка символов, представляющая собой дату или время. Способ представления задает строка формата s, например, строка dd/mm/yyyy задает, что значением функции является дата, а строка hh:mm – время |

**Пример1. Электронные часы**

Написать программу, на поверхности формы которой отображаются текущие время и дата.

unsigned short y,m,d,h,min,s,ms;

void \_\_fastcall TForm1::Timer1Timer(TObject \*Sender)

{

Label1->Caption=TimeToStr(Now());

Label2->Caption=DateToStr(Now());

DecodeDate(Now(),y,m,d);

Label3->Caption="год "+IntToStr(y);

Label4->Caption="месяц "+IntToStr(m);

Label5->Caption="день "+IntToStr(d);

DecodeTime(Now(),h,min,s,ms);

Label6->Caption="часы "+IntToStr(h);

Label7->Caption="минуты "+IntToStr(min);

Label8->Caption="секунды "+IntToStr(s);

Label9->Caption="миллисекунды "+IntToStr(ms);

}end;

**Пример2. Секундомер.**

При щелчке на кнопке Старт в первом текстовом поле отображается текущее время в момент нажатия кнопки. При нажатии на кнопке Стоп выводится текущее время в момент щелчка на кнопке. При щелчке на кнопке Время в третьем окне показывается, сколько времени прошло между щелчками на кнопках Старт и Стоп.

|  |  |
| --- | --- |
|  | TDateTime t1,t2,t;  void \_\_fastcall TForm1::Button1Click(TObject \*Sender)  {  t1=Now(); Edit1->Text=TimeToStr(t1);}  void \_\_fastcall TForm1::Button2Click(TObject \*Sender)  {  t2=Now(); Edit2->Text=TimeToStr(t2);}  void \_\_fastcall TForm1::Button3Click(TObject \*Sender)  {  t=t2-t1; Edit3->Text=TimeToStr(t);} |

**Операторы**

Все операторы языка C++ могут быть условно разделены на следующие категории:

* операторы условного перехода, к которым относятся оператор условия if и оператор выбора switch;
* операторы цикла (for, while, do while);
* операторы перехода (break, continue, return, goto).

Операторы могут быть простыми или составными. Простой не содержит в себе другие операторы. Операторы в программе могут объединяться в составные с помощью фигурных скобок. Любой оператор в программе может быть помечен. Метка состоит из имени и следующего за ним двоеточия. Все операторы, кроме составных, заканчиваются знаком «;».

**Операторы условного перехода Оператор if**

Условный оператор позволяет выполнять определенную часть программы только при выполнении некоего условия. В простейшем случае оператор принимает с дующий вид:

if (a>b) c=a-b; else c=a+b;

Проверяемое условие должно быть заключено в скобки. Если условие в скобках верно, то выполняется оператор, который следует за ними, иначе выполняется оператор, который следует за словом else. Если должно быть выполнено несколько операторов, то они оформляются в виде составного, то есть их нужно разместить в фигурных скобках.

Пример 1.

if (a>b){c=a-b; a++;} else {c=a+b; a--;}

Допускается наличие вложенных операторов if. Если не заключить отдельно каждый вложенный блок в фигурные скобки, то компилятор связывает каждое ключевое слово else с наиболее близким if.

Пример 2.

int x=17,y=14,z=5;

if (x>y)

if (x>z) max=x;

else max=z;

else if (y>z) max=y;

else max=z;

**Оператор switch**

Оператор switch предназначен для выбора из множества различных вариант.

Пример 1.

switch (ZNAC)

{

case'+': x=y+z; break;

case ‘-': x=y-z; break;

case '\*': x=y\*z; break;

case '/': x=y/z; break;

default : ;

}

Выражение в круглых скобках, следующее за ключевым словом switch, может быть любым допустимым в языке C++ выражением, значение которого должно быть целым. Все константные выражения, которые следуют за ключевым словом case, должны быть уникальными. Обычно в качестве константных выражений используются целые или символьные константы. Список операторов, предназначенных для обработки условия, может быть пустым либо содержать один или более операторов. В операторе switch не требуется заключать последовательно операторов в фигурные скобки.

Схема выполнения оператора switch следующая.

1. Вычисляется выражение в круглых скобках.
2. Вычисленные значения последовательно сравниваются с константными, следующими за ключевыми словами case.
3. Если одно из константных выражений совпадает со значением выражения, то управление передается оператору, помеченному соответствующим ключевым словом case.
4. Если ни одно из константных выражений не равно выражению, то управление передается оператору, помеченному ключевым словом default, а в случае отсутствия управление передается следующему после switch оператору.

Программист должен сам позаботиться о выходе из case, если это необходимо. Для этого используется оператор break. В данном примере анализируется введенный знак. Если он совпадает с одним из перечисленных, то выполняется обработка в соответствующей ветви case, а далее осуществляется выход из оператора.

Если не вставить оператор break, то вначале будет распознаваться введенный символ, затем выполнение перейдет на соответствующую ветвь, но выхода из оператора не будет. Далее будет выполняться следующая строка до завершения обработки.

В пределах одного оператора switch константные выражения, определяющие условие выполнения ветви, не могут иметь одинаковые значения.

**Операторы цикла**

При выполнении программы часто возникает необходимость повторения однотипных действий. Для этих целей используют циклы. Существуют три оператора цикла.

**Оператор for**

Оператор for применяется, когда требуется, чтобы какой-либо цикл выполнялся, фиксированное количество раз. Он имеет следующий формат:

for (выражение\_1; выражение\_2; выражение\_3) тело цикла;

Выражение\_1 обычно используется для установления начального значения переменных, управляющих циклом.

Выражение\_2 — это выражение, определяющее условие, при котором тело цикла будет выполняться.

Выражение\_3 определяет изменение переменных, управляющих циклом после каждого выполнения тела цикла.

Схема выполнения оператора for следующая.

1. Вычисляется выражение\_1.
2. Вычисляется выражение\_2.
3. Если значение выражения\_2 отлично от 0 (истина), выполняется тело цикла, вычисляется выражение\_3 и осуществляется переход к пункту 2. Ее выражение\_2 равно 0 (ложь), то управление передается на оператор, следующий за оператором for.

Например, вычисление суммы чисел от 1 до 10:

for (int sum = 0,i=1; i<=10;i + + ) sum=sum + i;

В следующем примере цикл никогда не выполнится, так как начальное значение переменной i меньше условия выполнения цикла:

for (int sum=0,i=10;i<10;i--) sum=sum+i;

**Оператор while**

Оператор цикла while используется в циклах с предусловием и имеет следующий формат:

while (выражение) тело;

В качестве выражения допускается использовать любое выражение языка С а в качестве тела — любой оператор, в том числе пустой или составной. Схема выполнения оператора while такова.

1. Вычисляется выражение.
2. Если выражение ложно, то выполнение оператора whi 1е заканчивается и полняется следующий по порядку оператор. Если выражение истинно, то полняется тело оператора while.
3. Процесс повторяется с пункта 1.

При выполнении оператора while, как и в операторе for, вначале проверяв условие. Поэтому while удобно использовать в ситуациях, когда тело операт не всегда нужно выполнять.

Внутри операторов for и while можно использовать локальные переменные, торые должны быть объявлены с определением соответствующих типов.

Сумма чисел от 1 до 10 может быть найдена с помощью оператора while:

int sum=0; int i=1; while (i<=10)

{

sum+=i; i++;

}

**Оператор do while**

Оператор do while называется оператором цикла с постусловием и используетсяся в тех случаях, когда необходимо выполнить тело цикла хотя бы один раз.

Схема выполнения оператора do while такова.

1. Выполняется тело цикла (которое может быть составным оператором).
2. Вычисляется выражение.
3. Если выражение ложно, то выполнение оператора do while заканчивается и выполняется следующий по порядку оператор. Если выражение истина, то выполнение оператора продолжается с пункта 1.

Можно найти сумму чисел от 1 до 10 с помощью этого вида цикла:

int sum=0; int i=1; do

{ sum+=i;

i++;

}

while (i<=10);

Однако если изменить условия выхода из цикла, то он все равно будет выполняться один раз, так как проверка выхода из цикла осуществляется только после его прохода:

int sum=0; int i=1; do

{ sum+=i;

i++;

}

while (i>10);

**Операторы перехода**

**Оператор break**

Оператор break влияет на выполнение операторов цикла (for, while, do while) и оператора switch. Выполнение оператора break внутри цикла приводит к выходу из цикла, в котором он содержится, и переходу к следующему за циклом оператору. Если оператор break выполняется внутри вложенных циклов, то выход осуществляется только из того цикла, внутри которого он находится. В операторе switch обработка информации на каждой из ветвей может заканчиваться oператором break, чтобы выйти из оператора, а не идти на обработку следующей ветви.

В следующем примере выход из цикла произойдет, когда переменная i будет равна 6.

for (int i=1;i<=10; i++) if (i>5) break;

**Оператор continue**

Оператор continue используется только внутри операторов цикла. Этот оператор позволяет перейти к следующей итерации цикла, не завершив текущей. В примере будет найдена сумма только четных чисел от 1 до 10:

for (int i=1, sum=0; i<=10; i++)

{

if (i%2) continue;

sum+=i;}

**Оператор return**

Оператор return завершает выполнение функции, в которой он задан, и возвращает управление в вызывающую функцию, в точку, непосредственно следующую за вызовом. Оператор return следует использовать либо для немедленного выхода из функции, либо для передачи возвращаемого значения. Формат оператора:

return [выражение];

Значение выражения, если оно задано, возвращается в вызывающую функцию в качестве значения вызываемой функции. Если выражение опущено, то возвращаемое значение не определено. Выражение может быть заключено в круглые скобки, хотя их наличие не обязательно.

Если в какой-либо функции отсутствует оператор return, то управление передается в вызывающую функцию после выполнения последнего оператора вызываемой функции. При этом возвращаемое значение не определено. Если функция не должна иметь возвращаемого значения, то ее нужно объявлять с типом void.

**Компоненты ввода и отображения чисел, дат и времени**

В библиотеке визуальных компонентов C++Builder существует ряд компонентов, позволяющих вводить, отображать и редактировать числа, даты и время.

|  |  |  |  |
| --- | --- | --- | --- |
| **Пиктограмма** | **Компонент** | **Страница** | **Описание** |
|  | **UpDown** (кнопка-счетчик) | Win32 | Кнопка-счетчик, в сочетании с компонентами **Edit** и другими позволяющая вводить цифровую информацию. Основное свойство — **Position.** |
|  | **CSpinEdit** (кнопка-счетчик с окном редакти рования) | Samples | Окно редактирования в комбинации с кнопкой-счетчиком. Почти то же, что комбинация **Edit** и **UpDown.** Основное свойство — **Value.** |
|  | **DateTimePicker** (окно ввода дати времени) | Win32 | Ввод даты (с выпадающим календарем) и времени. Основные свойства — Date и Time. |
|  | **MonthCalendar** (окно ввода дат) | Win32 | Ввод дат с выбором из календаря. |
|  | **Calendar** (календарь на указан ный месяц) | Samples | Отображение календаря на указанный месяц. Компонент **DateTimePicker** имеет больше возможностей по вводу дат, чем этот компонент. Основные свойства — **Month** и **Day.** |

**Ввод и отображение целых чисел — компоненты UpDown и CSpinEdit**

В C++Builder имеются специализированные компоненты, обеспечивающие ввод целых чисел — UpDown и CSpinEdit.

Компонент UpDown превращает окно редактирования Edit в компонент, в котором пользователь может выбирать целое число, изменяя его кнопками со стрелками. Если к тому же установить true в свойство Readonly для Edit, то пользователь просто не сможет ввести в окно какой-либо свой текст и вынужден будет ограничиться выбором числа.

Иначе это удобно, если требуемое число далеко от указанного по умолчанию, а шаг приращения Increment в UpDown мал. Но тут проявляется серьезный недостаток компонента UpDown: ничто не мешает пользователю ввести по ошибке не цифры, а какие-то другие символы. Чтобы избавиться от этого недостатка, лучше использовать компонент CSpinEdit. Компонент CSpinEdit представляет собой сочетание Edit и UpDown, оформленное как отдельный тип компонента.

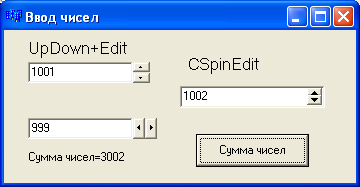
**Свойства компонента UpDown:**

|  |  |
| --- | --- |
| Свойства | Описание |
| Associate | Устанавливает связь кнопки со стрелками с одним из оконных компонентов |
| AlignButton | Определяет, слева или справа от окна будут размещаться кнопки. Значения udLeft или udRight |
| Orientation | Определяет, расположатся ли кнопки по вертикали или по горизонтали. Значения udHorizontal или udVertical |
| ArrowKeys | Определяет, будут ли управлять компонентом клавиши клавиатуры со стрелками. Значения true или false. |
| Thousands | Определяет наличие или отсутствие разделительного пробела между каждыми тремя цифрами разрядов вводимого число Значения true или false. |
| Min и Мах | Задают соответственно минимальное и максимальное значения чисел |
| Increment | Задает приращение числа при каждом нажатии на кнопку |
| Position | Определяет текущее значение числа. Это свойство можно читать, чтобы узнать, какое число задал пользователь. |
| Wrap | Определяет, как ведет себя компонент при достижении максимального или минимального значений. Если Wrap = false, то при увеличении или уменьшении числа до максимального или минимального значения это число фиксируется на предельном значении и нажатие кнопки, пытающейся увеличить максимальное число или уменьшить минимальное, ни к чему не приводит. Если же Wrap = true, то попытка превысить максимальное число приводит к его сбросу на минимальное значение. |

Свойства компонента CSpinEdit похожи на рассмотренные, только имеют другие имена: свойства Min, Max, Position называются соответственно MinValue, MaxValue, Value.

Компонент CSpinEdit во многих отношениях удобнее простого сочетания UpDown и Edit. Так что, если не требуются какие-то из описанных выше дополнительных возможностей UpDown (нестандартное расположение кнопок, «закольцовывание» изменений и т.п.), то лучше пользоваться компонентом CSpinEdit.

**Пример.** Найти сумму чисел, который вводятся в компоненты UpDown и CSpinEdit.



Код программы:

void \_\_fastcall TForm1::Button1Click(TObject \*Sender)

{

int y;

y=StrToInt(Edit1->Text)+StrToInt(Edit2->Text)+ CSpinEdit1->Value;

Label2->Caption="Сумма="+IntToStr(y);

}

**Ввод и отображение дат и времени — компоненты   
DateTimePicker, MonthCalendar, CCalendar**

Из этих компонентов наиболее удобным является DateTimePicker (на форме, слева вверху показан этот компонент в режиме ввода времени, а ниже — в двух вариантах режима ввода даты). Компонент очень эффектен за счет появления выпадающего календаря и обеспечивает безошибочный с точки зрения синтаксиса ввод дат и времени.

**Свойства DateTimePicker**

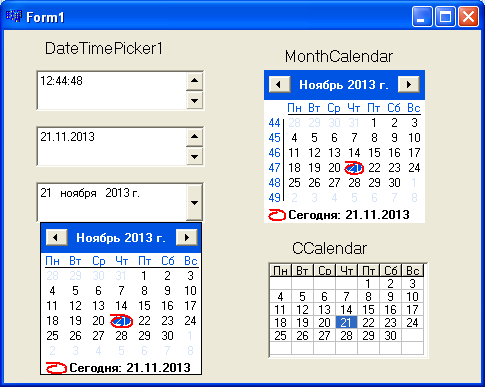
|  |  |
| --- | --- |
| Свойства | Описание |
| Kind | Определяет режим работы компонента: dtkDate - ввод даты, dtkTime - ввод времени. |
| DateMode | Значение dmComboBox - наличие выпадающего календаря. Значение dmUpDown - наличие кнопок увеличения и уменьшения С помощью кнопок пользователь может независимо устанавливать число, месяц и год. |
| DateFormat | Значения dfShort — краткий формат (например, «08.03.13»), или dfLong — полный формат (например, «8 Март 2013г.»). |
| Date | Задаёт значение даты по умолчанию. При чтении Date надо учитывать тип этого свойства - TDateTime, представляющий собой число с плавающей запятой, целая часть которого содержит число дней, отсчитанное от некоторого начала календаря, а дробная часть равна части 24-часового дня, т.е. характеризует время и не относится к дате. За начало календаря принята дата 12/30/1899 00 часов. |
| Time | Отображает введенное пользователем значение. |
| MaxDate и MinDate | Определяют соответственно максимальную и минимальную дату, которую может задать пользователь. |

Для преобразования значения свойства Date в строку можно воспользоваться функцией DateToStr. Например, оператор Memol->Lines->Add(“Дата: “ + DateToStr (DateTimePickerl->Date)); добавит в окно Memol строку вида "Дата: 08.03.13".

Преобразовать время в строку можно функцией TimeToStr.

Компонент MonthCalendar похож на компонент DateTimePicker, работающий в режиме ввода дат. В компоненте MonthCalendar предусмотрены некоторые дополнительные возможности:

|  |  |
| --- | --- |
| Свойства | Описание |
| MultiSelect | Допускает множественный выбор дат в некотором диапазоне |
| WeekNumbers | Указывать в календаре номера недель с начала года |
| FirstDayOfWeek | Перестраивает календарь, задавая первый день каждой недели |

Компонент CCalendar представляет собой менее красочный и более обыденно оформленный календарь на один месяц.

|  |  |
| --- | --- |
| Свойства | Описание |
| Year | Год (целое число) |
| Month | Месяц (целое число) |
| Day | День (целое число) |
| UseCurrentDate | Значение true устанавливает календарь на текущий месяц |
| StartOfWeek | Задает день, с которого начинается неделя. По умолчанию задано 1 — понедельник, как это принято в западных календарях. |

**Работа с несколькими формами**

**Задача.** Осуществить переход между формами по щелчку.

Для того, чтобы добавить новую форму, необходимо выполнить команду File/New/Form или щелкнуть на кнопке .

Тогда в разделе программного кода появится модуль Unit2, предназначенный для второй формы.

Запуск программы будет по умолчанию осуществляться с запуска первой формы.

Чтобы менять формы местами, их нужно поочередно показывать и скрывать. За данные действия отвечают соответственно методы Show и Hide.

Чтобы осуществить переход между формами по щелчку, необходимо запрограммировать следующие процедуры.

*В модуле первой формы:*

#include "Unit2.h"

……………………

void \_\_fastcall TForm1::FormClick(TObject \*Sender)

{

Form2->Show();

Form1->Hide();

}

*В модуле второй формы:*

#include "Unit1.h"

……………………

void \_\_fastcall TForm2::FormClick(TObject \*Sender)

{

Form1->Show();

Form2->Hide();

}

**Объект Shape (Фигура)**

Размещается на вкладке Additional (Дополнительная) .

Вид фигуры задается свойством shape, которое может принимать следующие значения:

stCircle — круг;

stEllipse — эллипс;

stRectangle— прямоугольник;

stRoundRect— прямоугольник со скругленными углами;

s.tRoundSquare — квадрат со скругленными углами;

stSquare — квадрат.

Для задания цвета и способа заливки используется свойство Brush (Кисть).

**Встроенные окна диалога**

Такие диалоги выгодно отличаются от нестандартных окон тем, что разработчику не составляет большого труда их создать или отобразить на экране. Простота использования объясняется тем, что они вызываются при помощи стандартных функций, параметры которых задают внешний вид и функциональные особенности того или иного диалога.

**Окно ввода данных**

Окно ввода – это стандартное диалоговое окно, которое появляется на экране в результате вызова функции InputBox. Значением данной функции является строка, которую ввел пользователь.

Общий вид:

**InputВох(заголовок, подсказка, значение)**

Здесь:

*заголовок –* строковое значение, задающее название окна;

*подсказка –* текст поясняющего сообщения;

*значение* - принимаемое по умолчанию значение, которое выводится автоматически в текстовом поле;

**Задание: *Создать проект «Сумма чисел»***

***void \_\_fastcall TForm1::Button1Click(TObject \*Sender)***

***{ int a,b;***

***a=StrToInt(InputBox("Слагаемые","Введите первое число", "3"));***

***b=StrToInt(InputBox("Слагаемые","Введите первое число", "4"));***

***Label1->Caption="Сумма чисел="+IntToStr(a+b);***

***}***

**Задание: *Создать проект «Фамилия, имя, отчество»***

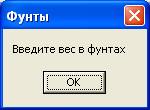
Описание задачи. Экранная форма проекта содержит три текстовых поля и командную кнопку ПУСК. После щелчка этой кнопки должно появится Окно ввода для ввода фамилии пользователя. После ввода своей фамилии пользователь должен щелкнуть кнопку ОК на этом окне, после чего должно появиться второе Окно ввода для ввода имени пользователя. Введя свое имя и щелкнув кнопку ОК, пользователь увидит новое (третье) Окно ввода – для своего отчества.

Все результаты должны отобразиться в текстовых полях (тип AnsiString).

**Окно вывода сообщений**

Окна сообщений используются для привлечения внимания пользователя. При помощи окна сообщения программа может, к примеру, проинформировать об ошибке в исходных данных или запросить подтверждение выполнения необратимой операции, например, удаления файла.

Вывести на экран окно с сообщением можно при помощи процедуры ***ShowMessage*** или функции ***MessageDlg***.

**1**. Процедура **ShowMessage** выводит на экран окно с текстом и командной кнопкой ОК.

Общий вид:

ShowMessage(Сообщение);

где Сообщение – это строка, которая будет выведена в окне

Например: ShowMessage(“Введите вес в фунтах”);

Замечание: В заголовке окна сообщения, выводимого процедурой ShowMessage указано название приложения, которое можно задать командой Project / Options / вкладка Application / поле Title. Если название приложения не задано, то в заголовке будет имя исполняемого файла.

2. Функция **MessageDlg** более универсальная. Она позволяет поместить в окно с сообщением один из стандартных значков, задать количество и тип командных кнопок и определить, какую из кнопок нажал пользователь.

Значение функции – число, которое соответствует командной кнопке, нажатием на которую был завершен диалог.

Общий вид

MessageDlg(сообщение, тип, кнопки, Контекст\_справки)

Первый задает строку, которая будет показываться в нашем сообщении (Как и все строки в C++Builder, заключаем ее в двойные кавычки).

Второй параметр определяет иконку на нашем Messagebox и сообщение в его заголовке. Возможные значения для этого параметра:

|  |  |  |
| --- | --- | --- |
| Значение | Вид сообщения | Пиктограмма |
| mtError | ошибка |  |
| mtConfirmation | вопрос |  |
| mtWarning | предупреждение |  |
| mtInformation | информация |  |
| mtCustom | пользовательское | Без значка |

Обратите внимание, что при записи любого из перечисленных значений Messagebox получит и соответствующий заголовок, кроме последнего. В нем заголовок будет совпадать с именем exe-файла.

Третий параметр задает командные кнопки диалогового окна (mbYes, mbNo, mbOK, mbCancel, mbHelp, mbAbort, mbRetry, mbIgnore и mbAll). Если кнопок несколько, то они объединяются посредством символов << .Так же необходимо написать ключевое слово TMsgDlgButtons().

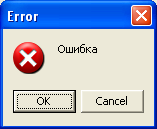
Пример: TMsgDlgButtons() << mbYes << mbNo.

Контекст\_справки задает раздел справочной системы программы, который появится в результате нажатия кнопки Help или клавиши <F1>. Если справочная система не используется, значение параметра должно быть 0.

Например:

int a=MessageDlg("Ошибка", mtError,TMsgDlgButtons() << mbOK<< mbCancel,0);

Возвращаемое значение – это целое число от 1 до 8. Оно зависит от того, какая из кнопок нажата.

В зависимости от возвращаемого функцией MessageDlg значения разработчик может предусмотреть выполнение приложением тех или иных действий.

Константы, возвращаемые диалогом MessageDlg

|  |  |  |
| --- | --- | --- |
| **Кнопка** | **Возвращаемое значение** | |
| OK | 1 | mrOk |
| Cancel | 2 | mrCancel |
| Abort | 3 | mrAbort |
| Retry | 4 | mrRetry |
| Ignore | 5 | mrIgnore |
| Yes | 6 | mrYes |
| No | 7 | mrNo |
| All | 8 | mrAll |

Если в окне диалога имеется кнопка Cancel, то в этом случае щелчок по ней эквивалентен нажатию клавиши Esc.

**Работа с элементами выбора**

При создании приложений для решения большинства практических задач, необходимо предусмотреть выбор пользователем определенных элементов или настроек из тех которые имеются на форме. Реализация данных возможностей осуществляется при помощи следующих элементов управления:

* CheckBox (флажок);
* RadioButton (переключатель);
* ListBox (список).
* Компонент TCheckBoxComboBox (комбинированный список);

**1. Флажок (CheckBox).** Распологается на вкладке Standard Палитры компонентов:

Этот компонет предназначен для включения/выключения некоторого параметра. Например, на форме можно расположить CheckBox для отметки того, получены деньги за некоторый заказ или нет, или, например, для включения/выключения звуков в настройках некоторой программы.

Компонент CheckBox (Флажок) используется для того, чтобы пользователь мог указать свое решение типа да/нет или да/нет/не уверен (в последнем случае флажок выглядит блеклым). Это решение отражается в свойстве State компонента, доступном как для чтения, так и для записи. В составе диалогового окна может быть несколько компонентов CheckBox, причем состояние любого из них никак не зависит от состояния остальных.

Некоторые флажки могут находиться также в частично установленном состоянии, когда признак установки флажка отображается приглушенным цветом. Такая возможность нужна, когда требуется сообщить пользователю о неполном соответствии указанному статусу (например, в ходе установки программ, таким образом, сообщается, что для установки выбраны не все доступные средства).

**Свойства компонента CheckBox**

|  |  |
| --- | --- |
| **Свойство** | **Описание** |
| Name | Имя компонента. Используется для доступа к свойствам компонента |
| Caption | Текст, который находится справа от флажка |
| Checked | Состояние, внешний вид флажка:  если флажок установлен (в квадратике есть "галочка"), то Checked = True;  если флажок сброшен (нет "галочки"), то Checked =False |
| State | Состояние флажка. В отличие от свойства Checked, позволяет различать установленное, сброшенное и промежуточное состояния. Состояние флажка определяет одна из констант:  cbChecked (галочка стоит), cbUnchecked (галочка не стоит) и cbGrayed (CheckBox находится в неопределенном состоянии) |
| AllowGrayed | Свойство определяет, может ли флажок быть в промежуточном состоянии:  если AllowGrayed = False, то флажок может быть только установленным или сброшенным;  если AllowGrayed =True, то допустимо промежуточное состояние (циклически меняет три состояния) |
| Left | Расстояние от левой границы флажка до левой границы формы |
| Top | Расстояние от верхней границы флажка до верхней границы формы |
| Height | Высота поля вывода поясняющего текста |
| Width | Ширина поля вывода поясняющего текста |
| Font | Шрифт, используемый для отображения поясняющего текста |
| ParentFont | Признак наследования характеристик шрифта родительской формы |

Свойства Height и Width определяют размеры прямоугольника, в котором выводится связанный с флажком текст, и никак не влияют на размеры самого флажка.

Чтобы реагировать на изменение состояния флажка, надо создать обработчик события OnClick Рассмотрим пример, когда при изменении состояния флажка его текущее состояние выводится в надпись Label1. Для этого надо разместить в форме соответствующий компонент, установить значение свойства AllowGrayed равным True, сформировать обработчик события OnClick и записать в нем оператор выбора текущего состояния флажка.

void \_\_fastcall TForm1::FormCreate(TObject \*Sender)

{

CheckBox1->State= cbGrayed;

Label1->Caption="Установлен частично";

}

//---------------------------------------------------------------------------

void \_\_fastcall TForm1::CheckBox1Click(TObject \*Sender)

{

switch (CheckBox1->State)

{

case cbUnchecked: Label1->Caption="Сброшен";break;

case cbChecked: Label1->Caption="Установлен";break;

case cbGrayed: Label1->Caption="Установлен частично"; break;

}

}

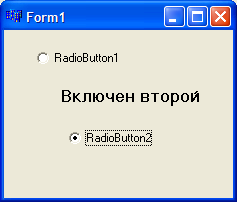
***2.* ПереключательRadioButton**

Переключатели используются в таких ситуациях, когда пользователю следует выбрать одну из нескольких имеющихся возможностей. При этом все добавляемые на форму элементы RadioButton принадлежат к одной группе, следовательно, при выполнении приложения может быть установлен только один из них.

**Свойства компонента *RadioButton***

|  |  |
| --- | --- |
| **Свойство** | **Описание** |
| Name | Имя компонента. Используется для доступа к свойствам компонента |
| Caption | Текст, который находится справа от кнопки |
| Checked | Состояние, внешний вид кнопки: если кнопка выбрана, то Checked = True; если кнопка не выбрана, то Checked = False |
| Left | Расстояние от левой границы флажка до левой границы формы |
| Top | Расстояние от верхней границы флажка до верхней границы формы |
| Height | Высота поля вывода поясняющего текста |
| Width | Ширина поля вывода поясняющего текста |
| Font | Шрифт, используемый для отображения поясняющего текста |
| ParentFont | Признак наследования характеристик шрифта родительской формы |

В форме размещается несколько переключателей, и во время проектирования один из переключателей желательно включить, а все остальные по умолчанию оставить выключенными.

Для динамического отслеживания состояния переключателей создайте обработчик события OnClick для каждого из них. Статус переключателя можно отобразить, например, с помощью надписи Label1.

void \_\_fastcall TForm1::RadioButton1Click(TObject \*Sender)

{

if (RadioButton1->Checked) Label1->Caption="Включен первый";

}

void \_\_fastcall TForm1::RadioButton2Click(TObject \*Sender)

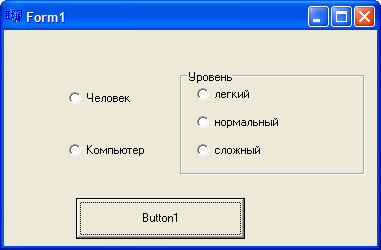
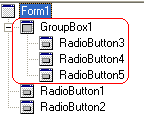
{

if (RadioButton2->Checked) Label1->Caption="Включен второй";

}

Пример. Создадим форму с двумя группами RadioButton на ней. В первой группе будет две RadioButton. Эта группа будет отвечать за выбор противника в некоторой воображаемой компьютерной игре (человек или компьютер). Во второй группе будет три RadioButton. Она будет отвечать за уровень сложности игры (легкий, нормальный и сложный).

Расположнение элементов можно заметить по окну Object TreeView:  
Если вы их расположили неправильно, то просто перетащите их мышкой в нужное место окна Object TreeView. Заголовки групп меняются свойством Caption.



Программный код:

TForm1::RadioButton1Click(TObject \*Sender)

{

String s1, s2;

if(RadioButton1->Checked)

s1="Человек";

if(RadioButton2->Checked)

s1="Компьютер";

if(RadioButton3->Checked)

s2="легкий";

if(RadioButton4->Checked)

s2="нормальный";

if(RadioButton5->Checked)

s2="сложный";

Form1->Caption=s1+" | "+s2;

}

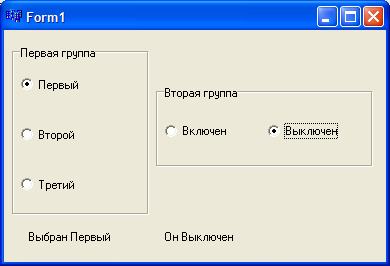
Если элемент управления находится в контейнере, то он является потомком элемента-контейнера. Следовательно, между ними существует тесная связь, в частности, при перемещении созданной таким образом рамки по форме, элементы-потомки также изменяют свое положение, однако их координатные свойства (Top, Left) при этом не изменяются. Если же в контейнер переместить уже имеющийся на форме элемент, то в этом случае связи типа предок-потомок не возникнет.

Если в программе требуется использовать несколько групп переключателей (например, одну для выбора типа устройства, а другую для выбора его состояния), то можно применить один из двух подходов. Первый состоит в выделении для каждой группы специального объекта (GroupBox), чтобы система могла понять, как объединяются переключатели. Второй подход состоит в использовании компонента TRadioGroup (Группа переключателей), который объединяет свойства и методы, обеспечивающие поддержку работы группы переключателей. После размещения в форме компонента TRadioGroup входящие в него переключатели задаются после перечисления их названий. Эти названия вводятся в свойство Items, имеющее тип TString.

Свойства компонента TRadioGroup

|  |  |
| --- | --- |
| **Свойство** | **Описание** |
| Columns | Определяет количество столбцов переключателей |
| ItemIndex | Содержит индекс установленного переключателя |
| Items | Содержит список строк с заголовками элементов. Добавление/удаление элементов достигается добавлением/удалением строк списка **Items** |
| Caption | Определяет заголовок группы (по умолчанию она называется **RadioGroup1** |

После размещения компонента на форме он пуст. Чтобы создать в нем хотя бы один переключатель, следует раскрыть редактор списка Items и ввести хотя бы одну строку: строки Items используются как поясняющие надписи справа от переключателей, а их количество определяет количество переключателей в группе. Заметим также, что после создания компонента его свойство ItemIndex по умолчанию имеет значение -1, это значит, что ни один переключатель в группе не установлен. Чтобы в момент появления компонента на экране какой-то переключатель был уже установлен, необходимо на этапе конструирования с помощью окна инспектора объектов или программно установить в свойство ItemIndex номер соответствующего переключателя (нумерация начинается с 0). Это же свойство позволяет программе проанализировать выбор пользователя.

    Для получения состояния переключателя в группе можно использовать обработчик событий OnClick. Например, чтобы отображать с помощью надписи Label1 название текущего переключателя, обратитесь к свойству Items (Список строк) и выделите тот элемент, номер которого записан в свойстве ItemIndex. Предварительно следует проверить, имеется ли вообще выделенный переключатель (значение свойства ItemIndex не равно -1).

Пример. Поместите на форму компоненты: RadioGroup1, RadioGroup2, Label1, Label2. Создайте переключатели для компонентов RadioGroup1, RadioGroup2. Для компонента RadioGroup1 создайте обработчик OnClick:

void \_\_fastcall TForm1::RadioGroup1Click(TObject \*Sender)

{

if (RadioGroup1->ItemIndex>-1)

Label1->Caption="Выбран "+RadioGroup1->Items->Strings[RadioGroup1->ItemIndex];

}

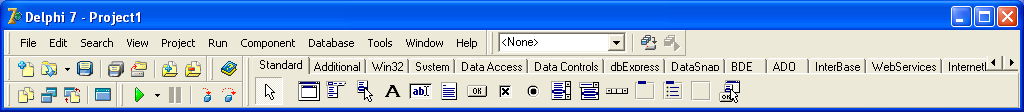
void \_\_fastcall TForm1::RadioGroup2Click(TObject \*Sender)

{

if (RadioGroup2->ItemIndex>-1)

Label2->Caption="Он "+ RadioGroup2->Items->Strings[RadioGroup2->ItemIndex];

}

***3. Простой список ListBox*** 

Компонент ListBox представляет собой список, в котором можно выбрать нужный элемент.

Первоначально компонент TListBox, размещенный в форме, изображается в виде пустого квадрата. Его размеры можно настроить с помощью мыши.

Список может иметь несколько столбцов. Это не означает, что каждый столбец представляет собой отдельный список, просто при заполнении видимой части списка строками донизу очередная строка отображается в следующем столбце. В остальном работать такой список будет так же, как обычный список.

**Свойства компонента *ListBox***

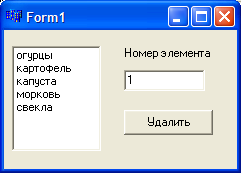
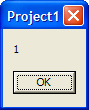
|  |  |
| --- | --- |
| **Свойство** | **Описание** |
| Name | Имя компонента. В программе используется для доступа к компоненту и его свойствам |
| Items | Элементы списка – массив строк |
| Count | Количество элементов списка |
| Sorted | Признак необходимости автоматической сортировки (True) списка после добавления очередного элемента |
| ItemIndex | Номер выбранного (выделенного) элемента списка (элементы списка нумеруются с нуля). Если в списке ни один из элементов не выбран, то значение свойства равно минус 1 |
| Left | Расстояние от левой границы списка до левой границы формы |
| Top | Расстояние от верхней границы списка до верхней границы формы |
| Height | Высота поля списка |
| Width | Ширина поля списка |
| Font | Шрифт, используемый для отображения элементов списка |
| ParentFont | Признак наследования свойств шрифта родительской формы |

Чтобы создать список, необходимо: вставить объект ListBox, выбрать его свойство Items, появится диалоговое окно String list Editor (Редактор списка строк). Введенные в него строки автоматически становятся элементами списка.Нумерация элементов списка идет с нуля.

Создание элементов (пунктов) списка компонента реализуется с помощью методов его свойства Items:

1. Если же вам надо добавить новый пункт на этапе работы программы, то надо воспользоваться методом **Add**:   
   ListBox1->Items->Add("New String");   
   Этот метод добавит новый элемент в конец списка.
2. Чтобы добавить новый элемент куда-нибудь в середину, то используем метод **Insert**:  
   ListBox1->Items->Insert(1,"Alex");
3. Для очистки всего списка вызываем метод **Clear**:  
   ListBox1->Items->Clear();
4. Если надо удалить i-й элемент ListBox, то используем метод **Delete**:  
   ListBox1->Items->Delete(i);

Пример. Напишите программный код, который при шелчке на ListBox'е покажет в MessageBox'е номер элемента, на котором был произведен щелчок и удалит элемент списка по введенному номеру:



void \_\_fastcall TForm1::ListBox1Click(TObject \*Sender)

{

ShowMessage(IntToStr(ListBox1->ItemIndex));

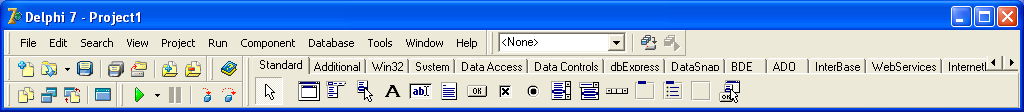
}

void \_\_fastcall TForm1::Button1Click(TObject \*Sender)

{

ListBox1->Items->Delete(StrToInt(Edit1->Text));

}

***4. Комбинированный список ComboBox*** 

Компонент ComboBox дает возможность ввести данные в поле редактирования путем набора на клавиатуре или выбором из списка.

Комбинированный список применяется в тех случаях, когда необходимо выбрать определенное значение какого-либо параметра в раскрывающемся окне или задать в текстовом поле собственное значение в случае его отсутствия в списке. Если все имеющиеся элементы не помещаются в границах стандартного окна, то на нем автоматически появляются полосы прокрутки, с помощью которых можно выбирать требуемый элемент в списке.

**Свойства компонента *СотbоВох***

|  |  |
| --- | --- |
| **Свойство** | **Описание** |
| Name | Имя компонента. Используется для доступа к свойствам компонента |
| Text | Текст, находящийся в поле ввода-редактирования |
| Items | Элементы списка – массив строк |
| Count | Количество элементов списка |
| ItemIndex | Номер элемента, выбранного в списке. Если ни один из элементов списка не был выбран, то значение свойства равно минус 1 |
| Sorted | Признак необходимости автоматической сортировки (True) списка после добавления очередного элемента |
| DropDownCount | Количество отображаемых элементов в раскрытом списке. Если количество элементов списка больше чем DropDownCount, то появляется вертикальная полоса прокрутки |
| Left | Расстояние от левой границы компонента до левой границы формы |
| Top | Расстояние от верхней границы компонента до верхней границы формы |
| Height | Высота компонента (поля ввода-редактирования) |
| Width | Ширина компонента |
| Font | Шрифт, используемый для отображения элементов списка |
| ParentFont | Признак наследования свойств шрифта родительской формы |

Компонент TComboBox (Комбинированный список) представляет собой вариант списка с присоединенным дополнительным полем, в котором отображается выбранный элемент списка. Это же поле может использоваться для ввода новых элементов или для быстрого поиска элемента по начальным символам. Если на экране отображается только поле без списка, то для раскрытия списка можно использовать комбинацию клавиш Alt+вниз, а для выбора элемента списка используются клавиши перемещения (клавиши со стрелками).

Компонент TComboBox работает в пяти разных режимах, определяемых значением свойства Style.

**Значения для свойства Style**

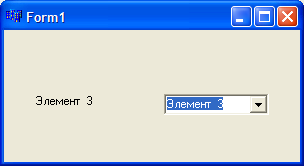
|  |  |
| --- | --- |
| **Значение** | **Механизм работы списка** |
| csDropDown | Создается раскрывающийся список с возможностью редактирования. Свойство MaxLength определяет максимально допустимое число символов, которое можно ввести в это поле (значение 0 указывает на отсутствие ограничений). Текст, введенный пользователем, доступен с помощью свойства Text |
| csDropDownList | Создается раскрывающийся список только для выбора значений, уже имеющихся в списке |
| csSimple | Отличается от стиля csDropDown только тем, что список не является раскрывающимся |
| csOwnerDrawFixed | Создается раскрывающийся список только для выбора значений с возможностью оформления пользователем |
| csOwnerDrawVariable | Создается раскрывающийся список только для выбора значений с возможностью оформления пользователем. Элементы могут иметь разную высот |

Понятия выделенная строка в раскрывающемся списке нет. В нем имеется только текущая выбранная строка (ее номер в списке хранится в свойстве ItemIndex). Соответственно нельзя и выделять строки списка. Единственный метод, связанный с выделением данных, - это процедура SelectAll, которая выделяет весь текст, введенный пользователем в присоединенное поле. При работе с раскрывающимся списком наиболее важными являются перечисленные ниже события.

Перечислим основные события компонента TComboBox.

   События компонента TComboBox

|  |  |
| --- | --- |
| **Название** | **Условия генерации** |
| OnChange | Пользователь изменил текст в присоединенном поле |
| OnDropDown | Список раскрывается. Это событие необходимо обрабатывать, если содержимое списка может меняться во время работы программы. Тогда в обработчике этого события можно заново сформировать содержимое списка (свойство Items) |

Пример программы с использованием поля со списком.

Поместите на форму компоненты: Label1, ComboBox1.

***Для Form1 создайте обработчик OnCreate:***

void \_\_fastcall TForm1::FormCreate(TObject \*Sender)

{ int i;

for (i=0;i<=5;i++)

ComboBox1->Items->Add ("Элемент "+IntToStr(i));

}

void \_\_fastcall TForm1::ComboBox1Select(TObject \*Sender)

{

int i=ComboBox1->ItemIndex;

Label1->Caption=ComboBox1->Items->Strings[i];

}

**Указатели**

*Указатель* — это переменная, содержащая адрес другой переменной. Значения указателей служат адреса участков памяти, выделенных для объектов конкретных типов. Именно поэтому в определении и описании указателя всегда присутствует обозначение соответствующего ему типа.

Указатели обеспечивают хранения адресов областей памяти. При обработке компилятором переменной, например, int i=20, выделяется память в соответствии с типом (int) и инициализируется значением (20). При этом все обращения к переменной компилятор заменяет адресом области памяти, в которой хранится значение переменной. Адреса получаются автоматически. Программист сам может определить собственные переменные для хранения адресов областей памяти. Такие переменные называются указателями.

Признаком указателя при определении переменной служит символ \*, помещенный перед именем. Например, выражение int \* ipl, \* ip2, i; вводит два указателя на объекты целого типа ipl, ip2 и одну переменную i целого типа. Переменная, объявленная как указатель, хранит адрес памяти и информацию о типе объекта, на который ссылается указатель. Размер памяти, требуемый для адреса, и формат этого адреса зависит от компьютера и реализации компилятора, а также от выбранной модели памяти.

Иногда целесообразно при определении указателя инициализировать его. В качестве начального значения можно задавать следующие:

* явно заданный адрес участка памяти;
* указатель, уже имеющий значение;
* выражение, позволяющее получить адрес объекта.

*Например:*

char ch='a'; //символьная переменная (типа char)

char \*pc1; //неинициализированный указатель на объект типа char

char \*pc2=&ch; //инициализированный указатель на объект типа char

char \*pc3=NULL; //нулевой указатель на объект типа char

char \*pc4=(char\*)0xB8000000; //адрес участка памяти

char \*pc5=pc2; //содержимое указателя рс2

Указатель содержит адрес объекта, что дает возможность косвенного доступа к этому объекту через указатель. Например, х — переменная типа int, а рх — указате Переменные, участвующие в программе, необходимо описать: •

int х, у;

int \*px;

px=&x; //чтение адреса переменной x

y=\*px;//присваивание у содержимого того, на что указывает px

Указатели могут появляться и в левой части присваивания. Если рх указывает на х, то выражение \*рх=0; полагает х равным 0.

**Задание 1.** Что обозначает каждая строка? (Подпишите комментарий)

int 1; //целая переменная

const int ci=1; //целая константа

int \*pi; //указатель на целую переменную

const int \*pci; //указатель на целую константу

int \*const ср=&i; //указатель-константа на целую переменную

const int \*const срс=&ci; //указатель-константа на целую константу

**Задание 2.** Что и почему неправильно в этих объявлениях? (объявления независимые)

int var; int \*iptr =&ivar; // ivar не объявлен

int ivar, \*iptr=&ivar; //неинициализированнa переменная ivar

float fvar; int \*iptr=&fvar; //разные типы - указатель на целое и указатель на вещественное.

int ivar, \*iptr; \*iptr = &ivar; //\*iptr имеет тип целое,а пытаются присвоить адрес.

*Получение адреса (&) и определение значения (\*)*

Операция получения адреса (&) дает результат, который зависит от расположения объекта в памяти. Операция & применима к объектам, имеющим имя и размещенным в памяти. Ее нельзя использовать для выражений, неименованных констант, битов полей структур и объединений, для регистровых переменных и внешних объектов (файлов). Следующие операции неверны: &5, &(к+1).

Операция определения значения (\*) иногда называется операцией разыменования. Результат операции зависит не только от значения указателя, но и от типа данных, на который он указывает. Например, указатель, объявленный как char \*ср, работает с участком памяти в 1 байт. Указатель, объявленный как int \* iр работает с участком памяти в 4 байта. Для указателей работает операция преобразования типов, поэтому можно написать следующий код:

unsigned long L=0x12345678L;

char \*cp;

cp=(char\*)&L; //чтение адреса переменной L

//и преобразование его в указатель на тип char

int \*ip;

ip=(int\*)&L; //чтение адреса переменной L

//и преобразование его в указатель на тип int

char c=\*cp; //имеется значение 0x78

int i=\*ip; //имеется значение 0x12345678

Явное преобразование типов необходимо при работе с разными указателя. Можно пользоваться указателем на неопределенный тип void \*, когда преобразование типов будет применимо по умолчанию. Данный тип указателя отличается от других тем, что в нем отсутствует информация о размере участка памяти:

void \*vp;

int i=1;

char ch='a';

vp=&i; //настройка на работу с типом int

vp=&ch; //настройка на работу с типом char

*Сложение и вычитание*

Особенностью арифметических операций с указателем является то, что результат увеличения или уменьшения значения зависит от размера того объекта, на который указатель ссылается. При добавлении к указателю целого числа содержимое указателя будет меняться на величину, кратную размеру его типа.

В операции сложения могут участвовать указатель и величина типа int. Oперация вычитания применима к двум указателям (результатом операции будет расстояние между двумя участками памяти) или к указателю и целой переменной (значение адреса будет уменьшаться на величину, кратную размеру типа).

Следует отметить, что объекты не всегда располагаются в памяти подряд, поэте неизвестно, на что будет указывать измененный указатель. Если работать с массивами, структурами и классами, то здесь есть уверенность в последовательном расположении элементов в памяти, поэтому имеет смысл использовать описанные операции.

int \*ptr1, \*ptr2, a[10];

int i=2;

ptr1=a+(i+4); //равно адресу элемента а[6]

ptr2=ptr1-i; //равно адресу элемента а[4]

Вычисления обычно используются, когда указатели ссылаются на элементы одного массива. Если ра и padr — ссылки на разные массивы, то их разность не имеет смысла.

*Инкремент и декремент*

При выполнении операций ++ и — значение указателя увеличивается или уменьшается на длину типа, на который ссылается используемый указатель. Можно выделить постфиксную и префиксную операции. Они действуют аналогично операциям с целыми числами.

int \*ptr. a[10];

рtг=&а[5];

ptr++; //равно адресу элемента а[б]

ptr—; //равно адресу элемента а[5]

*Сравнение указателей*

Значения двух указателей можно сравнивать в операциях ==, !=, <, <=, >, >=, при этом значения указателей рассматриваются просто как целые числа, а результат сравнения равен 0 (ложь) или 1 (истина).

int \*ра1. \*ра2, а[10];

ра1=а+5; //равно адресу элемента а[5]

ра2=а+7; //равно адресу элемента а[6]

if (pa1>pa2) a[3]=4; //pal меньше значения ра2, поэтому

//оператор а[3]=4 не будет выполнен

*Указатели и массивы*

Существует взаимосвязь между указателями и массивами. Любую операцию, которую можно выполнить с помощью индексов массива, можно произвести также с помощью указателей. Описание int A[10] определяет массив размера 10,то есть набор из десяти последовательных объектов, называемых А[0], А[1],...,А[9]. Запись А [ i ] соответствует элементу массива через i позиций от начала. Имя массива — это указатель на первый байт первого элемента массива поэтому, используя операцию \*, можно получить доступ к любому элементу массива.

intA[5];

int \*pA, с;

//операции присваивания адреса начала массива

рА=&А[0];

рА=А;

//получение адреса элемента с индексом i

pA=&A[i];

pA=A+i;

//чтение содержимого элемента с индексом 0

с=А[0];

с=\*А;

//чтение содержимого элемента с индексом i

c=A[i];

c=\*(A+i);

*Указатель на указатель*

Указатель — это объект в памяти, поэтому можно определить указатель на указатель.

int i=1;

int \*pi=&i;

int \*\*ppi=&pi;

int \*\*\*pppi=&ppi;

int k=\*\*\*pppi; //для к получится исходное значение i

*Операторы new и delete*

При создании переменных для них выделяется память, которая освобождает когда потребность в ней исчезает.

Однако при работе с большими массивами или массивами, размер которых определяется только во время выполнения программы, следует использовать динамическое распределение памяти. Эта память выделяется из свободной оперативной памяти или свободного пространства на жестком диске.

Динамическое выделение памяти осуществляется с помощью оператора new:

int \*mArray=new int[10];

При динамическом распределении памяти указателю присваивается адрес первого элемента массива. Дальнейшие операции с этим массивом ничем не отличаются от работы со статически созданным массивом.

for (int i=0;i<10;i++)

\*(mArray+i)=i;

После использования массива вся выделенная память должна быть освобождена. Освобождение памяти, выделенной с помощью new, осуществляется с помощью оператора delete:

delete mArray;

Пример динамического создания одномерного массива:

int i;

int dim1=5;

int \*pArrl; //одномерный массив

pArr1=new int[dim1]; //резервирование памяти

for(i=0; i<dim1; i++)

{pArr1[i]=i;} //присвоение значений

delete pArrl; //освобождение памяти

Пример динамического создания двумерного массива:

inti;

int dim1=5, dim2=2;

int \*\*pArr2; //двумерный массив (2 строки и 5 столбцов)

pArr2=new int\*[dim1]; //строки

for (i=0; i<dim1; i++)

{pArr2[i]=new int[dim2]; //столбцы для строки

pArr2[i][0]=i; //первый элемент строки

pArr2[i][1]=i\*i; //второй элемент строки

}

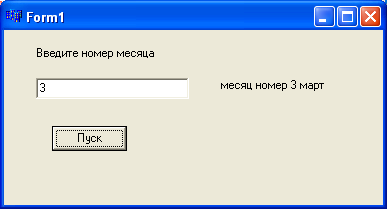
for (i=0; i<dim1; i++)

{delete pArr2[i]; //столбцы для строки

delete[] pArr2; //строки

}

**Пример 1.** Выведите название месяца по его названию.

void \_\_fastcall TForm1::Button1Click(TObject \*Sender)

{

int m=StrToInt(Edit1->Text);

char \*month\_name(int);

Label1->Caption="месяц номер "+IntToStr(m)+" "+month\_name(m);

}

char \*month\_name (int k) /\* Название k-го месяца \*/

{

static char \*name[] = {

"неверный месяц","январь",

"февраль","март","апрель",

"май","июнь","июль","август",

"сентябрь","октябрь","ноябрь",

"декабрь"

};

return (k<1||k>12)?name[0]:name[k];

}

**Задание 1.** Что напечатает следующая программа?

char \*c[]={ "ENTER","NEW","POINT","FIRST" };

/\*c - массив ссылок на символ. Элементы c инициализируются так, что они указывают на массивы символов "ENTER", "NEW", "POINT" и "FIRST".\*/

char \*\*cp[]={ c+3,c+2,c+1,c };

/\*Описатель \*\*cp[] соответствует символу, \*cp-ссылке на символ и cp[]-ссылке на ссылку символ. Элементы cp инициализируются так, что они указывают на элементы c.\*/

char \*\*\*cpp=cp;

/\*Описатель \*\*\*cpp дает символ, \*\*cpp - ссылку на символ, \*cpp-ссылку на ссылку на символ, наконец, cpp - ссылка, указывающая на ссылку на ссылку на символ.\*/

Label1->Caption=(\*\*++cpp);

Label1->Caption=Label1->Caption+((\*(--(\*(++cpp))))+3)+" "+(\*cpp[-2]+3)+(cpp[-1][-1]+1);

Ответ: POINTER STEW

*Решение:*

В рисунках, приведенных ниже, сплошными линиями обозначены указатели, а пунктирными - временные ссылки аргументов для вывода.

Рисунок 1 показывает взаимосвязь между cpp, cp и c.

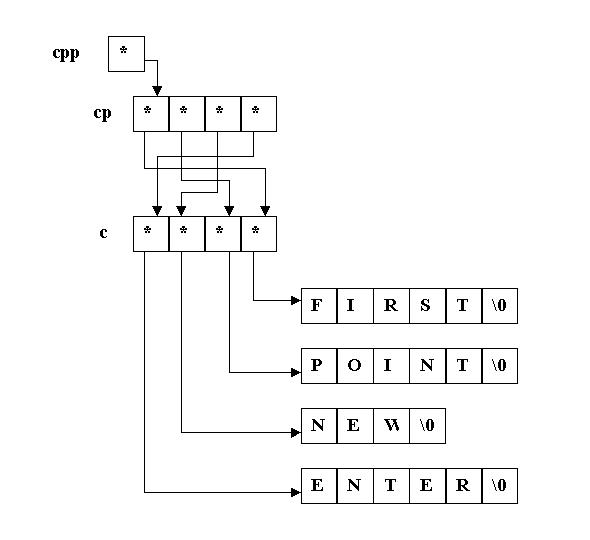
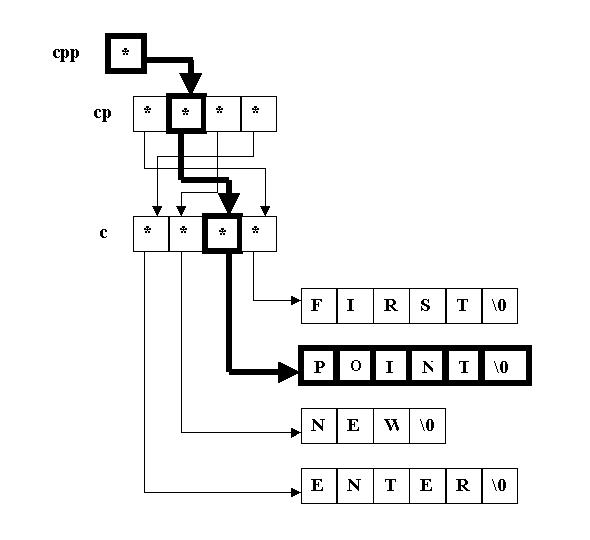
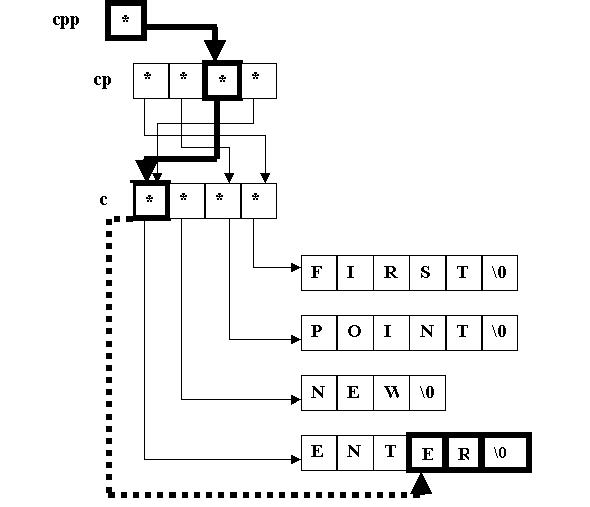


Рис.1. Взаимосвязь между cpp, cp и c

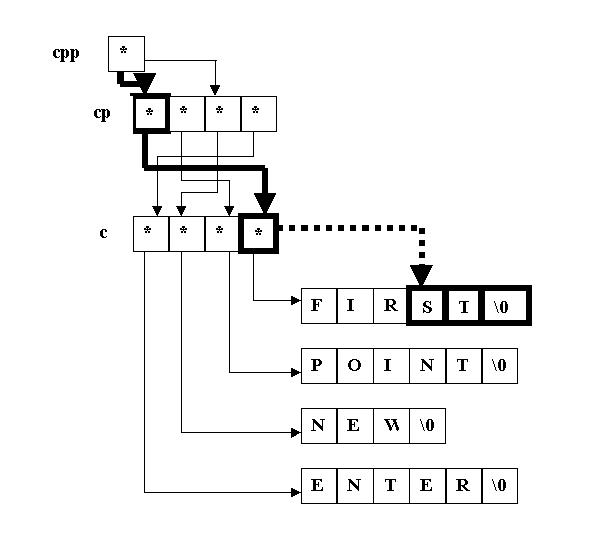
\*(\*(++cpp)) Увеличим cpp, а затем проследим цепочку ссылок. (Рис.2)

  
Рис.2. Цепочка ссылок

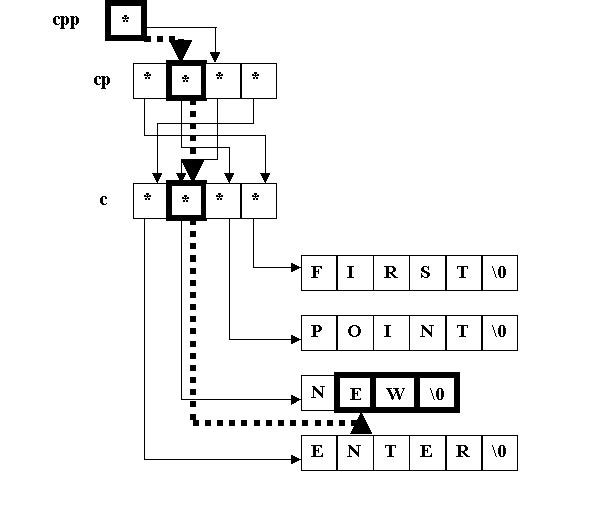
(\*(--(\*(++cpp))))+3 Увеличим cpp, по ссылке дойдем до cp[2], уменьшим cp[2], по ссылке дойдем до c[0] и эту ссылку (т.е. c[0]) увеличим на 3. (Рис.3)

  
Рис.3. Цепочка ссылок

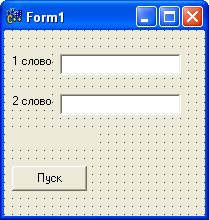
(\*(cpp[(-2)]))+3 Индексируя cpp значением -2, получим cp[0], по ссылке дойдем до c[3] и проиндексируем его значением 3. (Рис.4)

  
Рис.4. Цепочка ссылок

((cpp[-1])[-1])+1 Индексируя cpp значением -1, получим cp[1], снова индексируем с -1 и доходим до c[1], эту последнюю ссылку индексируем значением 1. (Рис.5)

  
Рис.5. Цепочка ссылок

**Задача 1.** Напишите программу для слияния (конкатенации) двух строк, введенных с клавиатуры. Конкатенацию строк описать в функции.

*Программный код:*

void TForm1::f(char \*s1,char \*s2)

{

while(\*s1) \*s1++;

while(\*s1++=\*s2++);

}

void \_\_fastcall TForm1::Button1Click(TObject \*Sender)

{

char \*a= new char[Edit1->Text.Length()+1];

strcpy(a, Edit1->Text.c\_str());

char \*b = new char[Edit1->Text.Length()+1];

strcpy(b, Edit2->Text.c\_str());

f(a,b);

Label1->Caption=a;

}

В Unit1.h в разделе private: необходимо прописать функцию в виде

void f(char \*s1,char \*s2);

**Задача 2.** Напишите программу, удваивающую каждую букву слова, введенного с клавиатуры.

*Программный код:*

void \_\_fastcall TForm1::Button1Click(TObject \*Sender)

{

char \*pa=Edit1->Text.c\_str();

int pr = 2\*strlen(pa);

char b[50],\*pb=b;

\*(pb+pr+1) = '\0';

while ((\*pb = \*pa)!='\0')

{

pb++; \*pb++ = \*pa++;

}

Label1->Caption=b;

}

**Задача 3.** Напишите программу для замены в слове всех букв “a” на сочетание “ку”.

*Программный код:*

void \_\_fastcall TForm1::Button1Click(TObject \*Sender)

{int k=0,i=0, n;

char x2[100];

char \*px1= new char[Edit1->Text.Length()+1];

strcpy(px1, Edit1->Text.c\_str());

n = strlen(px1)+1;

char \*px2; px2 = &x2[0];

while (k<n)

{

if (\*(px1+k)!='а')

{

\*(px2+i) = \*(px1+k); i++; k++;

}

else

{

\*(px2+i) = 'к'; \*(px2+i+1) = 'у'; k++; i += 2;

}

}

Label1->Caption=x2;

}

Дополнительные задачи:

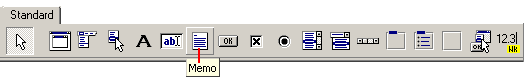
1. Напишите программу для замены в слове всех сочетаний “ку” на букву “a”.
2. Напишите программу, которая заменяет окончание “ing” каждого слова, встречающегося в предложении на “ed”.
3. Напишите программу, которая удваивает все буквы “н” в предложении, введенном с клавиатуры.
4. Напишите программу, которая вычеркивает из слова буквы, стоящие на четных местах.

**Массивы**

Компонент Memo

В некоторых случаях для ввода массива можно использовать компонент Memo. Он позволяет вводить текст, состоящий из достаточно большого количества строк, поэтому его удобно использовать для ввода символьного массива. Компонент находится на вкладке Standart.

Компонент Memo представляет собой элемент редактирования текста, который может состоять из нескольких строк.



|  |  |
| --- | --- |
| **Свойство** | **Описание** |
| **Name** | Имя компонента. Используется для доступа к свойствам компонента |
| **Text** | Текст, находящийся в поле Memo. Рассматривается как единое целое |
| **Lines** | Массив строк, соответствующий содержимому поля. Доступ к строке осуществляется по номеру. Строки нумеруются с нуля. Строка имеет тип AnsiString. |
| **Lines->Count** | Количество строк текста в поле Memo. |
| **ScrollBars** | Полоса прокрутки: ssBoth-горизонтальная и вертикальная, ssHorizontal-горизонтальная, ssVertical-вертикальная, ssNone-нет. |
| **Left** | Расстояние от левой границы поля до левой границы формы |
| **Top** | Расстояние от верхней границы поля до верхней границы формы |
| **Height** | Высота поля |
| **Width** | Ширина поля |
| **Font** | Шрифт, используемый для отображения вводимого текста |
| **Parent Font** | Признак наследования свойств шрифта родительской формы |

Для чистки окна используется метод Memo1->Сlеаr. Для того чтобы добавить новую строку в окно, используется метод Memo1->Lines->Add (переменная типа AnsiString).

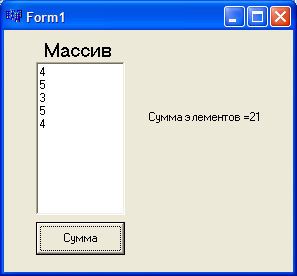
Получить доступ к находящейся в поле Memo строке текста можно при помощи свойства Lines: Memo1->Lines->Strings[i].

Если нужно вывести число, находящееся в переменной действительного или целого типа, то его надо предварительно преобразовать к типу AnsiString и добавить в массив. Например, если переменная u=100 целого типа, то метод Memo1->Line->Add(u) сделает это и в окне появится строка "Значение u=100". Если переменная u=-256,38666 действительная, то при использовании метода Memo1->Lines->Add("Значение u= "+FloatToStrF(u,ffFixed,8,2)) будет выведена строка "Значение u= -256.39". При этом под все число отводится восемь позиций, из которых две позиции занимает его дробная часть.

Если число строк в массиве Memo1 превышает размер окна, то для просмотра всех строк используется вертикальная полоса прокрутки. Если длина строки Memo1 превосходит количество символов в строке окна, то в окне отображается только начало строки. Для просмотра всей строки используется горизонтальная полоса прокрутки.

**Пример 1.** Ввести массив целых чисел через компонент Memo и найти сумму элементов.

*Программный код:*

void \_\_fastcall TForm1::Button1Click(TObject \*Sender)

{

int n=Memo1->Lines->Count;

int s=0;

for (int i=0;i<n;i++)

{s+=StrToInt(Memo1->Lines->Strings[i]);

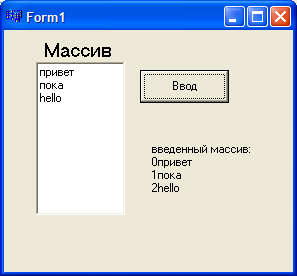
}

Label1->Caption="Сумма элементов ="+IntToStr(s);

}

**Пример 2.** Ввести символьный массив через компонент Memo и вывести его в метку.

*Программный код:*

#define size 5 //размер массива

…

void \_\_fastcall TForm1::Button1Click(TObject \*Sender)

{

AnsiString a[size];

int n=Memo1->Lines->Count;

int i,s=0,f=0;

AnsiString st;

if (n==0)ShowMessage("Исходные данные не введены");else f=1;

if (n>size)

{ShowMessage("Количество строк превышает размер массива");

n=size; }

if (f==1)

{for (i=0;i<n;i++)

{a[i]=Memo1->Lines->Strings[i];}

st="введенный массив:\n";

for (i=0;i<n;i++)

{st=st+IntToStr(i)+a[i]+"\n";}

}

Label1->Caption=st;

}

Дополнительные задачи:

1. Напишите программу для ввода массива целых чисел из 10 элементов с помощью объекта Memo. В массиве найти:
   1. Сумму отрицательных элементов массива;
   2. Произведение последних 5 элементов;
   3. Количество элементов, равных 0. Если таких нет, то вывести соответствующее сообщение;
   4. Заменить каждый второй элемент на противоположенный. Полученный массив вывести в метку;
   5. Найти мах и мin элементы и их номера.
2. Напишите программу для ввода символьного массива целых чисел из 7 элементов с помощью объекта Memo. В массиве найти самую длинную строку.

**Компонент StringGrid**

http://informatics.ssga.ru/_/rsrc/1322971584094/app-inform/pi-practics/pi-practics-lab-10/215.pngКомпонент StringGrid представляет собой таблицу, содержащую строки. Данные таблицы могут быть только для чтения или редактируемыми. Таблица может иметь полосы прокрутки, причем заданное число первых строк и столбцов может быть фиксированным и не прокручиваться. Таким образом, можно задать заголовки столбцов и строк, постоянно присутствующие в окне компонента. Каждой ячейке таблицы может быть поставлен в соответствие некоторый объект.

Компонент StringGrid  находится на вкладке Additional панели компонентов.

Таблица StringGrid состоит из выделенных серым FixedCols и FixedRows - зафиксированных ячеек-заголовков, и обычных, белых ячеек. Содержимое Fixed ячеек недоступно для редактирования, и меняется только программно. Для установки количества фиксированных ячеек можно изменить значение свойств FixedCols и FixedRows в окне свойств Object Inspector. Или поменять количество фиксированных строк и столбцов программно, написав в коде программы команды

StringGrid1-> FixedCols=1; // устанавливает 1 фиксированный столбец

StringGrid1-> FixedRows=1; // устанавливает 1 фиксированную строку

Общее количество строк и столбцов  в таблице можно установить вручную в окне Object Inspector, отредактировав свойства RowCount и ColCount, а можно изменить программно в ходе выполнения программного кода, написав следующие команды:

StringGrid1->RowCount=5 //устанавливаем общее кол-во строк равное 5

StringGrid1->ColCount=7 //устанавливаем общее кол-во столбцов равное 7

Эти свойства доступны как для записи (при установке количества строк и столбцов), так и для чтения (при проверке размер таблицы).

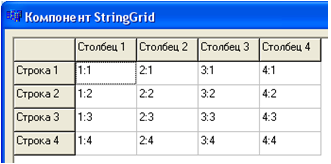
За многие свойства компонента StringGrid отвечает свойство Options. В Инспекторе Объектов Options - это раскрывающийся список, представляющий собой элементы данного множества. Если значение элемента равно True, то он присутствует в множестве, если False - то нет.

|  |  |
| --- | --- |
| Свойство | Значение |
| goFixedVertLine | Наличие вертикальных разделительных линий между "фиксированными" ячейками |
| goFixedHorzLine | Наличие горизонтальных разделительных линий между "фиксированными" ячейками |
| goVertLine | Наличие вертикальных разделительных линий между "обычными" ячейками |
| goHorzLine | Наличие горизонтальных разделительных линий между "обычными" ячейками |
| goRangeSelect | Возможность выделить диапазон ячеек |
| goDrawFocusSelected | Закрашивание ячейки с фокусом ввода |
| goRowSizing | Возможность менять высоту строк мышкой |
| goColSizing | Возможность менять ширину столбцов мышкой |
| goRowMoving | Возможность менять номер строки, то есть перемещать её, мышкой |
| goColMoving | Возможность менять номер столбца, то есть перемещать его, мышкой |
| goEditing | Возможность редактировать содержимое ячейки с клавиатуры |
| goTabs | При значении True фокус смещается на следующую ячейку в таблице, False - на следующий компонент |
| goRowSelect | Выделяется вся строка с "фокусированной" ячейкой |
| goAlwaysShowEditor | При значении True содержимое ячейки при получении фокуса сразу доступно редактированию, False - сначала необходимо щёлкнуть по ней мышкой (прим.: не действует при goRowSelect=True) |
| goThumbTracking | При значении True перемещение "бегунка" прокрутки мышкой вызывает немедленное перемещение ячеек, False - ячейки перемещаются только при отпускании "бегунка" |

Как следует из таблицы, за возможность редактировать содержимое ячеек с клавиатуры отвечает элемент goEditing свойства-множества Options. В Инспекторе Объектов установите его значение в True. Чтобы управлять этой возможностью программно, нужно включить или исключить из множества данный элемент:

StringGrid1->Options<<goEditing;  //Включаем редактирование, другие элементы не трогаем

StringGrid1->Options>>goEditing;  //Выключаем редактирование, другие элементы не трогаем

Компонент StringGrid имеет возможность адресации каждой отдельной ячейки по номеру столбца и строки.

Содержимое ячейки (i, j), где где i - номер столбца, j - номер строки, имеет вид:

StringGrid1->Cells[i][j]

и доступно как для чтения, так и для записи. Например:

StringGrid1->Cells[i][j]=”Hello”; - записывает слово в ячейку с координатами I и j.

A=StringGrid1->Cells[i][j]; - считывает содержимое ячейки с координатами I и j в переменную A.

Номера столбцов (i) и строк (j) отсчитываются от 0.

Выделенная ячейка таблицы имеет:

номер столбца:  StringGrid1->Col

номер строки:   StringGrid1->Row

поэтому содержимое выделенной ячейки будет адресоваться так:

S=StringGrid1->Cells[StringGrid1->Col][StringGrid1->Row];

Ячеек в таблице, как правило, много, и в рамках компонента видна только часть из них. Если ячейки не помещаются в таблице, появляются полосы прокрутки. При прокручивании:

StringGrid1->LeftCol                 Номер столбца, видимого самым левым

StringGrid1->TopRow                 Номер строки, видимой самой верхней

StringGrid1->VisibleColCount Количество столбцов, видимых в рамках таблицы

StringGrid1->VisibleRowCount Количество строк, видимых в рамках таблицы

У таблицы StringGrid также есть свойство и для управления размером ячеек.Для всех ячеек

DefaultRowHeight - высота строк по умолчанию

DefaultColWidth - ширина столбцов по умолчанию

Эти значения ширины и высоты принимают все новые ячейки. При необходимости индивидуально установить ширину и высоту столбцов и строк соответственно, пользуемся свойствами

RowHeights[i] - массив, содержащий высоты строк с номером i

ColWidths[i] - массив, содержащий ширины столбцов с номером i

Все эти свойства можно настроить в обработчике события OnCreate формы, так же как и надписи заголовков, располагающиеся в строках и столбцах "фиксированной" зоны таблицы. В результате таблица появляется уже в "настроенном" виде.

Так как таблица StringGrid, в отличие от, например, компонента ListBox, не имеет метода для очистки содержимого сразу всех ячеек, то для удаления внесённых в таблицу ранее данных приходится очищать каждую ячейку отдельно. Делается это двумя вложенными циклами for, пробегающими по столбцам и строкам:

int i, j;

{

    for (i=1;i<StringGrid1->RowCount;i++)  //Заголовки строк не трогаем

    for (j=1;j<StringGrid1->ColCount;j++)  //Заголовки столбцов не трогаем

    StringGrid1->Cells[j][i]='';

}

Хотя, есть метод для очищения содержимого целого столбца или строки:

  StringGrid1->Cols[i]->Clear(); //Очищается столбец с номером i

  StringGrid1->Rows[i]->Clear(); //Очищается строка с номером i

Очевидно, очищение этими методом происходит гораздо быстрее. Однако будут очищены и ячейки фиксированной зоны, содержащие, например, названия строк и столбцов, которые удалять не нужно. Их после очистки нужно просто "написать" заново, на глаз эта манипуляция совершенно незаметна. Для очистки всей таблицы достаточно последовательно очистить только строки или только столбцы:

int i, j;

{

    for (i=1;i<StringGrid1->RowCount; i++) //Заголовки столбцов не трогаем цикл от 1

      {

        StringGrid1->Rows[i]->Clear();

        StringGrid1->Cells[0][i]="Заголовок строки i";

      }

}

**ТЕСТИРОВАНИЕ ПРОГРАММНОГО ПРОДУКТА**

**Общая характеристика тестирования и его цикл**

Тестирование представляет собой деятельность по проверке программного кода и документации. Она должна заранее планироваться и систематически проводиться специально назначенным независимым тестировщиком. Работа тестировщика начинается до утверждения спецификаций требований. Он проверяет требования к ПП на полноту и возможность тестирования, определяет методы тестирования.

Одновременно с началом этапа планирования и создания спецификаций требований тестировщик разрабатывает стратегию тестирования. После утверждения спецификаций требований им разрабатывается и детализируется план тестирования. Тогда же тестировщик создает наборы тестов для проведения интеграционного и системного тестирований. Тестирование завершается созданием отчета о тестировании, в котором представляются все результаты его проведения.

Для каждого программного изделия должен существовать набор тестов, проверяющий его корректность. Существует несколько уровней тестирования, позволяющих полностью проверить программное изделие. Каждый уровень имеет свои цели и компоненты. Можно выделить пять уровней тестирования: модульное; интеграционное; системное; выходное; приемочное.

Тестирование первых четырех уровней проводится внутри организации, тогда как приемное тестирование выполняется совместно с представителями заказчика. Тестирование первого уровня осуществляет сам разработчик на этапе разработки, за остальные уровни отвечает независимый тестировщик.

Циклом тестирования называется совокупность действий, выполняемых тестировщиком с момента передачи базовой версии ПП тестировщику для интеграционного, системного или приемочного тестирования до момента успешного заверения тестирования (рис. 10.1). На каждом проходе цикла тестирования создаются:

* базовая версия ПП, подлежащего тестированию;
* отчет о ходе тестирования;
* метрики тестирования (заносятся в базу данных проекта).

Создание базовой

версии ПП

Исправление

ошибок

Проведение

тестирования

Анализ

результатов

Создание

отчета

Да

Выявлены ли

ошибки?

Нет

**Виды тестирования**

**Модульное тестирование**

Модульное тестирование — это тестирование программы на уровне отдельно взятых модулей, функций или классов. Цель модульного тестирования заключается в выявлении локализованных в модуле ошибок в реализации алгоритмов, а также в определении степени готовности системы к переходу на следующий уровень разработки и тестирования. Модульное тестирование проводится по принципу «белого ящика», т. е. основывается на знании внутренней структуры программы и часто включает те или иные методы анализа покрытия кода.

Модульное тестирование обычно подразумевает создание вокруг каждого модуля определенной среды, включающей заглушки для всех интерфейсов тестируемого модуля. Некоторые из них могут использоваться для подачи входных значений, другие — для анализа результатов, присутствие третьих может быть продиктовано требованиями, накладываемыми компилятором и сборщиком.

На уровне модульного тестирования проще всего обнаружить дефекты, связанные с алгоритмическими ошибками и ошибками кодирования алгоритмов, типа работы с условиями и счетчиками циклов, а также с использованием локальных переменных и ресурсов. Ошибки, связанные с неверной трактовкой данных, некорректной реализацией интерфейсов, совместимостью, производительностью и т. п., обычно пропускаются на уровне мо- дульного тестирования и выявляются на более поздних стадиях тестирования.

**Интеграционное тестирование.** Этот вид тестирования проводится для проверки совместной работы отдельных модулей и предшествует тестированию всей системы как единого целого. В ходе интеграционного тестирования проверяют связи между модулями, их совместимость и функциональность. Оно осуществляется независимым тестировщиком и входит в состав этапа тестирования. Элементами интеграционного тестирования являются:

проверка функциональности – соответствия отдельных функций, выполняемых совокупность модулей, функциям, заданным в спецификациях требований;

проверка промежуточных результатов – проверка всех промежуточных результатов и файлов на наличие и корректность;

проверка интеграции – проверка того, что модули передают другую информацию корректно.

Ошибки, выявленные в ходе интеграционного тестирования, заносятся в базу данных ошибок. Результаты интеграционного тестирования включаются в отчете тестирования при завершении цикла тестирования.

**Системное тестирование.** Этот вид тестирования предназначен для проверки программной системы в целом, ее организации и функционирования на соответствие спецификациям требований заказчика. Его проводит независимый тестировщик после успешного завершения интеграционного тестирования.

Элементами системного тестирования являются:

граничное тестирование – тестирование в граничных условиях;

прогоночное тестирование – тестирование всех функциональных характеристик реальной работы системы;

целевое тестирование – тестирование на целевой платформе (по возможности);

проверка документации – проверка пользовательской документации на корректность;

другие тесты, определяемые тестировщиком.

Ошибки, выявленные при системном тестировании, заносятся в базу данных проекта. Результаты системного тестирования включаются в отчет о ходе тестирования.

**Выходное тестирование**. Это завершающий этап тестирования, на котором проверяется готовность ПП к поставке заказчику. Данный вид тестирования проводит независимый тестировщик. Элементами выходного тестирования являются:

проверка инсталляции – проверка на ясность и корректность инструкций и инсталляции;

проверка документации – проверка того, что вся необходимая документация полностью подготовлена и готова к передаче заказчику.

Ошибки, выявленные при выходном тестировании, заносятся в базы данных проекта. При успешном завершении выходного тестирования ПП поставляется заказчику вместе с отчетом о результатах тестирования.

**Приемное тестирование.** Этот вид тестирования провидится организацией, отвечающей за инсталляцию, сопровождение программной системы и обучение конечного пользователя.

**Программные ошибки**

Одними из распространенных определений программной ошибки являются следующие два:

* программная ошибка – это расхождение между программой и ее спецификацией, причем тогда и только тогда, когда спецификация существует и она правильна;
* программная ошибка – это ситуация, когда программа не делает того, чего пользователь то нее вполне обоснованно ожидает.

Все программные ошибки модно разделить на соответствующие категории:

*Функциональные недостатки.* Данные недостатки присущи программе, если она не делает того, что должна, выполняет одну из своих функций плохо или не полностью. Функции программы должны быть подобно описаны в ее спецификации, и именно на основе утвержденной спецификации тестировщик строит свою работу.

*Недостатки пользовательского интерфейса.* Оценить удобство и правильность работы пользовательского интерфейса можно только в процессе работы с ним. Желательно, чтобы в этой работе принимал участие сам пользователь. Этого можно добиться с помощью разработки прототипа ПП, на котором проводятся обкатка и согласование всех требований к пользовательскому интерфейсу с дальнейшей фиксацией их в спецификации требований. После утверждения спецификации требований любые отклонения от нее или невыполнение являются ошибкой. Это в полной мере касается и пользовательского интерфейса.

*Недостаточная производительность.* При разработке некоторого ПП важной его характеристикой может оказаться скорость работы, иногда этот критерий задается в требованиях заказчика к ПП. Плохо, если у пользователя создается впечатление, что программа работает медленно, особенно если конкурирующие программы работают ощутимо быстрее, но еще хуже, если программа не удовлетворяет заданным в спецификации требований характеристикам. Это уже ошибка, которая должна быть устранена.

*Некорректная обработка ошибок.* Процедуры обработки ошибок – очень важная часть программы. Правильно определив ошибку, программа должна выдать о ней сообщение. Отсутствие такого сообщения является ошибкой в работе программы.

*Некорректная обработка граничных условий.* Существует много различных граничных ситуаций. Любой аспект работы программы, к которому применимы понятия «больше» или «меньше», «раньше» или «позже», «первый» или «последний», «короче» или «длиннее», обязательно должен быть проведен на границах диапазона. Внутри диапазонов программа может работать прекрасно, а вот на их границах могут происходить самые неожиданные ситуации, которые, в свою очередь, приводят к ошибкам в работе ПП.

*Ошибки вычислений.* К ошибкам вычислений относятся ошибки, вызванные неправильным выбором алгоритма вычислений, неправильными формулами, формулами, неприменимыми к обрабатываемым данным. Самыми распространенными среди ошибок вычислений являются ошибка округления.

*Ошибки управления потоком.* По логике работы программы вслед за первым действием должно быть выполнено второе. Если вместо этого выполняется третье или четвертое действие, значит, в управлении потоком допущена ошибка.

*Ситуация гонок.* Предположим, в системе ожидаются два события: А и Б. Если первым наступит событие А, то выполнение программы продолжится, а если событие Б, то в работе программы произойдет сбой. Разработчики предполагают, что первым всегда должно быть событие А, и не ожидают, что Б может выиграть гонки и наступить раньше. Такова классическая ситуация гонок.

Тестировать ситуации гонок довольно сложно. Наиболее типичны они для систем, где параллельно выполняются взаимодействующие процессы и потоки, а также для многопользовательских систем реального времени. Ошибки в таких системах трудно воспроизвести, и на их выявление обычно требуется очень много времени.

*Перегрузки.* Сбои в работе программы могут происходить из-за нехватки памяти или отсутствия других необходимых системных ресурсов. У каждой программы свои пределы, программа может не справляться с повышенными нагрузками, например со слишком большими объемами данных. Вопрос в том, соответствует ли реальные возможности программы и ее требования к ресурсам спецификации программы и как она себя поведет при перегрузках.

*Некорректная работа с аппаратурой компьютера.* Программы могут посылать аппаратным устройствам неверные данные, игнорировать их сообщения об ошибках, пытаться исправит устройства, которые заняты или вообще отсутствуют. Даже если нужное устройство просто сломано, программа должна понять это, а не «зависать» при попытке к нему обратиться.

**Тестирование документации**

Читая и анализируя документацию, тестировщик прежде всего уделяет внимание ее точности, полноте, ясности, простоте использования и тому, насколько она соответствует ПП. В ходе тестирования документации наверняка будут найдены проблемы по каждому из указанных критериев. Поэтому заранее следует запланировать многократное тестирование печатного руководства, интерактивной справки и других документов.

Тестировщик, работающий с документацией, отвечает за техническую точность каждого ее слова. Он обязан произвести самую тщательную проверку ее соответствия требованиям спецификации и поведению программы. Особо следует обращать внимание на сложные и запутанные места текста. Они могут отражать неудачно спроектированные элементы самой программы. Технический писатель обязан описать продукт таким, каким он является на самом деле, поэтому помочь устранить запутанные места может только изменение проекта. Настаивать на таких изменениях важно еще и потому, что, в конечном счете, они обеспечат не только простоту документирования продукта, но и легкость его использования.

Необходимо проверить, не пропущены ли в документации какие-нибудь функции продукта. Технические писатели опираются на спецификацию, собственные заметки и беседы с разработчиками. Разработчики стараются держать их в курсе дела, но иногда забывают сообщить о новых функциях, только что внесенных в программу. Поскольку тестировщики сталкиваются с этими функциями гораздо раньше технических писателей, стоит позаботиться, чтобы их описания попали в документацию. Кроме того, если определенная функция описана в руководстве, то это не значит, что она будет и в интерактивной справке. Вполне вероятно, что информация может потеряться.

Следует помнить, что тестировщик одинаково не имеет права требовать изменений как в руководстве к ПП, так и в самом ПП. Обязанность тестировщика выявить проблему, а что с ней делать, решать не ему. В частности, у тестировщика нет никакого права требовать стилистических изменений текста. Он может предложить такие изменения, но технический писатель вправе оставить все как есть и не обязан доказывать тестировщику, что поступает правильно. Для взаимодействия с техническими писателями формальная система отслеживания проблем обычно не применяется. Большинство комментариев вносятся прямо в копию руководства.

По договоренности с техническим писателем выбирается способ выделения в тексте правок и комментариев. Копии комментариев следует сохранять и проверять по ним очередные версии документации.

**Разработка и выполнение тестов**

**Требования к хорошему тесту**

Хороший тест должен удовлетворять следующим требованиям:

* *должна быть достаточной вероятность выявления текстом ошибки.* Целью тестирования является поиск возможных ошибок. Поэтому, разрабатывая тестовые примеры, необходимо проанализировать все возможные варианты сбоев программы или ее некорректной работы;
* *набор тестов не должен быть избыточным.* Если два теста предназначены для выявления одной и той же ошибки, то достаточно выполнить только один из них;
* *тест должен быть наилучшим в своей категории.* В группе похожих тестов одни могут быть эффективнее других. Поэтому, выбирая тест, нужно взять тот, который с наибольшей вероятностью выявит ошибку;
* *тест не должен быть слишком простым или слишком сложным.* Огромный и сложный тест трудно понять, трудно выполнить и долго создавать. Поэтому лучше всего придерживаться золотой середины, разрабатывая простые, но все же не совсем элементарные тестовые примеры.

**Классы эквивалентности и граничные условия**

Класс эквивалентности – это набор тестов, от выполнения которых ожидается один и тот же результат. В простейшем случае тест представляет собой набор входных данных, вводимых в тестируемую программу. В случае эквивалентных тестов эти данные обладают общими свойствами.

Группа тестов представляет собой класс эквивалентности, если выполняются следующие условия:

* все тесты предназначены для выявления одной и той же ошибки;
* если один из тестов выявит ошибку, то остальные тоже это сделают;
* если один из тестов не выявит ошибку, то остальные тоже это не сделают.

Кроме перечисленных абстрактных условий существуют еще и практические критерии, позволяющие отнести к одному классу конкретную группу тестов. В качестве таких критериев можно рассматривать соблюдения следующих условий:

* тесты включают в себя значения одних и тех же входных данных;
* для проведения тестов выполняются одни и те же операции программы;
* в результате всех тестов формируется одни и те же значения выходных данных;
* либо ни один из тестов не вызывает блок обработки ошибок программы, либо блок вызывается всеми тестами группы.

Поиск классов эквивалентности – процесс субъективный. Два человека, анализирующие одну и ту же программу, составят различные перечни классов. Необходимо все же стремиться выявить как можно больше классов эквивалентности. Это сэкономит время в дальнейшем и сделает тестирование более эффективным, избавив тестировщика от ненужного повторения эквивалентных тестов. Разбив все предполагаемые тесты на классы, можно затем выделить в каждом из них один или несколько тестов, которые кажутся наиболее эффективными; остальные выполнять ни к чему.

Рекомендации для поиска классов эквивалентности.

*Не забывайте о классах, охватывающих заведомо неверные или недопустимые входные данные.* Часто такие входные данные вызывают в программе самые разнообразные ошибки. Поэтому, чем больше вы выделите типов неверного ввода, тем больше найдете ошибок.

*Организуйте формируемый перечень классов в виде таблицы.* Обычно классов эквивалентности оказывается очень много, поэтому нужен удобный и продуманный способ организации собранной информации. Обычно всю информацию сводят в большую таблицу.

**Перечень классов эквивалентности**

|  |  |  |
| --- | --- | --- |
| Входное или выходное событие | Допустимые классы эквивалентности | Недопустимые классы эквивалентности |
| Ввод числа | Числа от 1 до 99 | Число 0.  Числа больше 99.  Выражение, результатом которого является недопустимое число (например, 5 – 5 результат которого равен 0).  Отрицательные числа.  Буквы и другие нечисловые символы |
| Ввод первой буквы имени | Первый символ является заглавной буквой.  Первый символ является прописной буквой | Первый символ не является буквой |
| Рисование прямой | От одной точки прямая линия длиной до  4 см | Отсутствие рисунка.  Линия длиннее 4 см.  Линия не является прямой |

Обратите внимание, что в перечень включены тесты не только допустимых, но и недопустимых входных данных. Информация в форме таблицы более понятна, ее легче и быстрее воспринимать, более очевидно разделение на допустимые и недопустимые варианты. Информацию в табличном виде легче анализировать, чтобы выяснить, все ли недопустимые варианты входных данных охвачены перечисленными классами эквивалентности.

*Определите диапазоны числовых значений.* С каждым новым диапазоном значений появляются и несколько новых связанных с ним классов эквивалентности. Обычно среди них имеются три недопустимых класса: все числа, которые меньше нижнего граничного значения диапазона; все числа, которые больше его верхнего граничного значения; нечисловые данные. Иногда один из этих классов отсутствует. Например, возможен ввод любого числа. В этом случае необходимо убедится, что это на самом деле так. Следует попробовать ввести очень большое число и посмотреть, что из этого получится.

Необходимо проверить также, нет ли у значений исследуемого параметра поддиапазонов. Каждый поддиапазон будет отдельным классом эквивалентности. Недопустимые классы будут располагаться ниже самого нижнего диапазона и выше верхнего из них.

*Если для полей или параметров существуют фиксированные перечни значений, выясните, какие из значений входят в перечень.* Если для параметра допускается только определенный перечень значений, один из классов эквивалентности может включать все значения из этого перечня, а другой – все остальные значения. В дальнейшем эти два класса можно разделить на ряд меньших классов.

*Проанализируйте возможные результаты выбора из списков и меню*. Любой элемент предложенного программой списка опций может представлять собой отдельный класс эквивалентности. Каждый элемент меню или списка опций обрабатывается программой особым образом, поэтому все они подлежат проверке. К классу недопустимых значений относятся ответы пользователя, которых нет в списке (если программа позволяет не только выбирать, но и вводить значения опций).

Например, если программа задает вопрос «Вы уверены? (Д/Н)», то один класс эквивалентности должен содержать ответ «Д» (также надо проверять и «д»), а второй ответ «Н» («н»). Все остальные ответы являются недопустимыми (хотя вполне возможно, что программа интерпретирует все, что не является положительным ответом, как отрицательный, т.е. как эквивалент ответа «Н»).

*Поищите классы значений, зависящих от времени.* Предположим, что вы нажимаете на клавишу пробела непосредственно перед тем, как система загрузит программу, во время загрузки и сразу после нее. Как ни странно, но подобные тесты могут разрушить систему. Какие классы эквивалентности можно выделить в подобной ситуации? К первому из них относятся все события, происходящие задолго до выполнения задания, ко второму – события, происходящие в короткий отрезок времени непосредственно перед выполнением задания, к третьему – события в период нго выполнения и т.д.

*Выявите группы переменных, совместно участвующих в определенных вычислениях, результат которых ограничивается конкретным набором или диапазоном значений.* Введите величины трех углов треугольника. К классу допустимых относятся значения, в сумме дающие 180°. Недопустимые значения можно разделить на два класса эквивалентности – с суммарным значением менее 180° и более 180°.

*Посмотрите, на какие действия программа отвечает эквивалентными событиями.* Выше рассматривались только входные события, поскольку анализировать из гораздо легче. Как пример выходного события рассмотрим события в табл. 10.1. Программа обеспечивает вычерчивание линии длиной до 4 см, причем предполагается, что линия прямая, хотя она может оказаться и другой формы.

Трудность состоит в том, чтобы определить, какие входные данные управляют длинной и формой линии. Иногда различные классы входных данных на входе дают один и тот же эффект. Если точный путь обработки каждого класса входных данных неизвестен, то лучше интерпретировать их как разные классы и тестировать по отдельности. Особенно это важно в тех случаях, когда в ответ на определенные входные данные при формировании выходной информации генерируется ошибка и управление передается блоку ее обработки.

*Продумайте варианты операционного окружения.* Бывает, что программа хорошо работает только при определенных типах мониторов, принтеров, модемов, дисковых устройств или любого другого подключенного к системе оборудования. Работа программы может зависеть даже от тактовой частоты компьютера. Поэтому, анализирую программу, особенно выполняющую низкоуровневые операции с оборудованием или ориентирующуюся на его определенные возможности, очень важно определить классы эквивалентных конфигураций системы.

**Граница классов эквивалентности.** Для каждого класса эквивалентности достаточно провести один-два теста. Лучшими из них будут те, которые проверяют значения, лежащие на границах класса. Эти значения могут быть наибольшими, наименьшими или какими-то другими, но в любом случае они должны быть предельными значениями параметров класса. Неправильные операторы сравнения (например > вместо >) вызывает ошибки только при граничных значениях аргументов. В то же время программа, которая сбоит при промежуточных значениях диапазона, почти наверняка будет сбоить при его граничных значениях.

Необходимо протестировать каждую границу класса эквивалентности, причем с обеих сторон. Программа, которая пройдет эти тесты, скорее всего, пройдет и все остальные, относящиеся к данному классу. Вот ряд примеров.

Если допустимые значения от 1 до 99, для тестирования допустимых данных можно выбрать 1 и 99, а для тестирования недопустимых – 0 и 100.

Если программа ожидает заглавную английскую букву, введите А и Z. Проверьте также символ @, поскольку его код предшествует коду символа А, и символ ], код которого следует за кодом символа Z. Кроме того, проверьте символы а и z.

Если программа обеспечивает вычерчивание от одной точки линии длиной до 4 см, нарисуйте одну точку и линию длиной ровно 4 см. Пусть также попробует обеспечить вычерчивание линии нулевой длины.

Если сумма выходных значений должна равняться 180, попробуйте ввести значения, дающие в сумме 179, 180 и 181.

Если программа получает определенное число входных данных, попробуйте ввести точно необходимое число, а также на единицу меньше и на единицу больше.

Если программа принимает ответы B, C и D, попробуйте ввести A и E.

Попробуйте отправить на печать файл непосредственно перед тем, кА принтер напечатает еще чье-либо задание, и сразу после того.

**Тестирование переходов между состояниями**

В каждой интерактивной программе осуществляются переходы из одного очевидного состояния в другое. Простейшим примером может служить меню. После запуска программы в нем имеется один перечень команд. После выбора одной из них состояние программы меняется, и в меню появляются команды, доступные в этом новом состоянии.

Необходимо протестировать каждую предлагаемую программой опцию, каждую команду меню. Команда 10 может быть доступна в режиме, открываемом по команде 9 или по команде 22. В этом случае команду 10 придется протестировать дважды – в обоих режимах. Однако команд меню, всевозможных режимов программы и путей перехода в эти режимы может быть так много, что протестировать их все просто нереально. Поэтому, отбирая тесты для проверки путей выполнения программы, лучше всего руководствоваться следующими принципами:

тестировать все наиболее вероятные последовательности действий пользователей;

если можно предложить, что действия пользователя в одном режиме могут влиять на представление данных или набор предоставляемых программой возможностей в другом режиме, тестировать эти действия;

кроме проведения самых необходимых тестов из тех, что описаны выше, поработать с программой в произвольном режиме, случайным образом выбирая путь ее выполнения. Переходы между состояниями могут быть гораздо более сложными, чем просто выбор команд меню. Содержимое и структура очередной формы ввода данных могут зависеть от информации, введенной в предыдущей форме, значения одних полей могут определять допустимые значения других, ввод определенной информации может инициировать серию дополнительных запросов. Например, при вводе чисел от 1 до 99 программа выводит одну форму запроса, обращенного к пользователю, а при вводе любых других чисел – другую. В этом случае вместе с классами эквивалентности и их граничными значениями следует проанализировать и возможные пути выполнения программы, чтобы составить действительно полноценный набор тестов.

Очень полезно составление схем меню. В подобной схеме отражаются все состояния программы и команды, вызывающие переходы между этими состояниями. В нее включаются команды, активизируемые через меню, графические средства (например, различные кнопки), и команды, выполняемые после нажатия определенных клавиш. Например, в схеме может быть показан путь от меню «Файл» к команде «Открыть», затем к диалоговому окну «Открытие файла» и назад, к основному состоянию программы. Особенно удобны подобные схемы в случае, если определенное диалоговое окно модно открыть несколькими способами и выйти из него в несколько различных режимов. В этом случае можно нарисовать на схеме все направления переходов и по ним протестировать программу. Это более надежный способ, чем работать с программой, без всякого плана с риском пропустить важные взаимосвязи ее состояний.

**Условия гонок и другие временные зависимости**

Попробуйте вмешаться в работу программы, когда она выполняет переход между двумя состояниями. Понажимайте на клавиши, особенно командные. Попробуйте понажимать на клавиши или повыбивать какие-либо пункты меню, когда программа выполняет операции обработки данных или ввода – вывода, предложите программе ввести вынести параллельно еще какую-нибудь информацию. Например, во время печати файла попросите ее распечатать еще один.

Если в программе определены ситуации тайм-аута, когда она ждет определенного события в течение заданного времени, а затем переходит в другое состояние, проверьте ее реакцию на действия пользователя, запросы системы или наступление ожидаемого события на границах интервала тайм-аута. Посмотрите, что будет, если событие произойдет за секунду до того, как программа должна прекратить ожидать его, или через секунду после этого.

Протестируйте систему при повышенной нагрузке. В мультизадачной среде запустите несколько других программ и посмотрите, как поведет себя ваша, успешно ли она справится со своей работой. Отправьте большой файл на принтер, чтобы процессор все время переключался на обслуживание печати. Подключите различные внешние устройства и заставьте их генерировать прерывания так часто, как только удастся. Короче говоря, замедлите и нагрузите компьютер, насколько это возможно. В результате ваша программа будет выполняться медленнее, и, быстро вводя данные, можно попробовать превысить ее возможности приема. Если в нормальном режиме работы сбоя программы добиться не удается, это может получиться при повышенной нагрузке.

Выполняя «стандартное» тестирование программы при сильно повышенной нагрузке, можно столкнуться с совершенно неожиданными ситуациями гонок. Если окажется, что программа в этом отношении уязвима, то необходимо провести в таких условиях полный цикл тестирования. Главная задача – обеспечить такую надежность разрабатываемого программного обеспечения, чтобы оно работало, пусть медленно, но без сбоев в любой системе и при любых дополнительных нагрузках. По крайней мере, необходимо совершенно точно выяснить, какие конфигурации системы являются предельными для эксплуатации программы.

**Нагрузочные испытания**

Важно не забывать протестировать те ограничения возможностей ПП, которые определены в его документации. Откройте максимальное число файлов или других структур данных, с которыми программа может работать, попробуйте подольше поэксплуатировать ее в таком состоянии. Если в документации ограничения не описаны, но существует логически допустимые значения каких-либо параметров, то следует проверить и их. Если программа не справляется с достаточно большим числовым значением, которое пользователь вполне может ввести, то составляется отчет об ошибке. Если же программа спокойно принимает и обрабатывает как очень маленькие, так и очень большие значение параметров, возможно, ограничений на них нет.

Необходимо проверить, как ведет себя программа, когда исчерпывается различные аппаратные ресурсы, например, переполняется диск или в принтере заканчивается бумага. Выясните, что будет, когда в системе останется очень мало свободной памяти, нагрузите компьютер как следует и посмотрите, что получится.

Нагрузочное тестирование – это, по сути дела, один из видов тестирования граничных условий. Схема его проведения абсолютно аналогична. Сначала программу запускают в условиях, в которых она должна работать, а затем в условиях, для которых она не предназначена. Имеет смысл проверить и различные комбинации условий. Вполне возможно, что, справившись с различными повышенными нагрузками по отдельности, программа не выдержит их все вместе. Нагрузив систему, проведите не просто один-два теста, а достаточно длительное и обстоятельное тестирование. Поэксплуатируйте программу в таких условиях некоторое время, возможно, сбой не сразу, но все же произойдет.

**Прогнозирование ошибок**

Иногда тестировщик предполагает, что определенный класс тестов вызовет сбой программы, хотя и не может это логически обосновать. Доверяйте своей интуиции и обязательно включайте подобные тесты в общий план. Существует целый ряд ситуаций и значений, которые, хотя и не являются граничными, но часто вызывают программные сбои. Типичным примером таких значений являются 0. Не стоит тратить время на поиски обоснований того, почему определенное входное значение или место программы кажется вам подозрительным. Просто протестируйте его.

Случается, что в сложных ситуациях интуиция подсказывает гораздо лучшую тактику тестирования, чем тривиальная логика. Бывает, что срабатывает и ассоциативная связь: вы уже находили ошибку в подобных обстоятельствах, хотя можете этого даже не помнить. Как бы там ни было, доверяйте своему внутреннему чувству и учитесь к нему прислушиваться: с опытом оно будет становиться все более развитым и надежным.

**Тестирование функциональной эквивалентности**

При тестировании функциональной эквивалентности сравниваются результаты вычислений разными программами одной и той же математической функции. Термин «функциональная эквивалентность» не имеет ничего общего с термином «классы эквивалентности». Если обе программы при вычислении одной и той же функции дают одинаковые результаты, значит, в них применены эквивалентные методы вычислений.

Предположим, что тестируется программа, которая вычисляет математическую функцию и печатает результат. Это может быть простая тригонометрическая функция или гораздо более сложная функция, инвертирующая матрицу или возвращающая коэффициенты для построения кривой, отражающей некоторый набор данных. Обычно в таких случаях можно найти другую программу, выполняющую те же действия, и при этом достаточно надежную и проверенную временем. Обеим программам предлагается обработать одинаковые наборы выходных данных. Если результаты совпадут, значит, тестируемая программа работает правильно.

**Автоматизация тестирования функциональной эквивалентности.** Везде, где возможно применить метод тестирования функциональной эквивалентности, он будет наилучшим выбором. Прежде всего, не придется вычислять значения вручную. Если функция сложна, это поможет сэкономить огромное количество времени и избежать ошибок, так часто возникающих при неавтоматизированных расчетах.

Процесс сравнивается также, скорее всего, удастся автоматизировать. Простейшим способом моет быть вывод результатов расчетов в файлы и их последующее сравнение с помощью соответствующей программы. Компьютер выполнит сравнение файлов и быстрее, и аккуратнее. Можно определить и допустимые расхождения результатов, например погрешности округлений.

Кроме того, вполне возможно автоматизировать и весь процесс тестирования, от ввода выходных данных до сравнения выходных. Если это получится, процедура тестирования будет выполняться практически мгновенно и исключительно надежно.

Хотя автоматизация подобных тестов – процесс не особенно сложный, он требует некоторого времени. Если программа может считать входные данные из файла, его необходимо подготовить. Кроме того, придется написать небольшие программки, выполняющие сравнение результатов.

Тестирование функциональной эквивалентности может потребовать некоторых затрат. Надежная эталонная программа, которая будет использоваться для сверки результатов, может оказаться не такой уж дешевой. К тому же, скорее всего, придется написать кое-какие программки. Может потребоваться и дополнительная техника, например второй компьютер. Разумеется, нельзя определить универсальные критерии того, сколько денежных средств имеет смысл потратить на проведение подобного тестирования. Однако есть определенные мерки.

Прежде всего, оцените, сколько дней потребуется на тестирование программы вручную. Включите в расчет время планирования, выполнения вычислительной и проведения тестов. Не забудьте и о том, что каждый тест провести не один раз, поскольку вы будете выявлять ошибки и повторять всю процедуру тестирования с самого начала. Прикиньте, сколько циклов тестирования потребуется провести.

Оцените также, сколько времени сэкономят автоматизированные средства тестирования. Снова учтите весь процесс: планирование, программирование и отладку тестов. Постарайтесь оценить необходимое время как можно более реалистично.

Число дней, которое предполагается сэкономить, умножьте на свой двойной оклад. Сумма оклада умножается на два потому, что в расчет берется еще выгода, которую компания получает от ускорения процесса тестирования и повышения его надежности. Если ваша оценка верна, полученная сумма – это минимум того, что можно сэкономить при функциональном тестировании с помощью эталонной программы. Если сама программа стоит меньше этой суммы, то любой разумный руководитель одобрит ее покупку. Подготовьте предложение и презентацию, поясняющие назначение покупаемой эталонной программы и основы расчетов. Если компания не настолько заинтересована в сокращении времени разработки, чтобы вкладывать в нее дополнительные средства, поясните, что вложенные средства гарантированно окупятся благодаря надежности и качеству продукта.

**Анализ чувствительности.** Автоматизация функционального тестирования позволяет выполнить гораздо больше тестов, чем вручную. Однако отбираться они должны не менее тщательно, поскольку число возможных значений входных данных, скорее всего, по-прежнему будет слишком велико, чтобы можно было провести абсолютно полное тестирование. У большинства функций число возможных значений аргументов бесконечно, так что справиться с ними не под силу даже компьютеру. Обязательно нежно будет проверить граничные значения. Как же отобрать наилучшие тесты? Чаще всего для этого применяется анализ чувствительности. Эта процедура заключается в следующем.

Прежде всего, получают общее представление о поведении функции, вычислив ее значения для ряда параметров, располагающихся по всей области определения. Затем ищут участки области определения, на которых небольшие изменения аргументов вызывают значительные скачки результирующих значений. Именно такие участки наиболее чреваты ошибками.

Значения, полученные в результате тестирования программируемой функции и ее эталона, могут не вполне совпадать. Если в процессе расчетов выполняются операции с плавающей запятой, то неизбежны округления или усечения результатов, а значит, и небольшие расхождения. Обычно это не страшно. Необходимо только правильно оценить допустимые погрешности округлений, чтобы можно было зафиксировать превышающие их расхождения, вызванные иными причинами.

Рекомендуется равномерно разделить каждый диапазон тестируемых входных значений на ряд поддиапазонов (их может быть около 100) и протестировать по одному значению внутри каждого из них. После ввода каждого значения проверяйте, правильный ли получился результат, чтобы не тратить зря времени, если что-то не так.

Получив общую картину поведения функции, проанализируйте его на предмет резких перемен. Если значения функции на отдельных участках области ее определения резко возрастают или уменьшаются либо наблюдаются разрывы и скачки, на них необходимо обратить более пристальное внимание.

Предположим, что при каком-то входном диапазоне значения функции (или их расхождение со значениями эталонной функции) резко возрастают. Разделите этот диапазон на 100 равных частей и проверьте по одному значению внутри каждой из них. Если все в порядке, то вы убедитесь ,что значения тестируемой и эталонной функции для всех тестируемых аргументов совпадают, а если нет, то вы нашли ошибку.

При профессиональном тестировании математических функций не обойтись без некоторых значений из теории вероятности. Если речь идет не об одной, а о целом ряде функций, необходимы более эффективные и научно обоснованные технологии поиска критических участков области определения функции, т.е. тех, где значения резко меняются или отличаются от эталонных.

**Случайный ввод.** Вместо разделения всей тестируемой области определения функции на определенное число равных участков можно воспользоваться другим способом подбора входных значений – случайны. Случайный выбор значений более эффективен, поскольку гарантирует их полную равноправность. Например, тестируя, токую последовательность входных значений, как 0,02; 0,04; 0,06 и т.д., вы никогда не узнаете, как программа обрабатывает нечетные числа (допустим, 0,03) или числа с большим количеством значащих цифр (например, 0,1415). В то же время при выборе входных значений случайным образом область определения функции покрывается гораздо более полно, все типы и диапазоны входных данных охватываются равномерно.

Если вы затрудняетесь в выборе методики подбора входных данных или не вполне уверены в поведении тестируемой функции, остановитесь на случайном способе. Он прекрасно подходит и для автоматизированного тестирования.

Отсутствие четкого обоснования для выбора конкретных входных значений можно компенсировать числом проводимых тестов. Здесь нет никаких ограничений – чем больше тестов проводится для каждого из классов эквивалентности, тем лучше. Обычно при автоматизированном тестировании со случайными входными значениями проводится, как минимум, 1 000 вычислений.

**Использование генератора случайных чисел.** Случайный ввод вовсе не означает «все, что происходит в голову», иначе он будет слишком предвзятым, чтобы претендовать на равномерный охват области определения функции. Больше подойдет компьютерная программа, которая может генерировать случайные числа в неограниченном количестве. Однако следует иметь в виду, что алгоритмы, используемые многими подобными программами, вовсе не случайны. Кроме того, нередко в программах даже базовый алгоритм реализован не точно. Поэтому, прежде чем выбрать конкретный генератор случайных чисел, даже созданный вполне авторитетной компанией или встроенный в один из стандартных языков программирования, необходимо выяснить, какой алгоритм положен в основу его работы и подходит ли он для ваших нужд. То, что годится для простой программы, позволяющей получить на экране разноцветный салют, может совершенно не подойти дл профессионального тестирования сложных инженерных программ. Нередко генераторы случайных чисел повторяют их последовательность через определенный интервал, пусть даже очень большой. В связи со всем этим прислушайтесь к следующим советам.

Не доверяйте какой-либо программе просто потому, что она у вас уже есть, а другую еще придется поискать. Попытайтесь разобраться в программе, и только когда она вам будет полностью ясна, используйте ее.

Если вы собираетесь воспользоваться генератором случайных чисел, встроенным в язык программирования, стоит его немного доработать. Сгенерировав с его помощью достаточно большое количество чисел (100 … 1 000), перемешайте их: измените их порядок с помощью последующих чисел, выдаваемых этим же генератором. Хотя это и замедлит работу, зато результат в случае плохого исхода генератора может быть уже вполне приемлемым.

**Применение технологии эквивалентности.** Тестирование математических функций – не единственная область применения эталонных программ. Путем сравнения с готовыми и проверенными ПП можно тестировать самые разные аспекты поведения программы. Вот несколько примеров.

Если тестируется программа проверки правописания, в основе которой лежит тот же алгоритм, что и в одной из уже существующих программ, можно предложить обеим программам проверить один и тот же набор слов.

Если тестируется программа автоматического переноса слов, особенно если отрабатывается модификация ее алгоритма для другого языка, можно взять для сверки проверенную программу, продаваемую на рынке программного обеспечения, подготовить узкий столбик текста и предложить его обеим программам. При тестировании программы, выполняющей выравнивание текста по ширине строки, необходимо проверить, насколько равномерно она разделяет слова пробелами. Для образца можно взять обычный текстовый процессор и обеим программам предложить один и тот же текст, набранный одинаковыми шрифтами.

Для отладки посылаемых на принтер двойных управляющих последовательностей можно перенаправить вывод в файл и то же самое сделать в эталонной программе, создав в ней точно такой же файл. Затем оба файла надо сравнить – они должны быть идентичны.

Во всех случаях, когда необходимо протестировать выходные данные, которые легко направлять в файл, также можно использовать эталонную программу, умеющую генерировать те же данные. Их легко можно сравнить.

В каждом конкретном случае существуют свои аргументы «за» и «против» этой технологии. Например, на ее реализацию может потребоваться слишком много времени, средств или усилий. Кроме того, эталонная программа тоже вполне может содержать ошибки. Но в любом случае методику тестирования эквивалентности следует иметь в виду – нередко ее применение значительно ускоряет работу и во много раз повышает ее эффективность.

Не забывайте включать в отчеты об ошибках выходные данные, полученные от обеих программ. Они важны для поиска причины ошибки.

**Регрессионное тестирование**

**Общие сведения.** Основной работой тестировщика является регрессионное тестирование. У этого термина два значения, которые объединяет идея повторного использования разработанных тестов.

Представьте, что после проведения теста была обнаружена ошибка, и программист ее исправил. Тестировщик снова проводит тот же тест, чтобы убедиться, что ошибки больше нет. Это и есть регрессивное тестирование. Можно провести несколько вариаций исходного теста, чтобы как следует проверить исправленный фрагмент программы. В данном случае задача регрессионного тестирования состоит в том, чтобы убедиться, что выявленная ошибка полностью исправлена программистом и больше не проявится.

Второй пример применения регрессионного тестирования. После выявления и исправления ошибки проводится стандартная серия тестов, но уже с другой целью: убедиться, что, исправив одну часть программы, программист не исправил другую. В этом случае тестируется целиком вся программа, а не какой-то исправленный ее фрагмент.

**Рекомендации по исправлению ошибок.** Получив отчет об ошибке, программист должен тщательно проанализировать исходный код программы, найти причину ошибки, исправить ее и протестировать результат. Это идеальный вариант. На практике некоторые программисты устраняют только описание в отчете симптомы наличия ошибки, а не саму ее. В результате какие-то проявления ошибки исчезают, но другие остаются. Бывает и так, что программист неправильно понимает отчет и исправляет не то, что надо. Некоторые недобросовестные сотрудники вообще не проверяют свою работу, не глядя, вносят исправления и немедленно возвращают программу тестировщикам. В результате ошибка остается и новые появляются. Опытный тестировщик должен быть готов к подобным ситуациям. Считается, что около трети вносимых в программу исправлений не срабатывают или даже «ломают» то, что уже работало.

Вот три задачи, должен поставить перед собой тестировщик, проверяющий внесенные программистом исправления:

убедиться, что ошибка исправлена, для чего выполнить тот же тест, в котором она проявилась и который был описан в отчете. Если программа его не пройдет, дальнейшее тестирование не имеет смысла;

постараться найти связные ошибки. Предположим, что программист устранил описанные в отчете симптомы ошибки, но саму ее не исправил. Можно попробовать спровоцировать симптомы ошибки каким-нибудь иным способом. В программе могут оказаться другие подобные ситуации? Следует провести как можно больше тестов;

протестировать оставшуюся часть программы. Неожиданные последствия исправления могут проявиться где-нибудь еще. Их поиск проводится неформально – без заранее подготовленного плана. Просто необходимо продумать , какие части программы могут быть затронуты внесенными исправлениями, и проверить их.

**Стандартная серия тестов.** Некоторое время спустя после начала тестирования ПП формируется *библиотека регрессионных тестов.* Это полный набор тестов, охватывающий всю программу и выполняющийся каждый раз, когда программисты сдают ее очередную рабочую версия. Лучше всего, если тесты полностью автоматизированы. В этом случае все равно приходится тратить на тестирование некоторое время, но вся процедура получается гораздо менее трудоемкой.

Когда для тестирования предоставляется очередная версия программы и наступает время повторить все тесты библиотеки, возникает ряд вопросов. Насколько велика библиотека? Действительно ли необходимо выполнять все тесты библиотеки снова? Не всегда легко заранее определить, какие тесты следует включать в регрессионную библиотеку. Поэтому первоначально в ней может оказаться больше тестов, чем это действительно необходимо. Как минимум, в нее должны входить примеры для проверки граничных условий и временных характеристик. Но стоит ли выполнять их все каждый раз?

Выполнять регрессионные тесты обычно не хочется, поскольку вероятность выявления ими ошибок не особенно велика. При первом проведении тестирования некоторые из этих тестов могут выявить ошибки, но после их окончательного исправления выполнение тех же тестов снова и снова кажется пустой потерей времени. Если ошибки больше нет, то какова вероятность, что она появится снова? А как быть с тестами, которые выполнялись уже несколько раз и ни разу не выявили ошибок? В некоторых компаниях такие тесты исключают из библиотеки, оставляя только те, в которых проявлялись ошибки.

Вместо того чтобы мучительно обдумывать каждый тест, лучше пойти более простым путем. Включите в регрессионную библиотеку все тесты, которые покажутся вам полезными. Периодически, примерно через каждые три цикла, пересматривайте эту библиотеку и удаляйте те тесты, отсутствие которых не снизит качества работы. Вот несколько полезных советов.

*Удалите тесты, которые эквивалентны другим тестам библиотеки.* В идеале такие тесты вообще не должны попадать в библиотеку, но когда она создается несколькими сотрудниками, подобные накладки вполне возможны.

*Уменьшите числа тестов, объектом которых является уже исправленная ошибка.* Если ошибка или некоторые ее разновидности проявляются в течение целого ряда циклов тестирования, то в библиотеку стоит добавить достаточное число тестов для выявления. Это вполне нормально и уместно. Соответствующую часть программы необходимо тщательнейшим образом тестировать до тех пор, пока в ней не останется и следа ошибки. Однако после этого большую часть тестов, направляемых на поиск исправленной ошибки, можно удалить из библиотеки.

*Комбинируйте тесты.* Если 15 тестов, которые программа скорее всего пройдет, можно объединить в один – сделайте это. В начале тестирования так поступать не стоит, но в дальнейшем объединение тестов позволит значительно ускорить работу.

*По возможности автоматизируйте тестирование.* Если вы уверены, что определенная группа тестов будет выполняться в течение пяти или десяти последующих циклов тестирования, то стоит потратить время на автоматизацию.

*Выделите часть тестов для периодического выполнения.* Все тесты регрессионной библиотеки не обязательно выполнять после каждого измерения программы. Это можно делать реже – на каждом втором или на каждом третьем цикле. На последней стадии тестирования лучше выполнить максимально возможное число тестов, чтобы убедиться, что программа готова к выпуску, на других циклах достаточно половины или даже трети всех тестов.

Регрессионная библиотека должна включать в себя все лучшие тесты из тех, что уже разработаны, но, если она будет слишком велика, не останется времени на разработку новых тестов. А ведь именно новейшие тесты с наибольшей вероятностью выявляют еще не найденные ошибки. Поэтому планировать работу надо так, чтобы регрессионная библиотека служила средством повышения эффективности тестирования, а не его тормозом.

**Выполнение тестов.** Придумать хороший тест – это только половина дела. Его еще нужно правильно выполнить. Вот несколько рекомендаций.

Если вы хотите, чтобы при установке программы на компьютер у пользователя была возможность выбора конфигурации, недостаточно просто запустить программу установки и посмотреть, предоставляет ли она необходимые опции. Выполните каждый вариант установки, всякий раз запуская саму программу и проверяя, действительно ли установлена выбранная конфигурация. Убедитесь, что программа при этом полностью работоспособна.

Если программа позволяет указать размер печатаемой страницы, отступы и другую подобную информацию, не считайте дело сделанным, увидев, что документ правильно выглядит на экране. Его необходимо распечатать.

Если в программе используются символы из расширенного набора ASCII, недостаточно увидеть их на экране. Они должны правильно печататься, пересылаться через модем и т.д. Необходимо учесть программное обеспечение, через которое будут проходить выходные данные: драйверы устройств, алгоритмы импорта.

Основное правило, вытекающее из приведенных рекомендаций, можно сформулировать так: тестовая процедура должна заставить программу использовать введенные данные и подтверждать, что они используются правильно.

**Отладка программ**

Локализация и исправление ошибок называется отладкой. Отладка — это процесс обнаружения причин возникновения ошибки и ее последующего исправления (в отличие от тестирования, являющегося процессом обнаружения самого факта существования ошибки).

Отладка включает в себя элементы тестирования и разработки. На некоторых проектах отладка может занимать до 50 % всего времени разработки. Для многих программистов отладка — это самая трудная часть программирования. При соответствующем подходе количество ошибок, требующих отладки, должно сократиться, и отладка должна стать самой легкой частью; при таком подходе все ошибки сводятся к небольшим недосмотрам или опечаткам.

Как и тестирование, отладка не является способом улучшения качества ПО. Отладка — это всего лишь способ исправления дефектов в программе. Качество программ должно обеспечиваться аккуратным анализом требований или прототипированием, грамотным проектированием и использованием лучших практик кодирования.

Ошибки синтаксиса языка

Как правило, в абсолютном большинстве случаев ловятся на стадии компиляции программы, или же, если вы работаете с интерпретируемым языком типа Perl или РНР, то при первом интерпретировании программы. Но есть один существенный момент — когда выражение допустимо, но зависит от конкретного компилятора или интерпретатора. Например, в языке Си вполне допустимыми по синтаксису, но не по смыслу, являются выражения:

s[i++]=i; printf ("%d %d", i++, i++) ;. Результат этих строк не определен, так как неизвестно, в каком порядке будет инкрементироваться и вычисляться значение переменной /. Переменная не может более 1 раза присутствовать в выражении, если ее значение изменяется в ходе вычисления этого выражения.

Ошибки во время выполнения

К ошибкам во время выполнения программы относятся ошибки в логике и алгоритмах программы. Ошибки в логике программы связаны с неправильной записью алгоритма, например, вместо && вы написали || или поставили точку с запятой там, где ее не должно быть. К этому же классу ошибок относится неправильное обращение с памятью. В языке Си неправильное обращение с памятью является наиболее распространенной ошибкой программистов, причем не только новички, но и профессионалы нередко допускают ошибки подобного типа. Для языка Си — это один из самых больных вопросов и больших недостатков. Ошибки в алгоритме связаны с неправильным проектированием программы или же с изначально неправильно заложенной бизнес-логикой и функциональными требованиями.

Пользователь ждет от программы то, что изначально от нее не требовалось. Ошибки во время выполнения — наиболее часто встречаемый тип ошибок, и, как правило, устранение таких ошибок не представляется чересчур сложной задачей.

Ошибки логики взаимосвязанных CGI-программ

Ошибки данного типа лежат во взаимосвязанных CGI-npoграммах. Рассмотрим в качестве примера тестовую систему (см. сайт http://test.itsoft.ru). При сдаче теста в цикле работают два скрипта. Первый показывает вопрос, а второй проверяет правильность ответа. Если в тесте 10 вопросов, то эти CGI-скрипты вызываются парно в цикле 10 раз. Но что будет, если пользователь нажмет кнопку «Обновить» в броузере? Скрипт, который показывает вопрос, вызовется повторно. Что будет при разрыве модемного соединения? Отладка в таких системах значительно сложнее, так как вам придется наблюдать за выполнением ряда взаимосвязанных скриптов.

Ошибки многопользовательского доступа

Ошибки многопользовательских систем связаны с неправильным разграничением доступа к совместным ресурсам.

Помимо файлов данных, записей в таблицах баз данных, общим ресурсов является генератор случайных чисел. На эти грабли нам пришлось наступить. Обязательно генерируйте случайные числа не только на основе времени, но и на основе уникального идентификатора процесса. В противном случае при выполнении в один и тот же момент времени двух копий одной CGI-программы вы получите одинаковые результаты для двух пользователей.

Ошибки многопользовательского доступа сложны тем, что могут не проявлять себя очень долго, до тех пор, пока в один и тот же момент времени системой ни будет запущено несколько копий одной CGI-программы.

Невоспроизводимые ошибки

Невоспроизводимые ошибки представляют собой наиболее сложный тип ошибок. Например, в високосном году 29 февраля ваша система вдруг начала давать сбои, которые сами собой исчезают в невисокосном году. Но бывают ошибки, которые мистическим образом появляются и исчезают. В той же тестовой системе была непонятная ошибка, которая проявлялась 1 раз на несколько сот случаев. Непонятным образом некоторые студенты после сдачи теста получали не результаты, а сбой системы.

На исправление этой ошибки ушло два рабочих дня. Оказалось, что проблема в скрипте на JavaScript, который отправлял данные HTML-формы на сервер после истечения допустимого времени ответа на вопрос. Проблема в том, что если время подходило к концу и пользователь нажимал кнопку «Ответить», а в это же время уже начала работать функция JavaScript form.submit(), то отправка данных HTML-формы происходила дважды, т. е. скрипт проверки правильности ответа вызывался 2 раза. А это за собой тянуло ошибку во взаимосвязанных CGI-скриптах, и внешнее проявление сбоя системы мы наблюдали уже при подсчете результатов, а не непосредственно сразу после двойной отправки HTML-формы. Сам код JavaScript был написан верно, и с теоретической точки зрения даже если пользователь нажимает кнопку «Отправить» в последнюю секунду, HTML-форма должна была отправляться только 1 раз. Но на практике все оказалось совсем по-другому. На самом деле ничего мистического нет, или, как говорится, чудес на свете не бывает. Просто невозможно воспроизвести условия, в которых наблюдалась невоспроизводимая ошибка. Надо искать в программе случайности: одновременный доступ к одному ресурсу, генератор случайных чисел, неинициализированные переменные, некорректная работа с памятью или преобразование типов, которые могут проявлять себя не каждый раз.

Ошибки инструментария и других компонентов системы

Ошибки самого компилятора или интерпретатора очень редки, но и такие бывают.

После классификации ошибок давайте рассмотрим методы их поиска. В первую очередь, один простой и, казалось бы, очевидный совет: «надо думать, анализировать, почему программа не работает так, как было задумано, что надо в ней исправить».

Никакой самый навороченный отладчик за вас ошибку не найдет. Никакая самая лучшая методика не найдет и не ускорит поиск ошибки, если вы не «включите» мозги по полной программе и не сосредоточитесь целиком и полностью на поимке ошибки. Итак, допустим, вами, пользователем или тестирующим, было зафиксировано некорректное поведение программы. Что делать? Ниже перечислены методы в порядке их приоритетности, которые используют многие программисты.

Локализация ошибки. Программа — это черный ящик, который что-то принимает на вход и выдает что-то на выходе. Первое, что вам необходимо сделать, — понять, где именно в этом черном ящике происходит сбой. Для этого уменьшайте количество параметров, передаваемых на вход программы. Добейтесь двух малоразличимых наборов входных параметров, чтобы при одном из них программа работала, а при другом нет.

Отключение ненужных модулей программы. Закомментируйте все лишнее. Тем самым вы упростите вашу программу. Имеет смысл комментировать отдельные куски программы до тех пор, пока ошибка не исчезнет.

Использование отладчика. Возможности современных отладчиков перечислены ниже:

• точки останова на конкретных строчках кода;

• остановка на я-й итерации цикла;

• остановка при изменении переменных;

• остановка при присваивании конкретного значения;

• прохождение кода строчка за строчкой;

• откат по программе (далеко не все);

• исследование всех данных в программе, включая типы, определенные пользователем;

• присваивание новых значений переменным;

• продолжение исполнения программы;

• многоязыковая отладка (язык1, язык2, ассемблер...);

• запоминание установок.

**Документация на программное обеспечение**

**Документация на программное обеспечение** — это документы, сопровождающие программное обеспечение (ПО) — программу или программный продукт. Эти документы описывают то, как работает программа и/или то, как её использовать.

Существует четыре основных типа документации на ПО:

* архитектурная/проектная — обзор программного обеспечения, включающий описание рабочей среды и принципов, которые должны быть использованы при создании ПО
* техническая — документация на код, алгоритмы, интерфейсы, API
* пользовательская — руководства для конечных пользователей, администраторов системы и другого персонала
* маркетинговая

**Архитектурная/проектная документация**

Проектная документация обычно описывает продукт в общих чертах. Не описывая того, как что-либо будет использоваться, она скорее отвечает на вопрос «почему именно так?» Например, в проектном документе программист может описать обоснование того, почему структуры данных организованы именно таким образом. Описываются причины, почему какой-либо класс сконструирован определённым образом, выделяются паттерны, в некоторых случаях даже даются идеи как можно будет выполнить улучшения в дальнейшем. Ничего из этого не входит в техническую или пользовательскую документацию, но всё это действительно важно для проекта.

**Техническая документация**

Это именно то, что подразумевают под термином документация большинство программистов. При создании программы, одного лишь кода, как правило, недостаточно. Должен быть предоставлен некоторый текст, описывающий различные аспекты того, что именно делает код. Такая документация часто включается непосредственно в исходный код или предоставляется вместе с ним.

Подобная документация имеет сильно выраженный технических характер и в основном используется для определения и описания API, структур данных и алгоритмов.

Часто при составлении технической документации используются автоматизированные средства — генераторы документации. Они получают информацию из специальным образом оформленных комментариев в исходном коде, и создают справочные руководства в каком-либо формате, например, в виде текста или HTML. Использование генераторов документации и документирующих комментариев многими программистами признаётся удобным средством, по различным причинам. В частности, при таком подходе документация является частью исходного кода, и одни и те же инструменты могут использоваться для сборки программы и одновременной сборки документации к ней. Это также упрощает поддержку документации в актуальном состоянии.

**Пользовательская документация**

В отличие от технической документации, сфокусированной на коде и том как он работает, пользовательская документация описывает лишь то, как использовать программу.

В случае, если продуктом является программная библиотека, пользовательская документация и документация на код становятся очень близкими, почти эквивалентными понятиями. Но в общем случае, это не так.

Обычно, пользовательская документация представляет из себя руководство пользователя, которое описывает каждую функцию программы, а также шаги, которые нужно выполнить для использования этой функции. Хорошая пользовательская документация идёт ещё дальше и предоставляет инструкции о том что делать в случае возникновения проблем. Очень важно, чтобы документация не вводила в заблуждение и была актуальной. Руководство должно иметь чёткую структуру; очень полезно, если имеется сквозной предметный указатель. Логическая связность и простота также имеют большое значение.

Существует три подхода к организации пользовательской документации. Вводное руководство (англ. *tutorial*), наиболее полезное для новых пользователей, последовательно проводит по ряду шагов, служащих для выполнения каких-либо типичных задач. Тематический подход, при котором каждая глава руководства посвящена какой-то отдельной теме, больше подходит для совершенствующихся пользователей. В последнем, третьем подходе, команды или задачи организованы в виде алфавитного справочника — часто это хорошо воспринимается продвинутыми пользователями, хорошо знающими, что они ищут. Жалобы пользователей обычно относятся к тому, что документация охватывает только один из этих подходов, и поэтому хорошо подходит лишь для одного класса пользователей.

Во многих случаях разработчики программного продукта ограничивают набор пользовательской документации лишь встроенной системой помощи (англ. *online help*), содержащей справочную информацию о командах или пунктах меню. Работа по обучению новых пользователей и поддержке совершенствующихся пользователей перекладывается на частных издателей, часто оказывающих значительную помощь разработчикам.

**Маркетинговая документация**

Для многих приложений необходимо располагать рядом рекламных материалов, с тем чтобы заинтересовать людей, обратив их внимание на продукт. Такая форма документации имеет целью:

1. подогреть интерес к продукту у потенциальных пользователей
2. информировать их о том, что именно делает продукт, с тем чтобы их ожидания совпадали с тем что они получат
3. объяснить положение продукта по сравнению с конкурирующими решениями

Одна из хороших маркетинговых практик — предоставление слогана — простой запоминающейся фразы, иллюстрирующей то что мы хотим донести до пользователя, а также характеризующей *ощущение*, которое создаёт продукт.

Часто бывает так, что коробка продукта и другие маркетинговые материалы дают более ясную картину о возможностях и способах использования программы, чем всё остальное.

**Генератор документации** — [программа](http://www.seobuilding.ru/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80%D0%BD%D0%B0%D1%8F_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B0) или пакет программ, позволяющая получать [документацию](http://www.seobuilding.ru/wiki/%D0%A2%D0%B5%D1%85%D0%BD%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B0%D1%8F_%D0%B4%D0%BE%D0%BA%D1%83%D0%BC%D0%B5%D0%BD%D1%82%D0%B0%D1%86%D0%B8%D1%8F), предназначенную для [программистов](http://www.seobuilding.ru/wiki/%D0%9F%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%81%D1%82) (документация на [API](http://www.seobuilding.ru/wiki/API)) и/или для конечных пользователей системы, по особым образом комментированному [исходному коду](http://www.seobuilding.ru/wiki/%D0%98%D1%81%D1%85%D0%BE%D0%B4%D0%BD%D1%8B%D0%B9_%D0%BA%D0%BE%D0%B4) и, в некоторых случаях, по [исполняемым модулям](http://www.seobuilding.ru/w/index.php?title=%D0%98%D1%81%D0%BF%D0%BE%D0%BB%D0%BD%D1%8F%D0%B5%D0%BC%D1%8B%D0%B9_%D0%BC%D0%BE%D0%B4%D1%83%D0%BB%D1%8C&action=edit&redlink=1) (полученным на выходе [компилятора](http://www.seobuilding.ru/w/index.php?title=%D0%9A%D0%BE%D0%BC%D0%BF%D0%B8%D0%BB%D1%8F%D1%82%D0%BE%D1%80&action=edit&redlink=1)).

Обычно, генератор анализирует исходный код программы, выделяя синтаксические конструкции, соответствующие значимым объектам программы (типам, классам и их членам/свойствам/методам, процедурам/функциям и т. п.). В ходе анализа также используется мета-информация об объектах программы, представленная в виде *документирующих комментариев*. На основе всей собранной информации формируется готовая документация, как правило, в одном из общепринятых форматов — [HTML](http://www.seobuilding.ru/wiki/HTML), [HTMLHelp](http://www.seobuilding.ru/wiki/HTMLHelp), [PDF](http://www.seobuilding.ru/w/index.php?title=PDF&action=edit&redlink=1), [RTF](http://www.seobuilding.ru/w/index.php?title=RTF&action=edit&redlink=1) и других.

**Документирующий комментарий** — это особым образом оформленный [комментарий](http://www.seobuilding.ru/wiki/%D0%9A%D0%BE%D0%BC%D0%BC%D0%B5%D0%BD%D1%82%D0%B0%D1%80%D0%B8%D0%B8_%28%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5%29) к объекту программы, предназначенный для использования каким-либо конкретным генератором документации. От того, какой генератор документации применяется, зависит [синтаксис](http://www.seobuilding.ru/w/index.php?title=%D0%A1%D0%B8%D0%BD%D1%82%D0%B0%D0%BA%D1%81%D0%B8%D1%81&action=edit&redlink=1) конструкций, используемых в документирующих комментариях.

В документирующих комментариях может содержаться информация об авторе кода, описываться назначение объекта программы, смысл входных и выходных параметров — для функции/процедуры, примеры использования, возможные исключительные ситуации, особенности реализации.

Преимущества генерации документации следующие:

* Документация становится в большей степени согласованной с программным кодом, то есть более точной и актуальной.
* Экономится значительное количество времени и усилий, требуемых для разработки документации. Освобождаемые время и ресурсы можно применить для улучшения качества документации.
* Более простым становится процесс изменения формата или стиля подачи документации. При наличии собственного генератора не надо менять формат каждого документа, достаточно изменить шаблон.
* Документация формируется значительно быстрее.

Практически для каждого популярного языка созданы программы, документирующие код. Для C#, VB и других языков платформы .Net имеется программа NDoc, позволяющая составлять документацию. Имеется приложение Doxygen, используемое в основном для C++, но можно генерировать документацию также для Си, Java, C#, PHP и других языков. Есть Javadoc для Java, PHPDoc для PHP и множество других.

Генераторы документации обычно используются для документирования назначения, дизайна кода, архитектуры, способа разрешения задач и т.д. При этом они обычно не применяются для:

* проектирования документации,
* создания пользовательской документации,
* документирования отдельных команд в приложении построчно.

При генерации документации необходимо:

* располагать текст документации рядом с кодом,
* сделать систему документирования простой и удобной для сопровождения,
* стараться интегрировать генераторы кода и документации,
* стараться собственноручно разработанные генераторы документации интегрировать с существующими аналогичными инструментами,
* вводить информацию только один раз.

Манипуляция данными

Генерация может применяться для создания кода, манипулирующего данными. Под манипуляцией будем понимать преобразование данных из одного формата в другой, проверку корректности данных, их соответствия правилам и другие операции с данными. Для генерации создается файл, содержащий информацию о структуре и форматах данных. Используя его, генератор формирует код для чтения и преобразования данных, а также для проверки их корректности и записи в различных форматах. Виды манипуляций, которые можно осуществлять:

1. Преобразование из одного формата в другой. Форматы могут быть совершенно разными. Они могут быть файлами, базами данных, вводом с клавиатуры, выводом на экран, оперативной памятью и т.п.
2. Проверка корректности данных, соответствия шаблонам и правилам.

В метаданных указывается, из каких форматов и в какие требуется переводить данные. Также указывается структура этих данных, шаблоны и правила. На основе этой информации генератор формирует методы для каждой пары форматов, преобразующие данные с указанной структурой из одного в другой.

Преимуществами применения генераторов кода для манипуляции данными являются:

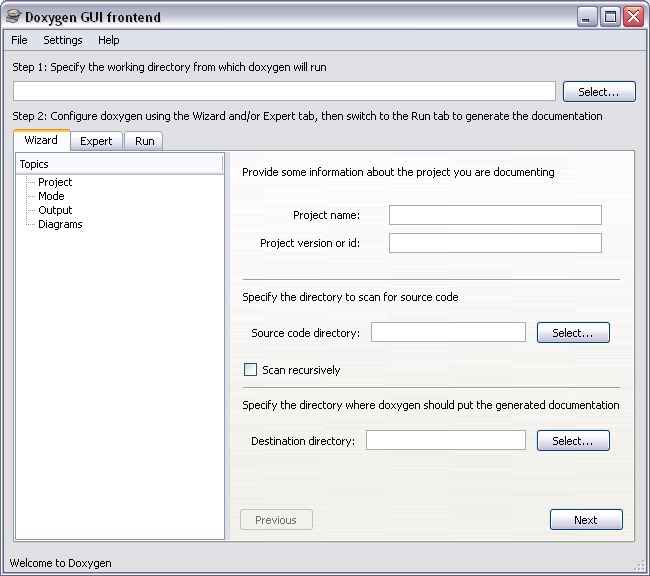
* высокая скорость разработки,
* одновременная поддержка множества форматов,
* гибкость применения различных иерархий данных,
* возможность генерации кода на нескольких языках,
* поддержка одновременной генерации нескольких вариантов реализации,
* сведение к минимуму ошибок данных, типов данных,
* возможность одновременной генерации документации и тестов для сгенерированного кода.

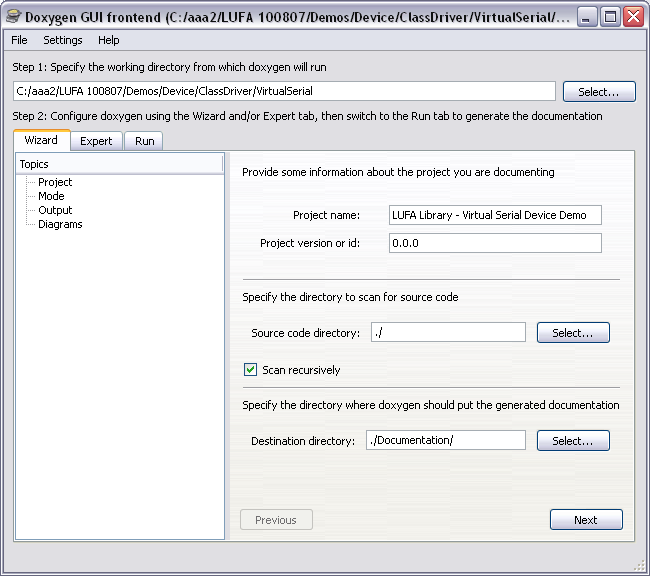
**Doxygen** - популярный генератор документации на основе исходных текстов. Что такое Doxygen, очень хорошо написано в Википедии, поэтому в статье я расскажу только о том, как быстро установить и начать использовать Doxygen.

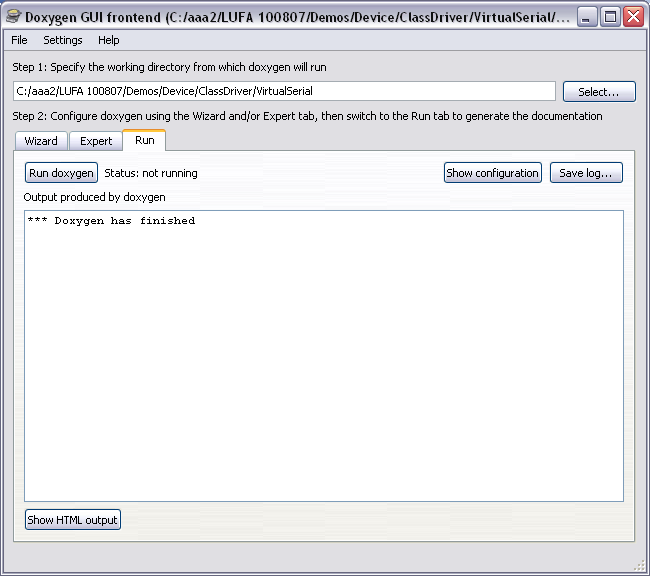
Итак, процесс по шагам:

**1**. Качаем по ссылке [1], доступны версии для Linux i386,  Mac OS X 10.6 (Snow Leopard), Mac OS X 10.4 (Tiger), Windows XP/Vista/7, а также исходники (Doxygen распространяется под лицензией GPL).

**2**. Запускаем **doxygen-1.7.2-setup.exe**. Отвечаем на несложные традиционные вопросы инсталлятора (можно тупо жать Next). После инсталляции появится папка c:\Program Files\doxygen\, где и находится вся система Doxygen, документация по ней, и утилиты.

**3**. Запускаем c:\Program Files\doxygen\bin\**doxywizard.exe**. Это программа, которая позволяет упростить создание и использование конфигурационного файла для создания документации (Doxygen GUI frontend).   


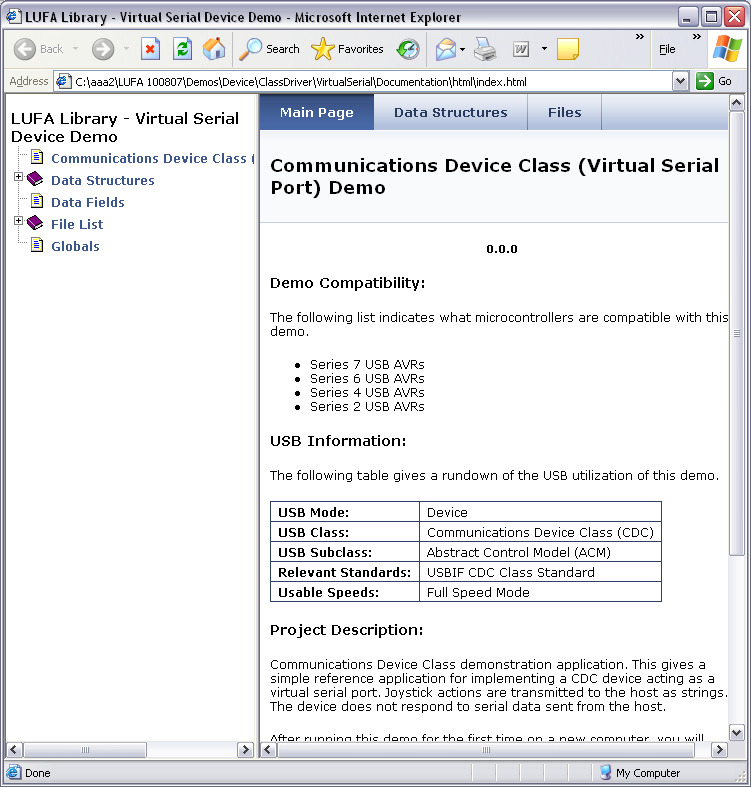
**4**. Предположим, что нам нужно создать документацию к каким-то исходникам. В этом примере я буду делать документацию к проекту LUFA 100807\Demos\Device\ClassDriver\VirtualSerial\. Теперь нужно посмотреть, есть ли в каталоге с исходниками готовая конфигурация Doxygen. Обычно такой файл называется Doxygen.conf или Doxyfile, и имеет примерно такое содержание: "# Doxyfile 1.6.2 # This file describes the settings ...". В моем случае папка с исходниками VirtualSerial уже содержала готовый конфигурационный файл Doxygen.conf. Теперь в программе меню doxywizard.exe выбрать File -> Open... -> файл **Doxygen.conf**.   


**5**. Теперь осталось перейти на закладку Run и нажать на кнопку Run doxygen:  


Если во время компиляции документации в текстах исходников встретятся ошибки, то они будут выведены в поле вывода "Output produced by doxygen". В сообщениях указаны номера строк, где найдены ошибки. Нумерация строк в сообщениях "tag without matching" может не совпадать со строками, где эти ошибки имеются на самом деле.

Запуск генерации документации можно также провести из командной строки вводом команды (config-file имя файла конфигурации doxygen):

doxygen

В моем примере в папке VirtualSerial\Documentation\html\ появятся файлы в формате html. Начинать просмотр документации нужно с файла VirtualSerial\Documentation\html\index.html.  
[](http://microsin.net/images/stories/programming/doxygen004.PNG)

Если конфигурационного файла нет, то можно вручную сгенерить конфигурацию. Нужно выполнить, как минимум, следующие шаги (для примера VirtualSerial):

**Закладка Wizard -> Project**:  
- в поле "Step 1: Specify the working directory from which doxygen will run" указать путь до каталога проекта. В моем случае это c:\aaa2\LUFA 100807\Demos\Device\ClassDriver\VirtualSerial\  
- в поле "Project name:" укажите имя проекта (LUFA Library - Virtual Serial Device Demo).  
- в поле "Project version or id:" укажите версию проекта (0.0.0).  
- в поле "Source code directory:" укажите ./  
- в поле "Destination directory:" укажите ./Documentation/  
- если Ваш проект содержит подпапки с исходниками, поставьте галочку "Scan recursiVely".

**Закладка Wizard -> Mode**:  
- Select the desired extraction mode: -> All Entities.  
- Select programming language to optimize the results for -> Optimize for C or PHP output

**Закладка Wizard -> Output** (выберем на этот раз формат **RTF**):  
- убираем галочку HTML  
- убираем галочку LaTeX  
- ставим галочку Rich Text Format (RTF)

**Закладка Wizard -> Diagrams**:  
- Diagrams to generate -> No diagrams

**Закладка Run**:  
- нажимаем кнопку Run Doxugen. В результате получим файл единственный файл VirtualSerial\Documentation\rtf\refman.rtf.

Полученный файл конфигурации можно сохранить для дальнейшего использования (File -> Save as... -> Doxyfile).

[**Проблема правильной обработки кодировки русского языка**]

По умолчанию Doxygen генерирует html-текст в кодировке **UTF-8** (используется мета-тег), и предполагает, что входной формат текста тоже UTF-8. С такими настройками поставляются большинство конфигов Doxygen. При попытке сгенерировать с таким конфигом html-документацию из текстового формата **windows-1251** (с таком формате имеется большинство файлов исходников, содержащих русскоязычные комментарии), на выходе получается html-файл, который отображается всеми браузерами с кракозябрами. Эти кракозябры исчезнут, если вручную переключить браузер из кодировки UTF-8 в кодировку windows-1251. Это, конечно, не решение проблемы - возникает неудобство, так как необходимо постоянно переключаться в кодировку windows-1251. Заставить doxygen генерировать тег с charset=windows-1251 невозможно. Однако проблема фиксится довольно просто - достаточно в конфигурационном файле doxygen указать правильную кодировку входного файла - вместо кодировки UTF-8 указать кодировку windows-1251. Делается это правкой переменной конфига **INPUT\_ENCODING**:

# раньше тут было указано INPUT\_ENCODING = UTF-8  
INPUT\_ENCODING         = windows-1251

Кроме того, если нужно правильно распознать русский текст, который есть в файле Doxygen.conf (например, это может быть текст имени проекта PROJECT\_NAME и другие строки), то необходимо отредактировать переменную конфига **DOXYFILE\_ENCODING** (указать кодировку windows-1251):

# раньше тут было указано DOXYFILE\_ENCODING = UTF-8  
DOXYFILE\_ENCODING      = windows-1251

После такого исправления html будет корректно генерироваться, и правильно отображаться всеми браузерами, русский текст будет без кракозябр.

[**Экранирование специальных символов doxygen**]

Для устранения предупреждений типа "имя\_файла:номер\_строки: warning: explicit link request to 'define' could not be resolved" нужно применять для экранирования спецсимволов обратный слеш '\' (backslash). Например, так нужно экранировать символ # вместе с ключевым словом define:

тут текст \#define тут дальше текст

Это устранит предупреждения типа request to '...' could not be resolved.

[**Ссылки**]

**1**. DoxyGen Latest Release site:stack.nl - отсюда можно скачать бинарники и исходники Doxygen.   
**2**. [Проект LUFA 100807 -> Demos -> Device -> ClassDriver -> VirtualSerial](http://depositfiles.com/files/dauqj5p4f), который использовался в статье (вместе с конфигами Dogygen и сгенерированной документацией).  
**3**. Дистрибутив [doxygen-1.7.2-setup.exe](http://depositfiles.com/files/b8tyzcbzg).