МИНИСТЕРСТВО НАУКИ И ОБРАЗОВАНИЯ

ГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
НИЖЕГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ
УНИВЕРСИТЕТ им. Р. Е. АЛЕКСЕЕВА

Э.С. СОКОЛОВА, Д.В. ДМИТРИЕВ, С.Н. КАПРАНОВ

Программирование на языке С++

Часть 2. Объектно-ориентированное программирование

Рекомендовано Ученым советом
Нижегородского государственного технического университета
им. Р.Е. Алексеева
в качестве учебно-методического пособия
для студентов всех форм обучения, включающих элементы
дистанционных технологий
по специальностям 230102 «Автоматизированные системы обработки
информации и управления», 230201 «Системы безопасности компьютеров»

Нижний Новгород
2010

УДК 004.4’242(075.8)
ББК 32.973.26-018.1я73

Соколова Э.С., Дмитриев Д.В., Капранов С.Н.
Программирование н языке С++. Ч.2 Объектно-ориентированное программирование: учеб.пособие /Э.С.Соколова, Д.В.Дмитриев, С.Н.Капранов; Нижегород.гос.техн.ун-т. – Н.Новгород, 2010. –107 с.

Учебное пособие предназначено для студентов 1, 2-го курсов специальностей 230102 «Автоматизированные системы обработки информации и управления», 230201 «Системы безопасности компьютеров».

В 2-й части учебного пособия дано введение в объектно-ориентированное программирование на языке С++. Представлены основные понятия и приемы объектно-ориентированного программирования: определение классов и объектов, инкапсуляция, наследование, полиморфизм, обработка исключительных ситуаций, перегрузка и переопределение. Пособие насыщено примерами, опробованными на современных системах разработки программного обеспечения. Представлены типичные ошибки, наиболее часто встречающиеся в программах.

Рис. 10. Библиогр.: 10 назв.

ББК 32.973.26-018.1я73

© Нижегородский государственный технический университет им. Р.Е. Алексеева, 2010
© Соколова Э.С., Дмитриев Д.В., Капранов С.Н., 2010

Содержание курса

1. Введение в ООП 5
   1.1. Абстракция данных 5
   1.2. Инкапсуляция 5
   1.3. Наследование 6
   1.4. Полиморфизм 6
   1.5. Раннее и позднее связывание 7
   1.6. Событийная управляемость. 8
   1.7. Достоинства ООП 8
   1.8. Недостатки ООП 9
2. Классы 11
   2.1. Описание класса 11
   2.2. Поля класса. Методы класса 13
   2.3. Описание объектов класса 16
   2.4. Хранение полей и методов класса. 17
   2.5. Доступ к элементам класса (private, public, protected) 18
   2.6. Конструкторы 19
   2.7. Конструкторы с параметрами 20
   2.8. Копирующий конструктор 22
   2.9. Деструкторы 23
   2.10. Объекты класса и указатель this 25
   2.11. Встраиваемые или Inline – функции 26
   2.12. Дружественные функции (friend) 29
   2.13. Статические поля класса 30
   2.14. Статические методы 33
   2.15. Константные методы 34
3. Наследование и композиция 35
   3.1. Композиция 35
   3.2. Базовые классы и производные классы 36
   3.3. Управление доступом к элементам базовых классов 40
   3.4. Переопределение элементов базового класса в производном классе 42
   3.5. Использование конструкторов и деструкторов в производных классах 43
   3.6. Функции, которые не наследуются автоматически 43
   3.7. Множественное наследование 43
4. Виртуальные функции и полиморфизм 47
   4.1. Виртуальные функции 48
   4.2. Позднее связывание 49
   4.3. Механизм вызова виртуальных функций 50
   4.4. Абстрактные классы и конкретные классы 52
   4.5. Виртуальные функции и деструкторы 53
   4.6. Отсутствие переопределения 54
   4.7. Заключение 56
5. Перегрузка и переопределение 57
   5.1. Перегрузка функций 57
   5.2. Указатели на перегруженные функции 58
   5.3. Безопасное связывание 58
   5.4. Три шага разрешения перегрузки 59
   5.5. Преобразования типов аргументов 60
   5.6. Перегрузка и ссылки: 61

6. Обработка ошибок и исключительные ситуации 63
6.1. Синтаксис исключений 64
6.2. Основная идея обработки исключений 66
6.3. Что такое "исключение" и как оно образуется 68
6.4. Генерация исключений 70
6.5. Перехват исключений 72
6.6. Использование обработчика catch(…) 79
6.7. Когда исключение считается обработанным? 80
6.8. Исключения связанные с ошибкой выделения памяти 81
6.9. Конструкторы и исключения 83
6.10. Деструкторы и исключения 84
6.11. Спецификация исключений 85
6.12. Алгоритм обработки исключительной ситуации 86
6.13. Функция set_unexpected() 87
6.14. Функция set_terminate() 87
6.15. Когда лучше обойтись без исключений 90
7. Шаблоны функций 90
8. Тестирование 94
8.1. Планирование работ по тестированию 95
8.2. Тестирование аналитических и проектных моделей 96
8.3. Основы тестирования классов 97
8.3.1. Способы тестирования классов 98
8.3.2. Оцениваемые факторы тестирования классов 99
8.3.3. Построение тестовых случаев для тестирования классов 99
8.3.4. Построение тестового драйвера 100
8.4. Тестирование взаимодействия и функционирования компонентов 100
8.4.1. Тестирование взаимодействий объектов 100
8.4.2. Выбор тестовых случаев 101
8.4.3. Тестирование протоколов 102
8.4.4. Тестовые шаблоны 102
9. Отладка 103
Список литературы 107

1. Введение в ООП

Объектно-ориентированное программирование основано на нескольких фундаментальных идеях:

- абстракция;
- инкапсуляция, заключающаяся в сокрытии реализации при открытом интерфейсе;
- наследование, обеспечивающее расширяемость кода и его повторное использование;
- полиморфизм, дающий возможность использования одинаковой подписи (прототипа) для функций, исполняющих разный код;
- позднее связывание, заключающееся в разрешении ссылки на функцию во время исполнения, а не на этапе компиляции;
- событийная управляемость.

Рассмотрим эти идеи по отдельности.

1.1. Абстракция данных

Абстракция данных позволяет выделить существенные характеристики некоторого объекта, отличающие его от всех других видов объектов. Её смысл состоит в том, чтобы посмотреть на объект, не заставляя себя разобраться в той совокупности сложных частей и взаимосвязей, из которых состоит данный объект. Абстракцию можно увидеть где угодно. Например, для того, чтобы ездить на машине, нам не нужно вникать, в то из каких частей она состоит и по каким принципам ездит.

1.2. Инкапсуляция

Формализуем понятие инкапсуляции в рамках объектно-ориентированного подхода к программированию.

В неформальной постановке вопроса под инкапсуляцией будем понимать доступность объекта исключительно посредством его свойств и методов.

Другими словами, концепция инкапсуляции призвана обеспечивать безопасность проектирования и реализации программного обеспечения на основе локализации манипулирования объектом в областях его полей и методов.

Иначе говоря, свойствами объекта можно оперировать исключительно посредством его методов. Это замечание касается как свойств, явно

определенных в описании объекта, так и свойств, унаследованных данным объектом от другого (других).

Практическая важность концепции инкапсуляции для современных языков объектно-ориентированного программирования (в том числе и для языка C++) определяется следующими фундаментальными свойствами.

Прежде всего, реализация концепции инкапсуляции обеспечивает совместное хранение данных (или, иначе, полей) и функций (или, иначе, методов) внутри объекта.

Как следствие, механизм инкапсуляции приводит к сокрытию информации о внутреннем "устройстве" объекта данных (или, в терминах языков ООП, свойств и методов объекта) от пользователя того или иного объектно-ориентированного приложения.

Таким образом, пользователь, получающий программное обеспечение как сервис, оказывается изолированным от особенностей среды реализации.

1.3. Наследование

В неформальной постановке под наследованием понимается свойство того или иного объекта, который является производным от некоего базового, сохранять поведение (а именно, атрибуты и операции над ними), характерное для родительского объекта.

С точки зрения языков программирования понятие наследования означает применимость всех или лишь некоторых свойств и/или методов базового (или родительского) класса для всех классов, производных от него. Кроме того, сохранение свойств и/или методов базового класса должно обеспечиваться и для всех конкретизаций (т.е. конкретных объектов) любого производного класса.

1.4. Полиморфизм

Полиморфизм даёт возможность использовать одинаковые имена (прототипы) для функций, исполняющих разный код.

Полиморфизмом это явление, при котором функции (методу) с одним и тем же именем соответствует разный программный код (полиморфный код) в зависимости от того, объект какого класса используется при вызове данного метода. Полиморфизм обеспечивается тем, что в классе-потомке изменяют реализацию метода класса-предка с обязательным сохранением сигнатуры метода.

Рассмотрим на примере с книгой. Пусть есть функция – открыть книгу на такой-то странице. Бумажную книгу надо пролистать, а в электронной – прокрутить ползунок.

1.5. Раннее и позднее связывание

Прежде чем коснуться самого применения виртуальных функций рассмотрим такие понятия, как **раннее** и **позднее связывание**. Сравним два подхода к покупке, к примеру, килограмма апельсинов. В первом случае мы заранее знаем, что нам надо купить 1 кг. апельсинов. Поэтому мы берем небольшой пакет, не много, но достаточно денег, чтобы хватило на этот килограмм. Во втором случае, мы, выходя из дома, не знаем что и как много нам надо купить. Поэтому мы берем машину (а вдруг будет много всего и тяжелое), запасаемся пакетами больших и малых размеров и берем как можно больше денег. Едем на рынок и выясняется, что надо купить только 1 кг. апельсинов.

Приведенный пример в определенной мере отражает смысл применения раннего и позднего связывания. Вполне очевидно, что для данного примера первый вариант оптимален. Во втором случае мы слишком много всего предусмотрели, но нам это не понадобилось. С другой стороны, если по дороге на рынок мы решим, что апельсины нам не нужны и решим купить 10 кг. яблок, то в первом случае мы уже не сможем этого сделать. Во втором же случае это не вызовет никаких затруднений.

А теперь рассмотрим этот пример с точки зрения программирования. При применении раннего связывания, мы как бы говорим компилятору: "Я точно знаю, чего я хочу. Поэтому жестко(статически) связывай все вызовы функций". При применении механизма позднего связывания мы как бы говорим компилятору: "Я пока не знаю чего я хочу. Когда придет время, я сообщу что и как я хочу".

Таким образом, во время раннего связывания вызывающий и вызываемый методы связываются при первом удобном случае, обычно при компиляции.

При позднем связывании вызываемого метода и вызывающего метода они не могут быть связаны во время компиляции. Поэтому реализован специальный механизм, который определяет, как будет происходить связывание вызываемого и вызывающего методов, когда вызов будет сделан фактически.

Очевидно, что скорость и эффективность при раннем связывании выше, чем при использовании позднего связывания. В то же время, позднее связывание обеспечивает некоторую универсальность связывания.

Позднее связывание заключается в разрешении ссылки на функцию во время исполнения, а не на этапе компиляции.

Это обеспечивает сохранение неизменным интерфейса класса-предка и позволяет осуществить связывание имени метода в коде с разными классами — из объекта какого класса осуществляется вызов, из того класса и берётся метод с данным именем.

Такой механизм называется динамическим (или поздним) связыванием — в отличие от статического (раннего) связывания, осуществляемого на этапе компиляции.

1.6. Событийная управляемость.

Передача управления внутри программы осуществляется не только путем явного указания последовательности обращений одних функций программы к другим, но и путем генерации сообщений различным объектам, разбора сообщений соответствующим обработчиком и передача их объектам, для которых данные сообщения предназначены.

1.7. Достоинства ООП

От любой методики разработки программного обеспечения мы ждем, что она поможет нам в решении наших задач. Но одной из самых значительных проблем проектирования является сложность. Чем больше и сложнее программная система, тем важнее разбить ее на небольшие, четко очерченные части. Чтобы справиться со сложностью, необходимо абстрагироваться от деталей. В этом смысле классы представляют собой весьма удобный инструмент:

- Классы позволяют проводить конструирование из полезных компонентов, обладающих простыми инструментами, что позволяет абстрагироваться от деталей реализации.
- Данные и операции над ними образуют определенную сущность, и они не разносятся по всей программе, как нередко бывает в случае процедурного программирования, а описываются вместе. Локализация кода и данных улучшает наглядность и удобство сопровождения программного обеспечения.
- Инкапсуляция позволяет привнести свойство модульности, что облегчает распараллеливание выполнения задачи между несколькими исполнителями и обновление версий отдельных компонентов.

ООП дает возможность создавать расширяемые системы. Это одно из основных достоинств ООП, и именно оно отличает данный подход от традиционных методов программирования. Расширяемость означает, что существующую систему можно заставить работать с новыми компонентами, причем без внесения в нее каких-либо изменений. Компоненты могут быть добавлены на этапе исполнения программы.

Полиморфизм оказывается полезным преимущественно в следующих ситуациях:

- Обработка разнородных структур данных. Программы могут работать, не различая вида объектов, что существенно упрощает код. Новые виды могут быть добавлены в любой момент.

- Изменение поведения во время исполнения. На этапе исполнения один объект может быть заменен другим, что позволяет легко, без изменения кода, адаптировать алгоритм в зависимости от того, какой используется объект.
- Реализация работы с наследниками. Алгоритмы можно обобщить настолько, что они уже смогут работать более чем с одним видом объектов.
- Создание "каркаса" (*framework*). Независимые от приложения части предметной области могут быть реализованы в виде набора универсальных классов, или каркаса (*framework*), и в дальнейшем расширены за счет добавления частей, специфичных для конкретного приложения.

Часто многоразового использования программного обеспечения не удается добиться из-за того, что существующие компоненты уже не отвечают новым требованиям. ООП помогает этого достичь без нарушения работы уже имеющихся клиентов, что позволяет извлечь максимум из многоразового использования компонентов:

- Сокращается время на разработку, которое может быть отдано другим задачам.
- Компоненты многоразового использования обычно содержат гораздо меньше ошибок, чем вновь разработанные, ведь они уже не раз подвергались проверке.
- Когда некий компонент используется сразу несколькими клиентами, улучшения, вносимые в его код, одновременно оказывают положительное влияние и на множество работающих с ним программ.
- Если программа опирается на стандартные компоненты, ее структура и пользовательский интерфейс становятся более унифицированными, что облегчает ее понимание и упрощает использование.

1.8. Недостатки ООП

Документирование классов - задача более трудная, чем это было в случае процедур и модулей. Поскольку любой метод может быть переопределен, в документации должно говориться не только о том, что делает данный метод, но и о том, в каком контексте он вызывается. Ведь переопределенные методы обычно вызываются не клиентом, а самим каркасом. Таким образом, программист должен знать, какие условия выполняются, когда вызывается данный метод. Для абстрактных методов, которые пусты, в документации должно говориться о том, для каких целей предполагается использовать переопределяемый метод.

В сложных иерархиях классов поля и методы обычно наследуются с разных уровней. И не всегда легко определить, какие поля и методы фактически относятся к данному классу. Для получения такой информации нужны специальные инструменты, вроде навигаторов классов. Если конкретный класс расширяется, то каждый метод обычно сокращают перед передачей сообщения базовому классу. Реализация операции, таким образом,

рассредоточивается по нескольким классам, и чтобы понять, как она работает, нам приходится внимательно просматривать весь код.

Методы, как правило, короче процедур, поскольку они осуществляют только одну операцию над данными, зато их намного больше. В коротких методах легче разобраться, но они неудобны тем, что код для обработки сообщения иногда "размазан" по многим маленьким методам.

Инкапсуляцией данных не следует злоупотреблять. Чем больше логики и данных скрыто в недрах класса, тем сложнее его расширять. Отправной точкой здесь должно быть не то, что клиентам не разрешается знать о тех или иных данных, а то, что клиентам для работы с классом этих данных знать не требуется.

Многие считают, что ООП является неэффективным. Как же обстоит дело в действительности? Мы должны проводить четкую грань между неэффективностью на этапе выполнения, неэффективностью в смысле распределения памяти и неэффективностью, связанной с излишней универсализацией.

Неэффективность на этапе выполнения:

В языках типа *Smalltalk* сообщения интерпретируются во время выполнения программы путем осуществления их поиска в одной или нескольких таблицах и за счет выбора подходящего метода. Конечно, это медленный процесс. И даже при использовании наилучших методов оптимизации *Smalltalk*-программы в десять раз медленнее оптимизированных С-программ.

В гибридных языках типа *Oberon-2*, *Object Pascal* и С++ отправка сообщения приводит лишь к вызову через указатель процедурной переменной. На некоторых машинах сообщения выполняются лишь на 10% медленнее, чем обычные процедурные вызовы. И поскольку сообщения встречаются в программе гораздо реже других операций, их воздействие на время выполнения влияния практически не оказывает.

Однако существует другой фактор, который влияет на время выполнения: это инкапсуляция данных. Рекомендуется не предоставлять прямой доступ к полям класса, а выполнять каждую операцию над данными через методы. Такая схема приводит к необходимости выполнения процедурного вызова каждый раз при доступе к данным. Однако если инкапсуляция используется только там, где она необходима (т.е. в тех случаях, когда это становится преимуществом), то замедление вполне приемлемое.

Неэффективность в смысле распределения памяти:

Динамическое связывание и проверка типа на этапе выполнения требуют по ходу работы информации о типе объекта. Такая информация хранится в дескрипторе типа и он выделяется один на класс. Каждый объект имеет невидимый указатель на дескриптор типа для своего класса. Таким образом, в объектно-ориентированных программах необходимая

дополнительная память выражается в одном указателе для объекта и в одном дескрипторе типа для класса.

Излишняя универсальность:

Неэффективность также может означать, что в программе реализованы избыточные возможности. В библиотечном классе часто содержится больше методов, чем это реально необходимо. А поскольку лишние методы не могут быть удалены, они становятся мертвым грузом. Это не влияет на время выполнения, но сказывается на размере кода.

Одно из возможных решений - строить базовый класс с минимальным числом методов, а затем уже реализовывать различные расширения этого класса, которые позволят нарастить функциональность. Другой подход - дать компоновщику возможность удалять лишние методы. Такие интеллектуальные компоновщики уже существуют для различных языков и операционных систем.

Но нельзя утверждать, что ООП неэффективно. Если классы используются лишь там, где это действительно необходимо, то потеря эффективности из-за повышенного расхода памяти и меньшей производительности незначительна. Кроме того, надежность программного обеспечения и быстрота его написания часто бывает важнее, чем производительность.

2. Классы

На данном этапе развития языков программирования передовой технологией разработки программного обеспечения является разработка, основанная на создании и поддержке абстрактных типов данных. Одновременно с вводом новых типов данных вводятся и новые типы операций, необходимые для манипулирования переменными абстрактных типов.

Таким образом, получаем, что программист сам вводит новые типы данных, называемые классами.

Класс – это производный структурированный тип, введенный программистом на основе уже существующих типов. Механизм классов позволяет создавать типы в полном соответствии с принципами абстракции данных, т.е. класс задает некоторою структурированную совокупность типизированных данных и позволяет определить набор операций над этими данными.

2.1. Описание класса

В языке С++ абстрактным типом данных, определяемым пользователем является класс. Этот класс представляет собой некую упрощённую модель объекта реального мира. В этой модели кроме данных также определяются функции и операции для работы с ними.

Объект класса это переменная этого типа.

Пример
Класс – книга.
Объект – Полный справочник по С++. Герберт Шилдт.

Простейшим образом класс можно определить с помощью конструкции:

ключ_класса имя_класса{ список_компонентов };

где:
ключ_класса - одно из служебных слов *class*, *struct*, *union*;
имя_класса - произвольно выбираемый идентификатор;
список_компонентов - определения и описания типизированных данных и принадлежащих классу функций.

Описание класса это такой же оператор, как и все остальные, поэтому оно должно заканчиваться точкой с запятой.

Например, рассмотрим следующий класс:

```
class Rectangle
{
    float Width;
    float Height;
public:
    int GetArea();

    void SetWidth(float Value)
    {
        Width = Value;
    }

    void SetHeight(float);
}
```

В проекте стандарта языка С++ указано, что компонентами класса могут быть данные, функции, классы, перечисления, битовые поля, дружественные функции, дружественные классы и имена типов. Вначале для простоты будем считать, что компоненты класса - это типизированные данные (базовые и производные) и функции. Заключенный в фигурные скобки список компонентов называют телом класса. Телу класса предшествует заголовок. В простейшем случае заголовок класса включает ключ класса и его имя. Определение класса всегда заканчивается точкой с запятой.

Принадлежащие классу функции мы будем называть методами класса или компонентными функциями. Данные класса - компонентными данными или элементами данных класса.

В качестве **ключа_класса** можно использовать служебное слово *struct*, но класс отличается от обычного структурного типа, по крайней мере, возможностью включения компонентных функций

Итак, класс - это тип, введенный программистом. Каждый тип служит для определения объектов. Для описания объекта класса используется конструкция:

имя_класса имя_объекта;

2.2. Поля класса. Методы класса.

Например, рассмотрим следующий класс:

```
class прямоугольник
{
    int storona_a;
    int storona_b;
public:
    int площадь()
    {
        return storona_a * storona_b;
}
}
```

Поля класса хранят всю необходимую информацию об объекте. Изменение состояния объекта класса связано с изменением его полей.

Поля класса могут иметь любой тип, кроме типа этого же класса (но могут быть указателями или ссылками на этот же класс).

При описании полей их инициализация не допускается.

Класс может содержать один или более методов, позволяющих осуществлять манипуляцию данными объекта

Метод объекта – программный код, выполненный в виде процедуры или функции, реагирующий на передачу объекту определенного сообщения.

Вызов метода объекта может приводить к изменению его состояния (значение полей-данных), а может и не приводить.

Метод может полностью определяется в теле класса (например «`SetWidth`»), хотя в большинстве случаев в классе методы описываются только прототипами. Тело же функции выносится за определение класса. Как у методов `GetArea` и `SetHeight`.

Так как в большинстве случаев у различных классов некоторые имена методов могут совпадать, то для того чтобы компилятор знал какой метод связан с каким классом, перед именем метода указывается соответствующий класс.

Это выглядит следующим образом:

тип_возвращаемого_значения имя_класса::имя_метода(список_аргументов)

Например:

```
int Rectangle::GetArea();
{
    return Width * Height;
}
```

Оператор *«::»* называется оператором разрешения области видимости. Он говорит о том что метод «*GetArea*» относится к классу «*Rectangle*». Если перед оператором *«::»* нет никаких имён, то метод или переменная в правой части относится к глобальным переменным.

Для того, что бы вызвать открытый метод класса или получить доступ к его открытым полям, необходимо указать имя объекта, за которым ставится оператор *«.»* и имя элемента класса, к которому мы получаем доступ.

Например:

```
int       S;
Rectangle LRect;

LRect.SetWidth(1);
LRect.SetHeight(3);

S = LRect.GetArea();
```

Внутри методов класса можно обращаться к любым элементам класса без оператора «.», т.к. компилятор знает, что обращение идет к элементу класса.

Классы могут целиком состоять из методов.

В определяемые объекты класса входят данные (элементы), соответствующие компонентным данным класса. Компонентные функции класса позволяют обрабатывать данные конкретных объектов класса. Но в отличие от компонентных данных компонентные функции не тиражируются при создании конкретных объектов класса. Если перейти на уровень реализации, то место в памяти выделяется именно для элементов каждого объекта класса. Определение объекта класса предусматривает выделение участка памяти и деление этого участка на фрагменты, соответствующие

отдельным элементам объекта, каждый из которых отображает отдельный компонент данных класса.

Как только объект класса определен, появляется возможность обращаться к его компонентам:

имя_объекта.**имя_класса**::**имя_компонента**

Имя класса с операцией уточнения области действия "::" обычно может быть опущено, и чаще всего для доступа к данным конкретного объекта заданного класса (как и в случае структур) используется уточненное имя:

имя_объекта.**имя_элемента**

При этом возможности те же, что и при работе с элементами структур.

Для того, что бы вызвать открытый метод класса или получить доступ к его открытым полям, необходимо указать имя объекта, за которым ставится оператор «.» и имя элемента класса, к которому мы получаем доступ.

имя_объекта.**имя_метода**(**параметры**)

Другой способ доступа к элементам объекта некоторого класса предусматривает явное использование указателя на объект класса и операции косвенного выбора компонента ('->'):

указателъ_на_объект_класса -> **имя_элемента**

Указатель на объект класса позволяет вызывать принадлежащие классу функции для обработки данных того объекта, который адресуется указателем. Формат вызова функции:

укаэатель_на_объект_класса -> .**имя_метода**(**параметры**)

Описание компонента класса с ключевым словом *static* определяет статический компонент класса.

Статические компоненты классов не "дублируются" при создании объектов класса, т.е. каждый статический компонент существует в единственном экземпляре. Доступ к статическому компоненту возможен только после его инициализации. Для инициализации используется конструкция:

тип имя_класса::**имя_компонента** = **инициализатор**;

Только при инициализации статический компонент класса получает память и становится доступным. Для обращения к статическому компоненту используется квалифицированное имя:

имя_класса::**имя_компонента**

Кроме того, статический компонент доступен "через" имя конкретного объекта:

имя_объекта.**имя_класса**::**имя_компонента**

либо

имя_объекта.имя_компонента

Поля класса могут иметь любой тип, кроме типа этого класса. Исключением могут служить указатели или ссылки на элементы данного класса.

Методы используются одни и те же для всех объектов данного класса.

```
///// Пример с адресами функций
class m
{
   int i;
   int f();
}
int main()
{
   m m1;
   m m2;
   printf(адрес функции m1.f);
   printf(адрес функции m2.f);
}
```

Классы могут целиком состоять из методов

2.3. Описание объектов класса

Описав класс как новый тип данных, мы уже можем его использовать. Заметим, что определив класс как тип, мы не определяем никакие объекты этого типа.

Для того чтобы определить объекты класса, которые ещё называются экземплярами класса, вводятся переменные с типом этого класса. Таким

образом, получаем что объект это имя указывающее на определённую область памяти. В С++ время жизни и видимость объектов класса подчиняется тем же законам как и любые другие обычные переменные.

```
Rectangle R1;       // объект класса «прямоугольник»

Rectangle R2(1,2); // объект класса «прямоугольник»
                    // с начальной инициализацией
                    // переменных класса

Rectangle R3[10];  // массив объектов класса
                    // «прямоугольник»
```

При создании экземпляра класса выделяется область памяти для хранения всех полей класса. После этого автоматически вызывается конструктор класса (его мы рассмотрим ниже). В одном классе могут присутствовать несколько конструкторов, которые призваны по-разному инициализировать объект.

При уничтожении объекта автоматически вызывается деструктор класса (его мы рассмотрим ниже). В отличие от конструктора, которых в классе может быть несколько, деструктор в классе может быть только один.

2.4. Хранение полей и методов класса.

При инициализации множества объектов некоторого класса под каждый выделяется область памяти, где хранятся элементы–данные этого класса.

Методы используются одни и те же для всех объектов данного класса. Методы класса создаются и помещаются в намять компьютера всего один раз — при создании класса. Это вполне логически оправданно: нет никакого смысла держать в памяти копии методов для каждого объекта данного класса, поскольку у всех объектов методы одинаковы. А поскольку наборы значений полей у каждого объекта свои, поля объектов не должны быть общими.

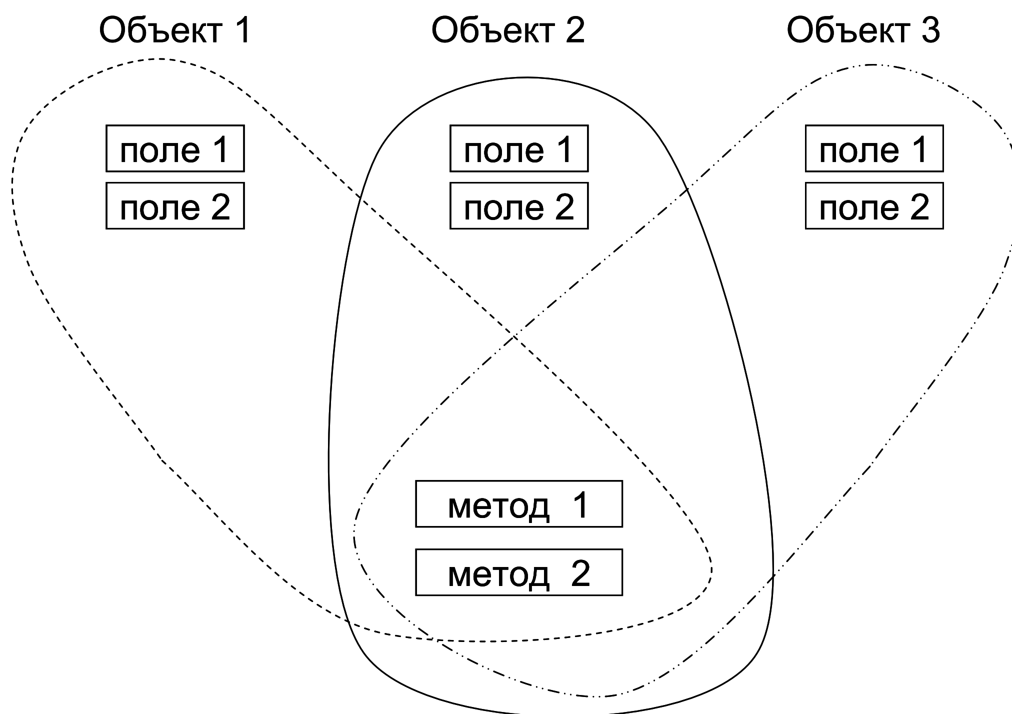


Рисунок 1 Объекты, данные, функции и память

При уничтожении объектов класса, память под полями освобождается. А методы остаются в памяти.

2.5. Доступ к элементам класса (private, public, protected)

В соответствии с правилами языка С++ все компоненты класса, введенного с помощью ключа класса *struct*, являются общедоступными. По умолчанию все члены класса *class* являются закрытыми, всегда есть возможность сделать некоторые элементы класса открытыми. Это необходимо для того что бы было возможно взаимодействие с другими объектами программы.

Для изменения видимости компонент в определении класса можно использовать спецификаторы доступа. Спецификатор доступа - это одно из трех служебных слов

private (собственный),
public (общедоступный),
protected (защищенный),

за которым помещается двоеточие.

Появление любого из спецификаторов доступа в тексте определения класса означает, что до конца определения либо до другого спецификатора доступа все компоненты класса имеют указанный статус.

Ключевое слово *public* определяет элементы класса к которым могут получить доступ все элементы программы без каких то ограничений.

Обычно в этом разделе находятся только методы класса, через которые внешние элементы программы общаются с закрытыми членами класса.

Защищенные (*protected*) компоненты классов нужны только в случае построения иерархии классов. При использовании классов без порождения на основе одних классов других (производных), применение спецификатора *protected* эквивалентно использованию спецификатора *private*.

Применение в качестве ключа класса служебного слова *union* приводит к созданию классов с несколько необычными свойствами, которые нужны для весьма специфических приложений. Пример такого приложения - экономия памяти за счет многократного использования одних и тех же участков памяти для разных целей. В каждый момент времени исполнения программы объект-объединение содержит только один компонент класса, определенного с помощью *union*. Все компоненты этого класса являются общедоступными, но доступ может быть изменен с помощью спецификаторов доступа *protected* (защищенный), *private* (собственный), *public* (общедоступный).

Изменить статус доступа к компонентам класса можно и с помощью использования в определении класса ключевого слова *class*. Все компоненты класса, определение которого начинается со служебного слова *class*, являются собственными (*private*), т.е. недоступными для внешних обращений. Так как класс, все компоненты которого недоступны вне его определения, редко может оказаться полезным, то изменить статус доступа к компонентам позволяют спецификаторы доступа *protected* (защищенный), *private* (собственный), *public* (общедоступный).

Итак, для сокрытия данных внутри объектов класса, определенного с применением ключа *struct*, достаточно перед их появлением в определении типа (в определении класса) поместить спецификатор *private*. При этом необходимо, чтобы некоторые или все принадлежащие классу функции остались доступными извне, что позволило бы манипулировать с данными объектов класса.

Правила хорошего программирования требуют, чтобы все поля класса были закрытыми, а доступ к необходимым полям осуществлялся косвенно через открытые методы класса.

Если же элементы класса описаны в разделе *protected* (защищённые), то к ним имеют доступ только объекты этого класса и его наследники.

2.6. Конструкторы

Перед использованием в программе любых переменных их необходимо инициализировать, т.к. компилятор просто выделяет память под переменную,

но не очищает её. Получаем что если переменная не проинициализированная, то она может принять любое значение, даже недопустимое.

Для защиты от неправильной инициализации переменных абстрактных типов в объектно-ориентированном программировании служит специальная функция, которая называется конструктор. Эта функция включается в класс как его метод.

Особенности конструкторов:

1) Конструктор имеет то же имя что и класс.

2) Конструктор не возвращает значение, даже типа *void*. Нельзя получить указатель на конструктор.

3) Класс может иметь несколько конструкторов с разными параметрами для разных способов инициализации объекта (при этом используется механизм перегрузки).

4) Конструктор, вызываемый без параметров, называется конструктором по умолчанию.

5) Параметры конструктора могут иметь любой тип, кроме этого же класса. Можно задавать значения параметров по умолчанию. Их может содержать только один из конструкторов.

6) Если в классе не указано ни одного конструктора, компилятор создает его автоматически. Такой конструктор вызывает конструкторы по умолчанию для полей класса и конструкторы по умолчанию базовых классов (занятее про «Наследование»). В случае, когда класс содержит константы или ссылки, при попытке создания объекта класса будет выдана ошибка, поскольку их необходимо инициализировать конкретными значениями, а конструктор по умолчанию этого делать не умеет.

7) Конструкторы не наследуются.

8) Конструкторы нельзя описывать с модификаторами `const`, `virtual` и `static`.

9) Конструктор вызывается в момент создания объекта.

10) Конструкторы глобальных объектов вызываются до вызова функции *main*. Локальные объекты создаются, как только становится активной область их действия. Конструктор запускается и при создании временного объекта (например, при передаче объекта из функции).

2.7. Конструкторы с параметрами

Конструкторам можно передавать аргументы, предназначенным для инициализации объекта. Параметры конструктора задаются также как и для любой другой функции.

Пример:

```
class Rectangle
```

```
{
    private:
        float Width;
        float Height;
    public:
        Rectangle () : Width(0), Height(0)
        { /*пустое тело функции*/ }

        Rectangle (float w) : Width(w)
        {
            Height = 0;
        }

        Rectangle (float w, float h) : Width(w),
Height(h)
        { /*пустое тело функции*/ }
}
```

Конструктор нельзя вызывать как обычную компонентную функцию. Для явного вызова конструктора можно использовать две разные синтаксические формы:

```
имя_класса имя_объекта (фактичаские_параметры_конструкора);
имя_класса(фактические_параметры_конструктора);
```

Первая форма допускается только при непустом списке фактических параметров. Она предусматривает вызов конструктора при определении нового объекта данного класса:

Пример:

```
Rectangle SS(1.3, 0.22);   // SS.Width = 1.3; SS.Height
= 0.22

Rectangle SS(2.3);         // SS.Width = 2.3;
                         // по умолчанию SS.Height = 0.0
```

Вторая форма явного вызова конструктора приводит к созданию объекта, не имеющего имени. Созданный таким вызовом безымянный объект может использоваться в тех выражениях, где допустимо использование объекта данного класса. Например:

```
Rectangle ZZ= Rectangle(4.0,5,0);
```

Этим определением создается объект ZZ, которому присваивается значение безымянного объекта (с элементами `Width` = 4.0, `Height` = 5.0), созданного за счет явного вызова конструктора.

Если конструктор имеет один параметр, то инициализировать объект можно следующим образом:

Пример:

```
Rectangle ZZ=7.6;
```

В этом случае компилятор считает этот оператор эквивалентный следующему:

```
Rectangle ZZ(7.6);
```

Здесь нужно быть очень осторожным, т.к. возможно вы хотели присвоить значение 7.6 какому то полю, а не вызывать конструктор, обнуляющий все остальные поля

Приёмом защитного программирования служит инициализация в конструкторе всех полей класса

2.8. Копирующий конструктор

Мы рассмотрели два способа инициализации объектов. Конструктор без аргументов может инициализировать поля объекта константными значениями, а конструктор, имеющий хотя бы один аргумент, может инициализировать поля значениями, переданными ему в качестве аргументов. Рассмотрим третий способ инициализации объекта, использующий значения полей уже существующего объекта. Для этого не нужно самим создавать специальный конструктор, поскольку такой конструктор предоставляется компилятором для каждого создаваемого класса и называется копирующим конструктором по умолчанию. Копирующий конструктор имеет единственный аргумент, являющийся объектом того же класса, что и конструктор. Пример:

```
class Rectangle
{
    private:
        float Width;
        float Height;
    public:
        Rectangle () : Width(0), Height(0)
        { /*пустое тело функции*/ }

        Rectangle (float w) : Width(w)
        {
             Height = 0;
```

```
        }

        Rectangle (float w, float h) : Width(w),
Height(h)
        { /*пустое тело функции*/ }
}
int main()
{
    Rectangle C1(9, 1.2);
    Rectangle C2(C1);
    Rectangle C3 = C1;

    return 0;
}
```

Мы инициализировали объект *C*1 значением `9 и 1.2` при помощи конструктора с двумя аргументами. Затем мы определяем еще два объекта класса *Rectangle* с именами *C*2 и *C*3, оба из которых инициализируются значением объекта *C*1. В обоих случаях был вызван копирующий конструктор но умолчанию.

Действие копирующего конструктора по умолчанию сводится к копированию значений полей объекта *C*1 в соответствующие поля объекта *C*2. То же самое выполняется и для объекта *C*3.

Кроме того, конструкторы копирования вызываются при передаче объекта в функцию по значению и при возврате объекта из функции.

В занятии по «Виртуальным функциям» мы рассмотрим, каким образом можно создать свой собственный копирующий конструктор с помощью перегрузки копирующего конструктора по умолчанию.

2.9. Деструкторы

В ходе своей работы объект может использовать определенные системные ресурсы, такие как динамическая память, открытые файлы, сетевые соединения и т.п.

Для освобождения этих ресурсов служит особый метод класса – деструктор.

Имя деструктора совпадает с именем класса, только перед ним указывается символ ~ (тильда).

Деструктор:

- не имеет аргументов и возвращаемого значения;
- не может быть объявлен как *const* или *static*;
- не наследуется;

- может быть виртуальным.

Данная функция вызывается автоматически при уничтожении экземпляра класса:

- для локальных объектов — при выходе из блока, в котором они объявлены;
- для глобальных — как часть процедуры выхода из *main*;
- для объектов, заданных через указатели, деструктор вызывается неявно при использовании операции *delete* или *delete* [].

Пример:

```
class MyFile
{
public:
    MyFile():m_pFile(NULL)
    {}

    ~MyFile()
    {
        Close();
    }
    bool Open(const char *fileName)
    {
        Close();
        m_pFile = fopen(fileName, "r");
        return m_pFile != NULL;
    }

    void Close()
    {
        if (m_pFile){fclose(m_pFile); m_pFile = NULL;}
    }
private:
    FILE *m_pFile;
};
```

Деструктор можно вызвать явным образом путем указания полностью уточненного имени, например:

```
MyFile *F;
F->~MyFile();
```

2.10. Объекты класса и указатель this

Когда функция, принадлежащая классу, вызывается для обработки данных конкретного объекта, этой функции автоматически и неявно передается указатель на тот объект, для которого функция вызвана. Этот указатель имеет фиксированное имя *this* и незаметно для программиста определен в каждой функции класса следующим образом:

имя_класса::**имя_компонентной_функции**
имя_класса * const this = **адрес_обрабатываемого_объекта**;

Имя *this* является служебным (ключевым) словом. Явно описать или определить указатель *this* нельзя. В соответствии с неявным определением *this* является константным указателем, т.е. изменять его нельзя, однако в каждой принадлежащей классу функции он указывает именно на тот объект, для которого функция вызывается. Говорят, что указатель *this* является дополнительным (скрытым) параметром каждой нестатической компонентной функции. Другими словами, при входе в тело принадлежащей классу функции указатель *this* инициализируется значением адреса того объекта, для которого вызвана функция. Объект, который адресуется указателем *this*, становится доступным внутри принадлежащей классу функции именно с помощью указателя *this*. При работе с компонентами класса внутри принадлежащей классу функции можно было бы везде использовать этот указатель.

Снятие неоднозначности в теле принадлежащей классу функции между одинаковыми именами формального параметра и компонента класса можно осуществить и без использования указателя *this*. Гораздо чаще для этой цели применяют операцию изменения видимости, т.е. используют выражение

имя_класса::**имя_компонента**

Почти незаменимым и очень удобным указатель *this* становится в тех случаях, когда в теле принадлежащей классу функции нужно явно задать адрес того объекта, для которого она вызвана. Например, если в классе нужна функция, помещающая адрес выбранного объекта класса в массив или включающая конкретный объект класса в список, то такую функцию сложно написать без применения указателя *this*. Действительно, при организации связных списков, звеньями которых должны быть объекты класса, необходимо включать в связи звеньев указатель именно на тот объект, который в данный момент обрабатывается. Это включение должна выполнить некоторая функция-компонент класса. Однако конкретное имя включаемого объекта в момент написания этой принадлежащей классу функции недоступно, так как его гораздо позже произвольно выбирает программист, используя класс как тип данных. Можно передавать такой

функции ссылку или указатель на нужный объект, но гораздо проще использовать указатель t*h*is.

Итак, повторим, когда указатель *this* использован в функции, принадлежащей классу, например, с именем *ZOB*, то он имеет по умолчанию тип *ZOB *const* и всегда равен адресу того объекта, для которого вызвана компонентная функция.

Для иллюстрации использования указателя *this* добавим в приведенный выше класс *Max_Area* новый метод, возвращающий ссылку на прямоугольник с наибольшей площадью, один объектов вызывает метод, а другой передается ему в качестве параметра (метод нужно поместить в секцию *public* описания класса):

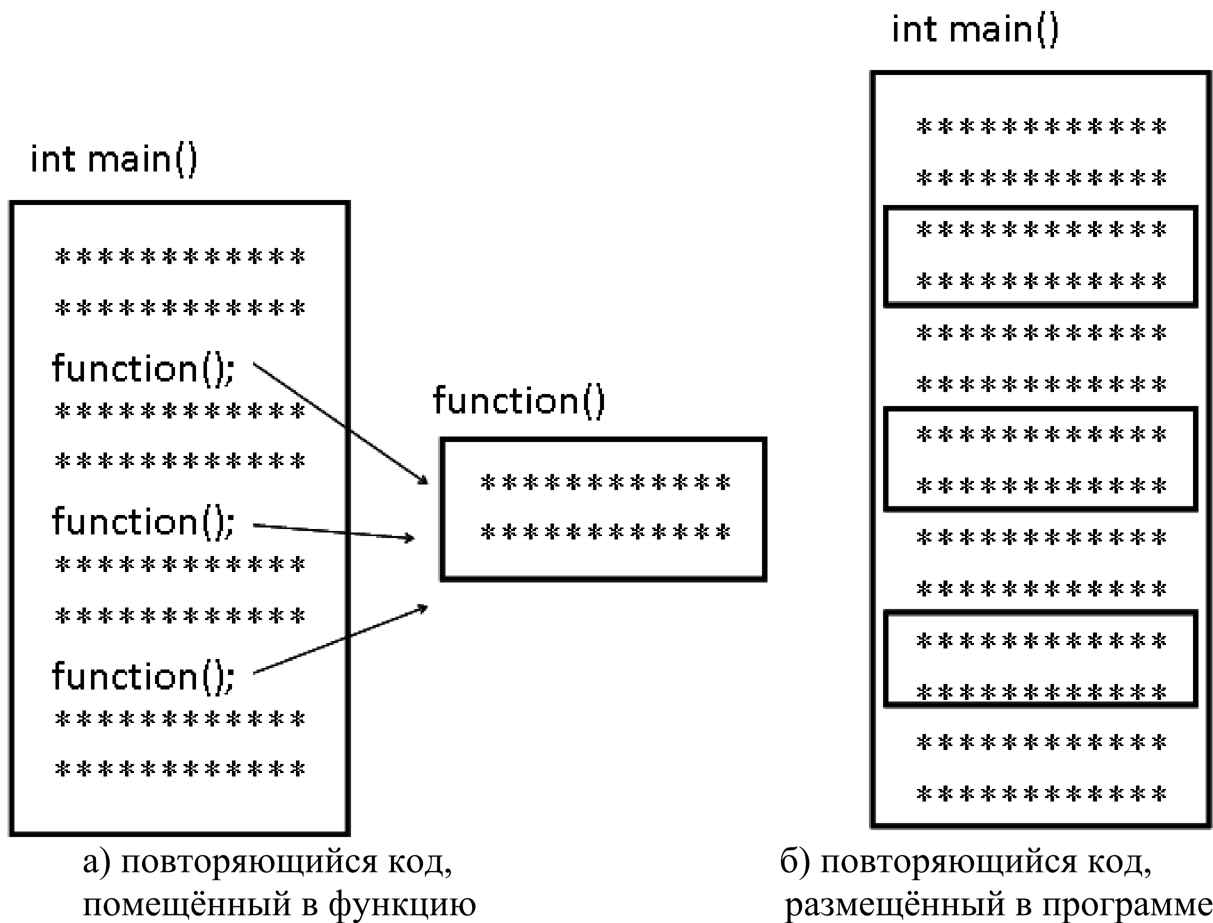
```
Rectangle & Max_Area(Rectangle &R)
{
    if( GetArea() > R.GetArea())
    {
       return *this;
    }
    else return R;
}
...
Rectangle Rect1(5, 1.7), Rect2(2, 4,8);
// Новый объект Largest инициализируется значениями полей Rect2
Rectangle Largest = Rect1.Max_Area(Rect2);
```

2.11. Встраиваемые или Inline – функции

Использование функций является экономичным с точки зрения использования памяти, поскольку вместо дублирования кода используется механизм вызовов функций. Когда компилятор встречает вызов функции, он генерирует команду перехода в эту функцию. После выполнения функции осуществляется переход на оператор, следующий за вызовом функции.

Использование функций, наряду с сокращением размера памяти, занимаемой кодом, увеличивает время выполнения программы. Для выполнения функции должны быть сгенерированы команды переходов (как правило, это инструкция языка ассемблера *CALL* или аналогичная ей команда), команды, сохраняющие значения регистров процессора, команды, помещающие в стек и извлекающие из стека аргументы функции (если они есть), команды, восстанавливающие значения регистров после выполнения функции, и наконец, команда перехода из функции обратно в программу. Кроме того, если функция возвращает значение, то необходимы дополнительные команды, работающие с этим значением. Выполнение всех перечисленных инструкций замедляет работу программы.

Для того чтобы сократить время выполнения небольших функций, вы можете дать указание компилятору, чтобы при каждом вызове такой функции вместо команды перехода производилась подстановка операторов, выполняемых функцией, в код программы. Различия между обычными и встраиваемыми функциями показаны на рис 2.



а) повторяющийся код, помещённый в функцию

б) повторяющийся код, размещённый в программе

Рисунок 2 Функции а) обычные б) встраиваемые

Длинные повторяющиеся последовательности действий лучше объединять в обычные функции, поскольку экономия памяти в этом случае оправдывает дополнительные накладные расходы на время выполнения программы. Но если вынести в функцию небольшой фрагмент кода, то выигрыш от экономии памяти будет мал, тогда как дополнительные временные затраты останутся почти на том же уровне, что и для функции большого объема. На самом деле при небольшом размере функции дополнительные инструкции, необходимые для ее вызова, могут занять столько же памяти, сколько занимает код самой функции, поэтому экономия памяти может в конечном счете вылиться в дополнительный ее расход.

В подобных ситуациях вы могли бы вставлять повторяющиеся последовательности операторов в те места программы, где это необходимо, однако в этом случае вы теряете преимущества процедурной структуры программы — четкость и ясность программного кода. Возможно, программа

станет работать быстрее и занимать меньше места, но ее код станет неудобным для восприятия.

Решением данной проблемы служит использование встраиваемых функций. Встраиваемые функции пишутся так же, как и обычные, но при компиляции их исполняемый код вставляется, или встраивается, в исполняемый код программы. Листинг программы сохраняет свою организованность и четкость, поскольку функция остается независимой частью программы. В то же время компиляция обеспечивает эффект встраивания кода функции в код программы.

Встраиваемыми следует делать только очень короткие функции, содержащие один-два оператора.

Пример:

```
// применение встроенных функций
#include <iostream>
using namespace std;
// функция lbstokg()
// переводит фунты в килограммы
inline float lbstokg(float pounds)
{
    return 0.453592*pounds;
}

int main()
{
    float lbs:
    cout <<"\n Введите вес в фунтах: ";
    cin >> lbs;
    cout << "\n Вес в килограммах" << lbstokg(lbs) << endl;
    return 0;
}
```

Для того чтобы сделать функцию встраиваемой, необходимо лишь указать ключевое слово *inline* в прототипе функции:

```
inline float lbstokg(float pounds)
```

Следует отметить, что ключевое слово *inline* является лишь рекомендацией компилятору, которая может быть проигнорирована. В этом случае функция будет скомпилирована как обычная. Такое может произойти, например, в том случае, если компилятор посчитает функцию слишком длинной для того, чтобы делать ее встраиваемой.

Если вы знакомы с языком С, то вы отметите, что встраиваемые функции являются аналогом широко используемого в языке С макроса *#define*. Преимуществом встраиваемых функций по сравнению с макросами является их более корректная работа с типами данных, а также удобный синтаксис.

2.12. Дружественные функции (friend)

Дружественной функцией класса называется функция, которая, не являясь его компонентом, имеет доступ к его защищенным и собственным компонентам. Функция не может стать другом класса "без его согласия". Для получения прав друга функция должна быть описана в теле класса со спецификатором *friend*. Именно при наличии такого описания класс предоставляет функции права доступа к защищенным и собственным компонентам.

Отметим особенности дружественных функций:

- Дружественная функция при вызове не получает указателя *this*. Объекты классов должны передаваться дружественной функции только явно через аппарат параметров.
- При вызове дружественной функции нельзя использовать операции выбора:

имя_объекта.**имя_функции**

и

указатель_на_объект -> **имя_функции**

Все это связано с тем фактом, что дружественная функция не является компонентом класса.

- На дружественную функцию не распространяется и действие спецификаторов доступа (*public*, *protected*, *private*). Место размещения прототипа дружественной функции внутри определения класса безразлично. Права доступа дружественной функции не изменяются и не зависят от спецификаторов доступа.

Итак, дружественная функция:

- не может быть компонентной функцией того класса, по отношению к которому определяется как дружественная;
- может быть глобальной функцией:

Пример:

```
class CL { friend int f1 (...); ... };
int f1(...) { тело_функции }
```

- может быть компонентной функцией другого ранее определенного класса:

```
class CLASS { ...  char f2(...); ... };
class CL    { ...  friend char CLASS::f2(...); ... };
```

В примере класс *CLASS* с помощью своей компонентной функции *f2*() получает доступ к компонентам класса *CL*. Компонентная функция некоторого класса (*CLASS*) может быть объявлена дружественной функцией

другому классу (*CL*), если только определение этого первого класса размещено раньше, чем определение второго.

- может быть дружественной по отношению к нескольким классам.

Использование механизма дружественных функций позволяет упростить интерфейс между классами. Например, дружественная функция позволит получить доступ к собственным или защищенным компонентам сразу нескольких классов. Тем самым из классов можно иногда убрать компонентные функции, предназначенные только для доступа к этим "скрытым" компонентам.

Дружественность требует разрешения, т.е. чтобы класс *B* стал другом класса *A*, класс *A* должен объявить, что класс *B* - его друг. Таким образом дружественность не обладает ни свойством симметричности, ни свойством транзитивности, т.е. если класс *A* - друг класса *B*, а класс *B* - друг класса *C*, то отсюда не следует, что класс *B* — друг класса *A*, что класс *C* — друг класса *B*, или что класс *A* — друг класса *C*.

2.13. Статические поля класса

Мы узнали, что каждый объект класса содержит свои собственные данные, теперь мы должны углубить свое понимание данной концепции. Если поле данных класса описано с ключевым словом *static*, то значение этого ноля будет одинаковым для всех объектов данного класса. Статические данные класса полезны в тех случаях, когда необходимо, чтобы все объекты включали в себя какое-либо одинаковое значение. Эти поля можно рассматривать как глобальные переменные доступные только в пределах области класса. Статическое поле по своим характеристикам схоже со статической переменной: оно видимо только внутри класса, но время его жизни совпадает со временем жизни программы. Таким образом, статическое поле существует даже в том случае, когда не существует ни одного объекта класса.

Следующий пример, иллюстрирует простое применение статического поля класса:

```
class Rectangle
{
    private:
        float Width;
        float Height;
        static int count; // общее поле для всех объектов
    public:
        Rectangle (float w, float h) : Width(w), Height(h)
        {
           count++;
```

```
        }
        ~Rectangle ()
        {
           count--;
        }
        int GetCount()    // возвращает значение count
        {
            return count;
        }

}

int Rectangle::count = 0;       // Определение count

//-----------------------------------------------------------
int main()
{
    Rectangle Rl, R2, R3;  // создание трех объектов
                           // каждый объект видит
                           //одно и то же значение
    cout << "Число объектов : " << Rl.GetCount() << endl;
    cout << "Число объектов : " << R2.GetCount() << endl;
    cout << "Число объектов : " << R3.GetCount() << endl;
    return 0;
}
```

В этом примере класс `Rectangle` содержит единственное поле *count*, имеющее тип stat*ic int*. Конструктор класса инкрементирует значение поля *count*. В функции *main*() мы определяем три объекта класса `Rectangle`. Поскольку конструктор в этом случае вызывается трижды, инкрементирование поля *count* также происходит трижды. Метод *`GetCount`*() возвращает значение *count*. Мы вызываем этот метод для каждого из объектов, и во всех случаях он возвращает одну и ту же величину:

```
Число объектов : 3
Число объектов : 3
Число объектов : 3
```

Если бы мы использовали не статическое, а автоматическое поле *count*, то конструктор увеличивал бы на единицу значение этого поля для каждого объекта, и результат работы программы выглядел бы следующим образом:

```
Число объектов : 1
Число объектов : 1
Число объектов : 1
```

Определение статических полей класса происходит не так, как для обычных полей. Обычные поля объявляются (компилятору сообщается имя и

тип поля) и определяются (компилятор выделяет память для хранения поля) при помощи одного оператора. Для статических полей эти два действия выполняются двумя разными операторами: объявление поля находится внутри определения класса, а определение поля, как правило, располагается вне класса и зачастую представляет собой определение глобальной переменной.

Поместив определение статического поля вне класса, мы обеспечили однократное выделение памяти под это поле до того, как программа будет запушена на выполнение и статическое поле в этом случае станет доступным всему классу. Каждый объект класса уже не будет обладать своим собственным экземпляром поля, как это должно быть с полями автоматического типа.

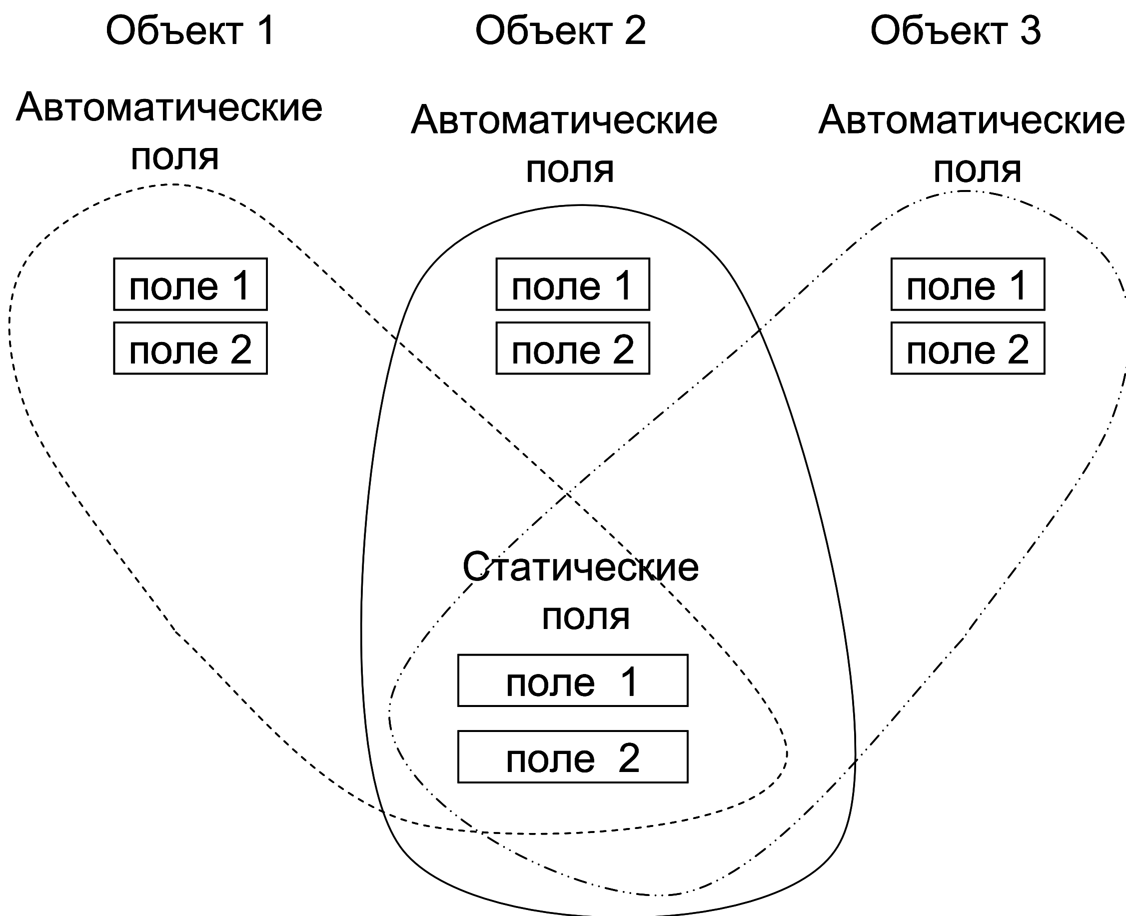


Рисунок 3 Статические и автоматические поля класса

Работая со статическими данными класса, легко совершить ошибки, которые компилятор будет не в силах распознать. Если вы объявите статическое поле класса, но забудете его определить, компилятор не выдаст предупреждающего сообщения. Ваша программа будет считаться корректной до тех пор, пока редактор связей не обнаружит ошибку и не выдаст сообщение о том, что вы пытаетесь обратиться к необъявленной глобальной переменной.

Основные моменты использования статических полей:

1) Память под статическое поле выделяется один раз при его инициализации независимо от числа созданных объектов (и даже при их отсутствии).

2) Статические поля доступны как через имя класса, так и через имя объекта:

Например:

```
Rectangle R;
cout << Rectangle::count;  // Будет выведено
cout << R.count;           // одно и то же
```

3) На статические поля распространяется действие спецификаторов доступа, поэтому статические поля, описанные как *private*, нельзя изменить с помощью операции доступа к области действия, как описано выше. Это можно сделать только с помощью статических методов (их мы рассмотрим дальше).

4) Память, занимаемая статическим полем, не учитывается при определении размера объекта с помощью операции *sizeof()*.

2.14. Статические методы

Статические методы предназначены для обращения к статическим полям класса. Они могут обращаться непосредственно только к статическим полям и вызывать только другие статические методы класса, потому что им не передается скрытый указатель *this*. Обращение к статическим методам производится так же, как к статическим полям — либо через имя класса, либо, если хотя бы один объект класса уже создан, через имя объекта.

```
class Rectangle
{
    private:
        . . .
        static int count;   // Поле count - скрытое
    public:
        . . .
        static void inc_count()
        {
            count++;
        }
        . . .
};
int Rectangle::count = 1;   // Определение в глобальной области

int main()
{
    Rectangle R;
```

```
    // R.count++ - нельзя, т.к. поле count скрытое
    // Изменение поля с помощью статического метода
    R.inc_count();
    cout << "Значение count : " << A::count << endl;
    R::inc_count();
    cout << "Значение count : " << A::count << endl;

    return 0;
}
```

Статические методы не могут быть константными (*const*) и виртуальными (*virtual*).

2.15. Константные методы

- В языке C++ методы объекта, не изменяющие его состояния (его данных) могут быть объявлены константными
  - Например, методы, возвращающие значения определенных полей данных
  - Изменить данные класса из константного метода нельзя
- Если объект был объявлен как константа, либо доступен по константной ссылке или указателю на const, то вызвать у него можно только константные методы
  - Это заставляет объявлять методы константными везде, где это только возможно

Пример:

```
class IntArray
{
public:
    …
    int GetSize()const
    {
        return m_numberOfItems;
    }
    void ClearElements()
    {
        delete [] m_pData;
        m_pData = NULL;
        m_numberOfItems = 0;
    }
private:
    int *m_pData;
    int m_numberOfItems;
};

void f(IntArray const& array)
{
```

```
    int i = array.GetSize();    // можно
    array.ClearElements();  // нельзя - неконстантные методы
                            //недоступны
}
```

3. Наследование и композиция

Начиная рассматривать вопросы наследования, нужно отметить, что обоснованно введенный в программу объект призван моделировать свойства и поведение некоторого фрагмента решаемой задачи, связывая в единое целое данные и методы, относящиеся к этому фрагменту. В терминах объектно-ориентированной методологии объекты взаимодействуют между собой и с другими частями программы с помощью сообщений. В каждом сообщении объекту передается некоторая информация. В ответ на сообщение объект выполняет некоторое действие, предусмотренное набором компонентных функций того класса, которому он принадлежит. Таким действием может быть изменение внутреннего состояния (изменение данных) объекта либо передача сообщения другому объекту.

Каждый объект является конкретным представителем класса. Объекты одного класса имеют разные имена, но одинаковые по типам и внутренним именам данные. Объектам одного класса для обработки своих данных доступны одинаковые компонентные функции класса и одинаковые операции, настроенные на работу с объектами класса. Таким образом, класс выступает в роли типа, позволяющего вводить нужное количество объектов, имена (названия) которых программист выбирает по своему усмотрению.

3.1. Композиция

Допустим, следующий класс почему-либо представляет ценность для нас:

```
// Класс, который должен использоваться многократно
#ifndef USEFUL H
#def1ne USEFUL_H
class X
{
    int i;
public:
    X() { 1 = 0;} // конструктор
    void set(int ii) { i = ii; }
    int read() const { return i; }
    int permute() { return i = i * 47; }
```

```
};
#endif // USEFUL_H //
```

Переменные класса объявлены закрытыми, поэтому мы можем абсолютно безопасно включить объект типа *X* в новый класс как открытый (*public*) объект, что значительно упрощает интерфейс класса:

```
// Многократное использование кода посредством композиции
#include "Useful.h"
class Y {
    int i; public:
    X x; // Внутренний объект
    Y() { i = 0: }
    void f(int ii) { i = ii; }
    int g() const { return i;}
};
int main()
{
    Y y;
    y.f(47);
    y.x.set(37): // Обращение к внутреннему объекту
} ///:-
```

Обращение к функциям внутреннего объекта (или подобъекта) просто требует дополнительного уточнения имени объекта. На практике внутренние объекты чаще объявляются закрытыми.

3.2. Базовые классы и производные классы

Объекты разных классов и сами классы могут находиться в отношении наследования, при котором формируется иерархия объектов, соответствующая заранее предусмотренной иерархии классов.

Иерархия классов позволяет определять новые классы на основе уже имеющихся. Имеющиеся классы обычно называют базовыми (иногда порождающими, родительскими), а новые классы, формируемые на основе базовых, - производными (порожденными), иногда классами-потомками, наследниками или дочерними классами. Производные классы "получают наследство" - данные и методы своих базовых классов - и, кроме того, могут пополняться собственными компонентами (данными и собственными методами). Наследуемые компоненты не перемещаются в производный класс, а остаются в базовых классах. Сообщение, обработку которого не могут выполнить методы производного класса, автоматически передается в базовый класс. Если для обработки сообщения нужны данные, отсутствующие в производном классе, то их пытаются отыскать автоматически и незаметно для программиста в базовом классе

Важнейшим свойством объектно-ориентированного программирования является наследование. Для того, чтобы показать, что класс *B* наследует класс *A* (класс *B* выведен из класса *A*), в определении класса *B* после имени класса ставится двоеточие и затем перечисляются классы, из которых *B* наследует:

```
class A
{
public:
      A();
      ~A();
      MethodA();
};

class B : public A
{
public:
      B();
      . . .
};
```

Термин "наследование" означает, что класс *B* обладает всеми свойствами класса *A*, он их унаследовал. У объекта производного класса есть все атрибуты и методы базового класса. Разумеется, новый класс может добавить собственные атрибуты и методы.

```
B b;
b.MethodA();  // вызов метода базового класса
```

Часто выведенный класс называют подклассом, а базовый класс – суперклассом. Из одного базового класса можно вывести сколько угодно подклассов. В свою очередь, производный класс может служить базовым для других классов. Изображая отношения наследования, их часто рисуют в виде иерархии или дерева.

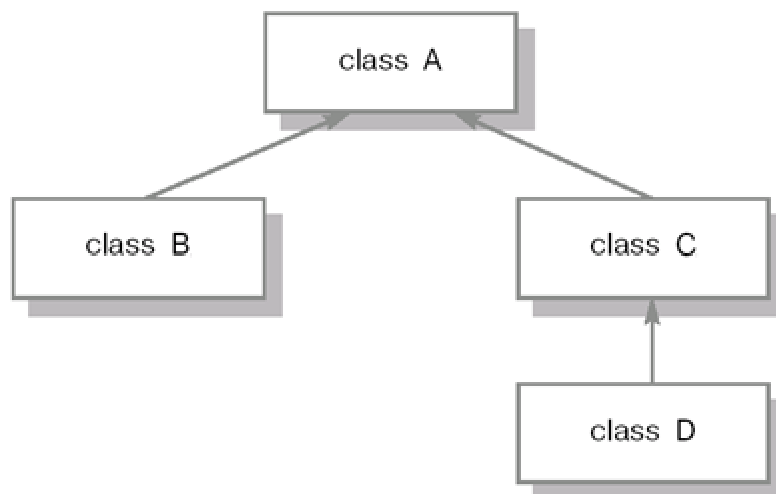


Рисунок 4 Пример иерархии классов.

Иерархия классов может быть сколь угодно глубокой. Если нужно различить, о каком именно классе идет речь, класс *C* называют непосредственным или прямым базовым классом класса *D*, а класс *A* – косвенным базовым классом класса *D*.

Производный класс имеет доступ к методам и атрибутам базового класса, объявленным во внешней и защищенной части базового класса, однако доступ к внутренней части базового класса не разрешен.

Механизм наследования классов позволяет строить иерархические деревья классов, Иерархия классов позволяет определять новые классы на основе уже имеющихся. Имеющиеся классы обычно называют базовыми (иногда порождающими), а новые классы, формируемые на основе базовых, - производными (порожденными), иногда классами-потомками или наследниками. Производные классы "получают наследство" – данные и методы базовых классов – и, кроме того могут пополняться собственными компонентами (данными и собственными методами). Наследуемые компоненты не перемещаются в производный класс, а остаются в базовых классах. Сообщение, обработку которого не могут выполнить методы производного класса, автоматически передается в базовый класс. Если для обработки сообщения нужны данные, отсутствующие в производном классе, то их пытаются отыскать автоматически и незаметно для программиста в базовом классе.

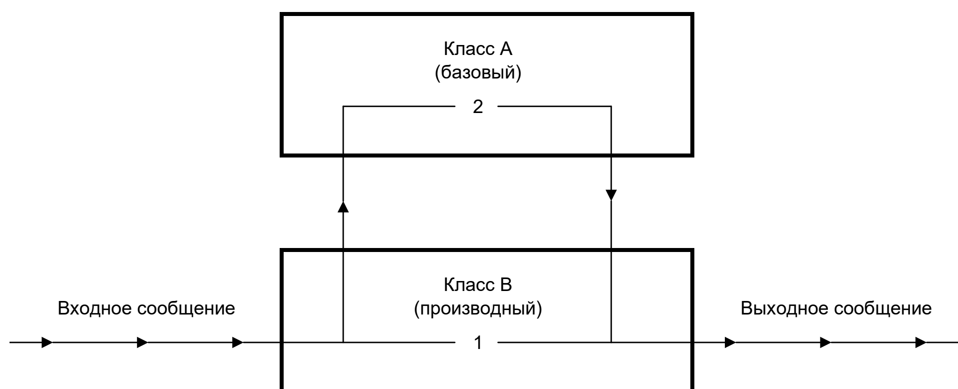


Рисунок 5 Схема обработки сообщений в иерархии объектов:
1 - обработка сообщения методами производного класса;
2 - обработка сообщения методами базового класса.

При большом количестве никак не связанных классов управлять ими становится крайне сложно. Наследование позволяет справиться с этой проблемой путем упорядочивания и ранжирования классов, то есть объединения общих для нескольких классов свойств в одном классе и использования его в качестве базового.

Классы, находящиеся ближе к началу иерархии, объединяют в себе наиболее общие черты для всех нижележащих классов. По мере продвижения вниз по иерархии классы приобретают все больше конкретных черт. Множественное наследование позволяет одному классу обладать свойствами двух и более родительских классов.

При наследовании некоторые имена методов (компонентных функций) и (или) компонентных данных базового класса могут быть по-новому определены в производном классе. В этом случае соответствующие компоненты базового класса становятся недоступными из производного класса. Для доступа из производного класса к компонентам базового класса, имена которых повторно определены в производном, используется операция «::» указания (уточнения) области видимости.

В дереве классов любой производный класс может, в свою очередь, становиться базовым для других классов, и таким образом формируется направленный граф иерархии классов и объектов. В иерархии производный объект наследует разрешенные для наследования компоненты всех базовых объектов. Другими словами, у объекта имеется возможность доступа к данным и методам всех своих базовых классов.

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

Любой производный класс может, в свою очередь, становиться базовым для других классов, и таким образом формируется направленный граф иерархии классов и объектов. В иерархии производный объект наследует разрешенные для наследования компоненты всех базовых объектов. Другими словами, у объекта имеется возможность доступа к данным и методам всех своих базовых классов.

Наследование в иерархии классов может отображаться и в виде дерева, и в виде более общего *направленного ациклического графа.* Допускается *множественное наследование* - возможность для некоторого класса наследовать компоненты нескольких никак не связанных между собой базовых классов.

3.3. Управление доступом к элементам базовых классов

При наследовании классов важную роль играет *статус доступа* (статус внешней видимости) компонентов. Для любого класса все его компоненты лежат в области его действия. Тем самым любая принадлежащая классу функция может использовать любые компонентные данные и вызывать любые принадлежащие классу функции. Вне класса в общем случае доступны только те его компоненты, которые имеют статус ***public***.

В иерархии классов соглашение относительно доступности компонентов класса следующее.

Собственные (***private***) методы и данные доступны только внутри того класса, где они определены.

Защищенные (***protected***) компоненты доступны внутри класса, в котором они определены, и дополнительно доступны во всех производных классах.

Общедоступные (***public***) компоненты класса видимы из любой точки программы, т.е. являются глобальными.

Если считать, что объекты, т.е. конкретные представители классов, обмениваются сообщениями и обрабатывают их, используя методы и данные классов, то при обработке сообщения используются, во-первых, общедоступные члены всех классов программы; во-вторых, защищенные компоненты базовых и рассматриваемого классов и, наконец, собственные компоненты рассматриваемого класса. Собственные компоненты базовых и производных классов, а также защищенные компоненты производных классов недоступны для сообщения и не могут участвовать в его обработке.

Еще раз отметим, что на доступность компонентов класса влияет не только явное использование спецификаторов доступа (служебных слов) - ***private*** (собственный), ***protected*** (защищенный), ***public*** (общедоступный), но и выбор ключевого слова ***class***, ***struct***, ***union***, с помощью которого объявлен класс.

При описании класса в его заголовке перечисляются все классы, являющиеся для него базовыми.

class имя : [private | protected | public] базовый_класс { тело класса };

Если базовых классов несколько, они перечисляются через запятую. Ключ доступа может стоять перед каждым классом.

Пример:

```
class  A { … };
class  B { … };
class  C { … };
class  D: A. protected B. public C { ... }:
```

По умолчанию для классов используется ключ доступа ***private***, (для структур - ***public***).

До сих пор мы рассматривали только применяемые к элементам класса спецификаторы доступа ***private*** и ***public***. Для любого элемента класса может также использоваться спецификатор ***protected***, который для одиночных классов, не входящих в иерархию, равносилен ***private***. Разница между ними проявляется при наследовании, что можно видеть из приведенных таблиц.

Спецификатор доступа к элементам в базовом классе	Тип наследования		
	public открытое наследование	protected защищенное наследование	private закрытое наследование
public	public в производном классе Может быть доступен непосредственно любым нестатическим функциям элементам, дружественным функциям и функциям, не являющимся элементами.	protected в производном классе Может быть доступен непосредственно любым нестатическим функциям - элементам и дружественным функциям.	private в производном классе Может быть доступен непосредственно любым нестатическим функция м - элемента м и дружественным функциям.
[protected	protected в производном классе Может быть доступен непосредственно любым нестатическим функциям-элементам и дружественным функциям.	protected в производном классе Может быть доступен непосредственно любым нестатическим функциям - элементам и дружественным функциям.	private в производном классе Может быть доступен непосредственно любым нестатическим функциям - элементам и дружественным функциям.
private	невидим в производном классе Может быть доступен нестатическим функциям-элементам и дружественным функциям через открытые или защищенные функции-элементы базового класса.	невидим в производном классе Может быть доступен нестатическим функциям-элементам и дружественным функциям через открытые или защищенные функции-элементы базового класса.	невидим в производном классе Может быть доступен нестатическим функциям-элементам и дружественным функциям через открытые или защищенные функции - элементы базового класса.

Доступ в базовом классе	Спецификатор доступа перед базовым классом	Доступ в производном классе	
		struct	class
public	отсутствует	**public**	**private**
protected	отсутствует	**public**	**private**
private	отсутствует	недоступны	недоступны
public	**public**	**public**	**public**
protected	**public**	**protected**	**protected**
private	**public**	недоступны	недоступны
public	**protected**	**protected**	**protected**
protected	**protected**	**protected**	**protected**
private	**protected**	недоступны	недоступны
public	**private**	**private**	**private**
protected	**private**	**private**	**private**
private	**private**	недоступны	недоступны

Объект производного класса с открытым наследованием можно обрабатывать как объект соответствующего базового класса. Это позволяет выполнять некоторые интересные манипуляции. Например, несмотря на то, что объекты целого ряда классов, полученных из некоторого базового класса, могут совершенно отличаться друг от друга, мы все же можем создать из них связанный список, — если мы будем рассматривать их как объекты базового класса. Однако обратное неверно: объекты базового класса не являются автоматически объектами производного.

Назначение защищенной (***protected***) части класса в том и состоит, чтобы, закрыв доступ "извне" к определенным атрибутам и методам, разрешить пользоваться ими производным классам.

Если одно и то же имя атрибута или метода встречается как в базовом классе, так и в производном, то производный класс перекрывает базовый.

3.4. Переопределение элементов базового класса в производном классе

Производный класс может переопределить функцию-элемент базового класса. При описании в производном классе функции с тем же именем, версия функции производного класса переопределяет версию базового класса. Чтобы сделать доступной для производного класса версию функции базового класса, нужно использовать операцию разрешения области действия.

Две типичные ошибки при использовании переопределения:

1. Рассмотрение объектов базового класса как объектов производного класса может вызвать ошибки.
2. Явное приведение типа указателя базового класса, который указывает на объект базового класса, к типу указателя производного класса и затем ссылка на элементы производного класса, которые не существуют в этом объекте.

При переопределении в производном классе функции-элемента базового класса принято вызывать версию базового класса и после этого выполнять некоторые дополнительные операции. При этом ссылка на функцию-элемент базового класса без использования операции разрешения области действия может вызвать бесконечную рекурсию, потому что функция-элемент производного класса будет в действительности вызывать сама себя.

При переопределении в производном классе функции-элемента базового класса нет необходимости получать такую же сигнатуру, как у функции-элемента базового класса.

3.5. Использование конструкторов и деструкторов в производных классах

Конструкторы не наследуются, поэтому производный класс должен иметь собственные конструкторы. Порядок вызова конструкторов определяется приведенными ниже правилами.

*Если в конструкторе производного класса явный вызов конструктора базового класса отсутствует, автоматически вызывается конструктор базового класса по умолчанию (то есть тот, который можно вызвать без параметров)

*Для иерархии, состоящей из нескольких уровней, конструкторы базовых классов вызываются начиная с самого верхнего уровня. После этого выполняются конструкторы тех элементов класса, которые являются объектами, в порядке их объявления в классе, а затем исполняется конструктор класса.

*В случае нескольких базовых классов их конструкторы вызываются в порядке объявления.

Ниже перечислены правила наследования деструкторов.

*Деструкторы не наследуются и если программист не описал в производном классе деструктор, он формируется по умолчанию и вызывает деструкторы всех базовых классов.

* В отличие от конструкторов, при написании деструктора производного класса в нем не требуется явно вызывать деструкторы базовых классов, поскольку это будет сделано автоматически.

*Для иерархии классов, состоящей из нескольких уровней, деструкторы вызываются в порядке, строго обратном вызову конструкторов: сначала вызывается деструктор класса, затем — деструкторы элементов класса, а потом деструктор базового класса.

Стоит особо подчеркнуть, что конструкторы и деструкторы принципиально отличаются от других функций: они вызываются все от начала и до конца иерархии, тогда как при вызове обычной функции вызывается только эта функция, но не ее версии из базовых классов. Если необходимо вызвать базовую версию обычной переопределенной функции, это придется сделать явно.

3.6. Функции, которые не наследуются автоматически

Нельзя наследовать конструкторы, деструкторы, отношение дружественности и некоторые неперегружаемые операции.

3.7. Множественное наследование

Класс называют непосредственным (прямым) базовым классом (прямой базой), если он входит в список базовых при определении класса. В то же время для производного класса могут существовать косвенные или непрямые предшественники, которые служат базовыми для классов, входящих в список базовых. Если некоторый класс *A* является базовым для *B* и *B* есть база для *C*, то класс *B* является непосредственным базовым классом для *C*, а класс *A* - непрямой базовый класс для *C*..

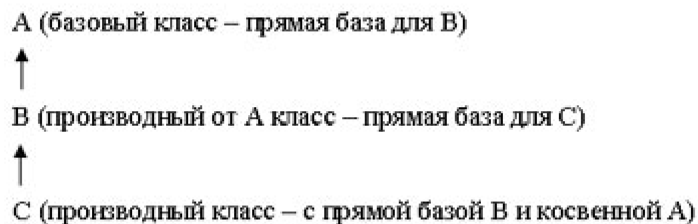


Рис. Прямое и косвенное наследование классов

Обращение к компоненту *XA*, входящему в *A* и унаследованному последовательно классами *B* и *C*, можно обозначить в классе C либо как *A*::*XA*, либо как *B*::*XA*. Обе конструкции обеспечивают обращение к элементу *XA* класса *A*.

Иерархию производных классов удобно представлять с помощью направленного ациклического графа (НАГ), где стрелкой изображают отношение "производный от". Производные классы принято изображать ниже базовых. Именно в таком порядке их объявления рассматривает компилятор и их тексты размещаются в листинге программы. Класс может иметь несколько непосредственных базовых классов, т.е. может быть порожден из любого числа базовых классов.

Наличие нескольких прямых базовых классов называют множественным наследованием.

Определения базовых классов должны предшествовать их использованию в качестве базовых.

При множественном наследовании никакой класс не может больше одного раза использоваться в качестве непосредственного базового. Однако класс может больше одного раза быть непрямым базовым классом.

```
class X { ...; f(); ... };
class Y: public X { ... };
class Z: public X { ... };
class D: public Y, public Z { ... };
```

В данном примере класс *X* дважды опосредованно наследуется классом *D*. Особенно хорошо это видно в направленном ациклическом графе (НАГ):

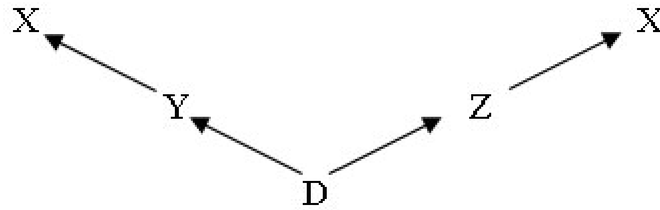


Рисунок 6 Двойное наследование класса *X*

Проиллюстрированное дублирование класса соответствует включению в производный объект нескольких объектов базового класса. В нашем примере существуют два объекта класса *X*, и поэтому для устранения возможных неоднозначностей вне объектов класса *D* нужно обращаться к конкретному компоненту класса *X*, используя полную квалификацию: *D*::*Y*::*X*::*f*() или *D*::*Z*::*X*::*f*(). Внутри объекта класса *D* обращения упрощаются: *Y*::*X*::*f*() или Z::*X*::*f*(), но тоже содержат квалификацию.

Чтобы устранить дублирование объектов непрямого базового класса при множественном наследовании, этот базовый класс объявляют виртуальным. Для этого в списке базовых классов перед именем класса необходимо поместить ключевое слово *virtual*. Например, класс *X* будет виртуальным базовым классом при таком описании:

```
class X { ... f(); ... };
class Y: virtual public X { ... };
class Z: virtual public X { ... );
class D: public Y, public Z { ... );
```

Теперь класс D будет включать только один экземпляр X, доступ х которому равноправно имеют классы Y и Z. Графически это очень наглядно:

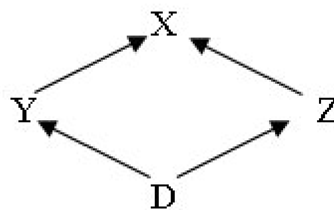


Рисунок 7 Исключение дублирования базового класса

Размеры производных классов при отсутствии виртуальных базовых равны сумме длин их компонентов и длин унаследованных базовых классов. "Накладные расходы" памяти здесь отсутствуют.

При множественном наследовании один и тот же базовый класс может быть включен в производный класс одновременно несколько раз, причем и как виртуальный, и как невиртуальный. Для иллюстрации этого положения изобразим направленный граф, а затем приведем структуру соответствующей ему иерархии классов:

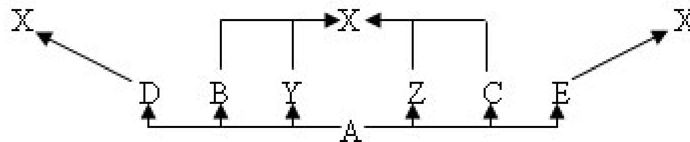


Рисунок 8 Структура иерархии классов при множественном наследовании

Пример включения базового класса как виртуального и невиртуального

```
class X  {...};
class Y: virtual public X { ...};
class Z: virtual public X { ...};
class B: virtual public X { ...};
class C: virtual public X { ...};
class E: public X { ...};
class D: public X { ...};
class A: public D, public B, public Y, public Z, public
C, public E {...};
```

В данном примере объект класса *A* включает три экземпляра объектов класса *X*: один виртуальный, совместно используемый классами *B*, *Y*, *C*, *Z*, и два невиртуальных относящихся соответственно к классам *D* и *E*. Таким образом, можно констатировать, что виртуальность класса в иерархии производных классов является не свойством класса как такового, а результатом особенностей процедуры наследования.

Возможны и другие комбинации виртуальных и невиртуальных базовых классов. Например:

```
 class BB  {   ...   };
 class AA: virtual public BB { ... };
 class CC: virtual public BB { ... };
 class DD: public AA, public CC, virtual public BB {
... };
```

Соответствующий НАГ имеет вид:

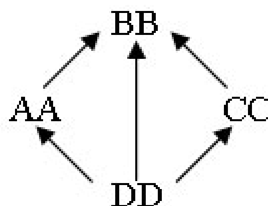


Рисунок 9 НАГ комбинации виртуальных и невиртуальных классов

При использовании наследования и множественного наследования могут возникать неоднозначности при доступе к одноименным компонентам

разных базовых классов. Простейший и самый надежный способ устранения неоднозначностей - использование квалифицированных имен компонентов. Как обычно, для квалификации имени компонента используется имя класса.

4. Виртуальные функции и полиморфизм

Третьим важнейшим аспектом объектно-ориентированных языков программирования (после абстрактного представления данных и наследования) является полиморфизм, реализованный в C++ при помощи виртуальных функций.

Программисты С обычно осваивают язык С++ в три этапа. На первом этапе они рассматривают его как «улучшенный язык С» — С++ требует объявлять все функции перед использованием и гораздо строже следит за применением переменных. Довольно часто ошибки в программах С обнаруживаются путем простой компиляции программ на С++.

На втором этапе они имеют дело с «объектно-базированным» языком С++. Речь идет об очевидных преимуществах: организации кода, при которой структуры данных группируются с функциями, работающими с этими данными; удобстве конструкторов и деструкторов, а также простейших применениях наследования.

Большинство программистов с опытом работы на С быстро осознают пользу объектно-базированного подхода, поскольку именно эти задачи обычно приходится решать при разработке библиотек. В С++ компилятор помогает в их решении.

Однако многие «застревают» на объектно-базированном уровне. До этого уровня легко добраться, причем программист обретает массу полезных инструментов без особых умственных усилий. Он создает классы и объекты, организует передачу сообщений объектам, и на первый взгляд все идет просто замечательно.

Но не стоит заблуждаться. Если остановиться на этом этапе, вы упустите самое главное в языке — «настоящее» объектно-ориентированное программирование. Переход к этому этапу можно сделать только при помощи виртуальных функций.

Виртуальные функции выводят концепцию типа за пределы простой инкапсуляции кода в структурах с ограничением доступа. Несомненно, именно эту концепцию будет труднее всего усвоить новичкам С++. С другой стороны, она становится решающим моментом в понимании сути объектно-ориентированного программирования. Если вы не используете виртуальные функции, значит, вы еще недоросли до этого.

Так как виртуальные функции тесно связаны с концепцией типа, а тип занимает центральное место в объектно-ориентированном программировании, виртуальные функции не имеют аналогов в традиционных процедурных языках. Возможности процедурных языков

можно понять на алгоритмическом уровне, но виртуальные функции представимы только с точки зрения архитектуры.

4.1. Виртуальные функции

К механизму виртуальных функций обращаются в тех случаях, когда в базовый класс необходимо поместить функцию, которая должна по-разному выполняться в производных классах. Точнее, по-разному должна выполняться не единственная функция из базового класса, а в каждом производном классе требуется свой вариант этой функции.

До объяснения возможностей виртуальных функций отметим, что классы, включающие такие функции, играют особую роль в объектно-ориентированном программировании. Именно поэтому они носят специальное название - полиморфные.

Большую гибкость (особенно при использовании уже готовых библиотек классов) обеспечивает позднее (отложенное), или динамическое связывание, которое предоставляется механизмом виртуальных функций. Любая нестатическая функция базового класса может быть сделана виртуальной, если в ее объявлении использовать спецификатор *virtual*.

Интерпретация каждого вызова виртуальной функции через указатель на базовый класс зависит от значения этого указателя, т.е. от типа объекта, для которого выполняется вызов. В противоположность этому интерпретация вызова через указатель невиртуальной функции зависит только от типа указателя.

Виртуальными могут быть не любые функции, а только нестатические компонентные функции какого-либо класса. После того как функция определена как виртуальная, ее повторное определение в производном классе (с тем же самым прототипом) создает в этом классе новую виртуальную функцию, причем спецификатор *virtual* может не использоваться.

В производном классе нельзя определять функцию с тем же именем и с той же сигнатурой параметров, но с другим типом возвращаемого значения, чем у виртуальной функции базового класса. Это приводит к ошибке на этапе компиляции.

Если в производном классе ввести функцию с тем же именем и типом возвращаемого значения, что и виртуальная функция базового класса, но с другой сигнатурой параметров, то эта функция производного класса не будет виртуальной. В этом случае с помощью указателя на базовый класс при любом значении этого указателя выполняется обращение к функции базового класса (несмотря на спецификатор *virtual* и присутствие в производном классе похожей функции).

При подмене виртуальной функции требуется полное совпадение сигнатур, имен и типов возвращаемых значений функций в базовом и производном классах.

Виртуальной не может быть глобальная функция. Функция, подменяющая виртуальную, в производном классе может быть описана как со спецификатором *virtual*, так и без него. В обоих случаях она будет виртуальной, т.е. ее вызов возможен только для конкретного объекта. Виртуальная функция может быть объявлена дружественной (*friend*) в другом классе.

Механизм виртуального вызова может быть подавлен с помощью явного использования полного квалифицированного имени. Таким образом, при необходимости вызова из производного класса виртуального метода (компонентной функции) базового класса употребляется полное имя.

4.2. Позднее связывание

Связыванием обычно называют привязку тела функции к месту ее вызова. В процедурных языках задача связывания решается на этапе компиляции (если определение функции находится в том же модуле, что и обращение к ней) или, самое позднее, на этапе компоновки (если определение функции находится в другом модуле). Поэтому этап компоновки часто называют редактированием связей. Соответственно, на момент исполнения программы каждый вызов функции в точности "привязан" к ее определению. Собственно, это и есть единственная модель связывания, используемая в процедурных программах. Такое связывание называют ранним, или статическим, связыванием.

Иначе обстоит дело в связи с управлением объектами. Переменная-указатель на объект базового типа может в действительности быть связана с объектом производного типа, что делает в общем случае невозможным установление действительного экземпляра метода, который нужно вызвать, в момент трансляции или компоновки программы. Рассмотрим эту проблему более детально.

Для начала рассмотрим следующий пример с функцией `Callprint()`:

```
void Callprint( Student *stud )
{
    stud->Print();
}
```

Во время компиляции невозможно установить, на объект какого типа будет указывать указатель stud во время выполнения: на объект типа student или на объект производного типа (например, *GraduateStudent*). Единственное, что гарантируется — так это то, что указатель, передаваемый функции `Callprint` в качестве аргумента, может быть безопасно и корректно преобразован к указателю на *student*. Таким образом, обработка объектов

типа *GraduateStudent* возможна, но только в рамках той функциональности, которую обеспечивает класс *student.*

Модель времени компиляции в этом случае является моделью времени выполнения. Как было сказано выше, такой вариант связывания называют ранним (или статическим) связыванием, т. е. связыванием, оперирующим типами, известными во время компиляции. Такие типы иногда называют статическими — отсюда и название модели связывания кода и данных. Языки, не являющиеся объектно-ориентированными, обычно предоставляют только такой механизм связывания.

Подведем промежуточный итог. Даже если в действительности указатель *stud* связан с объектом типа *Graduatestudent,* в соответствии с механизмом статического связывания вызываться всегда будет версия функции *Print()*, объявленная в классе *student*.

Основная проблема статического связывания заключается в том, что функция, аргументом которой является указатель или ссылка на объект базового класса, способна принять в качестве аргумента указатель или ссылку на объект производного класса, но это не позволяет обратиться к членам произвольного класса. Выясняется, что модель связывания, при которой все решения относительно типов объектов принимаются на стадии компиляции, недостаточна для реализации идей, заложенных в концепцию наследования.

Решение проблемы заключается в так называемом динамическом, или позднем, связывании, когда определение конкретного экземпляра функции-члена производного класса может происходить не во время компиляции, а во время выполнения. Динамическое связывание часто называют среди трех основных понятий объектно-ориентированного программирования наряду с абстрагированием и инкапсуляцией.

Если язык поддерживает позднее связывание, он должен предоставлять некоторый механизм, позволяющий определить тип объекта во время выполнения и вызвать подходящий метод в зависимости от типа объекта. В языке C++ подобный механизм основывается на использовании виртуальных функций. Объекты, определения классов которых допускают динамическое связывание, называют объектами, допускающими полиморфное поведение, или полиморфными объектами.

4.3. Механизм вызова виртуальных функций

Стандартное решение для обеспечения динамического связывания, используемое разработчиками компиляторов, заключается в следующем. Для каждого класса, допускающего полиморфное поведение своих объектов (т. е. содержащего хотя бы одну виртуальную функцию), создается специальная таблица, содержащая адреса виртуальных функций, объявленных в этом классе.

В определение класса добавляется скрытый член класса — указатель на таблицу виртуальных функций *Vtable*. Таким образом, размер памяти, занимаемой полиморфным объектом, увеличится ровно на один указатель по отношению к аналогичному объекту класса, не содержащего виртуальные функции.

Адреса одноименных виртуальных функций разных классов, находящихся в отношении наследования, помещаются в ячейки таблицы *VTable* с одинаковыми индексами.

Если какая-либо виртуальная функция не перезаписывается в производном классе (это означает, что для объектов этого класса должна использоваться версия базового класса), в таблицу виртуальных функций помещается адрес соответствующей функции базового класса.

В случае одиночного наследования накладные расходы, связанные с реализацией полиморфизма, составляют один указатель *vptr* для каждого объекта класса, содержащего виртуальные функции, и одна таблица виртуальных функций для каждого такого класса.

Рассмотрим пример генерации кода для вызова виртуальной функции *Erase*().

Имея в качестве *s*[i] указатель на объект типа *shape* (потому что все элементы массива являются указателями на *shape*), компилятор обращается к скрытому элементу *vptr*, который существует у любого класса, содержащего виртуальные функции (обычно этот элемент размещается в начале области памяти, занимаемой объектом). Указатель *vptr* содержит адрес начала таблицы виртуальных функций *Vtable*. Далее имя вызываемой функции преобразуется в индекс таблицы. Поскольку адреса виртуальных функций базового и производного класса хранятся в таблице в одинаковом порядке, то элемент таблицы с индексом 0 всегда содержит адрес функции *Draw()*, элемент таблицы с индексом 1 всегда содержит адрес функции *Erase*() и т. д. Таким образом, независимо от того, какого типа будет в действительности объект, связанный с указателем *s*[i], для вызова функции *Erase* то нужно извлечь ее адрес из элемента с индексом 1 таблицы *Vtable*, т. е. адрес вызываемой функции извлекается из ячейки памяти с адресом *vptr*+l.

Процесс вычисления адреса функции, таким образом, осуществляется во время выполнения программы — это и называется поздним связыванием.

Для того чтобы механизм, описанный в предыдущем разделе, работал корректно, нужно обеспечить инициализацию указателя *vptr* таким образом, чтобы он действительно указывал на корректную версию *VTable*. Как известно, задачу инициализации объекта класса решает конструктор. Когда вызывается конструктор, тип объекта известен, поэтому имеется возможность ориентировать указатель *vptr* на соответствующую типу объекта таблицу виртуальных функций.

Однако ни в одном классе из иерархии классов для представления геометрических фигур нет явного конструктора. Напомним, что если

конструктор по умолчанию отсутствует, и не объявлено ни одного явного конструктора, конструктор по умолчанию генерируется автоматически.

Дополнительно, следует отметить, что при вызове виртуальной функции из конструктора всегда будет вызываться локальная версия функции (т. е. версия, объявленная в данном классе). Это означает, что механизм полиморфных вызовов не работает в пределах конструктора. Это связано с тем, что полиморфный вызов обычно предполагает вызов версии функции, объявленной в производном классе, инициализация которого, возможно, еще не состоялась, а значит — и не инициализирована его таблица виртуальных функций. Кроме того, окончательное значение указатель *vprt* принимает после того, как закончит работу конструктор самого нижнего класса в иерархии классов. Это означает, что нет возможности полагаться на текущее значение указателя *vptr*.

4.4. Абстрактные классы и конкретные классы

Абстрактным классом называется класс, в котором есть хотя бы одна чистая (пустая) виртуальная функция. Чистой виртуальной называется компонентная функция, которая имеет следующее определение:

virtual тип **имя_функции** (**список_формальных_параметров**) = 0;

В этой записи конструкция "= 0" называется "чистый спецификатор". Пример описания чистой виртуальной функции:

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

Чистая виртуальная функция "ничего не делает" и недоступна для вызовов. Ее назначение - служить основой для подменяющих ее функций в производных классах. Исходя из этого становится понятной невозможность создания самостоятельных объектов абстрактного класса. Абстрактный класс может использоваться только в качестве базового для производных классов. При создании объектов такого производного класса в качестве подобъектов создаются объекты базового абстрактного класса.

Как всякий класс, абстрактный класс может иметь явно определенный конструктор. Из конструктора возможен вызов методов класса, но любые прямые или опосредованные обращения из конструктора к чистым виртуальным функциям приведут к ошибкам во время выполнения программы.

Механизм абстрактных классов разработан для представления общих понятий, которые в дальнейшем предполагается конкретизировать. Эти общие понятия обычно невозможно использовать непосредственно, но на их

основе можно, как на базе, построить частные производные классы, пригодные для описания конкретных объектов.

По сравнению с обычными классами абстрактные классы пользуются "ограниченными правами". Перечислим эти ограничения:

1. Невозможно создать объект абстрактного класса.
2. Абстрактный класс нельзя употреблять для задания типа параметра функции или в качестве типа возвращаемого функцией значения.
3. Абстрактный класс нельзя использовать при явном приведении типов. В то же время можно определять указатели и ссылки на абстрактные классы.
4. Объект абстрактного класса **не может быть формальным параметром функции, однако формальным параметром может быть указатель абстрактного класса**. В этом случае появляется возможность передавать в вызываемую функцию в качестве фактического параметра значение указателя на производный объект, заменяя им указатель на абстрактный базовый класс.

4.5. Виртуальные функции и деструкторы

При использовании полиморфизма для обработки динамически размещенных объектов иерархии классов может появиться одна проблема. Если объект уничтожается явным использованием операции *delete* над указателем базового класса на объект, то вызывается деструктор базового класса данного объекта. Это происходит вне зависимости от типа объекта, на который указывает указатель базового класса и вне зависимости от того факта, что деструкторы каждого класса имеют разные имена.

Существует простое решение этой проблемы: объявление деструктора базового класса виртуальным. Это автоматически приведет к тому, что все деструкторы производных классов станут виртуальными, даже если они имеют имена, отличные от имени деструктора базового класса. В этом случае, если объект в иерархии уничтожен явным использованием операции *delete*, примененной к указателю базового класса на объект производного класса, то будет вызван деструктор соответствующего класса. Вспомним, что когда производный класс уничтожен, часть базового класса, содержащаяся в производном классе, также уничтожается. Деструктор базового класса автоматически выполняется после деструктора производного класса.

Если у класса имеются виртуальные функции, предусматривайте создание виртуального деструктора, даже если он не требуется этому классу. Классы, производные от данного класса, могут содержать деструкторы, которые должны вызываться соответствующим образом.

Если производному классу не удается переопределить виртуальную функцию, класс использует версию функции базового класса. Если производный класс — часть длинной цепочки получения производных

классов, он будет использовать ту версию виртуальной функции, которая определена последней по времени. Исключением является ситуация, когда базовые версии скрыты, как описано далее.

В целом по виртуальным деструкторам можно сделать следующие выводы:

Следует использовать виртуальный деструктор в базовых классах и в классах, от которых возможно наследование в будущем, например, в классах с виртуальными методами

Не следует использовать виртуальные деструкторы в классах, от которых не планируется создавать производные классы в будущем

Также возможно в базовом классе объявить защищенный невиртуальный деструктор. Объекты данного класса удалить напрямую невозможно – только через указатель на класс-наследник

Данный деструктор будет доступен классам-наследникам

4.6. Отсутствие переопределения

Если производному классу не удается переопределить виртуальную функцию, класс использует версию функции базового класса. Если производный класс — часть длинной цепочки получения производных классов, он будет использовать ту версию виртуальной функции, которая определена последней по времени. Исключением является ситуация, когда базовые версии скрыты, как описано далее.

Переопределение скрывает методы

Предположим, что вы создаете следующую конструкцию:

```
class Dwelling
{
public:
virtual void showperks(int a) const;
• • •
};
class   Hovel   :   public  Dwelling
{
public:
void showperks();
};
```

Возможно, предупреждение и не будет получено. В любом случае код имеет следующие толкования:

```
Hovel trump ;
trump.showperks();                  //   допустимо
```

```
trump.showperks(5);              //    недопустимо
```

Новое объявление определяет функцию *showperks*(), которая не принимает никаких аргументов. Вместо того чтобы создавать две перегруженные версии функции, это переопределение скрывает версию базового класса, которая принимает аргумент *int*. Одним словом, переопределение унаследованных методов не является разновидностью перегрузки. В случае переопределения функции в производном классе она не просто замещает объявление базового класса сигнатурой той же самой функции. Вместо этого переопределение скрывает все методы базового класса с тем же самым именем, независимо от сигнатур аргумента.

Из этого вытекает несколько важных правил. Во-первых, если вы переопределяете унаследованный метод, убедитесь в точном соответствии первоначальному прототипу. Единственное исключение состоит в том, что возвращаемый тип, являющийся ссылкой или указателем на базовый класс, может быть заменен ссылкой или указателем на производный класс. Во-вторых, если объявление базового класса перегружается, переопределите все версии базового класса в производном классе:

```
class Dwelling
{
public:
// три перегруженные функции   showperks()
virtual void showperks(int a) const;
virtual void showperks(double x)  const;
virtual void showperks() const;
};

class Hovel : public  Dwelling
{
public:
// три переопределенные функции showperks()
void showperks (int a) const;
void showperks(double x)  const;
void showperks() const;
...
};
```

Если переопределить только одну версию, две другие становятся скрытыми и не могут использоваться объектами производного класса. Обратите внимание на то, что если никакое изменение не требуется, в результате переопределения может просто вызываться версия базового класса.

4.7. Заключение

В заключение перечислим основные свойства и правила использования виртуальных функций:

♦ виртуальный механизм поддерживает полиморфизм на этапе выполнения программы. Это означает, что нужная версия функции выбирается на этапе выполнения (а не компиляции). Класс, содержащий хотя бы одну виртуальную функцию, называется полиморфным;

♦ виртуальные функции позволяют строить различные версии функции для базового и производных классов;

♦ если базовый класс *B* содержит виртуальную функцию *vf*, и производный класс *D* (*B*<-*D*) содержит функцию *vf* того же типа, то вызов *vf* для объекта класса *D* приводит к вызову *D*::*vf*, даже если доступ осуществляется через указатель или ссылку на базовый класс;

♦ виртуальные функции можно объявить только в классах (*class*) и структурах (*struct*);

♦ виртуальные функции не могут быть статическими (их нельзя объявить со спецификатором *static*);

♦ виртуальные функции можно объявить со спецификатором *friend* для другого класса;

♦ виртуальными могут быть только компоненты-функции (глобальные функции не могут быть виртуальными);

♦ виртуальная функция может, в частности, использоваться только в базовом классе и не использоваться в производных классах. Поэтому функцию можно объявить виртуальной и тогда, когда предположительно позже можно рассмотреть ее новые версии, но на данном этапе абсолютной уверенности в этом нет;

♦ новая версия виртуальной функции в производном классе должна иметь то же число и те же типы аргументов, что и версия виртуальной функции для базового класса. В противном случае функция будет не виртуальной, а переопределенной;

♦ почти всегда виртуальные функции разных версий должны возвращать одни и те же значения. Однако из этого правила есть одно исключение. Виртуальная функция в производном классе может возвращать указатель или ссылку на некоторый объект другого производного класса *D*; виртуальная функция в базовом классе может возвращать указатель или ссылку на некоторый объект другого базового класса *B* и *B*<-*D* (см. последний пример);

♦ для вызова виртуальной функции требуется больше времени, чем для вызова невиртуальной функции. Кроме того, в первом случае требуется дополнительная память для хранения виртуальной таблицы

5. Перегрузка и переопределение

Две функции называются перегруженными, если они имеют одинаковое имя, объявлены в одной и той же области видимости, но имеют разные списки формальных параметров.

Перегрузка позволяет иметь несколько одноименных функций, выполняющих схожие операции над аргументами разных типов.

Ответственность за распознавание контекста и применение операции, соответствующей типам операндов, возлагается на компилятор, а не на программиста.

Для каждого перегруженного объявления требуется отдельное определение функции с соответствующим списком параметров.

5.1. Перегрузка функций

Если в некоторой области видимости имя функции объявлено более одного раза, то второе (и последующие) объявление интерпретируется компилятором так:

- если списки параметров двух функций отличаются числом или типами параметров, то функции считаются перегруженными.
- если тип возвращаемого значения и списки параметров в объявлениях двух функций одинаковы, то второе объявление считается повторным.

Имена параметров при сравнении объявлений во внимание не принимаются.

Если списки параметров двух функций одинаковы, но типы возвращаемых значений различны, то второе объявление считается неправильным (несогласованным с первым) и помечается компилятором как ошибка.

Если списки параметров двух функций разнятся только подразумеваемыми по умолчанию значениями аргументов, то второе объявление считается повторным.

Не следует производить перегрузку, когда присвоение функциям разных имен облегчает чтение программы.

Все перегруженные функции объявляются в одной и той же области видимости. К примеру, локально объявленная функция не перегружает, а просто скрывает глобальную.

Поскольку каждый класс определяет собственную область видимости, функции, являющиеся членами двух разных классов, не перегружают друг друга.

Использование *using*-объявлений и *using*-директив помогает сделать члены пространства имен доступными в других областях видимости. Эти

механизмы оказывают определенное влияние на объявления перегруженных функций.

5.2. Указатели на перегруженные функции

Можно объявить указатель на одну из множества перегруженных функций. Например:

```
extern void ff( vector<double> );
extern void ff( unsigned int );

void ( *pf1 )( unsigned int ) = &ff;
```

Поскольку функция *ff*() перегружена, одного инициализатора &ff недостаточно для выбора правильного варианта. Чтобы понять, какая именно функция инициализирует указатель, компилятор ищет в множестве всех перегруженных функций ту, которая имеет тот же тип возвращаемого значения и список параметров, что и функция, на которую ссылается указатель. В нашем случае будет выбрана функция *ff*(*unsigned int*).

Если не найдется функции, в точности соответствующей типу указателя, тогда компилятор выдаст сообщение об ошибке.

Присваивание работает аналогично. Если значением указателя должен стать адрес перегруженной функции, то для выбора операнда в правой части оператора присваивания используется тип указателя на функцию. И если компилятор не находит функции, в точности соответствующей нужному типу, он выдает сообщение об ошибке. Таким образом, преобразование типов между указателями на функции никогда не производится.

5.3. Безопасное связывание

При использовании перегрузки складывается впечатление, что в программе можно иметь несколько одноименных функций с разными списками параметров. Однако это лексическое удобство существует только на уровне исходного текста. В большинстве систем компиляции программы, обрабатывающие этот текст для получения исполняемого кода, требуют, чтобы все имена были различны. Редакторы связей, как правило, разрешают внешние ссылки лексически. Если такой редактор встречает имя *print* два или более раз, он не может различить их путем анализа типов (к этому моменту информация о типах обычно уже потеряна). Поэтому он просто печатает сообщение о повторно определенном символе *print* и завершает работу.

Чтобы разрешить эту проблему, имя функции вместе с ее списком параметров *декорируется* так, чтобы получилось уникальное внутреннее имя. Вызываемые после компилятора программы видят только это внутреннее имя. Как именно производится такое преобразование имен, зависит от реализации. Общая идея заключается в том, чтобы представить число и типы параметров в виде строки символов и дописать ее к имени функции.

5.4. Три шага разрешения перегрузки

Разрешением перегрузки функции называется процесс выбора той функции из множества перегруженных, которую следует вызвать. Этот процесс основывается на указанных при вызове аргументах. Рассмотрим пример:

Разрешение перегрузки функции – один и самых сложных аспектов языка С++. Пытаясь разобраться во всех деталях, начинающие программисты столкнутся с серьезными трудностями. Поэтому в данном разделе мы представим лишь краткий обзор того, как происходит разрешение перегрузки.

При разрешении перегрузки функции выполняются следующие шаги:

1. Выделяется множество перегруженных функций для данного вызова, а также свойства списка аргументов, переданных функции.
2. Выбираются те из перегруженных функций, которые могут быть вызваны с данными аргументами, с учетом их количества и типов.
3. Находится функция, которая лучше всего соответствует вызову.

На первом шаге необходимо идентифицировать множество перегруженных функций, которые будут рассматриваться при данном вызове. Вошедшие в это множество функции называются *кандидатами*. Функция-кандидат – это функция с тем же именем, что и вызванная, причем ее объявление видимо в точке вызова.

После этого идентифицируются свойства списка переданных аргументов, т.е. их количество и типы.

На втором шаге среди множества кандидатов отбираются *устоявшие* (*viable*) – такие, которые могут быть вызваны с данными аргументами, Устоявшая функция либо имеет столько же формальных параметров, сколько фактических аргументов передано вызванной функции, либо больше, но тогда для каждого дополнительного параметра должно быть задано значение по умолчанию. Чтобы функция считалась устоявшей, для любого фактического аргумента, переданного при вызове, обязано существовать преобразование к типу формального параметра, указанного в объявлении.

Если после второго шага не нашлось устоявших функций, то вызов считается ошибочным. В таких случаях мы говорим, что имеет место *отсутствие соответствия*.

Третий шаг заключается в выборе функции, лучше всего отвечающей контексту вызова. Такая функция называется *наилучшей из устоявших* (или *наиболее подходящей)*. На этом шаге производится *ранжирование* преобразований, использованных для приведения типов фактических аргументов к типам формальных параметров устоявшей функции. Наиболее подходящей считается функция, для которой выполняются следующие условия:

Преобразования, примененные к фактическим аргументам, *не хуже* преобразований, необходимых для вызова любой другой устоявшей функции.

Для некоторых аргументов примененные преобразования *лучше*, чем преобразования, необходимые для приведения тех же аргументов в вызове других устоявших функций.

Если на третьем шаге не удается отыскать единственную лучшую из устоявших функцию, иными словами, нет такой устоявшей функции, которая подходила бы больше всех остальных, то вызов считается *неоднозначным*, т.е. ошибочным.

При разрешении перегрузки следует также принимать во внимание функции, конкретизированные из шаблонов.

5.5. Преобразования типов аргументов

На втором шаге процесса разрешения перегрузки функции компилятор идентифицирует и ранжирует преобразования, которые следует применить к каждому фактическому аргументу вызванной функции для приведения его к типу соответствующего формального параметра любой из устоявших функций. Ранжирование может дать один из трех возможных результатов:

точное соответствие. Тип фактического аргумента точно соответствует типу формального параметра.

соответствие с преобразованием типа. Тип фактического аргумента не соответствует типу формального параметра, но может быть преобразован в него

отсутствие соответствия. Тип фактического аргумента не может быть приведен к типу формального параметра в объявлении функции, поскольку необходимого преобразования не существует.

Для установления точного соответствия тип фактического аргумента необязательно должен совпадать с типом формального параметра. К аргументу могут быть применены некоторые тривиальные преобразования, а именно:

- преобразование l-значения в r-значение;
- преобразование массива в указатель;
- преобразование функции в указатель;
- преобразования спецификаторов.

Категория соответствия с преобразованием типа является наиболее сложной. Необходимо рассмотреть несколько видов такого приведения: *расширение типов* (*promotions*), *стандартные преобразования* и *определенные пользователем преобразования*.

При выборе лучшей из устоявших функций для данного вызова компилятор ищет функцию, для которой применяемые к фактическим аргументам преобразования являются “наилучшими”. Преобразования типов ранжируются следующим образом: точное соответствие лучше расширения типа, расширение типа лучше стандартного преобразования, а оно, в свою очередь, лучше определенного пользователем преобразования.

Самый простой случай возникает тогда, когда типы фактических аргументов совпадают с типами формальных параметров. Например, есть две показанные ниже перегруженные функции *max()*. Тогда каждый из вызовов max() точно соответствует одному из объявлений:

```
int max( int, int );
double max( double, double );

int i1;

void calc( double d1 ) {
max( 56, i1 );    // точно соответствует max( int, int );
max( d1, 66.9 );  // точно соответствует max( double, double );
}
```

Если при разрешении перегрузки получается неоднозначный результат (фактические аргументы одинаково хорошо соответствуют двум или более устоявшим функциям), то для устранения неоднозначности можно применить явное приведение типа, заставив компилятор выбрать конкретную функцию.

5.6. Перегрузка и ссылки:

Фактический аргумент или формальный параметр функции могут быть ссылками. Рассмотрим, как это влияет на правила преобразования типов.

Рассмотрим, что происходит, когда ссылкой является фактический аргумент. Его тип никогда не бывает ссылочным.

```
int i;
int& ri = i;
void print( int );

int main() {
  print( i );   // аргумент - это lvalue типа int
  print( ri );  // то же самое
  return 0;
}
```

Фактический аргумент в обоих вызовах имеет тип `int`. Использование ссылки для его передачи во втором вызове не влияет на сам тип аргумента.

Рассмотрим, как влияет на преобразования, применяемые к фактическому аргументу, формальный параметр-ссылка:

1) фактический аргумент подходит в качестве инициализатора параметра-ссылки. В таком случае мы говорим, что между ними есть точное соответствие:

2)

```
void swap( int &, int & );

void manip( int i1, int i2 ) {
  // ...
  swap( i1, i2 );    // правильно: вызывается swap( int &, int & )
  // ...
  return 0;
}
```

3) фактический аргумент не может инициализировать параметр-ссылку. В такой ситуации точного соответствия нет, и аргумент нельзя использовать для вызова функции:

```
int obj;
void frd( double & );
int main() {
  frd( obj );   // ошибка: параметр должен иметь иметь тип const double &
  return 0;
}
```

Вызов функции `frd()` является ошибкой. Фактический аргумент имеет тип *int* и должен быть преобразован в тип `double`, чтобы соответствовать

формальному параметру-ссылке. Результатом такой трансформации является временная переменная. Поскольку ссылка не имеет спецификатора `const`, то для ее инициализации такие переменные использовать нельзя.

6. Обработка ошибок и исключительные ситуации

Существенным элементом любого приложения является обработчик ошибок. Ответственность за непредвиденное завершение программы полностью лежит на программисте, а не на пользователе. Надежная программа порождает правильный результат в тех случаях, когда обрабатываемые данные и действия пользователя корректны, а так же в тех случаях, когда это не так (в этом случае правильным результатом является завершение программы с выдачей информационных сообщений и (или) корректным разрушением среды исполнения приложения).

Исключительная ситуация, или *исключение* — это возникновение непредвиденного или аварийного события, которое может порождаться некорректным использованием аппаратуры. Например, это деление на ноль или обращение по несуществующему адресу памяти. Обычно эти события приводят к завершению программы с системным сообщением об ошибке. C++ дает программисту возможность восстанавливать программу и продолжать ее выполнение.

Исключения C++ не поддерживают обработку асинхронных событий, таких, как ошибки оборудования или обработку прерываний, например, нажатие клавиш Ctrl+C. Механизм исключений предназначен только для событий, которые происходят в результате работы самой программы и указываются явным образом. Исключения возникают тогда, когда некоторая часть программы не смогла сделать то, что от нее требовалось. При этом другая часть программы может попытаться сделать что-нибудь иное.

Исключения позволяют логически разделить вычислительный процесс на две части — обнаружение аварийной ситуации и ее обработка. Это важно не только для лучшей структуризации программы. Главной причиной является то, что функция, обнаружившая ошибку, может не знать, что предпринимать для ее исправления, а использующий эту функцию код может знать, что делать, но не уметь определить место возникновения. Это особенно актуально при использовании библиотечных функций и программ, состоящих из многих модулей.

Другое достоинство исключений состоит в том, что для передачи информации об ошибке в вызывающую функцию не требуется применять возвращаемое значение, параметры или глобальные переменные, поэтому интерфейс функций не раздувается. Это особенно важно, например, для конструкторов, которые по синтаксису не могут возвращать значение.

6.1. Синтаксис исключений

Синтаксис исключения выглядит следующим образом:

```
try
{
    контролируемый блок
}
catch(перехваченное исключение)
{
    блок обработки исключения
}
```

Например:

```
try
{
    x = x / y;
}
catch(...)
{
    cout >> "Деление на ноль" >> endl;
}
```

Ключевое слово *try* служит для обозначения *контролируемого блока* — кода, в котором может генерироваться исключение. Блок заключается в фигурные скобки:

```
try
{
...
}
```

Все функции, прямо или косвенно вызываемые из `try`-блока, также считаются ему принадлежащими.

Генерация (порождение) исключения происходит по ключевому слову `throw`, которое употребляется либо с параметром, либо без него:

```
throw [ выражение ];
```

Тип выражения, стоящего после `throw`, определяет тип порождаемого исключения. При генерации исключения выполнение текущего блока прекращается, и происходит поиск соответствующего обработчика и передача ему управления. Как правило, исключение генерируется не

непосредственно в *try*-блоке, а в функциях, прямо или косвенно в него вложенных.

Не всегда исключение, возникшее во внутреннем блоке, может быть сразу правильно обработано. В этом случае используются вложенные контролируемые блоки, и исключение передается на более высокий уровень с помощью ключевого слова *throw* без параметров.

Обработчики исключений начинаются с ключевого слова *catch*, за которым в скобках следует тип обрабатываемого исключения. Они должны располагаться непосредственно за *try*-блоком. Можно записать один или несколько обработчиков в соответствии с типами обрабатываемых исключений. Синтаксис обработчиков напоминает определение функции с одним параметром — типом исключения. Существует три формы записи:

```
catch(тип имя) { … /* тело обработчика */ }
catch(тип)     { … /* тело обработчика */ }
catch(…)       { … /* тело обработчика */ }
```

Первая форма применяется, когда имя параметра используется в теле обработчика для выполнения каких-либо действий — например, вывода информации об исключении. Вторая форма не предполагает использования информации об исключении, играет роль только его тип. Многоточие вместо параметра обозначает, что обработчик перехватывает все исключения. Так как обработчики просматриваются в том порядке, в котором они записаны, обработчик третьего типа следует помещать после всех остальных.

Пример:

```
catch(int i)
{
    . . .
    // Обработка исключений типа int
}
catch(const char *)
{
    . . .
    // Обработка исключений типа const char*
}
catch(Overflow)
{
    . . .
    // Обработка исключений класса Overflow
}
catch(...)
{
```

```
    . . .
    // Обработка всех остальных исключений
}
```

Пример:

```
// Простой пример обработки исключительной ситуации
#include "stdafx.h"
#include <stdio.h>
#include <conio.h>
#include <iostream>
using namespace std;


int main(int argc, char* argv[])
{
    cout << "Begin\i";
    try
    {
         cout << "Inside block try\n";
         throw 10;        //  excitation error
         cout << "This instraction isn't run";
    }
    catch (int i)
    {
        cout <<   "Number intercept error:    ";
        cout  << i <<  "\n";
    }
    cout  <<   "End";
    getch();
    return   0;

}
```

После обработки исключения управление передается первому оператору, находящемуся непосредственно за обработчиками исключений. Туда же, минуя код всех обработчиков, передается управление, если исключение в `try`-блоке не было сгенерировано.

6.2. Основная идея обработки исключений

Основная идея обработки исключений заключается в следующем (рис).

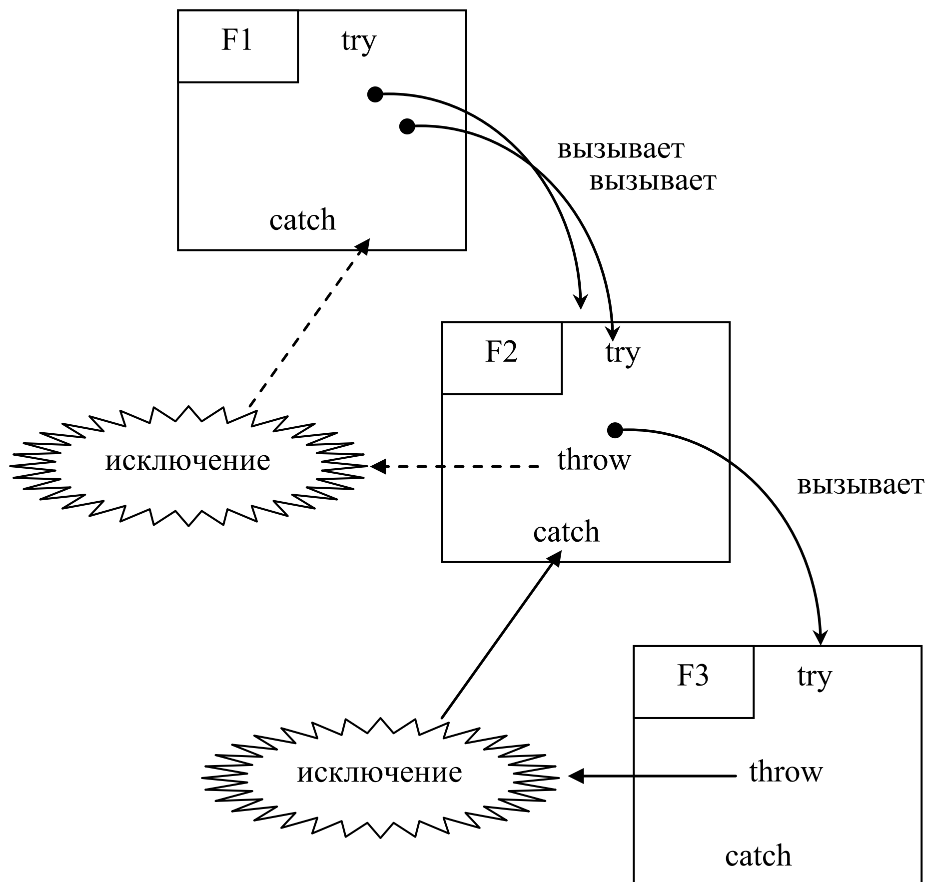


Рисунок 10 Уровни обработки исключительных ситуаций.

Пусть некоторая функция в иерархии вызовов (например, функция *F2*()) передает управление другой функции *F3*(). В ходе работы функции *F3*() выявляется некоторая ошибочная ситуация (т. е. функция *F3*() обнаруживает ошибку). Функция, обнаружившая ошибку, генерирует (*throws*) специальный объект, описывающий возникшую исключительную ситуацию. Иными словами, функция *F3*() создает объект-исключение. Выполнение функции, обнаружившей ошибку, на этом прекращается, и управление передается в вызвавшую её функцию (в нашем случае— в функцию *F2*()) для реализации действий по преодолению возникшей проблемы. В этом случае говорят, что функция *F2*() перехватывает (*catches*) исключение (рис.). При этом необходимо заметить, что обработкой исключения может заниматься как функция, непосредственно вызвавшая функцию-генератор исключения, так и предшествующие в иерархии вызовов функции **(**например**,** функция *F1*()).

Таким образом, с одной стороны, механизм обработки исключений способствует решению проблемы отделения кода, диагностирующего ошибку, от кода, обрабатывающего ошибку, некоторым единым способом, поддерживаемым языком.

С другой стороны, исключения предоставляют возможность коду, обнаружившему проблему, последствия которой не могут быть преодолены

на месте, передать эту проблему в другую часть системы. В частности, исключения предоставляют инструмент, позволяющий сообщить об ошибке, произошедшей в ходе выполнения конструктора. При этом следует заметить, что генерация исключений в конструкторе не лишена некоторых подводных камней.

Обработка исключений — тоже в своем роде механизм абстракции, поскольку позволяет абстрагироваться в основном коде от потенциальных проблем, не концентрируя все внимание на обнаружении и обработке ошибок.

6.3. Что такое "исключение" и как оно образуется

Как было отмечено выше, генерация исключения представляет собой создание объекта-исключения. Объект характеризуется типом, например:

```
// Класс, определяющий тип объекта-исключения
class Exception
{
    //   ...
    public:
    Exception()
    {
         // Конструктор
         ...
    }
};
```

Блок, обнаруживающий ошибку, инициирует процесс создания объекта-исключения с использованием специального синтаксиса:

```
// Функция, обнаруживающая ошибку и генерирующая
исключение
void F3()
{
    if( /*  ошибка...   */ )
    {
        throw Exception();   // Генерирует исключение
    }
    // Действия при отсутствии ошибок
    ...
}
```

Вызывающая функция может перехватить исключение и выполнить обработку ошибки:

```
// Функция,   перехватывающая исключение
void  F2()
{
    try
    {
        F3();   // Может генерировать исключение
    }
    catch( Exception )
    {
        // Обработка ошибки
        ...
     }
}
```

Блок *try* включает обращение к функциям, которые могут генерировать исключения. Поскольку обработка исключения осуществляется в блоке *catch*, то именно он и называется обычно обработчиком исключения. Если в контексте работы функции *F2*() исключение не может быть обработано полностью, эта функция может инициировать повторное исключение с тем, чтобы оно было обработано на следующих уровнях иерархии вызовов.

```
// Функция,   перехватывающая и повторно
// инициирующая исключение
void F2()
{
    try
    {
        F3{};   // Может генерировать исключение
    }
    catch( Exception )
    {
         // Частичная обработка ошибки
         . . .
         throw;   // Повторно генерирует исключение
    }
}
```

Если функция *F3*() может генерировать несколько типов исключений, то используется несколько последовательно записываемых друг за другом обработчиков исключений, которые в этом случае подобны перегруженным функциям, однако это подобие весьма условно. При перехвате исключения выбирается нужный обработчик в зависимости от типа сгенерированного исключения.

Таким образом, общая схема генерации и обработки исключения выглядит следующим образом. Функция, диагностировавшая ошибку, генерирует объект исключения с помощью инструкции *throw*. Далее происходит раскручивание стека приложения до тех пор, пока не будет обнаружен подходящий обработчик исключения. Еще раз обращаем внимание на то, что блок, начинающийся с ключевого слова *try*, и подходящий обработчик не обязательно располагаются в функции, непосредственно вызвавшей код, сгенерировавший исключение.

6.4. Генерация исключений

Выражение, формирующее исключение, может иметь две формы:

```
throw   выражение генерации исключения;
throw;
```

В первой из указанных форм исключение формируется как статический объект, значение которого определяется выражением генерации. Несмотря на то, что исключение формируется внутри функции как локальный объект, копия этого объекта передается за пределы контролируемого блока и инициализирует переменную, использованную в спецификации исключения обработчика. Копия объекта, сформированного при генерации исключения, существует, пока исключение не будет полностью обработано.

В некоторых случаях используется вложение контролируемых блоков, и не всегда исключение, возникшее в самом внутреннем контролируемом блоке, может быть сразу же правильно обработано. В этом случае в обработчике можно использовать сокращенную форму оператора:

```
throw;
```

Этот оператор, не содержащий выражения после служебного слова, "ретранслирует" уже существующее исключение, т.е. передает его из процедуры обработки и из контролируемого блока, в который входит эта процедура, в процедуру обработки следующего (более высокого) уровня. Естественно, что ретрансляция возможна только для уже созданного исключения. Поэтому оператор *throw*; может использоваться только внутри процедуры обработки исключений и разумен только при вложении контролируемых блоков.

Текст программы-примера:

```
// вложение   контролируемых   блоков
// и  ретрансляция  исключений
#include "stdafx.h"
```

```
#include <stdio.h>
#include <conio.h>
#include <iostream>
using namespace std;void compare(int k)  //Функция,
генерирующая  исключения
{
    if(k%2!=0) throw k;  // Нечетное значение   (odd)
    else throw "even";   // Четное   значение   (even)
}

// Функция с контролем и обработкой исключений:
void GG(int j)
{
    try
    {
        try
        {
            compare(j);
        }
        catch (int n)
        {
            cout << "\n Odd";
            throw;          // Ретрансляция исключения
        }
        catch (const char *)
        {
            cout  <<  "\n Even";
        }
    }
    //  Обработка ретранслированного  исключения;
    catch (int i)
    {
         cout <<  "\n Result =  " << i;
    }
}
int  main()
{
    GG(4);
    GG(7);
}
```

Результат работы программы:

```
Even
Odd
Result = 7
```

6.5. Перехват исключений

Когда с помощью *throw* генерируется исключение, функции исполнительной библиотеки С++ выполняют следующие действия:

1) создают копию параметра *throw* в виде статического объекта, который существует до тех пор, пока исключение не будет обработано;

2) в поисках подходящего обработчика раскручивают стек, вызывая деструкторы локальных объектов, выходящих из области действия;

3) передают объект и управление обработчику, имеющему параметр, совместимый по типу с этим объектом.

При раскручивании стека все обработчики на каждом уровне просматриваются последовательно, от внутреннего блока к внешнему, пока не будет найден подходящий обработчик.

Обработчик считается найденным, если тип объекта, указанного после *throw*:

- тот же, что и указанный в параметре *catch* (параметр может быть записан в форме `T`, *const* `T`, `T&` или *const* `T&`, где `T`— тип исключения);
- является производным от указанного в параметре `catch` (если наследование производилось с ключом доступа *public*);
- является указателем, который может быть преобразован по стандартным правилам преобразования указателей к типу указателя в параметре *catch*.

Последнее поясним подробнее.

При поиске обработчика, пригодного для "обслуживания" исключения, оно последовательно сравнивается по типу со спецификациями исключений, помещенными в скобках после служебных слов *catch*. Спецификации исключений подобны спецификациям формальных параметров функций, а набор обработчиков исключений подобен совокупности перегруженных функций. Если обработчик исключений (процедура обработки) имеет вид:

```
catch   (T x)
{
   …. // действия обработчика
}
```

где *T* - некоторый тип, то обработчик предназначен для исключений в виде объектов типа *T*.

Однако сравнение по типам в обработчиках имеет более широкий смысл. Исключение "захватывается" (воспринимается) обработчиком и в том случае, если тип исключения может быть стандартным образом приведен к типу формального параметра обработчика. Кроме того, если исключение есть

объект некоторого класса *T* и у этого класса *T* есть доступный в точке порождения исключения базовый класс *B*, то обработчик

```
catch(B x)
{
    ….//действия обработчика
}
```

также соответствует этому исключению.

Из выше изложенного следует, что обработчики производных классов следует размещать до обработчиков базовых, поскольку в противном случае им никогда не будет передано управление. Обработчик указателя типа `void` автоматически скрывает указатель любого другого типа, поэтому его также следует размещать после обработчиков указателей конкретного типа.

Пример:

```
#include "stdafx.h"
#include <stdio.h>
#include <conio.h>
#include <iostream>
using namespace std;


class Base
{
public:
    Base(){};
};

class Child : public Base
{
public:
    Child(){};
};


void main()
{
    try
    {
        throw Child();
    }
    catch(Base &B)
    {
```

```
        cout <<  "\n Base  " << endl;
    }
    catch(Child &C)
    {
        cout <<  "\n Child  " << endl;
    }
    getch();
}
```

Base

Пример:

```
#include "stdafx.h"
#include <stdio.h>
#include <conio.h>
#include <iostream>
using namespace std;


class Base
{
public:
   Base(){};
};

class Child : public Base
{
public:
   Child(){};
};


void main()
{
    try
    {
       throw Base();
    }
    catch(Child &C)
    {
        cout <<  "\n Child  " << endl;
    }
    catch(Base &B)
```

```
    {
          cout <<  "\n Base  " << endl;
    }

    getch();
}
```

Base

Пример:

```
#include "stdafx.h"
#include <stdio.h>
#include <conio.h>
#include <iostream>
using namespace std;


class Base
{
public:
    Base(){};
};

class Child : public Base
{
public:
    Child(){};
};


void main()
{
     try
    {
        throw Child();
    }
    catch(Child &C)
    {
          cout <<  "\n Child  " << endl;
    }
    catch(Base &B)
    {
          cout <<  "\n Base  " << endl;
```

```
    }

    getch();
}
```

```
Child
```

При возникновении исключительной ситуации исполнение функции, инициировавшей объект исключения, прекращается, поэтому, для того чтобы объект исключения не был потерян, создается копия объекта, которая и связывается с параметром обработчика *catch*. При следующей за этим передаче значения объекта-исключения в обработчик создается еще одна копия. Таким образом, копирование объекта-исключения происходит дважды. Данную ситуацию иллюстрирует следующая программа:

```
#include <iostream>
using namespace std;
class Exception
{
    public:
    Exception()
    {
        cout << "Exception constructed" << endl;
    }
    Exception( const Exception & )
    {
        cout << "Exception copied" << endl;
    }
    void Check()
    {
        cout << "Exception checked" << endl;
    }
};
void F3()
{
    throw Exception();
}
void F2()
{
    try
    {
        F3();
    }
    catch( Exception e )
    {
```

```
        cout << "Exception in F2 " << endl;
        throw e;
    }
}
void F1()
{
    try
    {
        F2 ();
    }
    catch( Exception e )
    {
        cout << "Exception in F1" << endl;
        e.Check ();
    }
}
void main()
{
    F1();
}
```

В программе класс исключения снабжен конструктором копирования, наличие которого позволяет наблюдать за возникновением копий объекта исключения. В результате исполнения программы будут выведены следующие сообщения:

```
Exception constructed
Exception copied
Exception copied
Exception in F2
Exception copied
Exception copied
Exception in Fi
Exception checked
```

Данный вывод наглядно демонстрирует, что при генерации и последующем перехвате объекта-исключения по значению (равно как и при повторной генерации) объект класса *Exception* копируется дважды.

Для того чтобы избежать повторного копирования, исключения следует перехватывать по ссылке.

Пример:

```
#include "stdafx.h"
#include <stdio.h>
```

```
#include <conio.h>
#include <iostream>
using namespace std;


class Exception
{
    public:
    Exception()
    {
        cout << "Exception constructed" << endl;
    };
    Exception( const Exception & )
    {
        cout << "Exception copied" << endl;
    };
    void Check()
    {
        cout << "Exception checked" << endl;
    };
};
void F3()
{
    throw Exception();
}
void F2()
{
    try
    {
        F3();
    }
    catch( Exception & e )
    {
        cout << "Exception in F2 " << endl;
        throw e;
    }
}
void F1()
{
    try
    {
        F2();
    }
    catch( Exception & e )
    {
```

```
        cout << "Exception in F1" << endl;
        e.Check();
    }
}
void main()
{
    F1();
    getch();
}
```

В программе класс исключения снабжен конструктором копирования, наличие которого позволяет наблюдать за возникновением копий объекта исключения. В результате исполнения программы будут выведены следующие сообщения:

```
Exception constructed
Exception copied
Exception in F2
Exception copied
Exception in Fi
Exception checked
```

6.6. Использование обработчика catch(…)

Обработчик данного вида позволяет перехватить исключения любого типа. Такой обработчик можно использовать, когда содержание обработки ошибки не зависит от типа исключения. Однако не следует увлекаться практикой использования таких "универсальных обработчиков" вместо нескольких отдельных обработчиков. Например, Рихтер отмечает, что если задуматься над смыслом данной конструкции, то она означает, что мы пытаемся убедить компилятор в способности написать код, обрабатывающий любое исключение, известное нам или неизвестное. Но это неверно и может служить источником трудно обнаруживаемых ошибок. Поэтому следует минимизировать использование таких обработчиков. Возможно, лучше позволить обработать исключение исполнительной системе (и мы, по крайней мере, увидим, что нештатная ситуация возникла), чем скрыть за подобной конструкцией отсутствующую обработку или обработку с ошибками.

Область применения обработчика `catch(...)`, таким образом, ограничена следующими типовыми ситуациями:

- освобождение занятых ресурсов и повторная генерация исключения;

- повторная генерация исключений, которые не могут быть обработаны данным исключением (обработчик *catch*(...) должен быть записан последним);
- обработка исключений в случае, если функция, генерирующая исключение, нарушила контракт, в соответствии с которым может генерировать только определенные виды исключений.

6.7. Когда исключение считается обработанным?

В соответствии с подходом, реализованным в С++, исключение считается обработанным сразу после входа в обработчик. Поэтому нижеследующая конструкция не приводит к бесконечному циклу:

```
try
{
    // ...
}
catch( Exceptions & )
{
    // ...
    throw Exception();
}
```

Следует обратить внимание, что генерация исключения типа *Exception* в данном обработчике не трактуется как повторная генерация исключения. Исключение перебрасывается во внешний блок *try-catch*.

Пример:

```
#include "stdafx.h"
#include <stdio.h>
#include <conio.h>
#include <iostream>
#include <climits>
#include <new>
using namespace std;


class Exceptions
{
public:
```

```
   Exceptions(){};
};

void main()
{
   try
   {
      try
      {
         throw Exceptions();
      }
      catch( Exceptions & e)
      {
         cout << "exception one" <<  endl;
         throw Exceptions();
      }
   }
   catch( Exceptions & e)
   {
      cout << "exception two" <<  endl;
   }

   getch();
}
```

6.8. Исключения связанные с ошибкой выделения памяти

До появления в C++ механизма обработки исключений, в случае, если выделение памяти завершалось неудачей, в результате операции *new* формировалось значение 0. В соответствии с требованиями современного стандарта C++ при ошибке выделения памяти *new* по умолчанию генерирует исключение *bad_alloc*. Исключение *bad_alloc*, равно как и соответствующая реализация *new*, определено в заголовочном файле *new* в пространстве имен *std*.

Поэтому рекомендации стандарта C++ предписывают преобразовывать код, использующий механизм с анализом значения указателя, к форме, ориентированной на обработку исключений. Если по каким-либо причинам желательно сохранить прежний стиль обработки ошибки резервирования динамической памяти, можно изменить программу, используя распределитель *nothrow*, при котором исключение не генерируется. Обе возможности иллюстрирует пример, представленный в примере.

Пример:

```
#include <iostream>
#include <climits>
#include <new>      // Подключает библиотеку, обеспечивающую
                    // реализацию new с генерацией
                    // исключения bad_alloc в случае ошибки
using namespace std;
// Функция запрашивает слишком много динамической памяти
void AskLotOfMemory()
{
    int  *p;
    p = new int[ UINT_MAX ];
}
int main()
{
    try
    {
        AskLotOfMemory();
    }
    catch( bad_alloc& e )
    {
        // Будет напечатано сообщение "bad allocation"
        cout << e.what() << endl;
    }
    // Использование nothrow подавляет генерацию исключения
    int *p = new ( nothrow ) int[ UINT_MAX ];
    if( p == 0 )
    {
        // Будет напечатано сообщение "no exception thrown"
        cout << "no exception thrown" <<  endl;
    }
}
```

В результате исполнения программы на консоль будут выведены следующие сообщения:

```
bad allocation
no exception thrown
```

В первом случае (при резервировании динамической памяти в функции *AskLotofMemory*()) реализуется подразумеваемое стандартом поведение с генерацией исключения *bad_alloc*, перехватываемого в функции *main*(). Если вы работаете в интегрированной среде программирования, то в режиме отладки возникшая исключительная ситуация (которая трактуется как опасная), возможно, будет перехвачена отладчиком. Например, при отладке в *Microsoft Visual Studio* появится окно ***Debug Assertion***. Для того чтобы убедиться, что программа работает, нажмите кнопку *Ignore*, дав программе возможность завершить работу, и убедитесь в том, что исключение успешно перехватывается.

Во втором случае (фрагмент резервирования памяти в функции *main*()) использование *nothrow* подавляет генерацию исключения, и работает "старый" механизм, подразумевающий необходимость анализа значения указателя p.

6.9. Конструкторы и исключения

Когда выполнение программы прерывается возникшим исключением, происходит вызов деструкторов для всех автоматических объектов, появившихся с начала входа в контролируемый блок. Если исключение было порождено во время исполнения конструктора некоторого объекта, деструкторы вызываются лишь для успешно построенных объектов. Например, если исключение возникло при построении массива объектов, деструкторы будут вызваны только для полностью построенных объектов.

Пример:

```
#include "stdafx.h"
#include <cstdlib>
#include <conio.h>
#include <iostream>
#include <exception>

using namespace std;

class A
{
public:
    A()
    {
        cout << "A::A(" << "Conctructor" << ")\n";
    }
    ~A()
    {
        std::cout << "A::~A(" << "Destructor" << ")\n";
    }
};
class B
{
public:
    B( size_t size)
        :m_a()
        ,m_pData(new int[size])
        ,m_size(size)
```

```
    {
        std::cout << "B::B(" << m_size << ")\n";
    }
    ~B()
    {
        delete [] m_pData;
        std::cout << "B::~B(" << m_size << ")\n";
    }
private:
    A m_a;
    size_t m_size;
    int * m_pData;
};

int main()
{
    try
    {
        B b2(1000000000);
    }
    catch (std::bad_alloc const & e)
    {
        std::cout << "Error: " << e.what() << "\n";
    }
    getch();
    return 0;
}
```

В результате исполнения программы будут выведены следующие сообщения:

```
A::A(Conctructor)
A::~A(Destructor)
Error: bad allocation
```

6.10. Деструкторы и исключения

В принципе, следует избегать завершения работы деструктора посредством генерации исключения (требования стандартной библиотеки предписывают считать это правилом). Это требование обусловлено следующими факторами. Во-первых, поскольку деструкторы в большинстве случаев вызываются неявно, пользователю класса может оказаться затруднительно встроить в подходящее место кода обработчик этого исключения. Во-вторых, деструкторы могут вызываться в процессе

раскрутки стека приложения (*stack unwinding*) при возникновении исключения и выходе объектов из областей видимости функций, находящихся в иерархии вызовов.

6.11. Спецификация исключений

Вообще говоря, вы не обязаны сообщать пользователям вашей функции, какие исключения она может запускать. Однако такое поведение считается нецивилизованным — оно означает, что пользователи не будут знать, как написать код перехвата потенциальных исключений. При наличии исходных текстов они смогут просмотреть их и поискать команды *throw*, однако библиотеки не всегда поставляются с исходными текстами. Хорошая документация поможет решить проблемы, но много ли найдется хорошо документированных программных проектов? Специальный синтаксис C++ позволяет сообщить пользователю, какие исключения запускаются данной функцией, чтобы он мог обработать их. Речь идет о необязательной спецификации исключений, указываемой в объявлении функции после списка аргументов.

Спецификация исключений состоит из ключевого слова *throw*, за которым в круглых скобках перечисляются типы всех потенциальных исключений, которые могут запускаться данной функцией. Объявление функции может выглядеть примерно так:

```
void f() throw(toobig, toosmall, divzero);
```

В отличие от этого объявления традиционное объявление функции означает, что функция может запускать исключения любых типов:

```
void f();
```

Однако следующая конструкция говорит о том, что функция не может запускать никаких исключений (проследите за тем, чтобы функции, находящиеся на очередном уровне в цепочке вызовов, не передавали исключения наверх!):

```
void f() throw();
```

Если хороший стиль программирования, полнота документации и удобства работы с функцией вам не безразличны, обязательно включайте спецификации исключений в те функции, которые их запускают.

Если функция вдруг сгенерирует исключение, не соответствующее списку спецификации исключений, тогда система вызовет обработчик `std::unexpected()`, который может попытаться сгенерировать свое

исключение[1], и если оно не будет противоречить спецификации, то продолжится поиск подходящего обработчика, в противном случае вызывается `std::terminate()`.

6.12. Алгоритм обработки исключительной ситуации

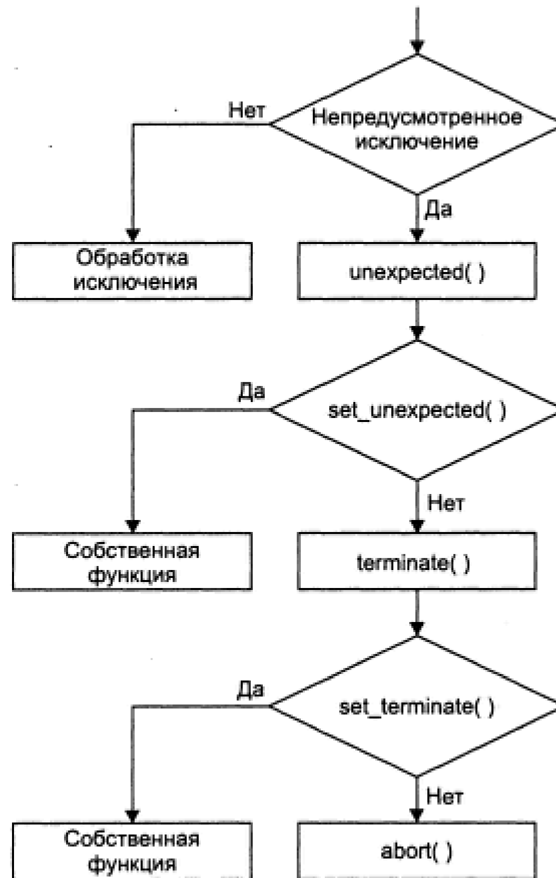


Рисунок 11 Обобщенный алгоритм обработки исключительной ситуации

Если функция породила исключение которое не может быть обработано, то эта ситуация обнаруживается во время исполнения программы и приводит к вызову стандартной функции *unexpected*, которая по умолчанию просто вызывает функцию *terminate*. С помощью функции *set_unexpected* можно установить собственную функцию, которая будет вызываться вместо *terminate* и определять действие программы при возникновении непредвиденной исключительной ситуации.

Функция terminate по умолчанию вызывает функцию *abort*, которая завершает выполнение программы. С помощью функции *set_terminate* можно установить собственную функцию, которая будет вызываться вместо *abort* и определять способ завершения программы.

6.13. Функция set_unexpected()

Функции *set_unexpected* и *set_terminate* описаны в заголовочном файле <*excepton*>.

Функция *void unexpected*() вызывается, когда некоторая функция порождает исключение, отсутствующее в списке ее исключений. В свою очередь функция *unexpected*() по умолчанию вызывает функцию, зарегистрированную пользователем с помощью *set_unexpected*(). Если такая функция отсутствует, *unexpected*() вызывает *terminate*(). Функция *unexpected*() не возвращает значения, однако может сама порождать исключения.

Функция *set_unexpected*() позволяет установить функцию, определяющую реакцию программы на неизвестное исключение. Эти действия определяются в функции, которая ниже поименована как *unexpected_func*(). Указанная функция специфицируется как функция типа *unexpected_function*. Этот тип определен в файле *except.h* как указатель на функцию без параметров, не возвращающую значения:

```
typedef void (*unexpected_function)();
unexpected_function set_unexpected(unexpected_function
unexpected_func);
```

По умолчанию, неожиданное (неизвестное для функции) исключение вызывает функцию *unexpected*(), которая, в свою очередь вызывает либо unexpected*_func*() (если она определена), либо *terminate*() (в противном случае).

Функция *set_unexpected*() возвращает указатель на функцию, которая была установлена с помощью *set_unexpected*() ранее. Устанавливаемая функция (*unexpected_func*) обработки неизвестного исключения не должна возвращать управление вызвавшей ее функции *unexpected*(). Попытка возврата приведет к неопределенным результатам.

Кроме всего прочего, *unexpected_func*() может вызывать функции *abort*(), *exit*() и *terminate*().

Иногда вызываемая функция порождает непредвиденные исключения, а иногда старая функция, не запускавшая исключений, заменяется новой функцией с исключениями, что приводит к вызову *unexpected*(). Каждый раз, когда вы вызываете функции, в которых есть исключения, подумайте о создании собственной функции *unexpected*). Такая функция регистрирует сообщение, а затем либо запускает исключение, либо завершает программу.

6.14. Функция set_terminate()

Функция *void terminate*() вызывается в случае, когда отсутствует процедура для обработки некоторого сформированного исключения. По умолчанию *terminate*() вызывает библиотечную функцию *abort*(), что влечет выдачу сообщения "*Abnormal program termination*" и завершение программы. Если такая последовательность действий программиста не устраивает, он может написать собственную функцию (*terminate_function*) и зарегистрировать ее с помощью функции *set_terminate*(). В этом случае *terminate*() будет вызывать эту новую функцию вместо функции *abort*().

Функция *set_terminate*() позволяет установить функцию, определяющую реакцию программы на исключение, для обработки которого нет специальной процедуры. Эти действия определяются в функции, поименованной ниже как *terminate_func*(). Указанная функция специфицируется как функция типа *terminate_function*. Такой тип в свою очередь определен в файле *except.h* как указатель на функцию без параметров, не возвращающую значения:

```
typedef void (*terminate_function)();
terminate_function  set_terminate(terminate_function
terminate_func);
```

Функция *set_tenninate*() возвращает указатель на функцию, которая была установлена с помощью *set_terminate*() ранее.

Следующая программа демонстрирует общую схему применения собственной функции для обработки неопознанного исключения:

```
#include "stdafx.h"
#include <cstdlib>
#include <conio.h>
#include <iostream>
#include <exception>

using  namespace  std;

//  Примеры типов исключений

class First{};

class Second{};

class Unknown{};


void g();
void f( int i )   throw  ( First,  Second )
{
    switch(i)
    {
        case 1:   throw First();
```

```
        case 2:   throw Second();
    }
    g();   //Эта функция генерирует "неожиданное" исключение
}
void g()
{
    throw Unknown ();
}
//Пользовательская функция аварийного завершения
void MyTermination()
{
    cout << "Unknown exception thrown" << endl;
    //exit(1);
}
int main()
{
    //  Регистрация  функции MyTermination () для
    //  вызова  вместо  tarminate()
    set_terminate(MyTermination);

    for(int i = 1 ; i <=3 ; i++ )
    {
        try
        {
            f(i);
        }
        catch(First&)
        {
            // (на  первой итерации цикла)
            cout << "First  exception caught" << endl;
        }
        catch (Second&)
        {
            //   (на второй итерации цикла)
            cout << "Second exception caught" << endl;
        }
        /* ЭТО  КОММЕНТАРИЙ, СОДЕРЖАНИЙ АЛЬТЕРНАТИВНЫЙ ВАРИАНТ!
        *
        *  Можно перехватить неизвестное исключение так:
        *  catch(...)
        *  {
        *          //(на третьей итерации цикла)
        *          cout << "Unknown exception  caught" << endl;
        *  }
        *   КОНЕЦ АЛЬТЕРНАТИВНОГО ВАРИАНТА */
    }
    getch();
}
```

Перечислим требования к создаваемой программистом функции для обработки неопознанного исключения:

- она не должна формировать новых исключений,

- эта функция должна завершать программу и не возвращать управление вызвавшей ее функции *terminate*(). Попытка такого возврата приведет к неопределенным результатам.

6.15. Когда лучше обойтись без исключений

Ошибки делятся на два вида:

1) те которые я ожидаю
2) и те которые маловероятны

Первые — лучше обрабатывать обычным способом (по возвращаемому результату)
Вторые — исключениями.

Например:

Работа с файлами:
Вполне вероятно, что может произойти ошибка при открытии файла и исключения лучше не использовать.
Если уж мы его благополучно открыли, то маловероятно, что произойдет ошибка при чтении/записи лучше использовать исключения.

7. Шаблоны функций

Цель введения шаблонов функций - автоматизация создания функций, которые могут обрабатывать разнотипные данные. В отличие от механизма перегрузки, когда для каждой сигнатуры определяется своя функция, шаблон семейства функций определяется один раз, но это определение параметризируется. Параметризировать в шаблоне функций можно тип возвращаемого функцией значения и типы любых параметров, количество и порядок размещения которых должны быть фиксированы. Для параметризации используется список параметров шаблона.

В определении шаблона семейства функций используется служебное слово *template*. Для параметризации используется список формальных параметров шаблона, который заключается в угловые скобки <>. Каждый формальный параметр шаблона обозначается служебным словом *class*, за которым следует имя параметра (идентификатор). Пример определения шаблона функций, вычисляющих абсолютные значения числовых величин разных типов:

```
template <class type>
type abs(type x) { return x > 0 ? x : -x; }
```

Шаблон семейства функций состоит из двух частей - заголовка шаблона:

template <**список_параметров_шаблона**>

и из обыкновенного определения функции, в котором тип возвращаемого значения и типы любых параметров обозначаются именами параметров шаблона, введенных в его заголовке. Те же имена параметров шаблона могут использоваться и в теле определения функции для обозначения типов локальных объектов.

В качестве еще одного примера рассмотрим шаблон семейства функций для обмена значений двух передаваемых им параметров.

```
template <class T>
void swap (T* x, T* y)
{
T z = *x;
*x = *y;
*y = z;
}
```

Здесь параметр *T* шаблона функций используется не только в заголовке для спецификации формальных параметров, но и в теле определения функций, где он задает тип вспомогательной переменной *z*.

Шаблон семейства функций служит для автоматического формирования конкретных определений функций по тем вызовам, которые транслятор обнаруживает в тексте программы. Например, если программист употребляет обращение *abs* (-10.3), то на основе приведенного выше шаблона компилятор сформирует такое определение функции:

```
double abs(double x) {return x > 0  ? x   :   -x;}
```

Далее будет организовано выполнение именно этой функции и в точку вызова в качестве результата вернется числовое значение 10.3.

Если в программе присутствует приведенный выше шаблон семейства функций *swap*() и появится последовательность операторов:

```
long k = 4,   d =  8;
swap   (&k,  &d) ;
```

то компилятор сформирует определение функции:

```
void swap(long*  x,   long*  y)
{
```

```
    long  z  = *x;
    *x =  *y;
    *y = z;
}
```

Затем будет выполнено обращение именно к этой функции и значения переменных *k*, *d* поменяются местами.

Если в той же программе присутствуют операторы:

```
double a = 2.44,   b =  66.3;
swap   (&a,   &b);
```

то сформируется и выполнится функция:

```
void swap (double* x, double* y)
{
    double z = *x;
    *x = *y;
    *y = z ;
 }
```

По существу механизм шаблонов функций позволяет автоматизировать подготовку определений перегруженных функций. При использовании шаблонов уже нет необходимости готовить заранее все варианты функций с перегруженным именем. Компилятор автоматически, анализируя вызовы функций в тексте программы, формирует необходимые определения именно для таких типов параметров, которые использованы в обращениях. Дальнейшая обработка выполняется так же, как и для перегруженных функций.

Можно считать, что параметры шаблона функций являются его формальными параметрами, а типы тех параметров, которые используются в конкретных обращениях к функции, служат фактическими параметрами шаблона. Именно по ним выполняется параметрическая настройка и с учетом этих типов генерируется конкретный текст определения функции. Однако, говоря о шаблоне семейства функций, обычно употребляют термин "список параметров шаблона", не добавляя определения "формальных".

Перечислим основные свойства параметров шаблона:

1. Имена параметров шаблона должны быть уникальными во всем определении шаблона.
2. Список параметров шаблона функций не может быть пустым, так как при этом теряется возможность параметризации и шаблон функций становится обычным определением конкретной функции.

3. В списке параметров шаблона функций может быть несколько параметров. Каждый из них должен начинаться со служебного слова *class*. Например, допустим такой заголовок шаблона:

```
template <class type1, class type2>
```

Соответственно, неверен заголовок:

```
template <class type1, type2, type3>
```

Недопустимо использовать в заголовке шаблона параметры с одинаковыми именами, т.е. ошибочен такой заголовок:

```
template <class t, class t, class t>
```

Имя параметра шаблона (в наших примерах *type1*, *type2* и т.д.) имеет в определяемой шаблоном функции все права имени типа, т.е. с его помощью могут специализироваться формальные параметры, определяться тип возвращаемого функцией значения и типы любых объектов, локализованных в теле функции. Имя параметра шаблона видно во всем определении и скрывает другие использования того же идентификатора в области, глобальной по отношению к данному шаблону функций. Если внутри тела определяемой функции необходим доступ к внешним объектам с тем же именем, нужно применять операцию изменения области видимости.

Итак, одно имя нельзя использовать для обозначения нескольких параметров одного шаблона, но в разных шаблонах функций могут быть одинаковые имена у параметров шаблонов. Ситуация здесь такая же, как и у формальных параметров при определении обычных функций, и на ней можно не останавливаться подробнее. Действительно, раз действие параметра шаблона заканчивается в конце определения шаблона, то соответствующий идентификатор свободен для последующего использования, в том числе и в качестве имени параметра другого шаблона.

Все параметры шаблона функций должны быть обязательно использованы в спецификациях параметров определения функции. Таким образом, будет ошибочным такой шаблон:

```
template <class A, class B, class C>
B func(A n, C m) {B value; ... }
```

В данном неверном примере остался неиспользованным параметр шаблона с именем B. Его применений в качестве типа возвращаемого функцией значения и для определения объекта *value* в теле функции недостаточно.

Определяемая с помощью шаблона функция может иметь любое количество непараметризованных формальных параметров. Может быть не параметризовано и возвращаемое функцией значение.

Как и при работе с обычными функциями, для шаблонов функций существуют определения и описания. В качестве описания шаблона функций используется прототип шаблона:

```
template <список_параметров_шаблона>
```

В списке параметров прототипа шаблона имена параметров не обязаны совпадать с именами тех же параметров в определении шаблона. Это и продемонстрировано в программе.

При конкретизации шаблонного определения функции необходимо, чтобы при вызове функции типы фактических параметров, соответствующие одинаково параметризованным формальным параметрам, были одинаковыми. Для определенного выше шаблона функций с прототипом

```
template <class  E> void swap(E,E);
```

недопустимо использовать такое обращение к функции:

```
int  n  =  4;   double d =  4.3;
swap(n,d); // Ошибка в типах параметров.
```

Для правильного обращения к такой функции требуется явное приведение типа одного из параметров. Например, вызов:

```
swap(double(n),d); // Правильные типы параметров.
```

приведет к конкретизации шаблонного определения функций с параметром типа *double*.

При использовании шаблонов функций возможна перегрузка как шаблонов, так и функций. Могут быть шаблоны с одинаковыми именами, но разными параметрами. Или с помощью шаблона может создаваться функция с таким же именем, что и явно определенная функция. В обоих случаях "распознавание" конкретного вызова выполняется по сигнатуре, т.е. по типам, порядку и количеству фактических параметров.

8. Тестирование

Тестирование и диагностирование объектно-ориентированного программного обеспечения является сложной задачей, не имеющей общего подхода, для решения которой не существует каких-то общих соглашений, приемов.

Определим проблемы тестирования, связанные с данными свойствами:

- *инкапсуляция* создает проблему доступа к данным, так как инкапсулированные в класс данные доступны только через операции. В процессе тестирования это может создать определенные проблемы с анализом значений обрабатываемых данных (некоторые методы не видны, т.е. закрыты для доступа средствами языка программирования). Закон инкапсуляции гласит, что внутреннее устройство объекта никого не интересует и должно быть скрыто внутри объекта, а данные и поведение прочно соединены в самом объекте. Чтобы обойти эту ситуацию и протестировать работу закрытого метода типа *private* (приватный), его тип определяют как *public* (общим). Т.е. теперь все внутреннее устройство объекта становится доступно тесту. Но такой подход может создать ряд непредвиденных проблем, например объектам одного типа становятся доступны данные и методы объекта другого типа и т.д.;

- *наследование* ставит вопросы о повторении тестирования наследуемых операций. Допустим, операция А принадлежит базовому классу, и она протестирована. Операция В принадлежит производному классу и вызывает операцию А. Встает вопрос - должна ли операция А тестироваться повторно? Решение этой проблемы частично может быть найдено в определении специальных отладочных методов, возвращающих информацию о состоянии класса, или в использовании низкоуровневых отладчиков программного кода, что так же создает определенные трудности при тестировании;

- *полиморфизм* приводит к неоднозначности с вызовом операций, которая может быть разрешена только на этапе выполнения. Следовательно, заранее построить и спланировать набор тестов для тестирования вычислительных алгоритмов становится просто невозможным. Даная проблема обусловлена, с одной стороны, вызовом виртуальных функций, а с другой тем, что программа манипулирует объектами таким образом, что класс, которому они принадлежат, заранее неизвестен. Тем не менее, концепция полиморфизма снимает ограничения, связанные с жестким контролем типов данных и позволяет гибко настроить код программы, что является мощным средством разработки программного обеспечения.

8.1. Планирование работ по тестированию

В процессе тестирования программного обеспечения осуществляются следующие виды деятельности:

Ручной прогон. На этом шаге программист с помощью карандаша и листа бумаги моделирует прохождение данных через программу. При изучении текста программы от начала до конца, трудно проверить всевозможные комбинации данных. Самое большее, что можно сделать на практике, - проверить всевозможные типы или наиболее вероятные наборы комбинаций данных. Если они дают правильные результаты, предполагается, что непроверенные комбинации также дадут правильные результаты.

Проектирование тестов. Является наиболее ответственным процессом. Очень часто тест создается вручную. Иногда применяют генераторы тестовых данных - специальные программы, формирующие данные в соответствии со спецификациями, задаваемыми программистом. Тестовые данные могут систематически или случайно выбираться из другого заданного набора данных для уменьшения их общего количества.

Выполнение тестов. На этом этапе осуществляется проверка всех возможных алгоритмов специально подготовленными тестами, а также выявляется, насколько интерфейс программы выдержит реальную нагрузку. Проблема заключается в том, что тестирование происходит на очень ограниченных объемах данных. Когда база данных будет насчитывать десятки, а то и сотни тысяч записей, скорость выполнения запросов пользователей может стать неприемлемой.

Изучение результатов тестирования. Выявление и устранение ошибок часто имеет циклический характер. Устранение одной ошибки может порождать другие ошибки. Особенно это касается работы с глобальными переменными, которые коварны тем, что нельзя сказать с полной уверенностью, что где-то на нижнем уровне подпрограмм изменение состояния переменной не приведет к новой ошибке

8.2. Тестирование аналитических и проектных моделей

Разработчики обычно моделируют создаваемые программные продукты, поскольку модель помогает им глубже понять решаемую задачу, а также в силу того, что они помогают успешно справляться со сложностью разрабатываемых ими систем. Модели аналитической и проектной информации в конечном итоге будут использованы для того, чтобы направить усилия по реализации проекта в правильное русло. Если модели обладают высоким качеством, они вносят неоценимый вклад в реализацию проекта, по если в них содержатся ошибки, они настоль же вредны. В этой главе рассматривается понятие целенаправленной проверки, представляющей собой метод расширенной проверки моделей по мере их создания и контроля этих моделей на предмет соответствия требованиям проекта. Основной недостаток стандартных ревизий заключаются в том, что центр тяжести проверки они переносят главным образом на то. что уже *имеется* (в модели), а не на то, что в ней *должно быть.* В ревизиях не предусмотрены средства систематического обнаружения того, чего не хватает в программном

продукте. Даже проверки Фагана, в которых применяются контрольные списки для обеспечения дальнейшей легализации процесса, не предусматривают средств для определения, чего в модели не хватает.

Целенаправленная проверка применяет перспективу тестирования на очень ранних стадиях процесса разработки. Традиционно тестирование начиналось на уровне реализации модулей и продолжалось по мере того, как сегменты программных кодов объединялись во фрагменты большего размера до тех пор, пока не вся система не оказывалась готовой к тестированию. Рассмотрим ситуацию, когда система представлена в виде аналитической или проектной информации. Новая версия традиционной *V*-образной модели тестирования, соотносит повторяющиеся применения целенаправленных проверок с различными уровнями тестирования.

Целенаправленная проверка требует затрат дорогостоящих ресурсов и внимания со стороны персонала, работающего нал проектом. Есть ли практический смысл в ее проведении? Исследования показывают, что отношение затрат на обнаружение и устранение неисправностей на ранней стадии процесса разработки к затратам по их обнаружению и устранению на этапе компиляции или системных испытаний колеблется в широких пределах. Например, устранение ошибки, обнаруженной во время системных испытаний, может стоить на два порядка дороже, нежели устранение той же ошибки на этапе анализа. Таким образом, даже умеренные затраты усилий на тестирование модели могут принести большую экономию.

8.3. Основы тестирования классов

Основным элементом объектно-ориентированной программы является класс. **Тестирование классов** охватывает виды деятельности, ассоциированные с проверкой реализации класса на точное соответствие спецификации этого класса. Если реализация корректна, то каждый экземпляр этого класса ведет себя подобающим образом.

Тестирование классов в первом приближении аналогично тестированию модулей в традиционных процессах тестирования — и для того, и для другого характерны одни и те же задачи, требующие решения. Тестирование классов должно решать некоторые вопросы комплексных испытаний, поскольку каждый объект определяет уровень области действия, в которой происходит взаимодействие множества методов на множестве атрибутов экземпляров. Основное внимание в данном случае уделяется тестированию классов в режиме прогона тестовых случаев. Основная цель заключается в том, чтобы описать базовые элементы и стратегии тестирования классов, при этом мы сосредоточимся на тестировании относительно простых классов.

Мы полагаем, что тестируемый класс обладает полной и корректной спецификацией, и что он успешно прошел тестирование в контексте моделей.

Если спецификация представлена в нескольких формах, то мы полагаем, что эти формы согласуются между собой, и требуемую информацию можно брать из той формы, которая окажется наиболее полезной в качестве основы для разработки тестовых случаев для конкретного класса. Для построения тестовых случаев мы отдаем предпочтение максимально формализованным спецификациям.

8.3.1. Способы тестирования классов

Эффективное тестирование программных колов конкретного класса достигается при помощи ревизий или выполнения тестовых случаев. В ряде ситуаций ревизия может оказаться вполне жизнеспособной альтернативой тестированию в режиме исполнения тестовых случаев, однако по сравнению с этим способом тестирования для ревизии характерны два следующих недостатка:

- При проведении ревизии возможны ошибки, обусловленные человеческим фактором.
- Ревизии требуют значительно больших затрат усилий, нежели регрессионное тестирование, при этом нередко для ее проведения требуются почти такие же затраты ресурсов, что и на полномасштабное тестирование.

В то время как тестирование в режиме прогона тестовых случаев может преодолеть указанные выше недостатки, существенные усилия могут потребоваться для отбора подходящих тестовых случаев и для разработки тестовых драйверов. В некоторых ситуациях трудозатраты, необходимые для построения тестового драйвера для конкретного класса, на несколько порядков превышают трудозатраты на разработку самого класса. В этом случае потребуется дать оценку затрат и выгоды, полученной от тестирования этого класса "за пределами" системы, в которой он будет применяться.

Такая ситуация характерна не только для объектно-ориентированного программирования. Подобная ситуация возникает в рамках традиционной процедурно-ориентированной разработки по отношению ко многим подпрограммам, вызываемым программами, находящимися на более высоком уровне структурной диаграммы.

После идентификации тестовых случаев для класса мы должны реализовать тестовый драйвер, обеспечивающий прогон каждого тестового случая, и запротоколировать результаты каждого такого прогона. Тестовый драйвер создает один или большее число экземпляров класса, осуществляющего прогон тестовых случаев. Важно не упускать из виду, что тестирование классов производится путем построения экземпляров этих классов и тестирования поведения построенных экземпляров. Тестовый драйвер может принимать множество форм. Мы отдаем предпочтение форме автономного "тестирующего"' класса перед всеми другими, поскольку он

предлагает удобную организацию упрощения драйверами и наследованием и с успехом может использоваться для улавливания общих для них свойств.

8.3.2. Оцениваемые факторы тестирования классов

Прежде чем приступать к тестированию того или иного класса, потребуется определить, тестировать ли его в автономном режиме как модуль или каким-то другим способом, как более крупный компонент системы. Мы принимаем решение на основе следующих факторов:

- Роль этого класса в системе, в частности, степень связанного с ним риска.
- Сложность класса, измеряемая количеством состояний, операций и связей с другими классами.
- Объем трудозатрат, связанных с разработкой тестового драйвера для тестирования этого класса.

Если какой-либо класс должен стать частью некоторой библиотеки классов, целесообразно выполнять всестороннее тестирование классов, даже если затраты на разработку тестового драйвера окажутся высокими, поскольку очень важным является ею корректное функционирование. Написание программ тестирования классов не влечет за собой принципиальных трудностей, поскольку по замыслу системы, они не должны взаимодействовать с другими классами.

8.3.3. Построение тестовых случаев для тестирования классов

Рассмотрим, как производится выявление и построение тестовых случаев, предназначенных для тестирования классов. Во-первых, мы выясним, как производится идентификация тестовых случаев на базе спецификации класса, сформулированной на языке OCL. Затем мы посмотрим на процесс построения тестового случая, взяв за основу диаграмму переходов.

Отбор необходимых тестовых случаев обычно производится на основании спецификации класса, причем сами спецификации могут быть сформулированы различными способами. Для этой цели применяется язык OCL, естественный язык и/или диаграммы переходов Тестовые случаи могут быть определены на базе реализации класса, однако использование только этого подхода может привести к распространению ошибок, которые разработчик класса допустил при интерпретации спецификации класса *в* процессе реализации программного обеспечения. Мы предпочитаем разрабатывать тестовые случаи, сначала беря за их основу спецификацию, а затем добавляя к ним тестовые случаи, необходимые для тестирования границ, привнесенных реализацией. Если в тестируемом классе спецификация отсутствует, до начала тестирования потребуется прибегнуть к

его "возвратному проектированию" и передать его на ревизию разработчикам.

8.3.4. Построение тестового драйвера

Тестовый драйвер представляет собой программу, которая осуществляет прогон тестовых случаев и сбор полученных при этом результатов

8.4. Тестирование взаимодействия и функционирования компонентов

8.4.1. Тестирование взаимодействий объектов

Объектно-ориентированная программа содержит совокупность объектов, взаимодействие которых приводит к решению конкретной проблемы. Способы взаимодействия этих объектов определяют то. что делает программа и, соответственно, правильность выполнения программы. Например, в заслуживающем доверия экземпляре примитивного класса ошибок может и не быть, однако если услуги этого экземпляра не используются должным образом другими компонентами программы, то сама эта программа содержит ошибки. Таким образом, для правильности программы весьма критично корректное сотрудничество, или *взаимодействие объектов.*

Основное назначение тестирования взаимодействий состоит в том, чтобы убедиться, что происходит правильный обмен сообщений между объектами, классы которых уже прошли тестирование в автономном режиме. Тестирование взаимодействий может быть выполнено над взаимодействующими объектами, вложенными в прикладную программу, или на примере взаимодействующих объектов в среде, предоставленной дли этой цели специальным средством отладки.

Прежде всего, подробно описывается, что собой представляют взаимодействия объектов и как взаимодействия идентифицируются в интерфейсе конкретного класса. Затем мы рассматривается тестирование взаимодействий вне контекста некоторой конкретной прикладной программы. И, наконец, анализируются некоторые из трудных для решения проблем, которые возникают при тестировании взаимодействий в контексте конкретной прикладной программы, и подходы к их решению.

8.4.2. Выбор тестовых случаев

Исчерпывающее тестирование, другими словами, прогон каждого возможного тестового случая, покрывающего каждое сочетание значений — вне всяких сомнений, надежный подход к тестированию Однако во многих ситуациях количество тестовых случаев достигает таких больших значений, что обычными методами с ними справиться попросту невозможно. Если имеется принципиальная возможность построения такого большого количества тестовых случаев, па построение и выполнение которых не хватит никакого времени, должен быть разработай систематический метод определения, какими из тестовых случаев следует воспользоваться. Если есть выбор, то мы отдаем предпочтение таким тестовым случаям, которые позволят найти ошибки, в обнаружении которых мы заинтересованы больше всего. Если у нас нет никакой предварительной информации, то, по-видимому, наилучшее, что можно предпринять в подобного рода ситуации — это случайный выбор. Рассмотрим общие принципы выборки, после чего применим их к тестированию взаимодействий.

При любом подходе к тестированию мы заинтересованы в том, чтобы систематически повышать уровень покрытия. Если тестировщик просто создает тестовые случаи без достаточного их обоснования, то увеличение их количества часто приводит на более поздних стадиях отладки к повторению проверок функциональных средств, которые уже подвергались тестированию. Современные методы позволяют получить строго очерченные наборы тестовых случаев и четко определенные методы увеличения степени покрытия.

Существуют различные способы определения, какой тестовый случай следует выбирать. Метод, который рассматривается первым, использует простой процесс выбора, основанный на распределении вероятностей. Распределение вероятностей определяет для каждого значения совокупности набор допустимых значений и вероятность выбора каждого из этих значений. В условиях равномерного распределения вероятностей каждое значение совокупности получает одно и то же значение вероятности выбора.

Мы можем остановить свой выбор на расслоенной выборке, в условиях которой тесты выбирают из некоторой последовательности категорий. Расслоенная выборка — это набор выборок, в котором каждая выборка представляет конкретную подсовокупность - например, мы можем провести отбор тестовых случаев, которые, как мы точно знаем, заставят работать каждый компонент архитектуры. Совокупность тестов разделена на поднаборы таким образом, что каждый поднабор содержит все тесты, которые затрагивают конкретный компонент. Выборка производится на каждом поднаборе независима от других.

Выборочный метод предлагает алгоритм для формирования тестового набора из множества возможных тестовых случаев. Это, однако, не выдвигает задачу определения совокупности тестовых случаев на первый

план. Процесс тестировании предусматривает определение совокупности тестов, в которых мы заинтересованы - например, функциональные тестовые случаи — с последующим определением метода выбора, какие из этих тестовых случаев будут построены и выполнены.

8.4.3. Тестирование протоколов

По мере того как некоторый объект вступает во взаимодействие с другими объектами, увеличивается количество сообщений, которые он получает. В соответствии со спецификацией, эти сообщения должны быть упорядочены в определенную последовательность. Тестирование по протоколу проверяет реализацию на соответствие спецификации. Различные протоколы, в которых принимает участие тот или иной объект, могут быть логически выведены из пред- и постусловий, регламентирующих выполнение отдельных операций, объявленных в классе этого объекта. Идентифицирующая последовательность вызовов методов, в которую объединяются методы, чьи постусловия удовлетворяют предусловиям следующего метода, образует протокол. Такие последовательности намного легче обнаружить на диаграмме состояний конкретного класса, нежели искать их. сопоставляя письменные формулировки пред- и постусловий.

Тестовый набор, предназначенный для тестирования взаимодействий, содержит тесты каждого протокола. По существу, это особая форма тестирования жизненного цикла. Каждый протокол представляет собой жизненный цикл тестируемых объектов классов в сочетании с другими классами. Каждый протокол соответствует последовательности состояний, которая начинается с исходных состояний двух объектов (как показано на диаграммах этих двух классов), из последовательностей состояний каждого объекта и завершается заключительными состояниями (опять-таки, обозначенными на диаграммах состояний). Соответствующий тестовый пример проводит эти два объекта через одну полную последовательность методов.

8.4.4. Тестовые шаблоны

Тестовые шаблоны - это проектные шаблоны, предназначенные для тестирования программных продуктов. Тестовые шаблоны захватывают и многократно используют знания из области проектирования программного обеспечения, которые получили широкое распространение в среде проектировщиков объектно-ориентированных программных продуктов. Каждый шаблон представляет собой конкретную конфигурацию взаимодействия некоторой совокупности объектов, которые образуют в масштабах всей разработки определенный кластер. Описание шаблона

содержит требования к контексту, в котором этот шаблон должен рассматриваться, совокупность факторов, которые играют существенную роль в анализе, проводимом с целью выбора компромиссных решений, а также инструкции, как построить соответствующие объекты. Для описания этих шаблонов применяются те же форматы, которые используются в среде разработчиков, однако при этом некоторым разделам описания придается особый смысл.

Успешно используется идея связывания тестовых шаблонов с конкретными проектными шаблонами. Когда разработчик прибегает к помощи конкретного проектного шаблона для структурирования некоторой части системы, тестировшик (в роли которого может выступать другой разработчик) знает, какой тестовый шаблон следует использовать для структурирования тестовых кодов.

9. Отладка

Существуют отладчики двух типов: отладчики пользовательского режима и отладчики уровня ядра.

Большинство программистов для отладки своих приложений применяют отладчики пользовательского режима. Именно такой отладчик встроен в среду *Microsoft Visual Studio*. Как следует из его названия, он служит для отладки приложений, которые работают в пользовательском режиме. С помощью этого встроенного средства можно осуществлять отладку программного кода пользовательского режима. В том числе динамические библиотеки, сервисы и т.п. В основу большинства таких отладчиков положен специальный отладочный *Application Programming Interface* (*API*), что переводится как интерфейс программирования для приложений. При запуске приложения с помощью API оно переходит в режим отладки. При этом отладчик получает полный контроль над выполнением такого приложения и полный доступ к его виртуальному адресному пространству. Кроме того, все процессы, запущенные отлаживаемым приложением, отладчик также может контролировать. В особую категорию следует отнести отладчики для виртуальных машин. Как правило, программы, выполняемые в виртуальной машине, должны отлаживаться с помощью *API* этой среды. Виртуальные машины интерпретируемых языков программирования (таких как *Java*, C#, *Managed* C++, *Perl*, *Visual Basic* и т.п.) представляют такое *API*.

Фактически это не отладчики уровня ядра, а уровня пользователей, поскольку используют не *API* операционной системы, а *API* виртуальной машины.

Код драйверов и ядра ОС позволяют отлаживать отладчики уровня ядра. Их используют, как правило, разработчики драйверов. Эти отладчики

путем прямого управления процессором управляют ходом выполнения программы (ядра и драйверов).

Поэтому при отладке на точке останова отлаживаемой программы происходит остановка всех процессоров в системе. В состоянии активности остается только отладчик. С этим связаны некоторые неудобства при их использовании. Например, в случае остановки отлаживаемого приложения невозможно воспользоваться другой программой.

Используя удаленную отладку, можно обойти данное ограничение. Для этого отладчик уровня ядра запускается на одном компьютере, который будет по-прежнему полностью блокировать систему. На другом компьютере запускается пользовательский интерфейс отладчика и отлаживаемое приложение. Интерфейс функционирует не на уровне ядра, а на пользовательском уровне, поэтому программист может пользоваться дополнительными утилитами, справочной информацией и т.п.

Следует отметить, что с помощью отладчиков уровня ядра можно отлаживать и код на пользовательском уровне, поскольку данные отладчики имеют доступ ко всем ресурсам компьютера. Это менее удобно, чем использование разработанных специально для этих целей отладчиков пользовательского режима. Данный метод часто используется взломщиками программного обеспечения с целью обойти защиту программы. Кроме того, такое использование оправдано в том случае, если есть необходимость отследить взаимодействия кода пользовательского режима и ядра ОС. В процессе отладки отслеживаются оба типа кода одновременно.

Существуют различные инструменты диагностирования, встроенные в большинство сред проектирования программного обеспечения.

Наиболее простым и одновременно очень мощным инструментом диагностирования является точка останова. Этот инструмент позволяет остановить выполнение программы при наступлении некоторого события. В простейшем варианте программа останавливается, когда выполнение достигает заданной строчки исходного кода. При наступлении такого события разработчик может изучить содержимое переменных, памяти процесса, стека вызова и т.п.

Использование точек останова на инструкциях является самым простым методом поиска ошибок, но не лишенным при этом недостатков. Пусть точку останова в процессе отладки поставили в середине большого цикла. Тогда программа будет прерываться на каждой итерации цикла до тех пор, пока не будет достигнута нужная ему итерация, что потребует много времени. Дальнейшее развитие этого направления поиска ошибок – использование условных точек останова. Отличие условных точек останова от обычных в том, что отладчик самостоятельно проверяет некоторое условие каждый раз, когда он достигает точки останова. Отладчик либо продолжает выполнение программы, либо приостанавливает ее работу в зависимости от результатов проверки.

Точки останова, описанные выше, позволяют отслеживать зависимости программы по управлению в процессе ее исполнения. Точки останова при этом ставятся на инструкции. Для того чтобы отследить зависимости по данным применяются точки останова, которые приостанавливают работу программы, если ячейка памяти была изменена. Данный инструмент применяется для поиска строчек программного кода, которые нарушают целостность данных. Такой тип точек останова использует аппаратные возможности процессора, что позволяет в полной мере реализовывать отладчики уровня ядра. Это обуславливает высокую производительность точек останова, которые ставятся на данных для отладчиков уровня ядра. Что касается точек останова на данных для отладчика уровня пользователя, то их возможности несколько скромнее, но и они позволяют выполнять такую задачу.

Общей проблемой метода отладки «грубой силы» является то, что его применение не предполагает процесса обдумывания отладки программы. Данный метод сводится к перебору вариантов и подозрению на каждой строке программного кода.

Экспериментально доказано, что средства отладки не всегда помогают процессу отладки (это справедливо как для начинающих, так и для опытных программистов), и бесцельные прогоны программы под отладчиком неэффективны по сравнению с вдумчивым подходом.

Одним из методов отладки, применяющих вдумчивый подход, является индуктивный анализ, означающий движение от частного к целому. Смысл метода заключается в том, что, отслеживая симптомы ошибки и взаимосвязи между симптомами, ошибку можно найти значительно быстрее.

Процесс индукции разбивается на следующие шаги:

1) Определение данных, имеющих отношение к ошибке.
2) Организация данных.
3) Выдвижение гипотезы.
4) Доказательство или опровержение гипотезы.

В плохо организованной системе диагностики анализу подвергаются все имеющиеся данные, что приводит к неоправданному увеличению сложности процедуры диагностирования и локализации ошибок.

На первом шаге индуктивного метода должны быть перечислены все действия, свидетельствующие о правильном выполнении программы и всех ее неправильных действиях. На втором шаге осуществляется структурирование данных, имеющих отношение к ошибке, с целью выявления некоторых закономерностей. Особое внимание на этом шаге следует обратить на противоречивость, когда ошибка проявляется при правильных входных значениях и отсутствует при вводе неправильных.

В основу индуктивного метода диагностирования положена теория взаимосвязи между признаками ошибки и гипотезами о ее причинах. Если выдвинуть гипотезу не представляется возможным, то необходимы

дополнительные данные, которые можно получить с помощью дополнительных тестов.

Путем сравнения гипотезы с первоначальными симптомами ошибки или данными гипотеза доказывается или опровергается. Основными критерием корректности гипотезы является то, что она полностью объясняет существование выявленных симптомов ошибки. Если такое объяснение получить не удалось – гипотеза считается неверной, или, как минимум, неполной.

Другой метод вдумчивого диагностирования – дедуктивный. Данный метод работает на основании некоторых общих теорий и предпосылок и позволяет, используя операции исключения и уточнения, прийти к определенному заключению, то есть обнаружить место ошибки. Он состоит из следующих этапов:

1) Перечисление возможных причин и гипотез.
2) Использование данных для исключения возможных причин.
3) Уточнение выбранной гипотезы.
4) Доказательство выбранной гипотезы.

На первом шаге составляется список всех возможных причин возникновения проявления ошибки в процессе тестового прогона. С помощью списка этих причин можно структурировать и анализировать имеющиеся в распоряжении данные. На втором шаге проводится тщательный анализ данных, в результате которого из списка причин исключаются все возможные причины, кроме одной. Если в процессе анализа будут исключены все причины, то это говорит о том, что потребуются дополнительные тестовые прогоны для получения дополнительных данных с целью выдвижения новых гипотез возникновения ошибок программе. В случае, когда остается более чем одна причина, то из всего множества выбирается наиболее вероятная из них или проводятся дополнительные тестовые прогоны с целью более тщательного анализа данных.

Выявленная в процессе анализа причина может быть определена верно, но при этом может в недостаточно полной мере отражать специфику ошибки. Например, ошибки, возникающие при работе с файлами при обращении к последней записи файла. Например, ошибка записи, проявившаяся в силу некоторых особенностей работы файловой системы, когда происходит затирание признака конца файла последней записью буфера.

Отдельным направлением в области отладки программного обеспечения являются методы, направленные на локализацию конкретных видов ошибок.

Таким образом, современные средства разработки программного обеспечения обладают достаточно мощными инструментами тестирования и отладки, тем не менее, задача выявления ошибок в программе остается сложной и ресурсоемкой.

Список литературы

1. Брюс Эккель Философия С++. Введение в стандартный С++ Издательство: Питер, 2004 г.
2. Брюс Эккель, Чак Эллисон Философия С++. Практическое программирование Издательство: Питер, 2004 г. Мягкая обложка, 608 стр.
3. Бьерн Страуструп Язык программирования С++. Специальное издание Издательства: Бином, Невский Диалект, 2008 г. , 1104 стр.
4. Герберт Шилдт Полный справочник по С++ Издательство: Вильямс, 2007 г. 800 стр.
5. Х. М. Дейтел, П. Дж. Дейтел Как программировать на С++ Издательство: Бином-Пресс, 2010 г. 1456 стр
6. Лафоре Р. Объектно-ориентированное программирование в С++.Питер.2004, 928 стр.
7. Т. А. Павловская С/С++. Программирование на языке высокого уровня Издательство: Питер, 2009 г. Твердый переплет, 464 стр.
8. Т. А. Павловская, Ю. А. Щупак С++. Объектно-ориентированное программирование. Практикум Издательство: Питер, 2008 г. Мягкая обложка, 272 стр.
9. Е. В. Пышкин Основные концепции и механизмы объектно-ориентированного программирования Издательство: БХВ-Петербург, 2005 г. Твердый переплет, 640 стр.
10. Стивен Дьюхерст С++. Священные знания Издательство: Символ-Плюс, 2007 г. , 240 стр.