Содержание

Объектно-ориентированное мышление

Объектно-ориентированное программирование (ООП) основная методология программирования 90-х годов. Она является продуктом 25 летней практики и включает ряд языков: Simula 67, Smalltalk, Lisp, Clu, Actor, Eiffel, Objective C, C++. Это стиль программирования, который фиксирует поведение реального мира таким способом, при котором детали его реализации скрыты.

Почему ООП так популярно:

- надежда, что ООП может просто и быстро привести к росту продуктивности и улучшению надежности программ, помогая тем самым разрешить кризис в программном обеспечении;
- желание перейти от существующих языков программирования к новой технологии;
- вдохновляющее сходство с идеями, родившимися в других областях.

ООП является лишь последним звеном в длиной цепи решений, которые были предложены для разрешения "кризиса программного обеспечения". Кризис программного обеспечения означает, что те задачи, которые мы хотим решить, опережают наши возможности.
Несмотря на то, что ООП действительно помогает в проектировании сложных и больших программ, ООП не панацея. Чтобы стать профессионалом в программировании, необходим талант, способность к творчеству, интеллект, знания, логика, умение строить и использовать абстракции и опыт, даже, если используются лучшие средства разработки.
ООП является новым понимание того, что называется вычислениями. Чтобы стать профессионалом в ООП, недостаточно просто добавить новые знания, необходимо полная переоценка привычных методов разработки программ.

Язык и мышление

Язык, на котором мы говорим, непосредственно влияет на способ восприятия мира. Это касается не только естественных языков, но и для искусственных языков - языков программирования. Чтобы эффективно использовать ООП требуется глядеть на мир иным способом, не как с точки зрения структурных языков. Само по себе использование C++ не делает программу ООП. "Программа фортрановского типа " может быть написана на любом языке".
Гипотеза Сапиро-Ворфа утверждает, что индивидуум, использующий некоторый язык, в состоянии вообразить или придумать нечто, что не может быть переведенным или даже понятым индивидуумом из другой языковой среды. Существует и прямо противоположная концепция: **принцип Черча**:

> *Любое вычисление, для которого существует эффективная процедура, может быть реализовано на машине Тьюринга.*

Это утверждение не доказуемо, поскольку не имеется строго определения "эффективная процедура". Но не найдено опровержения этого принципа.
В 1960-х годах было показано, что машина Тьюринга может быть смоделирована на любом языке, где есть условный оператор и оператор циклов. (Отсюда создалось утверждение, что оператор goto не нужен). Принцип Черча утверждает, что по своей сути все языки программирования идентичны. Но объектно-ориентированный подход решает проблему проще, приближая решение к естественному восприятию.

Новая парадигма

ООП часто называют новой *парадигмой* программирования. Другие парадигмы: директивная (структурное программирование - Pascal, C), логическая - Prolog, функциональное - Lisp, Effel.

Парадигмы в программировании определяют как проводить вычисления, как работа, выполняемая компьютером должна быть структурирована и организована.
Новички в информатике часто могут освоить парадигму лучше, чем опытные профессионалы, так как этот способ решения задач ближе к естественному восприятию.

Способ видения мира

Рассмотрим ситуацию из обыденной жизни. Например, Вам надо сообщить поздравить своего родственника, живущего в другом городе с днем рождения. Для это Вы идете на почту и посылаете телеграмму. Вы сообщаете оператору, что хотите переслать данный текст по некоторому адресу. И Вы можете быть уверены, что ваше поздравление попадет по нужному адресу.
Итак, для решения своей проблемы Вы нашли *агента* (почту) и передали ему *сообщение*, содержащее запрос. Обязанностью почты (или работников почты) будет удовлетворить Ваш запрос любым известным только им способом. Вас совершенно не интересует каким именно. Имеется некий *метод* - т.е. алгоритм, или последовательность операций, которые используют почтовые работники для выполнения запроса. Вам не надо знать какой конкретно метод используется.
Итак, **первым принципом ОО подхода** к решению задачи является способ задания действий.

> *Действие в ООП инициируется посредством передачи сообщений агенту (объекту), ответственному за действия. Сообщение содержит запрос на осуществление действия и сопровождается дополнительной информацией (аргументами), необходимой для его выполнения. Получатель - это агент, посылается сообщение. Если он принимает сообщение, то на него автоматически возлагается ответственность за выполнение указанного действия. В качестве реакции на сообщение получатель запустит некоторый метод, чтобы удовлетворить принятый запрос.*

Скрытие информации является важным принципом и в традиционных языках программирования. Чем же пересылка сообщения отличается в ООП и в традиционных подходах.
Первое отличие состоит в том, что имеется вполне определенный *получатель* - агент, которому послано сообщение. При вызове процедуры нет столь явно выделенного получателя.

Второе отличие состоит в том, что *интерпретация* сообщения (а именно метод, вызываемый после приема сообщения) зависит от получателя и является различной для различных получателей. Вы можете попросить своего товарища, летящего в город, где живут ваши родственники, поздравить их, и метод, который он изберет для решения этого запроса будет отличаться от того, который использовали на почте. Хотя родственники будут поздравлены. Если же Вы попросите коменданта общежития поздравить Ваших родственников, то у нее вероятно вообще не найдется метода для решения этой задачи, а если она и примет сообщение, то выдаст диагностическое сообщение об ошибке.

Таким образом, различие между вызовом процедуры и пересылкой сообщения состоит в том, что в последнем случае существует определенный получатель и интерпретация (т.е. выбор подходящего метода, который запускается в ответ на сообщение) может быть различной для различных получателей. Обычно конкретный получатель неизвестен вплоть до выполнения программы, и , следовательно, не известен какой метод будет вызван. В таком случае говорят, что имеет место позднее связывание между сообщением и методом, который вызывается в ответ на это сообщение. Эта ситуация противопоставляется раннему связыванию на стадиях компоновки, присущих традиционному программированию.
Фундаментальной концепцией в ООП является понятие обязанности или *ответственности* за выполнение действий. Ваш запрос на поздравление родственников выражает лишь желаемый результат. Почта вольна сама выбирать средство для удовлетворения этого запроса. Полный набор обязанностей,

связанных с определенным объектом называют протоколом.
Различие между взглядом структурного подхода и ООП можно выразить как

Задавайтесь вопросом не о том, что Вы можете сделать для своих структур, а о том, что структуры данных могут сделать для Вас.

Мы можем рассматривать почту как некоторое средство связи. Объединим в средства связи все, что позволяет связываться. Эта операция является **вторым принципом ООП:**

> *Все объекты являются представителями, или экземплярами, классов. Метод активизируемый объектом в ответ на сообщение, определяется классом, к которому принадлежит получатель сообщения. Все объекты одного класса используют одни и те же методы в ответ на одинаковые сообщения.*

Рис.1.1. Весь мир - это объекты, которые передают друг другу сообщения

О почте Вы знаете, больше, чем-то, что нужно, чтобы сделать запрос. Вы знаете, что у Вас попросят деньги, что Вам выдадут квитанцию. Все это справедливо и для магазинов, ресторанов. Поскольку категория Post более узкая, чем Service , то любое знание, которым Вы обладаете для категории Service будет справедливо и для Post.
Работников почты можно представить в виде, например, такой иерархии категорий. Работник почты - это продавец услуг, продавец услуг - это просто продавец, продавец - это человек, человек - это млекопитающее, млекопитающее - это животное, животное - это материальные объекты.
Принцип, в соответствии с которым знание о более общей категории разрешается использовать для более узкой категории, называется *наследованием.*

> *Классы могут быть организованы в иерархическую структуру с наследованием свойств. Дочерний класс (или подкласс) наследует атрибуты родительского класса (или надкласса), расположенного выше в иерархическом дереве.*

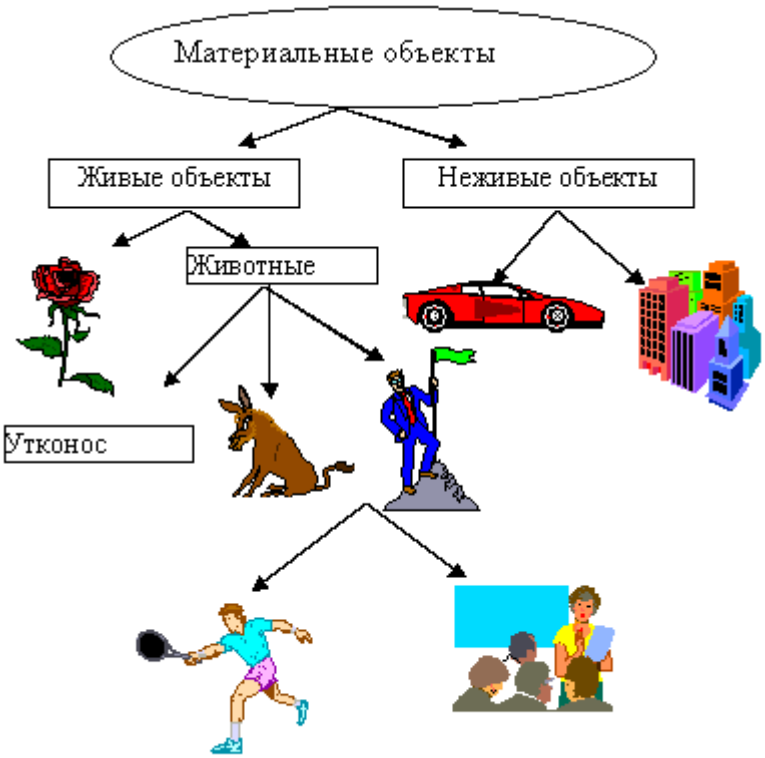

Рис. 1.2. Иерархическое дерево.

Утконос представляет проблему для структуры на рис. 1.2. Утконос млекопитающее, но откладывает яйца, следовательно, необходимо переопределить способ рождения. Таким образом, необходимо разрешать переопределять информацию, наследуемую из родительских классов. Для поиска метода, подходящего для обработки сообщения, используется следующее правило.

> *Поиск метода, который вызывается в ответ на определенное сообщение, начинается с методов, принадлежащих классу получателя. Если подходящий метод не найден, то поиск продолжается для родительского класса. Поиск продвигается вверх по цепочке родительских классов до тех пор, пока не будет найден нужный метод или пока не будет исчерпана последовательность родительских классов. В первом случае выполняется найденный метод, во втором - выдается сообщение об ошибке. Если выше в иерархии классов существуют методы с тем же именем, что и текущий, то говорят, что данный метод переопределяет наследуемое поведение.*

Тот факт, что Ваш друг студент и оператор почты будет реагировать по разному на просьбу поздравить родственников, является признаком *полиморфизма*.
Характеристики ООП

1. Все является объектом.
2. Вычисления осуществляются путем взаимодействия (обмена данными) между объектами, при котором один объект требует, чтобы другой объект выполнил некое действие. Объекты взаимодействуют, посылая и получая сообщения. Сообщения - это запрос на выполнение действия, дополненный набором аргументов, которые могут понадобиться при выполнении действия.
3. Каждый объект имеет независимую память, которая состоит из других объектов.
4. Каждый объект является представителем класса, который выражает общие свойства объектов.

5. В классе задается поведение (функциональность) объекта. Тем самым все объекты, которые являются экземплярами одного класса, могут выполнять одни и те же действия.
6. Классы организованы в единую древовидную структуру с общим корнем, называемую иерархией наследования. Память и поведение, связанное с экземплярами определенного класса, автоматически доступны любому классу, расположенному ниже в иерархическом дереве.

Вычисление и моделирование

Традиционная модель, описывающая выполнение программы на компьютере, базируется на дуализме - *процесс-состояние*. С этой точки зрения компьютер является администратором данных, следующим некоторому набору инструкций. В рамках ООП мы никогда не поминаем адреса ячеек памяти, переменные, присваивания.

> Вместо процессора, который перемалывает биты и шарит по карманам структур данных, мы получаем вселенную благовоспитанных объектов, которые любезно просят друг друга о выполнении тех или иных своих желаний.

При ООП мы считаем, что *вычисление есть моделирование*.
Конечно, объекты не могут во всех случаях реагировать на сообщение только тем, что вежливо обращаются друг к другу с просьбой выполнить некоторое действие. Это приведет к бесконечному циклу запросов, аналогично тому, как два джентльмена так и не вошли в дверь, уступая друг другу дорогу. На некоторой стадии объекты должны выполнять некоторые действия перед пересылкой запросов другим объектам.

Сложность

На заре информатики большинство программ писалось на ассемблере, затем с увеличением сложности задач стало невозможным помнить обо всех состояниях регистров, стеков и т.д. Появились такие языки как Fortran, Cobol. По мере дальнейшего развития методов программирования было замечено, что ту задачу, которую программист решает за один месяц, два программиста не решают за 1 месяц. "Рождение ребенка занимает 9 месяцев независимо, сколько женщин занято этим".
При традиционном подходе сложность программы усложняется большим числом перекрестных ссылок. Перекрестные ссылки обозначают зависимость одного фрагмента кода от другого. Даже самый независимый фрагмент кода часто невозможно понять в изоляции от других.
Главный механизм борьбы со сложностью - это *абстрагирование*, т.е. способность отделить логический смысл фрагмента программы от проблемы его реализации. От процедур к модулям, далее к абстрактным данным и, наконец, к объектам.
Процедуры и функции были первыми механизмами абстрагирования, примененными в языках программирования. Процедуры позволяют сконцентрировать в одном месте работу, которая выполняется многократно, а затем многократно использовать этот код. Кроме того, процедуры обеспечили возможность *маскировки информации*. Программист мог написать процедуру, которое используют другие программисты, не вдаваясь в детали. Но они не решали всех проблем: нет эффективного механизма маскировки деталей организации данных.

Рассмотрим пример организации стека.

Листинг 1.1.

```
int datastack[100];
int datatop=0;

void init() {                                   void push(int val)
```

```
datatop=0;
}
int top()
{ if(datatop > 0)
return datastack[datatop-1];
return 0;
}
{ if (datatop < 100)
datastack[datatop++]=val;
}
int pop()
{ if(datatop > 0)
return datastack[--datatop];
return 0;
}
```

Видно, что данные не могут быть локальными, потому что являются общими для всех процедур. Данные должны содержаться в глобальных переменных. Однако, если переменные глобальные нет способа ограничить доступ к ним или их видимость. Если имя переменной datastack, об этом должны знать все программисты, чтобы не создать переменные с таким именем. Переменные с таким именем запрещены, даже если они больше нигде не используются.
Модули можно рассматривать как улучшенный метод создания и управления совокупности имен и связанными с ними значениями. Если рассматривать модуль как абстрактную концепцию, то ее суть состоит в разбиении пространства имен на две части. Открытая (public) часть является доступной извне модуля, закрытая (private) часть доступна только внутри модуля. Типы, данные и процедуры могут быть к любому из этих двух частей.

1. Пользователя, который намеривается использовать модуль, следует снабдить всей информацией, необходимой, чтобы делать это корректно, и не *более того*.
2. Разработчика следует снабдить всей информацией, необходимой для создания модуля, и не *более того*.

Модули решают некоторые, но не все проблемы разработки программного обеспечения. Например, если захочется иметь два экземпляра стека. Механизм модуля, позволяя маскировать данные, не позволяет размножать экземпляры.

Абстрактный тип данных задается программистом. С данными абстрактного типа можно манипулировать так же, как и сданными типов, встроенных в систему. Пользователю разрешается создавать переменные, которые принимают значения из допустимого множества, и манипулировать ими, использовать имеющиеся операции. Чтобы построить абстрактный тип данных, мы должны иметь:

1. Экспортировать определение типа данных
2. Делать доступным набор операций, использующихся для манипулирования экземплярами типа данных.
3. Защищать данные, связанные с типом данных, чтобы с ними можно было работать только через указанные подпрограммы.
4. Создавать несколько экземпляров абстрактного типа данных.

Объекты - это почти абстрактный тип данных, но дополненный некоторыми новшествами.

ООП добавляет идеи к абстрактному типу данных. Главная из них - пересылка сообщений. Действие инициируется по *запросу,* обращенному к конкретному объекту, а не через вызов функции. К пересылки сообщения добавляется переопределение имен и совместного/многократного использования кода. *Интерпретация* сообщений меняется для различных объектов.
Добавляется механизмы *наследования* и *полиморфизма.* Наследование позволяет различным типам данных совместно использовать один и тот же код, приводя к уменьшению его размера и повышению функциональности. Полиморфизм обеспечивает, чтобы общий код удовлетворял конкретным особенностям отдельных типов данных.

Люди строят дома, машины, самолеты, собирая из отдельных деталей, не изготовляя заново для каждого отдельного случая. Можно ли сконструировать ПО таким же образом?
Многократное использование - это цель, к которой все стремятся, но редко достигают. ООП обеспечивает механизм для отделения существенной информации от специализированной (например, конкретный формат данных).

Резюме

ООП - это не просто набор некоторых свойств, добавленных в уже существующие языки. Это новый шаг в *осмыслении* задач и разработки ПО.
Программы - это совокупность взаимодействующих объектов. Каждый объект отвечает за конкретную задачу. Вычисление осуществляется посредством взаимодействия объектов. Объект получается в результате инкапсуляции состояния (данных) и поведения (операций), во многом аналогично абстрактному типу данных - АТД. Поведение объекта диктуется классом. Данные и поведение представлены в виде классов, экземпляры которых - объекты. Все экземпляры одного класса будут вести себя одинаковым образом в ответ на одинаковые запросы.

Объект проявляет свое поведение путем вызова метода в ответ на сообщение. Интерпретация сообщения зависит от объекта и может быть различной для различных классов объектов.
Для удобства создания нового типа из уже существующих типов, определенных пользователем используется механизм *наследования*. Классы могут быть организованы в виде иерархического дерева наследования.
С помощью уменьшения взаимозависимости ООП позволяет разрабатывать системы, пригодные для многократного использования.
ООП - это взгляд на программирование, сосредоточенный на данных; в котором данные и поведение жестко связаны. Для этого необходимо, чтобы объекты определялись вместе с сообщениями, на которые они могут реагировать.

Объектно-ориентированная парадигма предлагает новый подход к разработке программного обеспечения. Фундаментальная концепция объектно-ориентированной парадигмы состоит в *передаче сообщений объектам*.
Таким образом, Объектно-ориентированный язык должен обладать свойствами *абстракции, инкапсуляции, наследования и полиморфизма.*

1. Инкапсуляция с сокрытием данных - способность отличать внутреннее состояние объекта и поведение от его внешнего состояния и поведения
2. Абстракция - расширяемость типов - способность добавлять типы, определяемые пользователем для того, чтобы дополнить ими встроенные типы. Один из принципов ООП заключается в том, чтобы типы, определяемые пользователем, должны обладать теми же привилегиями, что и встроенные типы.
3. Наследование - способность создавать новые типы, повторно используя, описание существующих типов.
4. Полиморфизм с динамическим (поздним) связыванием - способность объектов быть ответственными за интерпретацию вызова функции

Принципы Объектно-ориентированного подхода:

1. Действие в объектно-ориентированном программировании инициируется посредством передачи сообщений объекту. Сообщение содержит запрос на осуществление действия. В качестве реакции на сообщение получатель запустит некоторый метод, чтобы удовлетворить принятый запрос.

2. Все объекты являются экземплярами, классов. Все объекты одного класса используют одни и те же методы в ответ на одинаковые сообщения.
3. Принцип наследования. Классы могут быть организованы в иерархическую структуру с наследованием свойств. Дочерний класс наследует атрибуты родительского класса.
4. Принцип полиморфизма. Объекты реагируют на одно и тоже сообщение строго специфичным для них образом.

Упражнения

1. Приведите пример иерархии из повседневной жизни со свойством, что следующий уровень является более специализированной формой предыдущего. Укажите пример иерархии без свойств наследования.
2. Возьмите задачу из повседневной жизни (типа поздравления родственников) и опишите ее решение в терминах агентов (объектов) и обязанностей.

Объектно-ориентированное проектирование

Само по себе проектирование на объектно-ориентированном языке не обеспечивает объектно-ориентированного программирования. Самое важное в ООП – техника проектирования, основанная на выделении и распределении обязанностей.
Объектно-ориентированное проектирование - это тема отдельного курса и можно отослать к книге Г. Буча "Объектно-ориентированный анализ и проектирование". В библиотеки ВГУЭС находится первое издание этой книги. Второе издание с примерами приложений на C++ можно заказать на Мистрале.

В лекции рассматривался пример проектирования системы - справочник по рецептам.
Первый шаг в проектировании - уточнение спецификации.
Второй шаг - идентификация компонент. Компонента - это просто абстрактная единица, которая может выполнять определенную работу. Компонента должна иметь небольшой набор четко определенных обязанностей и должна взаимодействовать с другими компонентами настолько слабо, насколько это возможно. Затем строим диаграмму взаимодействия. Компонента определяется в терминах *поведения и состояния.* Поведение - это набор действий, ею осуществляемых. Полное описание компоненты можно назвать протоколом. Состояние компоненты - это ее внутреннее содержание. Состояние может изменяться с течением времени.

Интерфейс и реализация модуля - принцип Парнаса.

Программист *знает*, как *использовать* компоненту, написанную другим программистом, но не должен знать, как она *реализована.*

Говорят, что компонента *инкапсулирует* поведение объекта, если она умеет выполнять некоторые действия, но подробности, как именно она это делает, скрыты. Это приводит к двум представлениям о программной системе. Вид со стороны интерфейса - это лицевая сторона. В интерфейсной части описывается, *что умеет делать* компонента. Вид со стороны реализации - это изнанка, она определяет, *как компонента выполняет задание*.

Разделение интерфейса и реализации является наиболее важной идеей в программировании. Итак **принцип Парнаса**

- Разработчик программы должен предоставлять пользователю всю информацию, которая нужна для эффективного использования приложения, и *ничего* кроме того.
- Разработчик программного обеспечения должен знать только требуемое поведение компоненты и *ничего* кроме того.

Классы и методы

Инкапсуляция

В программировании, основанном на абстрактных типах данных, информация сознательно прячется в небольшой части программы. Каждый объект имеет два лица. С внешней точки зрения АТД представляет собой совокупность операций, которые определяют поведение абстракций. Программист, который определяет этот АТД, видит значения переменных, которые используются для поддержания внутреннего состояния объекта.
Например, для абстрактного типа данных stack пользователь видит только описание допустимых операций – pop, push, top. С другой стороны программисту, реализующему stack, необходимо манипулировать конкретными структурами данных. Конкретные детали инкапсулированы в более абстрактный объект.
Каждый экземпляр имеет свою собственную совокупность переменных. Эти значения не должны изменяться клиентами напрямую, а только с помощью методов, ассоциированных с классом.
Объект является комбинацией состояния и поведения. Состояние описывается переменными экземпляра, поведение характеризуется методами. Снаружи клиенты могут узнать только о поведении объектов. Изнутри доступна полная информация о том, как методы обеспечивают необходимое поведение, изменяют состояние и взаимодействуют с остальными объектами.

Разновидности классов

Классы в ООП имеют несколько различных форм и используются для различных целей. Следующие категории охватывают большую часть классов.

- Управление данными
- Источник данных и посредники в передаче данных
- Классы для просмотра данных
- Вспомогательные или упрощающие проектирование классы.

Это не исчерпывающий список. Если оказывается , что класс разрывается между двумя категориями, то его можно разбить на два класса.
Классы администраторы данных Data Manager (часто называют Data или State) – это классы, основной обязанностью которых является поддержка данных или информации о состоянии чего-либо. Например, для игры в карты основная задача класса Card состоит в том, чтобы хранит масть и ранг карты. Классы администраторы – это фундаментальные блоки проекта, существительные.
Источник данных Data Source – это классы, которые генерируют данные. Посредники при передаче данных (Data Sinks), служат для приема и дальнейшей передачи данных (например, запись в файл). Источники и посредники не хранят внутри себя данные, в отличии от классов-администраторов, они генерируют их по запросу (источники данных) или обрабатывают их при вызове (посредники данных).
Классы просмотра также незаменимы почти в любом приложении. Все программы осуществляют вывод информации (обычно н экран). Т.е. можно изолировать внутренние данные и методы, осуществляющие их вывод. Полезно отделять собственно объект от его визуализации. Благодаря этому принципу системы, обеспечивающие вывод графической информации могут быть значительно упрощены. В идеальном случае модель не требует и не содержит информации о своем визуальном представлении. Это позволяет одну и ту же модель применить в разных приложениях.
К вспомогательным классам можно отнести те классы, которые не содержат полезной информации, но облегчают выполнение сложных заданий. Например, связный список карт – колода.

Пример: Игра в карты

Класс Card мало, что знает о своем предполагаемом использовании и может применяться в карточной игре любого типа.

Класс Card

Хранить масть и ранг карты
Возвращает цвет карты
Хранит состояние картинка вверх или вниз
Рисует карту на экране
Удаляет карту с экрана

Класс Card является администратором данных, который хранит и возвращает значения масти и ранга, и рисует карту. Рассмотрим уточнение класса Card

Класс Card

Suit возвращает масть и ранг карты
color- Возвращает цвет карты
faceUp - проверяет состояние картинка вверх или вниз
draw рисует карту на экране
erase - удаляет карту с экрана

Интерфейс и реализация

Идеи Парнаса в терминах объектов:

- Объявление класса должно обеспечивать клиента информацией, необходимой для успешной работы и ни какой другой.
- Методам должна быть доступна вся информация, необходимая для выполнения их обязанностей, и никакая другая.

Принцип Парнаса делит мир на две части. Имеется внешний образ наблюдаемый пользователем объекта, и мы будем называть это представление об объекте *интерфейсом*.(interface), оно описывает как объект взаимодействует с внешним миром. Обратная сторона объекта связана с его реализацией (implementation). Пользователю разрешен доступ только к тому, что описано в интерфейсной части. Реализация определяет, как достигается выполнение обязанностей, заявленных в интерфейсной части.

Классы и методы в языке C++

Классы C++ предлагают расширение предопределенных системных типов. Класс - это определяемый пользователем тип. Каждый класс представляет уникальное множество объектов и операций над ними (методов). Класс предоставляет средства для создания, манипулирования и уничтожения таких объектов.
Что представляет собой хороший класс? Нечто, имеющее небольшое и хорошо определенное множество действий. Нечто, что можно рассматривать как "черный ящик", которым манипулируют только посредством этого множества действий. Нечто, чье фактическое представление можно любым мыслимым способом изменить, не повлияв на способ использования множества действий. Нечто, чего

можно хотеть иметь больше одного.
В общем случае, класс представляется в следующей форме:

class имя класса{список членов};

Описание класса начинается с ключевого слова class. Список членов класса определяет собственные элементы класса.
При описании членов классов возможно указание атрибутов управления доступом к элементам классов. Такими атрибутами являются:

- *public:* члены класса видны извне класса
- *private:* соответствующие элементы могут использоваться только внутри класса

По умолчанию элементы класса имеют тип *private*. Указанный в описании спецификатор доступа распространяется на все последующие определения, пока не встретится другой спецификатор.

Поскольку пользователя часто интересует открытая часть класса, то в описании она должна идти первой.

Функция card(suit,int) в описании класса является уникальной – она имеет тоже имя, что и класс и не имеет возвращаемого значения. Эта функция называется *конструктором*. Она используется при инициализации создаваемых экземпляров класса. Методы draw и halfdraw описаны в стиле прототипа функции.

Листинг 3.1 Описания класса card на C++
```
enum suits {diamond, club, heart, spade};
enum colors {red, black};
class card {
public:
card(suits,int);
// доступ к атрибутам
colors color ();
bool faceUp ();
int rank ();
suits suit ();
// выполняемые действия
void draw (window &,int x, int y);
void halfdraw (window &,int x, int y);
void flip ();
private:
bool faceup;
int r; // ранг
suits s; // масть
};
```
Метод suit возвращает значение масти в s. A rank значение ранга r.

Файл для реализации данного класса должен обеспечить работу методов, описанных в интерфейсном файле.
```
#include “card.h”
card::card(suits sv, int rv)
{
```

```
s=sv; // масть инициализируется
r=rv; // ранг инициализируется
faceup=true; // начальное положение картинкой вверх
}
int card::rank() { return r; }
```

Комбинация имени класса и имени метода образует полное имя.

При программировании с использованием классов очень часто используется много маленьких функций. Это может страшно понизить эффективность, потому что стоимость вызова функции (хотя и вовсе не высокая по сравнению с другими языками) все равно намного выше, чем пара ссылок по памяти, необходимая для тела функции. Чтобы справиться с этой проблемой, был разработан аппарат inline-функций (встраиваемые функции). Компилятор преобразует вызов функции в ее код. Функция, определенная (а не просто описанная) в описании класса, считается inline. Кроме того, функцию можно определить с ключевым словом inline вне протокола класса.

```
inline int card::rank() { return r; }
```

Листинг 3.2. Описание класса card с inline методами

```
class card {
public:
// конструкторы
card(suits,int);
card();
card(const card &c);
// доступ к атрибутам
colors color ();
bool faceUp () {return faceup; }
int rank () { return r; }
suits suit () { return s; }
// выполняемые действия
void draw (window &, int x,int y);
void halfdraw (window &,int x, int y);
void flip () { faceup =!faceup; }
private:
bool faceup;
int r; // ранг
suits s; // масть
};
```

Многие компиляторы требуют описание переменных, используемых во встраиваемых функциях, до определения функций. Поэтому либо переносят поля private до public, либо выносят встраиваемые функции в файл реализации с описателем inline.

Язык C++ поддерживает неполное описание классов: *class* X. Неполное описание класса позволяет использовать ссылки на эти классы до их полного определения. Объекты типа класс могут использоваться как аргументы функций и как возвращаемые функциями значения.

Классы и методы в языке Java

Трудно сказать, возможно язык Java (не путать с технологией Java) можно описать как диалект языка C++. В них много общего, но и много отличий. Язык Java не имеет указателей, ссылок (точнее там все ссылки), структур оператора goto, функций (только методы) и перегрузки операторов. Он поддерживает

строки как примитивный тип данных и выполняет сборку мусора.
Сам Java язык общего назначения, но основное его использование в области Internet и сетевых распределенных приложений. Мы не затрагиваем эти аспекты совсем и только касаемся языка.
Рассмотрим класс на Java

Листинг 3.3.

```
class card {
// статические значения цветов и мастей
final public int red =0;
final public int black =0;
final public int spade =0;
final public int heart =1;
final public int diamond =2;
final public int club =3;
// поля данных
private boolean faceup;
private int r;
private int s;
// конструктор
card(int sv,int rv) {
s=sv; r=rv; faceup=false; }
// доступ к атрибутам карты
public int rank() { return r; }
public int suit() { return s; }
public int color () {
if( suit()==heart || suit()==diamond)
return red;
else
return black; }
public boolean faceUp() { return faceup; }
public void draw (Graphics g, int x, int y)
{
/* */
}
public void flip()
{
faceup != faceup;
}
};
```

Описание класса на языке Java очень похоже, за исключением нескольких моментов

- Отсутствие препроцессора, глобальные переменные, перечисляемые типы данных. Символьные константы могут быть созданы и инициализированы с помощью ключевого слова final. Такие значения не могут изменяться.
- Реализация методов проводится непосредственно внутри определения класса. Вместо разбиения описания класса на public и private эти ключевые слова присоединяются в явном виде к каждой переменной или методу.
- Логический тип данных, именуемый boolean вместо bool.
- За исключением конструкторов все методы должны иметь возвращаемое значение

Ключевое слово this

Нестатические функции-члены класса оперируют с тем объектом типа класса, из которого они вызваны. Ключевое слово *this* обозначает специальную локальную переменную, доступную в теле любой функции-члена класса, описанной без спецификации *static*. Переменная *this* не требует описания и всегда содержит указатель соответствующего объекта.

- *this->имя члена* указывает на объект, членом, которого он является.
- **this* представляет собой сам объект и, в зависимости от контекста, может быть лево- или правосторонней величиной
- *this* представляет собой адрес объекта.

Пример

```
class C {
int c1,c2;
public:
void init(int b) { c2=b; c1=b+1; }
C & inc() { c1++; c2++; return *this; }
void *adress() { return this; }
void print() { cout << c1 << c2; }
};
void main(void) {
C a;
a.init(10); a.print(); cout << " adress=" << a.adress<< " inc " << a.inc().print() << endl;
}
```

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

```
x* this;
```

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

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

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

В языке Java **this** обозначает не указатель на объект, а собственно объект.

Функции-члены

Метод в C++ , J++ принято называть функцией-членом класса, о пересылке сообщений говорят как о вызове функции-члена класса.

- Функции-члены - это, функции которые манипулируют данными-членами класса.
- Функции-члены имеют доступ ко всем полям своего класса.
- Функции-члены могут быть в закрытой, защищенной и открытой части класса.
- Функции-члены могут быть определены внутри или вне (C++) объявления класса. Функции-члены, определенные вне класса, могут быть сделаны inline.
- Функции-члены могут обращаться к полям или функциям-членам, объявленным после них.

- Функции-члены имеют неявно объявленную переменную this.
- Функции-члены могут быть static. Такие функции могут непосредственно обращаться и изменять статические поля класса. Статические Функции-члены класса не могут быть объявлены const или virtual. К таким функциям можно обращаться через имя класса, а не через имя конкретного экземпляра объекта.

```
Class Shape {
static int num;
Point center;
public:
Shape(Point c): center(c) { ++num; }
~Shape() { --num; }
static int Num() { return num; }
};
int Shape::num=0;
void main()
{
cout << Shape::Num(); // 0
Shape s;
cout << Shape::Num(); // 1
for(int n=Shape::Num(); n>0; n--) { }
}
```

- Функции-члены могут быть объявлены как const, что не позволяет им изменять значение и возвращать неконстантную ссылку или указатель на любое поле класса. Такие Функции-члены не могут быть статическими.

```
class Int {
int *v, size, top;
public:
int Pop();
int Top() const; // Не может изменять что-либо
};
void f(const int &s) {
int a =s.Top(); // Правильно: Top() константный член
int b=s.Pop(); // Ошибка : s не может быть модифицирован
}
```

```
class A {
int a;
public:
A(){a=10; }
Int & get() const; // Ошибка возвращает
ссылку
};
int & A::get() const { return a; }
int main(int argc, char* argv[]){
A b;
cout<< " " << b.get();
return 0;
}
```

Данные члены и управление доступом к элементам классов.

- Данные-члены - это набор взаимосвязанной информации, возможно различных типов, объединенной в один объект.
- Данные-члены могут находиться в закрытой (private), защищенной (protected) или открытой (public) части класса.
- Данные-члены могут иметь статический класс памяти (static). Поля, имеющие статический класс памяти, совместно используются всеми объектами класса. К ним возможен доступ через имя класса (с использованием операции разрешения доступа), а не через контекстный объект класса. Статические поля могут быть инициализированы; если нет, то инициализируется значением ноль.

- Данные-члены могут быть объявлены как const. Константные данные должны быть инициализированы в каждом определении конструктора. Имена полей и начальные значения заключаются в скобки, отделяются от списка аргументов конструктора двоеточием.
- Данные-члены могут быть переменными другого класса. В этом случае требуется явная инициализация, если только поля не имеют конструктора по умолчанию.

Способ создания и инициализация

Рассмотрим ряд моментов связанных с созданием и уничтожением. \

Первый это стек и куча. Вопрос о выделении памяти через стек или кучу связан с тем как выделяется и освобождается память. Разделяют автоматические и динамические переменные. Для автоматических переменных память создается при входе в процедуру или блок, управляющие этими переменными. При выходе из блока память (автоматически) освобождается. Многие языки используют термин статический (static) для обозначения переменных, автоматически размещаемых в стеке. (В данном случае речь не идет о переменных типа static в C++). Для динамического выделения памяти используется оператор new. Память выделяется по специальному запросу со стороны пользователя.

Автоматические переменные уничтожаются автоматически, динамические по запросу со стороны пользователя ключевым словом delete в C++. В Java нет освобождения памяти, осуществляется процесс сборки мусора – автоматически отслеживается ситуация, когда к объекту нет доступа.

Указатели – это эффективное средство работы с динамической информацией. В языке Java объекты представлены внутренним образом как указатели, но при этом никогда не используются в таком качестве.

Создание и инициализация в C++. Конструкторы.

Использование для обеспечения инициализации объекта класса функций вроде set_date() (установить дату) неэлегантно и чревато ошибками. Поскольку нигде не утверждается, что объект должен быть инициализирован, то программист может забыть это сделать, или (что приводит, как правило, к столь же разрушительным последствиям) сделать это дважды. Есть более хороший подход: дать возможность программисту описать функцию, явно предназначенную для инициализации объектов. Поскольку такая функция конструирует значения данного типа, она называется конструктором. Конструктор распознается по тому, что имеет то же имя, что и сам класс. Например:

```
class date {

date(int, int, int);

};
```

Метод конструктор автоматически и неявно вызывается каждый раз, когда создается объект, принадлежащий к соответствующему классу. Это происходит или во время объявления переменной или, когда объект создается динамически с помощью оператора new или когда применяются временные переменные.
Часто бывает хорошо обеспечить несколько способов инициализации объекта класса. Это можно сделать, задав несколько конструкторов. Например:

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

date(int, int, int); // день месяц год
date(char*); // дата в строковом представлении
date(int); // день, месяц и год сегодняшние
date(); // дата по умолчанию: сегодня
};

data::data(){
month=01, day=22, year=1999;
}

date today(4);
date july4("Июль 4, 1983");
date now; // инициализируется по умолчанию
```

Тело конструктора часто представляет собой последовательность операторов присваивания. Они могут быть заменены инициализаторами в заголовке функции.

```
Data(int x,int y,int z):month(x),day(y),year(z) {}
```

Динамическое создание переменной выглядит следующим образом.

```
Data *d, *g;
D=new data(1,3,1987);
G=new data;
```

Конструкторы подчиняются тем же правилам относительно типов параметров, что и перегруженные функции. Если конструкторы существенно различаются по типам своих параметров, то компилятор при каждом использовании может выбрать правильный.
Размножение конструкторов в примере с date типично. При разработке класса всегда есть соблазн обеспечить "все", поскольку кажется проще обеспечить какое-нибудь средство просто на случай, что оно кому-то понадобится или потому, что оно изящно выглядит, чем решить, что же нужно на самом деле. Последнее требует больших размышлений, но обычно приводит к программам, которые меньше по размеру и более понятны. Один из способов сократить число родственных функций - использовать параметры со значением по умолчанию, пример. В случае date для каждого параметра можно задать значение по умолчанию, интерпретируемое как "по умолчанию принимать: today" (сегодня).

```
class date {
int month, day, year;
public:
date(int d =0, int m =0, int y =0);
date(char*); // дата в строковом представлении
};
```

Констукторы и массивы объектов

Чтобы описать вектор объектов класса, имеющего конструктор, этот класс должен иметь конструктор, который может вызываться без списка параметров. Нельзя использовать даже параметры по умолчанию. Например:

```
table tblvec[10];
```

будет ошибкой, если нет конструктора без параметров так как для table::table(int sz=10) требуется целый параметр. Нет способа задать параметры конструктора в описании вектора. Чтобы можно было описывать вектор таблиц table, можно модифицировать описание table, например, так:

```
class table {
void init(int sz); // как старый конструктор
public:
table(int sz) { init(sz); } // как раньше, но без по умолчанию
table() { init(10); } // по умолчанию
};
```

Когда вектор уничтожается, деструктор должен вызываться для каждого элемента этого вектора. Для векторов, которые не были размещены с помощью new, это делается неявно. Однако для векторов в свободной памяти это не может быть сделано неявно, поскольку компилятор не может отличить указатель на один объект от указателя на первый элемент вектора объектов. Например:

```
void f() {
table* t1 = new table;
table* t2 = new table[10];
table* t3 = new table[10];
delete t1; // одна таблица
delete t2; // неприятность: 10 таблиц
delete[] t3;
}
```

Компилятор не может найти число элементов вектора из объема выделенной памяти, потому, что распределитель свободной памяти не является частью языка и может быть задан программистом. Конструкторы могут инициализировать массивы объектов класса таким образом, как и массивы встроенных типов. Максимальное число элементов может быть опущенно, если оно равно числу инициализирующих значений. Если максимальное число элементов больше числа значений, то оставшиеся значения инициализируются при помощи конструктора по умолчанию. Если это не конструктор по умолчанию, то все же значения должны быть указаны. Если заданы все значения для данного размера массива, то фигурные скобки при указании списка значений могут быть опущены.

```
class Phone {
int a,b,c;
public:
Phone(int a1,int b1,int c1):a(a1),b(b1),c(c1) {}
};

Phone office[]= { // Компилятор вычислит размер за нас
900, 800, 905,
678,456,546 };
Phone office[3]= { // Требуетс конструктор по умолчанию, которого у Phone нет
890, 790,343,
238, 279, 564
};
```

Деструктор

Определяемый пользователем тип чаще имеет, чем не имеет, конструктор, который обеспечивает надлежащую инициализацию. Для многих типов также требуется обратное действие, деструктор, чтобы обеспечить соответствующую очистку объектов этого типа. Имя деструктора для класса X есть ~X()

("дополнение конструктора"). В частности, многие типы используют некоторый объем памяти из свободной памяти, который выделяется конструктором и освобождается деструктором. Вот, например, традиционный стековый тип, из которого для краткости полностью выброшена обработка ошибок:

```
class char_stack {
int size;
char* top, * s;
public:
char_stack(int sz) { top=s=new char[size=sz]; }
~char_stack() { delete []s; } // деструктор
void push(char c) { *(top++) = c; }
char pop() { return *(--top);}
};
```

Когда char_stack выходит из области видимости, вызывается деструктор:

```
void f(){
char_stack s1(100), s2(200);
s1.push('a'); s2.push(s1.pop());
char ch = s2.pop();
cout << char(ch) << "\n";
}
```

Когда вызывается f(), конструктор char_stack вызывается для s1, чтобы выделить вектор из 100 символов, и для s2, чтобы выделить вектор из 200 символов. При возврате из f() эти два вектора будут освобождены.

Конструктор копирования

Объект класса без конструкторов можно инициализировать путем присваивания ему другого объекта этого класса. Это можно делать и тогда, когда конструкторы описаны. Например:

```
date d = today; // инициализация посредством присваивания
```

По существу, имеется конструктор по умолчанию, определенный как почленная (в ранних версиях C++ - побитовая) копия объекта того же класса. Если для класса X такой конструктор по умолчанию нежелателен, его можно переопределить конструктором с именем X(X&).
Семантика вызова по значению требует, чтобы локальная копия типа параметра создавалась и инициализировалась от значения выражения, переданного как фактический параметр. Для этого необходим *конструктор копии.* Компилятор предоставляет такой конструктор по умолчанию. Его сигнатура

```
stack::stack(const stack&);
```

Компилятор производит копирование путем почленной инициализации. Это не всегда применимо. В большинстве случаев, указатель является адресом объекта, удаляемого при выходе из контекста. Однако, код, в котором производится действие по дублированию значения указателя, а не объекта, на который он указывает, может быть ошибочным. Дело в том, что удаление воздействует на другие экземпляры, все еще предполагающие, что объект существует. Поэтому важно, чтобы класс имел свою собственную явно определенную копию конструктора.

```
Stack::stack(const stack& str) {
s=new char [str.max_len]; max_len=str.max__len; top=str.top;
```

```
memcpy(s,str.s,max_len);
}
```

Конструктор копирования вызывается при возвращении значения объекта из функции

```
stack f(stack r) // вызов конструктора копирования при создании копии
{
return r; // вызов конструктора копирования для создании копии при возвращении
}
```

Конструктор копирования вызывается в случае инициализации при объявлении.

```
Stack s(10); // конструктор с одним параметром
Stack d=s; // конструктор копирования.
```

Все хорошо разработанные классы должны иметь конструктор копирования, особенно те, которые имеют дело с динамической памятью, открытыми файлами, любыми указателями.

Создание и инициализация на языке Java

Основное различие между C++ и Java состоит в том, что Java использует автоматическую сборку мусора. Все переменные типа “объект” в языке Java получают значение типа null. Объекты создаются оператором new. В отличии от C++ в Java всегда используются () в конструкторе.

```
Card a=new card(card.spade,5); // создать 5 пик.

Card b=new card();
```

Понятие конструктора аналогично C++, за исключением того, что не поддерживаются инициализаторы. В отличии от C++ конструкторы Java могут вызывать другие конструкторы того же объекта.

```
class newclass
{
newclass(int i) { }
newclass(int i,int j) { this(i); }
};
```

Деструкторов в Java нет, но есть функция finalize. Она автоматически вызывается системой, когда обнаруживается, что объект больше не используется. Память занятая объектом помечается как свободная. Не известно когда будет, если вообще будет вызван метод finalize. Поэтоу не стоит на него полагаться, но можно его использовать как средство оптимизации.

Резюме

В C++ существуют специальные функции-члены класса, которые определяют как объекты класса создаются, инициализируются, копируются и уничтожаются. Важнейшие из них - конструкторы и деструкторы. Им свойственны многие черты обычных функций-членов класса, но существуют и свои особенности:

- Конструкторы и деструкторы не могут описываться как функции, возвращающие значение (даже типа void).
- Конструкторы могут иметь аргументы. В качестве параметров могут быть элементы, получающие значения по умолчанию. Деструктор не имеет параметров.
- Имя конструктора совпадает с именем класса. Имя деструктора - имя класса, которому предшествует символ ~.
- Нельзя получить адрес конструктора или деструктора.

- Если они явно не описаны, то конструкторы и деструкторы автоматически создаются компилятором.
- Конструктор не может быть вызван как обычная функция. Дестуктор может быть вызван как обычная функция с указанием полного имени.

```
Int s(100), *ps=new Int(50);
s.~Int(); // явное разрушение s
this->~Int(); // Явное разрушение *this (may be error this=ps ?)
```

- Компилятор автоматически вставляет вызовы конструктора и деструктора при описании и уничтожении объекта.
- Конструкторы могут быть и в закрытой, и в защищенной, и в открытой части класса. Если конструктор находится в закрытой части класса, то объект этого класса с использованием такого конструктора могут создавать только те, кто имеет доступ к закрытой части.
- Деструктор автоматически вызывается компилятором:
  - при выходе из области видимости;
  - при создании временных объектов для преобразования в выражениях или для передачи функциям аргументов;
  - когда возвращенные функцией значения более не нужны
  - при выполнении операции delete для объектов, размещенных в динамической памяти.
- Конструктор не может быть ни const, ни volatile, ни static, ни virtual.
- Конструктор может быть вызван тремя эквивалентными способами (пользователю предоставляется право выбора):

```
Int good(100);
Int bad=Int(100);
Int ff=100;
```

- Конструкторы используются для инициализации константных полей и полей, которые являются переменными класса. Константное поле должно быть инициализировано в каждом определении конструктора. Классы с константными полями должны иметь по крайне мере один конструктор.

```
class Int {
const int size;
public:
Int(int i): size(i) { }
};
```

Если у класса есть конструктор, то он вызывается всегда, когда создается объект класса. Если у класса есть деструктор, то он вызывается всегда, когда объект класса уничтожается. Объекты могут создаваться как:

- Автоматический объект: создается каждый раз, когда его описание встречается при выполнении программы, и уничтожается каждый раз при выходе из блока, в котором оно появилось;
- Статический объект: создается один раз, при запуске программы, и уничтожается один раз, при ее завершении;
- Объект в свободной памяти: создается с помощью операции new и уничтожается с помощью операции delete;
- Объект член: как объект другого класса или как элемент вектора.

Примет класса - Строка

```
#include <string.h>
class StringHolder {
char *contens;
public:
StringHolder(char *aString=0);
~StringHolder(void);
char *getContens(void);
void setContens(char *aString);
};
StringHolder::StringHolder(char *aString) {
if(aString) { contens=new char[strlen(aString)+1]; strcpy(contens,aString); }
else {contens=new char[1]; *contens='\0'; }
}
StringHolder::~StringHolder(void) { delete contens; }
char *StringHolder::getContens(void) { return contens; }
void StringHolder::setContens(char *aString) {
delete contens;
contens=new char[strlen(aString)+1]; strcpy(contens,aString);
}
void main()
{
StringHolder str1,str2("Object2"), *str3, *str4;
str3=new StringHolder("object3");
str4=new StringHolder;
cout << str1.getContens(); cout << str2.getContens();
cout << str3->getContens(); cout << str4->getContens();
}
```

Упражнения

1. Предположим Вам требуется программа на традиционном языке вроде C. Как бы вы смоделировали методы и классы?
2. Как вы думаете., что лучше, иметь ключевое слово public, protected или private присоединенное к каждому объекту (как в Java), или к области данных (как в C++).

Дружественные функции и перегрузка операций. Преобразование данных.

План

1. Преобразования, определяемые классом
2. Перегрузка и выбор функций
3. Дружественные функции
4. Перегружаемые операции
5. Перегрузка унарного оператора
6. Перегрузка бинарного оператора
7. Перегрузка оператора присваивания и индексирования
8. Перегрузка операций new, delete, ->

Полиморфизм- это средство для придания различных значений одному и тому же сообщению в зависимости от типа обрабатываемых данных. *Преобразование* - это явное или неявное изменение значения в зависимости от типа. Преобразование обеспечивает форму полиморфизма. Перегрузка функций дает одному и тому же имени функции различные значения. Одно и то же имя имеет различные интерпретации, которые зависят от выбора функции. Выбранная функция удовлетворяет алгоритму соответствия сигнатур для C++. Такая форма полиморфизма называется *специальный полиморфизм.*

Операторы перегружаются и выбираются на основании алгоритма соответствия сигнатур. Перегрузка операторов придает им новые значения. Например, выражение **a+b** имеет различные значения, в зависимости от типов переменных **a** и **b.** Перегрузка оператор + для типов, определяемых пользователем, позволяет использовать их в дополнительных выражениях в большинстве случаев так же, как и встроенные типы. При определении функции преобразования допустимы также выражения смешанного типа.

Один из принципов ООП состоит в том, что определяемые пользователем типы должны иметь те же привилегии, что и встроенные типы. Типы, встроенные в базовый язык, могут смешиваться в выражениях, поскольку это удобно, но, с другой стороны, при этом обременительно определять последовательность необходимых преобразований.

Преобразование, определяемые классом

Операция преобразования типов предоставляет механизм явного преобразования объектов данного класса в другой тип.

Явное преобразование типов выражения применяется тогда, когда неявное преобразование нежелательно или когда без него выражение недопустимо. Одна из задач C++ - интеграция Абстрактного Типа Данных (АТД) и встроенных типов. Чтобы достигнуть этого существует механизм функции-члена, обеспечивающей явное преобразование. Функциональная запись:

Имя типа (выражение)

эквивалентна приведению. Тип должен быть выражаем как идентификатор. Таким образом два выражения
x=(float)i;

x=float(i);
эквивалентны. Выражение
p=(int *)x; // допустимое приведение
не может непосредственно функционально выражаться как
p=int *(x);
Однако, для этого может использоваться typedef
typedef int * int_ptr;
p=int_ptr(x);
Конструктор с одним аргументом фактически представляет собой преобразование из типа аргумента к типу конструируемого класса.

string(const char *p) { len=strlen(p); s=new char[len+1]; strcpy(s,p); }

Представленное выражение - автоматическое преобразование типов от char * к string. Оно доступно как явно, так и неявно. Явно оно используется или как операция преобразования, или в приведении, или в функциональной форме. Таким образом, возможны два рабочих варианта кода:
string s;
char * logo=" Hello";
s=string(logo); // выполняет преобразование, затем присвоение
и
s=logo; // неявный вызов преобразования
Данный код - преобразование из уже определенного типа к типу, определяемому пользователем. Однако, пользователь не может добавлять конструкторы встроенных типов, таких как int. Может возникнуть необходимость в преобразовании из string в char *, что может быть выполнено с помощью определения специальной функции преобразования внутри класса string следующим образом:

operator char * () { return s; }

Общая форма записи такой функции-члена

operator тип() {...}

Такая функция преобразования должна быть нестатической функцией-членом без возвращаемого типа и с пустым списком аргументов. Преобразование происходит неявно в выражениях присвоения, при передаче параметров функциям, и в значениях, возвращаемых функциями.
Преобразующая функция-член в форме A::operator B() и конструктор в форме B::B(const A&) обеспечивает преобразование из типов объекта A в тип объекта B.
Неявное преобразование типов происходит тогда, когда имеется конструктор преобразования данного типа в объект класса:
Int int1,int2; int2=int1+6;
Если класс содержит конструктор с одним параметром вида:
Int::Int(int a) { b=a; },
то код будет работать нормально, если же нет, то будет ошибка.
Int int1,int2; int2=6+int1;
Запись такого вида недопустима. Она означает:. передать объекту 6 сообщение +с параметром int1.
Возможно ли, обеспечить такую запись. Да, если написать перегруженный оператор + как дружественные функции.
Рассмотрим преобразование объектов одного класса в объекты другого класса. Если мы хотим преобразовать объект класса OldType в объект класса NewType, то класс NewType должен иметь конструктор с параметром типа OldType или OldType& :
class OldType{

```
};
class NewType {
public:
NewType(void) {}
NewType(OldType &old) { }
};
void main() {
OldType oldtype;
NewType newtype=oldtype;
}
```

либо с использованием операции преобразования типов

```
operator NewType(void);
class NewType {
public:
NewType(void) {}
};
class OldType {
publci:
operator NewType() { return NewType(); }
};
void main() {
OldType oldtype;
NewType newtype=oldtype;
}
```

Использование одновременно конструктора и перегрузку операции приведение невозможно.
Так как, операции преобразования не имеют ни явных аргументов, ни возвращаемого значения, то они не могут быть перегружены. Может быть несколько операторов преобразования для различных типов. Операции преобразования наследуются и могут быть виртуальными.

Перегрузка и выбор функции

Перегруженные функции - важная особенность языка C++. Перегружаемая функция выбирается в соответствии со списком параметров при обращении к функции и при объявлении функции. При вызове перегруженных функций компилятор должен иметь алгоритм для выбора соответствующей функции. Этот алгоритм зависит от того, какие преобразования типов доступны. Наилучшее соответствие должно быть уникально. Оно должно быть лучше всех, по крайне мере, для одного параметра, а для всех остальных параметров так же хорошо, как любое другое соответствие.
Алгоритм соответствия для каждого параметра следующий:

1. Использовать точное соответствие, если оно найдено
2. Проверить поддержку стандартных типов
3. Проверить стандартные преобразования типов
4. Проверить преобразования, определяемые пользователем
5. Использовать соответствие для аргументов, если оно найдено

Рассмотрим пример.

```
class complex

{
        double real,imag;
public:
```

```
                complex(double r) { real=r; imag=0; }
                void assign(double r,double i) { real=r; imag=i; }
                void print() { cout << real << "+"<<imag<<"i"; }
                // обеспечивает преобразование complex в double
                operator double() { return sqrt(real*real+imag*imag); }
};
inline int greater(int i,int j) { return (i > j) ? i:j; }
inline double greater(double i,double j) { return (i > j) ? i:j; }
inline complex greater(complex i,complex j) { return (i > j) ? i:j; }
void main() {
                int i=5,j=10;
                float x=7;
                double y=14;
                complex w(0),z(0);
                w.assign(x,y); // требует преобразования float x в double
                z.assign(i,j); // оба параметра преобразуются в double
                // выбирает первое определение greater по правилу соответствия
                cout << greater(i,j) << endl;
                // Выбирает второе определение greater, так как использовано стандартное расширяющее
                преобразование из float в double
                cout << greater(x,y) << endl;
                // Второе определение greater выбирается из-за точного соответствия //правилу.
                cout << greater(y,double(z)) << endl;
                // точно соответствует третьему определению
                cout << greater(w,z) << endl;
}
```

Функция-член преобразования **operator double** требуется для оценки условия **w<z**. **Complex** переменные **w z** преобразуются в **double**. Для возвращаемого типа не необходимости в преобразовании. Для избежания неоднозначности необходимо явное преобразование **double(z)**. Если вместо этого будет использоваться вызов функции:

greater(y,z); // error

то он будет иметь два доступных преобразования, достигших соответствия. Преобразования параметра **y** из **double** в **complex**, определяемое пользователем, соответствует третьему определению. Преобразование **z** из **complex** в **double**, также определяемое пользователем, соответствует второму определению. Но второе определение функции имеет лучшее соответствие для первого параметра, между тем как третья функция имеет лучшее соответствие для второго параметра. Это нарушает требование о том, чтобы "Максимальное соответствие должно быть уникально. Оно должно быть лучше всего для, по крайне мере, одного параметра и одинаково хорошо для всех остальных".

Дружественные функции

Класс может предоставлять особые привилегии определенным внешним функциям или функциям-членам другого класса. Эти функции получили названия дружественных. Если функция или класс объявлены как дружественные данному классу, то такие функции или функции-члены такого класса могут осуществлять непосредственный доступ ко всем полям класса, для которого они дружественны. Дружественные функции и классы могут осуществлять прямой доступ к закрытым полям класса без использования функций-членов этого класса.

Ключевое слово **friend** - спецификатор функции, который дает функции- не члену класса доступ к скрытым членам класса. Он используется для того, чтобы выйти за строгие рамки типизации и сокрытия данных в C++.

Одна из причин их использования состоит втом, что некоторые функции нуждаются в привилегированном доступе более, чем к одному классу. Вторая причина - **friend**-функция передает все параметры через список параметров, и значение каждого из них подчинено преобразованию, совместимому с назначением. Такие преобразования применяются к явно переданным аргументам-классам и поэтому особенно полезны в случаях перегрузки оператора.

Объявление **friend** функции должно появляться внутри объявления класса, которому она дружественна. Имени функции предшествует ключевое слово **friend**, и ее объявление может находится как в **public** так и в **private** части класса, что не повлияет на значение. Функция-член одного класса может быть **friend**-функцией другого класса. Это происходит тогда, когда функция-член объявлена в **friend** классе с использованием оператора разрешения контекста для определения имени функции дружественного класса. Если все функции-члены одного класса являются **friend**-функциями другого класса, то это можно определить записью:

```
freind class имя класса
class t1 {
friend void a(); // friend-функция
int b(); // функция-член
};
class t2 {
friend int t1::b(); // функция-член класса t1 имеет доступ ко всем скрытым полям класса t2
};
class t3 {
friend class t1; // все функции-члены класса t1 имеют доступ ко всем полям класса t3
};
```

Рассмотрим класс **matrix** и класс **vector**. Функция умножения вектора на матрицу должна иметь доступ к **private-членам** обоих классов. Эта функция будет **friend** для обоих классов.

```
class matix;
class vect {
int *p;
int size;
friend vect mpy(const vect &,const matix &);
public:
};
class matrix {
int **base;
int row,column;
friend vect mpy(constr vect&,const matirx &);
};
vect mpy(const vect &v,const matrix &m) {
if(v.size!=m.row) { exit(1); }
vect ans(column);
//.....
return ans;
}
```

Второстепенное значение требует предварительного описания класса **matrix**. Оно необходимо потому, что функция **mpy** должна появляться в обоих классах, и использует каждый класс как тип аргумента.

Friend-функции можно рассматривать как часть общего интерфейса класса. Существует ряд ситуаций, в которых они могут быть альтернативой функциям-членам. Использование **freind**-функций спорно, потому что они нарушают инкапсуляцию, окружающую **private** члены классов. Парадигма ООП утверждает, что объекты (в C++ они - переменные класса) доступны через их **public** члены. Только функции-члены должны иметь доступ к скрытой реализации АТД. Это ясный и строгий принцип проектирования. **Friend**-функция находится на самой его границе, поскольку имеет доступ к **private** членам, сама не являясь функцией-членом. С ее помощью можно организовать быстрый код для доступа к подробностям реализации класса.

Перегружаемые операции

Для перегрузки встроенных операторов C++ можно использовать ключевое слово **operator.** Ему может быть присвоен ряд значений, зависящих от параметров. Таким образом оператору типа + можно присвоить дополнительные значения. Это облегчает написание программ и делает их более читабельными.

vect oprator *(const vect &,const matix &)

где * является двухместным оператором умножения.

matix t; vect s,r; s=t*r; // s=mpy(r,t);
Перегруженный оператор можно вызвать и использованием функциональной формы записи:
s=operator*(t,r);
Хотя операторам и могут добавляться новые значения, но их приоритет остается прежним. Например, операция умножения имеет более высокий приоритет, чем сложение. Перегружены могут все операторы. Нельзя использовать аргументы по умолчанию.

Доступны все арифметические, логические операторы, операторы сравнения, равенство, присвоение, операторы поразрядных операций, префиксные и постфиксные формы операторов приращения и декремента. могут быть перегружены операторы индексации “[]” и обращения к функции “()”. Также могут быть перегружены оператор указателя класса “->“ и оператор указателя на член “->*”. Возможна перегрузка **new, delete.**

Перегрузка унарного оператора

Одноместные операторы типа “!”, “++”, “~” “[]”.

```
class clock
{
        unsigned long sec;
public:
        clock(unsigned long s):sec(s) { }
        void tick() { sec++; }
        clock operator++() { tick(); return *this; }
};
void main() { clock t(100); ++t; }
```

Этот класс перегружает префиксный оператор приращения ++. Перегруженный оператор представляет собой функцию-член. Перегруженный **operator++()** также обновляет неявную переменную **clock** и возвращает обновленное значение.

Префиксную операцию ++ можно перегрузить, используя **friend**-функцию.
frien clock operator++(clock &s1) { s1.tick(); return s1; }
Так как переменная **clock** должна увеличиваться мы передаем ее по ссылке. Решение о выборе между представлением friend и функцией-членом обычно зависит от того, насколько необходимы и доступны операторы неявного преобразования. Явная передача аргумента, как в friend-функции, позволяет автоматическое его приведение.

Операции инкремента и декримента могут быть перегружены как для префиксной, так и для постфиксной формы записи.
Префиксная форма объявляется как операция-член класса, которая не имеет аргументов, либо как дружественная операция, которая имеет один аргумент, представляющий собой ссылку на объект данного класса.
Постфиксная форма объявляется как операция-член класса, которая имеет один аргумент типа int, либо как дружественная операция, которая имеет два аргумента: ссылку на объект данного класса и аргумент типа int. Аргумент типа int на самом деле не используется. Фактически он имеет нулевое значение.

```
class X {
public:
X & operator ++(); //Префиксная форма
X & operator ++(int); // Постфиксная форма
};
class Y {
public:
friend Y & operator ++(Y &); //Префиксная форма
friend Y & operator ++(Y &,int); // Постфиксная форма
};
```

@x интерпретируется как x.opertor@(), если operator @ - функция-член и operator@(x), если operator @ - дружественная функция.

Перегрузка бинарного оператора

Если бинарный оператор перегружается с помощью функции-члена, то в качестве своего первого аргумента он получает неявно переданную переменную класса, а второго - единственный из списка аргументов. При объявлении **friend**-функции или обычной функции в списке параметров определяют оба аргумента.

```
class clock {
friend clock operator+(const clock &c1,const clock &c2) ;
};
clock operator+(const clock &c1,const clock &c2) {
clock temp(c1.sec+c2.sec); return temp;
}
```

Оба параметра явно определяются и являются кандидатами для преобразования назначением.
Используя это определение имеем:

```
int i=5; clock c(100);
c+i; // допустимо: i - преобразуется в clock
i+c; // допустимо: i - преобразуется в clock
```
В противоположность этому, перегрузим двухместный - функцией-членом.

```
class clock {
clock operator -(const clock &c) { clock temp(sec-c.sec); return temp; }
};
```

Существует первый неявный аргумент. В него попадает некоторый используемый параметр. Это может вызвать асимметричное поведение двухместных операторов.

```
int i=5; clock c(100);
c-i; // допустимо i преобразуется в clock; c.operator-(i)
i-c; // недопустимо i не относится к типу clock; i.operator-(c)
Определим операцию умножения: long и clock
clock operator *(long m, const clock &c) {
clock temp(m*c.sec); return temp;
```

Такая реализация вынуждает операцию умножения иметь фиксированный порядок выполнения, зависящий от типа. Для избежания этого обычно пишется второй перегруженный функциональный оператор.

```
clock operator *(const clock &c,long m) {
                clock temp(m*c.sec); return temp;
}
```

y@x интерпретируется как y.opertor@(x), если operator @ - функция-член и operator@(y,x), если operator @ - дружественная функция.

Операция вызова функции

Операция вызова функции должна быть объявлена как нестатическая функция-член класса. Она позволяет пользователю определять число операндов.

```
class X {
int a,b,c;
public:
X(int a1, int b1, int c1): a(a1),b(b1),c(c1) {}
void operator() { int i,int j, int k);
};
X ex(8,20,17); // Вызывается конструктор
ex(10,15,57); // Вызывается операция ()
```

Перегруженные операции - функции-члены против дружественных функций

Если перегруженная операция реализована как функция-член, то ей либо вообще не передаются явно параметры, либо передается один параметр. Дружественной функции - перегруженной операции - передается один, либо два параметра.
Если бинарная операция перегружена как функция-член, то ее первым операндом является объект, который принимает сообщение. Следовательно, явно передается только один параметр.
Если унарная перегруженная операция реализована как функция-член, то ее операндом является принимающий сообщение объект. Таким образом, эта функция-член не имеет явных параметров.
Перегруженная операция не может иметь более двух параметров.

Фунция-член
```
Class X {
X operator-() const; // Унарный минус
X operator&() const; // Вычисление адреса
X operator^() const; // Ошибка: операция ^ - бинарная
        };

Class X {
        X operator-(const X&) const; // бинрный минус
X operator&( const X&) const; // побитовое И
        };
```

Дружественная функция
```
class Y {
        friend Y operator-(const Y&);
friend Y operator&(const Y&);
        freind Y operator^(const Y&);
};

class Y {
        friend Y operator-(const Y&,const Y&);
friend Y operator&(const Y&, const Y&);
        };
```

Перегружаемые оператор присваивания и индексированные операторы

```
class vect {
private:
                int *p;
                int size;
public:
                vect(void); // создает массив из 10 элементов
                vect(int mysize); // создает массив из mysize элеметов
                vect(int *a,int n); // инициализируется массивом
                vect(const vect &v); // инициализируется vect
                ~vect() { delete p; }
                int & operator[](int i);
                vect& operator=(const vect &v); // перегруженное присваивание
                vect operator+(const vect &v); // перегруженное сложение
};
vect::vect(int n) { // преобразует обычный массив
                p=new int[n]; size=n;
                for(int index=0; index < size; index++) p[i]=index;
}
vect::vect(int *a,int n) { // преобразует обычный массив
                p=new int[n];
                size=n;
                for(int index=0; index < size; index++) p[i]=a[i];
}
vect::vect(const vect &v) { // конструктор копии
                p=new int[v.size]; size=v.size;
                for(int index=0; index < size; index++) p[i]=v.p[i];
}
```

Перегруженный оператор индексации берет целый аргумент и проверяет, находится ли значение в пределах диапазона. Если да, он использует это значение для того, чтобы возвратить адрес индексированного элемента.

```
int &vect::operator[](int i) {
if(i < 0 || i > =size) { cerr << “”;}
```

```
return p[i];
}
```

Перегруженный оператор индексации имеет возвращаемый тип и один параметр. Он должен быть нестатической функцией-членом.

Вопросы:

1. Что если перегруженная операция [] будет иметь тип возвращаемого значения int, а не int&?

Ответы:

Произойдет ошибка при компиляции. Использование ссылки в качестве возвращаемого значения позволяет использовать операцию индексирования как с правой, так и слевой стороны от операции присваивания.

Оператор присваивания

В C++ версии 2.0 имеется операция присваивания по умолчанию, которая выполняет рекурсивное по членное копирование одного объекта в другой. В более ранних версиях использовался механизм побитового копирования.

```
vect first(200),second(200);
second=first; cout << second[10]; // =9
first[10]=-50; cout << second[10]; //=-50
```

Когда назначение не перегруженно, оно определяется по умолчанию с семантикой, представляющей собой присвоения значения. Это называется “семантикой поверхностного копирования” и не всегда допустимо. Перегрузка назначения желательна, а иногда и необходима, особенно, если класс содержит указатели динамически распределяемой памяти:

```
vect &vect::operator=(const vect &) {
int s=(size<v.size) ? size : v.size;
for(int i=0; i<s; i++) p[i]=v.p[i]; return *this; }
```

Если необходимо запретить операцию присваивания для объектов данного класса, то ее нужно просто объявить в закрытой части объявления.

```
Class IntStack {
int *v, size,top;
public:
IntStack &operator=(const IntStack&);
};
IntStack & IntStack::operator=(IntStack &from) {
if(this != &from) { // Проверка на from=from
delete [] v;
v= new int [size=from.size]; for(int i=0; i<size; i++) v[i]=from.v[i];
top=from.top;
}
return *this; // Позволяет выполнять множественное присваивание
}
IntStack a(100),b,c;
b=c=a;
```

Присваивание против инициализации

Для того, чтобы операция = обозначала присваивание, в левой части выражения, содержащего эту операцию, должен находиться объект (т.е. для него должна быть выделена память) операция присваивания должна выглядеть следующим образом:
C& C::operator=(C&);
Так как операция присваивания возвращает ссылку на объект, она может использоваться в цепочке таких операций.
Инициализация происходит либо при по членном копировании, либо при использовании конструктора следующего вида:
C::C(&C);
Желательно включение конструктора инициализации.
Инициализация происходит в следующих случаях:
Объявление переменной vect second=first;
Возврат автоматической переменной из функции. Возвращаемое значение - это новый объект, который инициализируется при помощи конструктора.
Передача значения функции в качестве параметра. Параметр действует как локальная переменная и инициализируется при помощи конструктора.
Рассмотрим примеры инициализации:

```
class demo {
        int i;
public:
        demo(void) {i=0; }
        demo(int i1):i(i1) {}
        demo(demo &a) { i=a.i; }
        demo &operator=(demo &a) { i=a.i; return *this; }
};
demo copydemo(demo a) {
        demo temp; // конструктор без параметров
        temp=a; // оператор присваивания
        return temp; // вызов конструктора инициализации, временная копия
}
void main(void) {
        demo *ptr=new demo(5); // вызывается конструктор с одним параметром
        demo demo2=demo1; // вызов инициализирующего конструктора
        demo demo3; // вызов конструктора без параметров
        demo3=demo2=demo1; // перегруженная операция присваивания вызывается 2 раза
        demo4=copydemo(demo1); // вызывается конструктор инициализации для передачи
        аргумента, по выходе оператор присваивания.
}
```

Конструктор копирования должен быть определен, если класс содержит поля , являющиеся указателями на динамически распределяемую память. При отсутствии конструктора копирования, конструктор копирования по умолчанию сделает так, что указатель b объекта v и указатель b объекта a будут иметь одинаковые значения. По той же причине необходимо перегружать операцию присваивания, если конструктор копирования перегружен пользователем. Инициализация объекта из другого объекта того же класса может быть запрещена, если конструктор копирования просто объявить в закрытой части класса.

```
Int a(100);
Int b(a); // Объявить b, которое равно a
Int c=a; // То же в альтернативной форме
```

Перегруженные операции new delete, ->

Операции new и delete предоставляют классу механизм управления собственной памятью. Тип возвращаемого значения для new - void *, для delete - void. Неявно они являются статическими функциями-членами и, следовательно, не могут быть не константными не виртуальными.

```
class X {
public:
void *operator new(size_t sz) { return malloc(sz); }
void operator delete(void *p) { free(p); }
};
```

Если операции new и delete определены для класса, не могут быть вызваны при выделении и освобождении памяти для массивов объектов данного класса. Операция new может быть перегружена, чтобы принимать дополнительные аргументы. Передаваемые аргументы, заключенные в скобки, помещаются при вызове после ключевого слова new.

```
class T {
public:
// Добавить дополнительный аргумент buf , который будет использоваться для
// выделения памяти для T по указанному адресу
void * operator new(size_t size,void *buf) { /* Разместить T по адресу buf */}
};
char buffer[1000];
T *p=new (buffer) T;
```

Для объектов класса X вместо операций new, delete будет использоваться X::operator new, X::operator delete.

Перегруженная операция new с дополнительными аргументами скрывает глобальную операцию ::operator new. Теперь при обращении к new с обычным синтаксисом произойдет ошибка.

```
T *p=new T; // ошибка: отсутствует аргумент
```

Чтобы снова использовать обычный синтаксис оператора new, для класса должна быть предусмотрена функция-член new, имеющая только один аргумент - size_t size. Она явно вызывает глобальную операцию ::operatot new.

```
class T {
public:
void * operator new(size_t size,void *buf) { /* Разместить T по адресу buf */}
void * operator new(size_t size) { return ::operator new(size); }
};
```

Перегруженная операция -> обычно используется в классах, которые содержат в качестве поля указатель на другой класс.

```
struct A { int *intptr; };
class B { A *a;
public:
B(void) { a=new A; } A *operator->(void) { return a; }
```

```
};
B b; *(b->inptr)=15; // = b.a->inptr;
```

Производные классы : одиночное наследование

План

1. Порожденный класс
1. Типизированные преобразования и видимость
1. Производные классы и правила наследования
1. Правила доступа для объектов и классов
1. Закрытые части классов
1. Защищенная часть классов
1. Открытая часть классов
1. Обобщение правил доступа
1. Иерархия классов против контейнерных классов
1. Правила доступа для друзей классов и производных классов

Наследование - механизм создания нового класса из старого. Т.е., к существующему классу можно что-либо добавить, или изменять его каким-либо образом для создания нового (порожденного) класса. Это мощный механизм для повторного использования кода. Наследование позволяет создавать иерархию связанных типов, совместно использующих код и интерфейс.

Большинство полезных типов представляют собой различные варианты друг друга, поэтому утомительно выписывать для каждого один и тот же код. Порожденный класс наследует описание основного класса. Затем он может изменяться с добавлением членов, изменения функций существующих членов и изменения привилегий доступа.

В терминах ООП, оператор “слон есть млекопитающее” описывает отношение. Выражение “цирк содержит слона” описывает отношение подразделения.

C++ поддерживает **virtual** функции-члены, которые объявлены в основном классе и переопределены в порожденном классе. Иерархия классов, определенная общим наследованием, создает связанный набор типов пользователя, на которые можно ссылаться с помощью указателя базового класса. При обращении к виртуальной функции через этот указатель в C++ выбирается соответствующее функциональное определение во время выполнения. Объект, на который указывается, должен содержать в себе информацию о типе, поскольку различия между ними может быть сделано динамически. Это особенность типична для ООП кода. Каждый объект “знает” как на него должны воздействовать. Эта форма полиморфизма называется чистым полиморфизмом.

Способность к наследованию должна быть встроена в программное обеспечение для того, чтобы максимизировать многократное использование кода и позволить естественное моделирование

предметной области. С использованием наследования ключевыми элементами методологии ООП становятся:

- разработка соответствующего набора типов
- проектирование из возможных связей и применение механизма наследования для совместного использования кода
- использование виртуальных функций для полиморфической обработки связанных объектов.

Порожденные класс

Для порождения нового класса от существующего может использоваться следующая форма записи:

class *имя класса* : (public | protected | private) *имя базового класса* {

объявления членов

};

Одна из особенностей порожденного класса - видимость унаследованных членов. Для определения доступности членов основного класса порожденному классу используются ключевые слова **public, protected** и **private**. Рассмотрим пример:

```
class student {

protected:

int id;

char name[30];

public:

sudent(int i,char *nm);

void print();

};

class grad_student : public student {

protected:

char thesis[30];

public:

grad_student(int i,char *nm, char *t);

void print();

};
```

В этом примере **grad_student** - порожденный класс, а **student** - базовый класс. Использование ключевого слова **public** в заголовке описания порожденного класса означает, что **protected** и **public** члены класса **student** должны быть унаследованы как **protected** и **public** члены grad-student соответственно. Члены **private** недоступны. Общее наследование также означает, что полученный класс grad-student - подтип класса student. Следовательно, студент окончивший институт - студент.

Порожденный класс представляет собой модификацию основного класса, которая наследует общие и защищенные члены базового класса. Таким образом, в примере grad_student члены student - id,name, print унаследованы. Функция-член print перекрывается. Это означает, что порожденный класс имеет реализацию функции-члена, отличающуюся от базового класса. Не следует путать с перегрузкой, где то же самое имя функции может иметь различные значения для каждой уникальной сигнатуры.

Такой механизм выгоден по следующим соображениям:

Код используется многократно. Тип grad_student использует существующий проверенный код из student.

Иерархия отражает отношения, существующие в предметной области. Различные полиморфные механизмы позволяют коду пользователя обрабатывать grad_student как подтип student, что упрощает код пользователя, в тоже время предоставляя ему выгоды при обработке этих различий между подтипами.

Типизированные преобразования и видимость

В общем случае порожденный класс представляет собой подтип базового класса. Переменная порожденного класса может обрабатываться так, как будто она была базового класса. Указатель, тип которого - указатель на базовый класс, может указывать на объекты, имеющие тип порожденного класса. Из-за этого может возникнуть беспорядок, поскольку встречаются неуловимые косвенные преобразования между типами основного и порожденного классов. Иногда трудно уследить за тем, к какому члену производится обращение, если базовый и порожденный классы перегружают член с одним и тем же именем.

Рассмотрим пример конструктора базового и порожденного класса:

```
student::student(int i,char *nm):id(i) { name=new char [strlen(nm)+1]; strcpy(name,nm); }

grad_student::grad_student(int i,char *nm,char *t): student(i,nm) {

thesis=new char[strlen(t)+1]; strcpy(thesis,t); }
```

Конструктор для **student** вызывается, как часть списка инициализаторов. Логично создавать объект базового класса прежде, чем может быть завершен полный объект.

Ссылка на порожденный класс может быть неявно преобразована в ссылку на общий базовый класс.

```
grad_student gs(1,”NNN”,”Tn”);

student &rs=&gs;
```

Переменная **rs** - ссылка на **student**. Базовый класс для **grad_student - student**. Следовательно, такое преобразование ссылки допустимо.

Функция-член print - перегружена.

void student::print() { cout << id << name; }

void grad_student::print() { student::print(); cout << thesis; }

Для того, чтобы **grad_student::print** вызвала функцию **student::print**, должен быть использован идентификатор с разрешенной областью **student::print**. Иначе произойдет бесконечный цикл. Рассмотрим пример отношения между базовым и порожденным классом.

void main(void) {

student s(1,"Ma"), *ps=&s;

gard_student gs(2,"Jh","Pet"), *pgs;

ps->print(); // student::print()

ps=pgs=&gs;

ps->print(); // student::print()

pgs->print(); // gard_student::print()

}

Правило заключается в том, что указатель на порожденный класс может быть неявно преобразован в указатель на базовый класс. Переменная **ps** может указывать на объекты обоих классов, но переменная указатель **pgs** может указывать только на объекты типа **grad_student.**

Производные классы и правила наследования

Производные классы и объекты этих классов полностью наследуют протокол своего базового класса. Возможность прямого доступа к полям и функциям-членам базового класса через протокол производного класса зависит от того, в каком из разделов описания базового класса (**private**, **protected**, **public**) они определены. Кроме того, возможность доступа к протоколу базового класса зависит от того, является ли производный класс открытым или закрытым. (По умолчанию производный класс является **private**).

Рассмотрим пример базового и двух производных классов открытого и закрытого. Когда говорят, что объекты имеют доступ к полям или функциям-членам, - это означает, что к таким объектам можно обращаться непосредственно, используя операцию ::, либо передавая сообщение соответствующее определенной функции-члену.

Конструкторы производных классов автоматически вызывают соответствующий конструктор базового класса.

class A {

int i,k;

```
public:
A(void) { i=0; k=0; }
A(int i1,int k1=0): i(i1),k(k1) {}
int geti(void) { return i; }
void print(void) { cout << " A "<< i <<k; }
};
```

```
class B: public A {
float s;
public:
B(void): A() { s=0.; }
B(int i1,int k1=0,float s1=0.): A(i1,k1),s(s1) {}
float gets(void) { return s; }
void print(void){A::print(); cout << " s="<< s; }
};
```

```
class C: private A {
public:
C(void):A() {}
C(int i1,int k1=0):A(i1,k1) { }
int geti(void) { return A::geti(); }
void print(void) { A::print(); }
};
```

```
void main() {
A a1, a2(10);
B b1,b2(20,30,40);
C c1,c2(50,60);
}
```

Вопросы:

1. *Что, если попытаться осуществить доступ к i через протокол класса C или класса B ?*

   Произойдет ошибка компиляции, потому что i находится в private части класса A и недоступны протоколу производных классов.

1. *Что, если убрать определитель A перед оператором :: в методе geti класса C?*

   Результат - ошибка компиляции. Неопределенна функция ::geti.

1. *Что, если убрать явный вызов конструктора базового класса A() и A(i1) из конструкторов порожденных классов.*

   Это не окажет влияния на функционирование этих классов и конструкторов. Конструкторы в классе наследнике автоматически вызывают конструктор базового класса.

1. *Что, если во втором конструкторе классов, B, C убрать полный вызов родительского конструктора A(i1)?*

   Удаление вызова конструктора не повлияет на функционирование классов. Компилятор выдаст предупреждение и параметр i будет установлен 0, так как вызывается конструктор без параметров базового класса.

1. *Что, если убрать void конструктор для классов B, C? Является ли void конструктор для класса A наследуемым и доступным для пользователя?*

   Нет! Если удалить эти void конструкторы, то в не инициализирующих объявлениях, таких как B b1, C c1 компилятор выдаст ошибку. Void конструктор можно убрать, если есть конструктор с параметрами, которые все определены по умолчанию.

1. *Что, если в конструкторы для классов B и C не включать значения по умолчанию для k? Будет ли по-прежнему работать умолчание в родительском классе A?*

   Нет! Значения по умолчанию должны быть переопределены и в наследуемых классах. Иначе в объявлениях вида B b1(10), компилятор выдаст ошибку.

1. *Что, если у класса-наследника есть конструктор, а у базового класса его нет?*

   Это ситуация зависит от ситуации, системы и компилятора.

Правила доступа для объектов и классов

- Элементы, объявленные как **private**, доступны функциям-членам и друзьям класса.
- Элементы, объявленные как **protected**, доступны функциям-членам, друзьям класса и функциям-членам производных классов.
- Элементы, объявленные как **public**, доступны любым функциям.
- Когда класс используется как открытый (public) базовый класс, его открытые элементы становятся открытыми элементами производного класса, а защищенные (protected) элементы - защищенными элементами производного класса.
- Когда класс используется как закрытый (private) базовый класс, его открытые и защищенные элементы становятся закрытыми элементами производного класса.

```
class Base {

private:

type data1;

returntype function1();

protected:
```

```
type data2;
returntype function2();
public:
type data3;
returntype function3();
};
class PublicDerived: public Base {
private:
type data4;
returntype function4();
protected:
type data5;
returntype function5();
public:
type data6;
returntype function6();
};
class PrivateDerived: private Base {
private:
type data7;
returntype function7();
protected:
type data8;
returntype function8();
public:
type data9;
```

returntype function9();

};

main() {

Base base;

PublicDerived pub;

PrivateDerived priv;

}

Закрытые части классов

Доступ к протоколу класса

Описание класса имеет доступ ко всем своем собственным частям. Любая функция-член в закрытой (**private**), защищенной(**protected**) или открытой (**public**) части описания класса имеет прямой доступ ко всем полям данных и всем функциям-членам описания класса. Таким образом, любая функция-член в классе Base имеет прямой доступ к data1, data2, data3, function1, function2, function3. Аналогично, любая функция-член класса PublicDerived имеет прямой доступ к data4, data5, data6, function4, function5, function6. Любая функция в классе PrivateDerived имеет прямой доступ к data7, data8, data9, function7, function8, function9.

Доступ через протокол производного класса

Производные классы наследуют все протоколы (включая все поля данных и функции-члены) от своего базового класса. Однако производный класс не имеет прямого доступа к полям данных или функциям-членам, определенным в закрытой части его базового класса. Классы PublicDerived и PrivateDerived наследуют data1, function1 от своего базового класса (Base), но не имеют прямого доступа к ним. Доступ можно обеспечить через функцию, определенную в защищенной или открытой части класса Base. Эта функция будет наследоваться открытыми и защищенными производными классами и будет доступна им.

Доступ через объекты класса

Объекты класса не имеют доступа к защищенной части описания класса. Таким образом, объект aBase не имеет доступа к своим собственным закрытым элементам data1, function1.

Вопрос:

Что если, программист захочет обеспечить доступ к закрытой части класса Base через объекты этого класса.

Это можно сделать, определив функцию доступа в открытой части описания класса:

//добавить в public часть класса Base

returntype privateAccessFunction(type avalue) {

function1(); data1=avalue; }

abase1.privateAccessFunction(newValue);

Доступ через объекты public-производного класса

Объекты public-производного класса не имеют доступа к закрытой части базового класса. Если в классе Base определены открытые функции для доступа к закрытым полям через объекты классов, то эти функции наследуются public-производными классами и могут использоваться объектами этих классов. Объект public-производного класса может получить доступ к закрытой части его базового класса с помощью следующего выражения:

aPublicDerived.privateAccessFunction(newValue);

Однако объекту aPublicDerived недоступны его собственные закрытые элементы data4, function4. Чтобы сделать их доступными необходимо добавить метод доступа в открытой части класса PublicDerived.

Доступ через объекты private-производного класса

Объекты private-производного класса не имеют доступ к закрытой части их базового класса. Элементы data1, function1 недоступны объекту aPrivateDerived. Даже добавление функции privateAccessFunction к открытой части Base не обеспечит этот доступ, так как объектам private-производного класса недоступны элементы открытой части их базового класса.

Вопрос:

Что если, действительно необходимо предоставить доступ к закрытой части базового класса через объекты private-производного класса.

//добавить открытую функцию privateAccessFunction в Base

// переопределить функцию PrivateAccessFunction в открытой части

// класса PrivateDerived1

returntype privateAccessFunction(type aValue) {

Base::privateAccessFunction(aValue); }

aPrivateDerived.privateAccessFunction(newValue);

Защищенная часть классов

Защищенная часть класса интерпретируется как открытая для его протокола и протоколов производных классов (**public**-, **private**-производных классов). Защищенная часть интерпретируется как закрытая для доступа через объекты класса и объекты производных классов (**public, private -** производных классов).

Доступ через протокол класса

Протокол класса имеет доступ ко всем его частям. Любая функция-член класса Base имеет прямой доступ к data2,function2.

Доступ через протокол производного класса

Протокол производного класса имеет прямой доступ к защищенным частям его базового класса и ко всем частям его собственного протокола. Класс PublicDerived1 имеет прямой доступ к data2, data5, function2, function5. Класс PrivateDerived1 имеет прямой доступ к data2, data8, function2, function8.

Доступ через объекты класса

Объекты класса не имеют прямого доступа к защищенным частям протокола. Через объекты aBase нельзя получить прямой доступ к data2, function2. Возможно осуществить непрямой доступ через определение метода доступа в открытой части класса Base.

Доступ через объекты public-производного класса

Нельзя осуществить доступ к защищенной части любого производного класса через его объекты. Объект aPublicDerived1 не имеет прямого доступа к data2,data5,function2, function5.

Доступ через объекты private-производного класса

Нельзя осуществить доступ к защищенной части любого производного класса через его объекты. Объект aPrivateDerived1 не имеет прямого доступа к data2,data8,function2, function8.

Открытая часть классов

Открытая часть классов доступна любому файлу, в который включено описание этого класса. Прямой доступ к открытой части базового класса через объекты производного класса зависит от того, является ли этот класс public или private-производным классом.

Доступ через протокол класса

Любая функция-член описания класса имеет прямой доступ к его открытой части. Протокол класса Base имеет прямой доступ к data3, function3

Доступ через протокол производного класса

Любая функция-член производного класса имеет прямой доступ к открытой части его базового класса. Это справедливо как для **public-**,так и для **private-**производных классов. Кроме того, производный класс имеет прямой доступ к собственной открытой части. Протокол класса PublicDerived имеет прямой доступ к data3,function3, data6,function6. Протокол класса PrivateDerived1 имеет прямой доступ к data3,data9, function3, function9.

Доступ через объекты класса

Все объекты какого-либо класса имеют прямой доступ к открытой части протокола своего класса. Объект aBase имеет прямой доступ к data3, functuon3

Доступ через объекты public-производного класса

Объекты public-производного класса имеют прямой доступ к открытой части своего класса и его базового класса. Объекты aPubcliDerived имеет прямой доступ к data3, data6, function3, function6.

Доступ через объекты private-производного класса

Объекты private-производного класса имеют доступ к открытой части своего класса, но не имеют доступа к открытой части базового класса. Объект aPrivateDerived имеет прямой доступ к data9, function9, но не имеет прямого доступа к data3, function3.

Обобщение правил доступа

- Элементы, объявленные как закрытые (**private**), могут использоваться функциями-членами и друзьями класса.
- Элементы, объявленные как защищенные (**protected**), могут использоваться функциями-членами, друзьями класса, а также функциями-членами производных классов.
- Элементы, объявленные как открытые (**public**), могут использоваться любой функцией.
- Когда класс используется как открытый базовый класс, его открытые элементы становятся открытыми элементами производного класса, а защищенные элементы становятся защищенными элементами производного класса.
- Когда класс используется как закрытый базовый класс, его открытые и защищенные элементы становятся закрытыми элементами производного класса.

Иерархия классов против контейнерных классов

Контейнерные классы используются тогда, когда необходимо содержать большое количество индивидуальных элементов.

Рассмотрим примеры противопоставления концепции иерархии классов и контейнерного класса. В данном примере они реализованы через определение базового класса Canine, производного класса Dog, и класса Tail, экземпляр которого содержится в Canine.

```
class Tail {

int length;

public:

Tail(int alength): length(alength) {}

~Tail(void) {}

int getlength(void) { return length; }

};

class Canine {

Tail tail; // содержится в собачьей

int weight;
```

```
public:

Canine(int alength,int aweight): tail(alength), weight(aweight) { } // необходимо вызвать конструктор для класса, который содержится в данном классе

~Canine(void) {}

Tail getTail(void) { return tail; }

int getWeight(void) { return weight; }

};

class Dog: private Canine {

char *name;

public:

Dog(int alength,int aweight,char *aname): Canine(alength,aweight) {

                name = new char[strlen(aname)+1]; strcpy(name,aname); }

~Dog (void) { delete name; }

char *getname(void) { return name; }

void print(void) { cout <<“ name” << name; “ weight” << getweight();

                cout << “ length” << getTail().getlength();

}

};

main() {

Dog myDog(12,48.7,Alex”);

myDog.print();

}
```

Инициализация контейнерного объекта внутри класса представлена в многопараметровом конструкторе класса Canine. В этом конструкторе за двоеточием следует специальный вызов параметризованного конструктора класса Tail - tail(length).

Вопросы:

1. Что если, из конструктора Canine исключить специальный вызов конструктора класса Tail()

Компилятор выдаст предупреждение о том, что не используется alength. При инициализации объекта класса Dog происходит вызов void конструктора для класса Tail, если последний существует, иначе будет ошибка компилятора.

Экземпляры переменных, объявленных в описании класса, инициализируются при создании объекта, а не при объявлении этих переменных.

2. Что если, попытаться проинициализировать экземпляр переменной tail в класса Canine в месте его объявления

Компилятор выдаст ошибку. Экземпляры переменных нельзя инициализировать в точке их объявления. Это применимо как к объектам, так и к обычным С-типам.

Для того, чтобы напечатать длину собачьего хвоста, необходимо послать сообщение getTail(). Это сообщение возвращает экземпляр Tail, который посылает сообщение getlength().

Объекты myDog не имеют прямого доступа к открытой части Canine. Следовательно они не имеют прямого доступа к tail, weight, через открытые методы доступа. Если такие доступы желательны, то необходимо объявить класс Dog как public-производный класс от Canine.

Использование иерархии классов

Основная цель данного примера объяснить отношения “является”, которое представлено производными классами.

```
class Vehicle {

int weight;

public:

Vehicle(void) {}

Vehicle(int aweight): weight(aweight) {}

~Vehicle(void){}

int getweight(void) { return weight; }

void print() { cout << “ weight” << weight;}
```

```
};

class Car: public Vehicle {

int horsepower;

public:

Car(void) {}

Car(int aweight,horse): Vehicle(awright),horsepower(horse){}

~Car(void) {}

int gethorse(void) { return horsepower; }

void print(void) { Vehicle::print(); cout << " horse" << horsepower; }

};

class SportCar: public Car {

float timetosixty;

public:

SportCar(void){}

SportCar(int aweight,int horse,float times):Car(aweight,horse),timetosixty(times){}

float gettimes(void) { return timetosixty; }

void print(void) { Car::print(); cout << " times" << timetosixty; }

};
```

Использование контейнерных классов

Рассмотрим другой подход к организации класса, описывающего спортивные автомобили. Все три класса находятся на одном иерархическом уровне и связаны посредством отношения "содержит". Спортивный автомобиль "содержит" двигатель и "содержит" характеристики. Таким образом, класс SportCar содержит в себе объекты классов Engine и Performance.

```
class Engine {

int horse;

public:
```

```
Engine(void) {}
Engine(int ahorse): horse(ahorse), {}
~Engine(void) {}
int gethorse(void) { return horse; }
void print(void) { cout << " horse" << horse; }
};
class Performane {
float times;
public:
Performance(void) {}
Performance(float atimes): times(atimes) {}
~Performance(void) {}
float gettimes(void) { return times; }
void print(void) { cout << " times" << times; }
};
class SportCar {
int weight;
Engine engine;
Performance performance;
public:
SportCar(void) {}
SportCar(int aweight,int adisp,float atimes): engine(ahorse),
                performance(times), weight(aweight) {}
int getweight(void) { return weight; }
int gethorse(void) { return engine.gethorse(); }
float gettimes(void) { return performance.gettimes(); }
```

```
void print(void) {cout << " weight" << weight; engine.print(); performance.print();}

};
```

Правила доступа для друзей классов и производных классов

Класс SportCar и содержащиеся в нем классы Egine и Performance переопределяются с использованием концепции дружесвенных классов. В данном случае классы Engine и Performance объявляют SportCar дружественным классом. Это обеспечивает доступ через SportCar к закрытым данным каждого из этих двух классов и исключает необходимость в специальных методах доступа.

```
class Engine {

friend class SportCar; // позволяет SportCar получить доступ к закрытым данным

int horse;

};

class Performane {

friend class SportCar; // позволяет SportCar получить доступ к закрытым данным

float times;

};

class SportCar {

int weight;

Engine engine;

Performance performance;

public:

SportCar(void) {}

SportCar(int aweight,int ahorse, float atimes): weight(aweight) {

                engine.horse=ahorse; performance.times=atimes;

}

int getweight(void) { return weight; }

int gethorse(void) { return engine.horse; }
```

```
float gettimes(void) { return performance.times; }

void print(void) { cout << weight << engine.horse<<performance.times; }

};
```

Класс SportCar имеет прямой доступ к закрытым данным классов Engine и Performance через их экземпляры engine и performance.

Дружественные и производные классы

Приведены три класса Person, Student и Trustee. Student -**private**-производный от Person и Trustee. Класс Trustee объявлен как дружественный Person. Таким образом, класс Trustee не имеет доступ к новым закрытым и защищенным полям класса Student. Он имеет доступ только к закрытым и защищенным полям, определенным в классе Person.

Студент описывается протоколами двух классов - Student и Person. Поэтому дружественный класс Trustee имеет прямой доступ только к части закрытых данных студента.

Интерфейс производного и дружественного классов

```
class Person {

friend class Trustee;

int i;

protected:

char name[20];

public:

Person(char *aname,int i1): i(i1) { strcpy(name,aname); }

~Person(void) { }

char *getname(void) { return name; }

int geti(void) { return i; }

void addtoi(int i1) { i+=i1; }

};

class Student:private Person {

int s;

public:
```

```
Student(char *aname,int i1,int s1): Person(aname,i1),s(s1) {}
int geti(void) { return Person::geti(); }
char *getName(void) { return name; }
void addtos(int s1) { s+=s1; }
int gets(void) { retunr s; }
};
class Trustee {
Person *client;
public:
Trustee(void) {}
Trustee(Person *aclient) { client=aclient; }
char *getclient(void) { return client->name; }
void addtoi(int i1) { client->i+=i1; }
};
main() {
Person *p1=new Person(“Lei”,10000);
Trustee t1(p1);
Student *s1=new Student(“Alice”,3500,550);
Trustee t2((Person *)s1);

cout <<“ name” << p1->getname() << “ IRA” <<p1->geti();
p1->addtoi(5000);
cout << “ IRA “ << p1->geti();

int amount=p1->subi(6400);
cout << “ IRA “ << p1->geti();
```

```
cout << “name “ << s1->getname(); cout << “ IRA” << s1->geti();

t2.addtoi(3500); cout << “ IRA” << s1->geti();


cout << “ balance” << s1->gets();

s1->addtoi(500);

cout << “ add” << s1->gets();

}
```

Объект студент не имеет доступа к открытой части протокола, определенной в Person, т.к. Student - private-производный класс от Person. Ограничения:

aPerson может:

- проверить счет IRA

- класть на счет IRA

- не может снимать со счета IRA

aStudent может:

- проверить счет IRA

- не может класть или снимать со счета IRA

- проверять, класть или снимать со счета save

aTrustee может:

- снимать и класть на счет IRA для aPerson и aStudent

- не имеет никакого доступа ко счету студента save

Следующий пример иллюстрирует правила дружественности для классов производных от дружественного. Класс Trustee - дружественный классу Person, класс JuniorTrustee - private-производный от класса Trustee.

aPerson может:

- проверять и класть на счет (но не снимать со счета) IRA

aTrustee может:

- класть и снимать со счета IRA для aPerson

aJuniorTrustee может:

- класть на счет (но не снимать со счета) IRA для aPerson

Интерфейс класса производного от дружественного

```
class Person {
friend class Trustee;
private:
int i;
protected:
char name[30];
public:
Person(char *aname,int i1): i(i1) { strcpy(name,aname); }
~Person(void) { }
char *getname(void) { return name; }
int geti(void) { return i; }
void addto i(int i1) { i+=i1; }
int sub(int i1) { cout << " Only the trustee can withdraw"; return 0; }
};
class Trustee {
private:
Person *client;
public:
```

```
Trustee(void) {}

Trustee(Person *aclient) { client=aclient; }

char *getclient(void) { return client->name; }

void addtoi(int i1) { client->i+=i1; }

};

class JuniorTristee: private Trustee {

public:

JuniorTrustee(void) {}

JuniorTrustee(Person *aclient):TruStee(aclient) {}


// методы доступа должны быть переопределены как для private-//производного класса

char *getclient(void) { return Trustee::getclient(); }

void addtoi(int i1) { Trustee::addtoi(i1); }

};

main()

{

Person *p1=new Person(“Lei”,10000);

JuniorTrustee t1(p1);

}
```

Вопросы:

1. Что если, класс aJuniorTrustee сделать public-производным классом от класса Trustee.

Тогда объекты aJuniorTrustee получат доступ к открытым методам класса Trustee, которые имеют прямой доступ к закрытым данным aPerson. Методы доступа наследуемы, но дружественность -нет.

Суммарные правила доступа для дружественных и производных классов к закрытым, защищенным и открытым частям описания класса

На рис. Показаны все изменения в возможности доступа через дружественные классы и их производные классы к базовому классу и его производным классам.

На рис. не освещены возможности доступа к полям базового класса, которые имеют дружественные функции производных классов. Так как открытые и защищенные элементы открытого базового класса становятся открытыми и защищенными элементами производного класса, дружественная функция производного класса получает прямой доступ к этим полям. Это справедливо и для закрытого базового класса. В этом случае защищенная часть базового класса включается в закрытую часть производного класса. Но дружественная функция производного класса имеет прямой доступ к закрытой части производного класса и, следовательно, получает прямой доступ к защищенным полям базового класса. Она не имеет доступа к закрытой части базового класса.

```
class Base {

protected:

int i;

public:

Base(void) { i=12; }

};

class Child: private Base {

friend void tester(Child child);

public:

Child(void) { i=17; }

int geti() { return i; }

};

void tester(Child child) { cout << " in child " << child.i; }

void main() {

Child c;

cout << c.geti() ; tester(c); }
```

Объявления доступа

- Объявления доступа дают возможность сделать опять защищенными или открытыми защищенные или открытые члены закрытого базового класса в производном классе соответственно

- Объявления доступа позволяют сделать снова открытыми открытые члены базового класса в защищенном производном. (Ни этот, ни предыдущий пункт не выполняется, если в производном классе объявлены члены с теми же именами, что и в базовом классе.
- Для перегруженных функций позволяют вернуть им первоначальное ограничение доступа, которое они имели в базовом классе, если все они имели одинаковое ограничение доступа.

```
class Base {

int a; void g();

public:

int b,c; void f(), f(int), g(int);

};

class Derived : private Base {

public:

Base::a; // Ошибка нельзя сделать a -public

Base::b; // Вновь делает b public

int c;

Base::c; // Ошибка: нельзя с объявлять дважды

Base::f; // Вновь делает все f() public

Base::g; // Ошибка: функции g() имеют различное ограничение доступа

};
```

Виртуальные функции и полиморфизм

План

1. Виртуальные функции и полиморфические кластеры
1. Проверка ошибок при использовании виртуальных функций
1. Техническая реализация виртуальных функций
1. Абстрактные базовые классы
1. Преимущества позднего связывания

Термин **полиморфический** в словаре определен, как “имеющий, принимающий или встречающийся в различных формах, символах или стилях”. В применении к объектно-ориентированным языкам, полиморфизм рассматривается как существенное свойство, позволяющее некоторое сообщение передавать различными путями. Следовательно, сообщение может иметь множество различных реализаций.

До этого связывание сообщения, посылаемого объекту, с конкретным методом (функцией-членом) осуществлялось на этапе компиляции (т.е. до запуска программы). Такое **раннее связывание** во многих случаях желательно, так как это позволяет получить наиболее оптимальный код. Кроме раннего связывания компилятор C++ выполняет проверку типа, гарантируя, что каждому объекту послано законное сообщение. В действительности, объектно-ориентированный язык предлагает программисту выбор между проверкой типов и поздним связыванием.

Позднее связывание позволяет ассоциировать сообщение с методом во время выполнения программы. Программист определяет специфические действия, которые должен выполнять объект, получив сообщение. Во время выполнения, программа интерпретирует эти действия и связывает сообщение с соответствующим методом. Просмотр списка методов и выбор нужного возлагается не на программиста, а на программу. Это позволяет писать более надежный код и облегчает его использование и модификацию.

Виртуальные функции и полиморфические кластеры

В C++ позднее связывание охватывает ряд функций-членов (методов), называемых **виртуальными функциями**. Виртуальная функция объявляется в базовом или в производном классе и, затем, переопределяется в наследуемых классах. Совокупность классов (подклассов), в которых определяется и переопределяется виртуальная функция, называется полиморфическим кластером, ассоциированным с некоторой виртуальной функцией. В пределах полиморфического кластера сообщение связывается с конкретной виртуальной функцией-членом во время выполнения программы.

В Smalltalk, чисто объектно-ориентированном языке, полиморфизм охватывает все методы в системе, поэтому вся система Smalltalk является полиморфическим кластером.

Обычную функцию-член можно переопределить в наследуемых классах. Однако без атрибута **virtual** такая функция-член будет связана с сообщением на этапе компиляции. Атрибут **virtual** гарантирует позднее связывание в пределах полиморфического кластера.

Чтобы добиться позднего связывания для объекта, его нужно объявить как указатель или ссылку на объект соответствующего класса. Для открытых производных классов указатели и ссылки на объекты этих классов совместимыми с указателями и ссылками на объекты базового класса (т.е. к объекту производного класса можно обращаться, как будто это объект базового класса). Выбранная функция-член зависит от класса, на объект которого указывается, но не от типа указателя.

Часто возникает необходимость передачи сообщений объектам, принадлежащим разным классам в иерархии. Каждый класс в иерархии отвечает на сообщение по-своему. В этом случае требуется реализация механизма позднего связывания. В C++ этот механизм реализован для виртуальных функций (объявление с использованием ключевого слова virtual).

C++ поддерживает **virtual** функции-члены, которые объявлены в основном классе и переопределены в порожденном классе. Иерархия классов, определенная общим наследованием, создает связанный набор типов пользователя, на которые можно ссылаться с помощью указателя базового класса. При обращении к виртуальной функции через этот указатель в C++ выбирается соответствующее функциональное определение во время выполнения. Объект, на который указывается, должен содержать в себе информацию о типе, поскольку различия между ними может быть сделано динамически. Это особенность типична для ООП кода. Каждый объект “знает” как на него должны воздействовать. Эта форма полиморфизма называется чистым полиморфизмом.

Виртуальные функции позволяют управляемым классам определять различные версии функций базового класса. В C++ функции-члены класса с различным числом и типом параметров есть действительно различные функции, даже если они имеют одно и тоже имя. Виртуальные функции позволяют переопределять в управляемом классе функции, введенные в базовом классе, даже если число и тип аргументов то же самое. Для виртуальных функций нельзя переопределять тип функции. Если две функции с одинаковым именем будут иметь различные аргументы, C++ будет считать их различными и проигнорирует механизм виртуальных функций. Для определения виртуальной функции служит ключевое слово *virtual*. Виртуальная функция обязательно член класса. Например:

```
struct B {

virtual void vf1();

virtual void vf2();

virtual void vf3();

void f();

};

class D:public B {

public:

virtual void vf1(); // виртуальная функция

void vf2(int); // не виртуальная функция

// другой список аргументов
```

```
char vf3(); // недопустимо - изменен тип
void f();
};
void main() {
D d; B *bp=&d;
bp->vf1(); // вызов D::vf1
bp->vf2(); // вызов B::vf2
bp->f(); // вызов B::f
}
```

Рассмотрим механизм создания полиморфического кластера виртуальной функции.

```
class Parent {
protected:
char *lastName;
public:
Parent(void) { lastName=new char[5]; strcpy(lastName,"None"); }
Parent(char *a) { lastName=new char [strlen(a)+1]; strcpy(lastName,a); }
Parent(Parent &a) {
        lastName=new char[strlen(a.lastName)+1]; strcpy(lastName,a.lastName);
}
char *getLastName(void) { return lastName; }
void setLastName(char *a) {
        lastName=new char[strlen(a)+1]; strcpy(lastName,a);
}
virtual void answerName(void) { cout << "My last name is" << lastName << " }
~Parent(void) { delete lastName; }
};
```

```
class Child: public Parent {
protected:
char *firstName;
public:
Child(void) { firstName=new char[5]; strcpy(firstName,”None”); }
Child(char *a,char *a1) : Parent(a){
        firstName=new char[strlen(a1)+1]; strcpy(firstName,a1);
}
Child(Child &a): Parent(a) { setLastName(a.getLastName());
        firstName=new char[strlen(a.firstName)+1]; strcpy(firstName,a.firstName);
}
char *getFirstName(void) { return firstName; }
void setFirstName(char *a) {
        firstName=new char[strlen(a)+1]; strcpy(firstName,a);
}
~Child(void) { delete firstName; }
virtual void answerName(void) {
        Parent::answerName(); cout << “My first name is ” << firstName << “\n”;
}
};
class GrandChild: public Child {
private:
char *grandFatherName;
public:
GrandChild(char*a,char*a1,char*a2): Child(a,a1){
grandFatherName=new char[strlen(a2)+1]; strcpy(grandFatherName,a2);
```

```
}
~GrandChild(void) { delete grandFatherName; }
virtual void answerName(void) { Child::answer();
cout << “My grandfather name is “ << grandFatherName << “\n”; }
};
void main(void) {
Parent *family[3];\
Parent *p=new Parent(“ Иванов ”);
Child *c=new Child(“ Иванов ”,”Сергей”);
CrandChild *g=new GrandChild(“Иванов”,”Андрей”,”Владимир”);
family[0]=p; family[1]=c; family[2]=g;
for(int index=0; index<3; index++)
family[index]->answerName();
}
```

Результат работы программы

```
My last name is Иванов
My last name is Иванов
My first name is Сергей
My last name is Иванов
My first name is Андрей
My grandfather’s name is Владимир
```

Вопросы:

Что, если в производных классах Child *и* GrandChild *опустить ключевое слово* virtual *перед функцией-членом* answerName*?*

Программа будет работать отлично. Ключевое слово virtual необходимо в полиморфическом кластере только перед виртуальной функцией самого верхнего уровня. Для всех остальных переопределений этой функции ключевое слово virtual необязательно.

Что, если убрать ключевое слово virtual *перед всеми функциями-членами* answerName?

На выходе получим:

My last name is Иванов

My last name is Иванов

My last name is Иванов

Для реализации полиморфизма можно использовать ссылки на объекты вместо указателей.

```
void main(void) {
Parent p(“Иванов”);
Child c(“Иванов”,”Сергей”);
CrandChild g(“Иванов”,”Андрей”,”Владимир”);
Parent &f0=p, &f1=c; &f2=g;
f0.answerName();f1.answerName();f2.answerName();
}
```

Рассмотрим тонкости виртуальной функции

```
class P {
private:
virtual void method1(void) {}
void method2(void){}
void method3(void){method1(); method2();} // this->method1, this->method2
void method5(void){}
public:
void method6(void){method3();}
void method4(void){}
void method7(void){method5(); }
```

```
};

class C:public P {

private:

void method1(void) {}

void method5(void) {}

public:

C(){}

void method4(void) {}

void method7(void) { method5(); }

};

void main() {

P pop;

C me;

pop.method6(); // P::method1, P::method2

me.method6(); // C::method1, P::method2

pop.method4(); // P::method4

me.method4(); // C::method4

pop.method7(); // P::method5

me.method7(); // C::method5

}
```

Сообщение *method6()* посылается объекту *me*. Это сообщение унаследовано из класса *P*. Вызывается защищенный метод *method3()* и, затем, защищенные методы *method1()* и *method2()*. Так как *method1()* объявлен в базовом классе *P* виртуальным, система на этапе выполнения программы связывает сообщение *this->method1()* с *method1()* класса *C*.

При перегрузке функции-члена виртуальный функции способны вызвать смешивание и беспорядок.

```
class B {

public:
```

```
virtual foo(int);

virtual foo(double);

};

class D: public B {

public:

foo(int);

};

void main() {

D d; B b, *pb=&d;

b.foo(9); // B::foo(int);

b.foo(9.5); // B::foo(double)

pb->foo(9.5); // B::foo(double);

pb->foo(9); // D::foo(int)

}
```

Функция-член базового класса B::foo(int) в порожденном переопределена. Функция-член базового класса B::foo(double) в порожденном классе скрыта.

Virtual могут быть только нестатические функции-члены. Функция порожденного класса автоматически становится виртуальной. Специальным ограничением является: деструкторы могут быть виртуальными, а конструкторы - нет.

Рассмотрим пример преимущества использования виртуальных функций для упрощения кода:

```
class shape {

protected:

double x,y;

public:

virtual double area() { return 0.; }

};

class rectangle: public shape {
```

```
double height,width;

public:

double area() { return (height*width); }

};

class circle: public shape {

double radius;

public:

double area() { return (PI*radius*radius); }

};
```

Вычисление площади является локальной ответственностью порожденного класса. Код пользователя, который использует полиморфное вычисление площади, может выглядеть так:

```
shape *p[N];

.....

for(i=0; i<N; i++) tot_area+=p[i]->area();
```

Здесь главное преимущество состоит в том, что код пользователя не нуждается в изменении, даже если к системе добавляются новые формы. Изменения управляются локально и распространяются автоматически, благодаря полиморфному характеру кода пользователя.

Проверка ошибок при использовании виртуальных функций

В некоторых объектно-ориентированных языках, которые поддерживают позднее связывание, проверка ошибочных сообщений осуществляется на этапе выполнения программы. C++, посредством виртуальных функций, поддерживает и статический контроль сообщений, и позднее связывание.

```
class P {

public:

P(void){}

virtual void hello(void) {}

};

class C: public P {

public:
```

```
C(void) {}

virtual void hello(void) { cout << “Hello world”; }

};

void main() {

P *p; C c;

p=&c;

p->hello(“Hello”); // ошибка

}
```

Компилятор выдаст ошибку на строке p->hello(“hello”). Компилятор может определить, что виртуальная функция hello не имеет параметров, даже если сообщение связывается с методом hello класса C во время выполнения программы. Виртуальные функции, определенные в родительском или потомственных классе имеют одинаковые списки параметров, так как позднее связывание не оказывает влияния на контроль типов параметров.

Техническая реализация виртуальных функций

Объект C++ представляет собой непрерывный участок памяти. Указатель на такой объект содержит начальный адрес этого участка. Когда вызывается функция-член (объекту посылается сообщение), этот вызов транслируется в обычный вызов функции с дополнительным аргументом, который содержит указатель на объект.

Например,

```
ClassName *object;

object->message(10);
```

преобразуется в

```
ClassName_message(object,10);
```

При создании объектов производных классов их поля сцепляются с полями родительских классов. Эту функцию выполняет компилятор.

Виртуальные функции реализованы с использованием таблиц функций. Рассмотрим следующие классы.

```
class P {

int value;

public:

virtual int method1(float r):
```

```
virtual void method2(void);

virtual float method3(char *s);

};

class C1:public P {

public:

void method2(void);

};

class C2: public C1 {

public:

float method3(char *s);

};
```

Таблица виртуальных функций, virtualTabl, содержит функции-члены каждого класса в полиморфическом кластере. Указатель на эту таблицу имеют все объекты классов и подклассов полиморфического кластера.

Типичный объект, приведенных выше классов, выглядит, примерно, так:

```
int value;

virtual_table->P::method1

C1::method2

C2::method3
```

Компилятор преобразует вызов виртуальной функции в косвенный вызов через virtualTable. Например,

```
C2 *c;

c->method3(“Hello”);
```

преобразуется в

```
(*(c->virtual_table[2]))(c,”Hello”);
```

Абстрактные базовые классы

Базовый класс иерархии типа обычно содержит ряд виртуальных функций, которые обеспечивают динамическую типизацию. Часто в базовом классе эти виртуальные функции фиктивны и имеют пустое тело. Определенное значение им придают в порожденных классах. В C++ для этой цели применяется

чистая виртуальная функция. Чистая виртуальная функция - виртуальная функция-член, тело которой обычно не определено. Запись этого объявления внутри класса следующая:

virtual *прототип функции = 0;*

Чистая виртуальная функция используется для того, чтобы “отложить” решение о реализации функции. В ООП терминологии это называется *отсроченным методом.*

Класс, имеющий, по крайней мере, одну чистую виртуальную функцию - *абстрактный класс*. Для иерархии типа полезно иметь базовый абстрактный класс. Он содержит общий базовые свойства порожденных классов, но сам по себе не может использоваться для объявления объектов. Напротив, он используется для объявления указателей, которые могут обращаться к объектам подтипа, порожденным от абстрактного класса.

Объясним эту концепцию при помощи разработки примитивной формы экологического моделирования. В нашем примере будем иметь различные формы взаимодействия жизни с использованием абстрактного базового класса **living**. Создадим fox (лису) как типичного хищника, и rabbit (кролика) как его жертву. Rabbit есть grass (траву).

```
const int N = 40, STATES=4 ; // размер квадратной площади

enum state { EMPTY, GRASS, RABBIT, FOX };


class living;

typedef living *world[N][N]; // world будет моделью


class living { // что живет в мире

protected:

int row, column; // местоположение

void sums(world, int sm[]); // sm[#states] используется next

public:

living(int r,int c):row(r),column(c) {}

virtual state who()=0; // идентификация состояний

virtual living* next(world w)=0; // расчет next

};
```

```
void living::sums(world w,int sm[]) {
sm[EMPTY]=sm[GRASS]=sm[RABBIT]=sm[FOX]=0;
for(int i=-1; i <= 1; i++)
for(int j=-1; j <= 1; j++) sm[w[row+i][column+j]->who()]++;
}

// текущий класс - только хищники
class fox: public living {
protected:
int age; // используется для принятия решения о смерти
public:
fox(int r,int c,int a=0):living(r,c),age(a) {}
state who() { return FOX; } // отложенный метод для FOX
living* next(world w); // отложенный метод для FOX
};
// текущий класс - только жертвы
class rabbit: public living {
protected:
int age; // используется для принятия решения о смерти
public:
rabbit(int r,int c,int a=0):living(r,c),age(a) {}
state who() { return RABBIT; } // отложенный метод для RABBIT
living* next(world w); // отложенный метод для RABBIT
};
// текущий класс - только растения
class grass: public living {
```

```
public:

grass(int r,int c):living(r,c) {}

state who() { return GRASS; } // отложенный метод для GRASS

living* next(world w); // отложенный метод для GRASS

};

// жизнь отсутствует

class empty: public living {

public:

empty(int r,int c):living(r,c) {}

state who() { return EMPTY; } // отложенный метод для EMPTY

living* next(world w); // отложенный метод для EMPTY

};
```

Обратите внимание на то, что проект позволяет развивать и другие формы хищника, жертвы и растительной жизни, используя следующий уровень наследования. Характеристики поведения каждой формы жизни фиксируется в версии next().

```
Living* grass::next(world w) {

int sum[STATES];

sums(w,sum);

if(sum[GRASS] > sum[RABBIT) // есть траву

return (new grass(row,column));

else

return (new empty(row,column));

}
```

Grass может быть съеден Rabbit. Если в окрестности имеется больше grass, чем rabbit, grass остается, иначе - grass будет съедена.

Преимущества позднего связывания

В C++ преимущество различной реакции объекта на сообщение можно получить в полиморфическом кластере. Можно создать коллекцию ссылок или указателей на объекты. Эти ссылки(указатели) должны

быть объявлены как ссылки или указатели на объекты базового класса или одного из производных классов в полиморфическом кластере. Любая попытка загрузить указатель или ссылку на объект, не входящий в полиморфический кластер, закончится неудачей (компилятор выдаст сообщение о несоответствии типов). Ссылки и указатели на объекты базового класса совместимы только с ссылками или указателями на объекты public-производного класса.

В языках поддерживающих только раннее связывание ответственность за вызов соответствующей версии функции лежит на программисте. Обычно для выполнения такого контроля, используются операторы множественного выбора, такие как операторы switch, if else. Каждый дополнительный выбор, вводимый в систему, необходимо включить во все операторы множественного выбора. При этом программа изменяется во многих местах..

В языках, поддерживающих позднее связывание, как C++, при необходимости внести дополнение в программу, создаются новые производные классы. Дополнительные возможности реализуются через виртуальные функции этих классов. Такая возможность позволяет минимизировать модификацию программы.

Виртуальные деструкторы

C++ позволяет об’явить деструктор виртуальным, так же как и обычную член-функцию. Тогда деструкторы всех классов, порожденных из класса, в котором об’явлен виртуальный деструктор, так же будут виртуальными.

Использование виртуальных деструкторов позволяет обеспечить вызов соответствующего дестрктора при разрушении об’ектов оператором delete, даже если тип разрушаемого об’екта неизвестен на стадии компиляции..

```
class Figure {

public:

Figure();

virtual ~Figure();

};

typedef Figure *PFigure;

class Circle:public Figure {

public:

Circle(int centerx, int centery,int radius);
```

```
virtual ~Circle();
};
class Rectangle:public Figure {
public:
Rectangle(int left, int top, int right, int bottom);
~Rectangle();
};
int main();
{
const ALLFigures = 2;
PFigures figures[ALLFigures];
figures[0]=new Circle(100,100,10); // массив указателей на фигуры
figures[1]=new Rectangle(100,100,200,300);
for(int count=0; count<ALLFigures; count++)
delete figures[count]; // уничтожение массива
}
```

Производные классы: множественное наследование

План

1. Первый пример
1. Конфликт имен
1. Порядок вызова конструкторов
1. Виртуальные базовые классы

Иерархию простого наследования можно описать, используя структуру дерева, где каждый узел представляет подкласс, который может порождать любое количество дополнительных подклассов. Как и в случае простого наследования, определители *private, protected, public* в родительском классе можно использовать для управления доступом к экземплярам переменных и методам, которые унаследованы производным классом (подклассом) от базового (родительского) класса. Кроме того, спецификатор *public* или *private*-производного класса, как и при простом наследовании, определяют производные классы, объекты которых имеют простой доступ к открытым данным или функциям-членам базового класса.

Для описания иерархии множественного наследования можно использовать прямой ациклический граф (ПАГ) (Ориентированный граф наследования без петель). Подкласс может унаследовать протокол одного или более родительских классов. При этом помимо спецификаторов *public* и *private*-производных классов, используется дополнительная опция *virtual*.

Первый пример

Класс *Derived* является производным от обоих базовых классов *Base1* и *Base2*.

```
class Base1 {

int id;

public:

Base1(void) { cout << " Constructor Base1"; id=0; }

Base1(int anid) { cout << " Constructor Base1"; id=anid; }

void assignid(int anid) { id=anid; }

int accessid(void) { return id; }

};

class Base2 {

char name[20];
```

```
public:
Base2(void) { cout << " Constructor Base2"; strcpy(name,"void"); }
Base2(char *str) { cout << " Constructor Base2"; strcpy(name,str); }
void assignName(char *str) { strcpy(name,str); }
char *accessName(void) { return name; }
};
class Derived: public Base1,public Base2 {
char ch;
public:
Derived(void) { cout << " Construct Derived"; ch='a'; }
Derived(char c,int anid,char *str):Base1(anid),Base2(str) {
cout << " Construct Derrived"; ch=c; }
void assign(char c) { ch=c; }
friend ostream& operator<<(osream& o,Derived& d);
};
ostream& operator<<(ostream&o,Derived& d) {
o<<"id"<<d.accessid()<<"name"<<d.accessName() <<" ch"<<c; return o;
}
main() {
Derived object1; cout << object;
Derived object2('e',26,"Robert:); cout << object2;
}
```

Результаты работы программы

```
Constructor Base1 Constructor Base2 Construct Derived id 0 name void ch a
Constructor Base1 Constructor Base2 Construct Derived
id 26 name Robert ch e
```

Когда *object1* объявлен при помощи выражения *Derived object1*, вызываются *void*-конструкторы двух базовых классов в той последовательности, в которой определено наследование (*Base1*-конструктор первый, *Base2*-конструктор второй).

Вопрос:

1. Что, если переписать конструктор Derived с тремя параметрами, как приведено ниже:

```
Derived(char c,int anint,char *str): Base2(str),Base1(anint) { ch=c; }
```

Результат работы программы будет таким же.

2. Что, если переписать перегруженный оператор <<, как показано:

```
ostream & operator <<(ostream&o,Derived& d) {
o<<“id”<<d.id<<“name”<<d.name<<“ ch”<<d.c; return o;
}
```

Компилятор выдаст сообщения об ошибке. Производный класс в иерархии множественного наследования не имеет доступа к защищенным экземплярам переменных любых его базовых классов. Оператор вывода << является дружественным классу *Derived*, но не *Base1* и *Base2*. Поэтому он не имеет прямого доступа к защищенным членам *Base1* и *Base2*.

3. Что, если экземпляры переменных в классах Base1 и Base2 перенести в защищенные секции и изменить перегруженный оператор <<, как предложено во втором вопросе.

Программа будет отлично работать. Защищенные экземпляры переменных одного или более базовых классов полностью доступны производным классам.

Конфликт имен

А что, если один или более элементов двух базовых классов имеют одинаковые имена. Предположим, например, что экземпляры переменных в базовых классах *Base1* и *Base2* (вариант, в котором вместо *private* использовали *protected*) переименованы в *instance*.

Появится неоднозначность в определении перегруженного оператора <<. Компилятор не сможет отличить *d.instnace*, ссылающийся на класс *Base1*, от *d.instance*, ссылающийся на класс *Base2*. Если нет обращения на прямую, то все будет работать нормально.

Если данные описаны как *private* (либо, если для доступа к ним все равно используются функции доступа), то программа будет работать, так как протокол класса *Derived* может получить доступ к экземплярам переменных двух базовых классов, только используя методы доступа *accessid*, *accessName*. Только при попытке обратиться к экземплярам напрямую вы потерпите неудачу из-за конфликта имен.

Рассмотрим пример, когда два базовых класса содержат поля с одинаковыми именами.

```
class Base1 {

protected:

int id;

public:

Base1(void) { id=0; }

Base1(int anid) { id=anid; }

};

class Base2 {

protected:

int id;

public:

Base2(void) { id=0; }

Base2(int anid) { id=anid; }

};

class Derived: public Base1,public Base2 {

char ch;

public:

Derived(void) { ch='a'; }
```

```
Derived(char c,int anid,int aid):Base1(anid),Base2(aid) { ch=c; }

void assign(char c) { ch=c; }

friend ostream& operator<<(osream& o,Derived& d);

};

ostream& operator<<(ostream&o,Derived& d) {

o<<“id Base1”<<d.Base1::id;

o<<“id Base2”<<d.Base2::id;

o<<“ ch”<<c; return o;

}

main() {

Derived object1; cout << object;

Derived object2(‘e’,120,-150); cout << object2;

}
```

Результаты работы программы

```
id Base1 0 id Base2 0 ch a

id Base2 120 id Base2 ch e
```

Объекты класса *Derived* содержат по два экземпляра переменных каждый. Одна *id* из класса *Base1*, а другая - *id* из класса *Base2*. Доступ к той или иной переменной в функции-члене может быть осуществлен *Base1::id* или *Base2::id*, в дружественной функции через квалификатор *d.Base1::id* или *d.Base2::id.*

Рассмотрим еще один пример.

```
class Base1 {

public:

int cost();

};

class Base2 {
```

```
public:

int cost();

};

class Der: public Base1, public Base2 {

public:

int tot_cost() { return (Base1::cost()+Base2::cost()); }

};

void main() {

Der der;

der.cost(); // неоднозначность

}
```

Компилятор выдаст сообщение об ошибке - неоднозначное обращение. Эта проблема может быть устранена или правильной квалификацией **cost** с использованием оператора разрешения контекста, или добавлением члена **cost** к порожденному классу.

Порядок вызова конструкторов

При использовании множественного наследования, в протоколе производного класса необходимо вызывать конструктор базовых классов для инициализации полей данных, наследуемых от двух или более базовых классов, и инициализировать различные элементы объектов. Четко определенный порядок инициализации приведен ниже:

- Инициализация осуществляется в порядке, определяемом порядком объявления
- Члены инициализируются в порядке объявления
- Void-конструкторы базовых классов, которые явно не указаны в инициализирующем списки, вызываются после конструкторов явно инициализируемых базовых классов, в том порядке, в котором они следуют в объявлении класса. Эти void-конструкторы вызываются перед любыми конструкторами полей данных.

```
class Parent1 {

int p1;

public:

Parent1(void) { cout << " Construct Parent1 free parameter"; p1=0; }

Parent1(int ap):p1(ap) { cout << "Construct Parent1 1 parameter"; }
```

```
~Parent1(void) { cout << " destruct Parent1"; }
};
class Parent2 {
int p2;
public:
Parent2(void) { cout << " Construct Parent2 free parameter"; p2=0; }
Parent2(int ap):p2(ap) { cout << "Construct Parent2 1 parameter"; }
~Parent2(void) { cout << " destruct Parent2"; }
};
class Parent3 {
int p3;
public:
Parent3(void) { cout << " Construct Parent3 free parameter"; p3=0; }
Parent3(int ap):p3(ap) { cout << "Construct Parent3 1 parameter"; }
~Parent3(void) { cout << " destruct Parent3"; }
};
class Member {
int m;
public:
Member(void) { cout << " Construct Member free parameter"; m=0; }
Member(int ap):m(ap) { cout << "Construct Member 1 parameter"; }
~Member(void) { cout << " destruct Member"; }
};
class Child: public Parent3,public Parent2,public Parent3 {
Member mem1,mem2 ;
public:
```

```
Child(void) { cout << " construct child free parameter"; }
Child(Member achildMember): childMember(achildMember) {
cout << " Construct Child with 1 parameter"; }
Child(int v1,int v2,int v3,int m1,int m2): mem2(m1),
Parent1(v1),Parent3(v3),Parent2 (v2),mem1(m2){
cout <<" construct child with 3 parameter"; }
~Child(void) { cout << "destruct Child"; }
};
main() {
Member m(16); // Construct Member with 1 parameter
// destruct Member
Child child1; // Construct Parent3 free parameter
// Construct Parent2 free parameter
// Construct Parent1 free parameter
// Construct Member free parameter
// Construct Member free parameter
// construct child free parameter
// destruct Child
// destruct Member
// destruct Member
//destruct Parent1
// destruct Parent2
// destruct Parent3

Child child2(m); // Construct Parent3 free parameter
// Construct Parent2 free parameter
```

// Construct Parent1 free parameter

// Construct Member free parameter

// Construct Child with 1 parameter

// destruct Child

// destruct Member

// destruct Member

// destruct Parent1

// destruct Parent2

// destruct Parent3

Child child3(10,20,30,40,50); // Construct Parent3 with 1 parameter

// Construct Parent2 free parameter

// Construct Parent1 with 1 parameter

// Construct Member 1 parameter 50

// Construct Member 1 parameter 40

// construct child with 3 parameter

// destruct Child

// destruct Member

// destruct Member

// destruct Parent1

// destruct Parent2

// destruct Parent3

}

Виртуальные базовые классы

В прямом ациклическом графе наследования класс может появиться более чем один раз. Рассмотрим ПАГ множественного наследования, приведенный на рис. Элементы данных (экземпляры переменных) класса *Parent* появляются дважды в классе *GrandChild*. Первый набор наследуется через *Child1*, второй

- через *Child2*. Такое наследование бывает нежелательно. Виртуальные базовые классы обеспечивают механизм для избежания дублирования элементов в классе, таком как *CrandChild*

Допустим, класс *Parent* на самом деле называется *Vehicle* (транспортное средство). В его протоколе содержится закрытое поле данных *int topSpeed*. Класс *Child1* на самом деле называется *Boat*(Корабль), а класс *Child2* -называется *Plane* (самолет). Наконец, класс *GrandChild* на самом деле называется *SeaPlane*. Для *SeaPlane* желательно наследовать *Boat::topSpeed* и *Plane::topSpeed*. Объект класса *SeaPlane* имеет различную предельную скорость в зависимости от того, действует он как корабль или самолет.

С другой стороны, допустим, класс *Parent* назван *DomesticAnimal* (домашнее животное), класс *Child1* назван *Cow* (корова), класс *Child2* назван *Buffalo* (бык), а класс *GrandChild* назван *Beefalo*(теленок). Допустим, протокол класса *DomesticAnimal* включает экземпляры переменных *int weight, float price, char color[20]*. Нежелательно, чтобы протокол класса *Beefalo* имел по два экземпляра переменных *weight*, *price*, *color*. Избежать дублирования позволяет использование виртуальных базовых классов.

```
class DomesticAnimal {

protected:

int weight; float price; char color[20];

public:

DomesticAnimal(void) {weight=0; price=0.; strcpy(color,”none”); }

DomesticAnimal(int aweight, float aprice,char *acolor) {

weight=aweight; price=aprice; strcpy(color,acolor);
```

```
}
virtual void print(void) { cout << weight << price << color; }
};
class Cow: public virtual DomesticAnimal {
public:
Cow(void) { }
Cow(int aweight,float aprice,char *acolor) {
weight=aweight; price=aprice; strcpy(color,acolor); }
void print(void) { cout << " Cow has propeties"; DomesticAnimal::print(); }
};
class Buffalo: public virtual DomesticAnimal {
public:
Buffalo(void) { }
Buffalo(int aweight,float price,char *acolor) {
weight=aweight; price=aprice; strcpy(color,acolor); }
void print(void) { cout<< " Buffalo has propeties";
DomesticAnimal::print(); }
};
class Beefalo: public Cow,public Buffalo {
public:
Beefalo(int aweight,float aprice,char *acolor) {
weight=aweight; price=aprice; strcpy(color,acolor); }
void print(void) { cout << " beefalo has propeties";
DomesticAnimal::print(); }
};
main() {
```

```
Cow aCow(1400,375.0,”black and white”); // void const Domestic

// const par Cow

Beefalo aBeefalo(1700,525.0,”Brown and black”); // void cons Domestic

// void const Cow

// void const Buffalo

// const par Beefalo

DomesticAnimal& myCow=aCow;

DomesticAnimal& myBeefalo=aBeefalo;

my.Cow.print();

myBeefalo.print();

}
```

Результат работы

```
Cow has propeties

weight=1400 price=375.0 color=black and white

beefalo has propeties

weight=1700 price=525.0 color=brown and white
```

Код приведенной выше программы решает проблему класса *Beefalo*, который наследует поля данных *weight, price, color*. Объекты класса *Beefalo* имеют по одному полю данных для веса, цены и цвета.

- Виртуальные базовые классы инициализируются (вызывается *void*-конструктор) перед любыми не виртуальными базовыми классами и в том порядке, в котором они появляются в ПАГе наследования при просмотре его снизу-вверх и слева направо.
- Если виртуальный базовый класс имеет хотя бы один конструктор, то он должен иметь *void*-конструктор.

Ключевое слово *virtual* в классе *Cow* и классе *Buffalo* предотвращает многократное копирование полей данных *weight,price,color* из предков класса B*eefalo*.

Вопрос:

1. Что, если из объявлений классов Cow *и* Buffalo *убрать ключевое слово* virtual.

Компилятор выдаст сообщение об ошибке. В *Beefalo* унаследовано несколько копий полей *weight,price,color*, следовательно, конструктор с несколькими параметрами в классе *Beefalo* становится недействительным.

2. *Что, если конструктор класса* Cow *заменить на*

Cow(int aweight,float aprice,char *acolor):

DomesticAnimal(aweight,aprice,acolor) { }

Вместо void конструктора DomesticAnimals вызовется конструктор с параметрами.

3. *Что, если конструктор с параметрами класса* Beefalo *заменить на*

Beefalo(int aweight,float aprice,char *acolor) :

DomesticAnimal(aweight,aprice,acolor){}

Все будет работать нормально. Вначале вызовется конструктор базового виртуального класса с параметрами, затем конструкторы наследуемых классов без параметров. Вызова конструктора без параметра виртуального базового класса не будет.

Другой пример

class W{

public:

virtual void f(void) { cout << “ W::f”; }

```
virtual void g(void) {cout << " W::g"; }
virtual void h(void) {cout << " W::h"; }
virtual void k(void) {cout << " W::k"; }
};
class A: public virtual W {
public:
void g(void) { cout << " A::g";}
};
class B: public virtual W {
public:
void f(void) { cout << " B::f"; }
};
class C: public A,public B, public virtual W {
public:
void h(void) { cout << "C::h"; }
void f(void) { B::f(); }
void g(void) { A::g(); }
};
main() {
C *pc=new C;
pc->f(); // C::f() ( B::f)
pc->g(); // C::g() ( A::g)
pc->h(); // C::h
((A *)pc)->f(); // C::f (B::f)
((W *)pc)->f(); // C::f (B::f)
```

```
B *pb=new B;

pb->f(); // B::f

pb->g(); // W::g

((W *)pb)->f(); // B::f


A *pa=new A;

pa->f(); // W::f

pa->g(); // A::g

((W *)pa)->g(); // A::g

}
```

При вызове *((A *)pc)->f()* выполняется функция *f* класса *C*. Тоже происходит и при вызове *((W *)pc)->f()*.Таблица виртуальных функций использует определение *f* в классе *C*. Это демонстрирует то, что при вызове “верхней” виртуальной функции по одному пути ПАГа может закончиться вызовом функции по другому пути.

Вопрос

1. *Что, если в Классе A определить функцию член f.*

Последняя часть работы программы измениться

A::f A::g A::g

Если виртуальный базовый класс и производный класс разделяют имя какого-либо поля, функции-члена или перечисления, то имя в производном классе скрывает имя в виртуальном базовом классе.

```
class V { public: int i; void f(); };

class A: virtual public V {public: int i; int f(); };

class B: virtual public V { };

class C: public A, public B {
```

```
void g() { i=f(); } // Правильно A::i и A::f() скрывают V::i V::f()

};
```

Их конструкторы вызываются конструктором последнего класса в цепочке производных классов.

```
V::V(int n): i(n) {}

A::A(int i): V(i) {}

B::B(int i): V(i) {}

C::C(int i): V(i), A(i), B(i) {}
```

В объявлении класса могут быть смешаны с невиртуальными базовыми классами. Виртуальные базовые классы могут вызывать нежелательные множественные вызовы функций-членов в виртуальном базовом классе.

```
ckass V { public: void f(); };

class A: virtual public V {

public:

void f() { V::f(); }

};

class B: virtual public V {

public:

void f() { V::f(); }

};

class C: public A,public B {

public:

void f() { A::f(); B::f(); } // V::f() вызывается дважды

};
```

Для преодоления указанного эффекта можно определить функцию real_f(), которая выполняет действия, специфичные для функции f() данного класса. Тогда функция f() будет выполнять последовательные вызовы сначала функции real_f(), а затем - функции f().

```
class V {

protected:
```

```
void real_f() {}
public:
void f() { real_f(); }
};
class A: virtual public V {
protected:
void real_f() {}
public:
void f() { real_f(); V::f(); }
};
class B: virtual public V {
protected:
void real_f() { }
public:
void f() { real_f(); V::f(); }
};
class C: public A, public B {
protected:
void real_f() {}
public:
void f() { real_f(); A::reaal_f(); B::real_f(); V::real_f(); }


};
```

Параметрический полиморфизм. Шаблоны.

План

1. Шаблон класса
1. Шаблон функций
1. Совпадение сигнатуры и перегрузка
1. Шаблоны классов - Дружественность
1. Шаблоны классов - Статические члены
1. Шаблоны классов - Аргументы шаблона класса
1. Шаблоны классов - Наследование.
1. Исключения
1. Исключения в C++
1. Философия восстановления после ошибок

C++ использует ключевое слово **template** для того, чтобы обеспечить *параметрический полиморфизм*. Параметрический полиморфизм позволяет одному и тому же коду использоваться относительно различных типов, где *тип - параметр тела кода*. Параметрический полиморфизм особенно полезен при определении контейнерных классов. Обработка данных в контейнерном классе имеет одну и ту же форму, независимо от типа. Шаблоны определения класса и шаблоны определения функции дают возможность многократно использовать код простым способом, безопасным по отношению к типу, который позволяет компилятору автоматизировать процесс реализации типа. Полиморфизм обеспечивает многократное использование кода.

Шаблон класса

Рассмотри пример реализации шаблона stack

```
template <class TYPE>

class stack {

enum { EMPTY=-1};

TYPE *s;

int len, top;

public:
```

```
stack: len(1000) { s=new TYPE[1000]; top=EMPTY; }

stack(int size): len(size) { s=new TYPE[size]; top=EMPTY; }

~stack() { delete s; }

void reset() { top=EMPTY; }

void push(TYPE c) { s[++top]=c; }

TYPE pop() { return s[top--]; }

TYPE top_of() { return s[top]; }

boolean empty() { return boolean(top==EMPTY); }

boolean full() { return boolean(top==len); }

};
```

Синтаксис объявления класса предваряется

template <class *идентификатор*>

где *идентификатор* - параметр шаблона, который, по существу, заменяет произвольный тип. Через определение класса аргумент шаблона может использоваться как имя типа. Этот аргумент создается при реализации экземпляра в фактических объявлениях. Вот пример использующего его объявления стека:

```
stack <char> stk_ch; // stack из 1000 элементов char

stack <char *> stk_str(200); // stack из 200 элементов char *

stack <complex> stk_cmplx(100); // stack из 100 элементов complex
```

Этот механизм спасает от трудоемкого переписывания объявлений классов в том случае, когда единственным отличием является объявление типа. Эта схема - альтернатива к использованию void * в качестве универсального типа указателя.

При обработки такого типа код всегда должен содержать угловые скобки в виде части объявления.

```
// Реверсирование последовательности char * представляемых строками

void reverse(char *str[], int n) {
```

```
stack <char *>stk(n);

for(int i=0; i<n; i++) stk.push(str[i]);

for(i=0; i<n; i++) str[i]=stk.pop();

}

// Инициализация стека комплексными числами из массива

void init(complex c[], stack <complex> stk, int n) {

for(int i=0; i<n; i++) stk.push(c[i]);

}
```

Функции-члены, объявленные и определенные внутри класса, как и ранее, обычно являются inline. При внешнем определении должно использоваться полное объявление в угловых скобках. Так top_of() при определении вне класса шаблона, будет записано как

```
template <class TYPE> TYPE stack<TYPE>::top_of() { return s[top]; }
```

Шаблон функций

Большинство функций имеют одно и то же тело кода, независимо от типа. Например, инициализация содержимого одного массива от другого того же самого типа. Обычно это код выглядит так:

```
for(i=0; i<n; i++) a[i]=b[i];
```

Данный код можно на С автоматизировать простой макрокомандой.

```
#define COPY(A,B,N) { int i; for(i=0; i<N; i++) A[i]=B[i]; }
```

Она работает, но не безопасно по отношению к типу. Пользователь легко смешивает типы при несоответствующих преобразованиях. Для достижения подобных эффектов на С++ можно использовать различные формы преобразования и перегрузки. Однако, при отсутствии соответствующих преобразований и сигнатур, не будет предприниматься никаких действий. Шаблоны обеспечивают для этого следующий полиморфный языковый механизм:

```
template <class TYPE>

void copy(TYPE a[],TYPE b[],int n) {

for(int i=0; i<n; i++) a[i]=b[i];
```

}

Вызов copy() со специфическими параметрами заставляет компилятор на основании этих параметров, генерировать действительную функцию. Если это невозможно, то возникает ошибка во время компиляции.

```
double f1[50], f2[50];

copy(f1,f2,50);

char c1[25],c2[50];

copy(c1,c2,10);

int i1[75],i2[75];

copy(i1,i2,40);

char *ptr1, *ptr2;

copy(ptr1,ptr2,100);

copy(i1,f2,50); // error

copy(ptr1,f2,50); // error
```

Последние два вызова приведут к ошибке компиляции. Типы фактических параметров не соответствую шаблону. Запись следующего вида не вызовет ошибку

```
copy(i1,(int *)f2,50);
```

Однако при этом будет получена несоответствующая форма копирования. Дело в том, что родовая процедура копирования должна получать в виде параметров два класса отличающегося типа.

```
template class T1, class T2>

void copy(T1 a[], T2 b[], int n) { for(int i=0; i<n; i++) a[i]=b[i]; }
```

В этой форме существует поэлементное преобразование. Оно обычно соответствует и наиболее безопасному преобразованию.

Совпадение сигнатуры и перегрузка

Часто родовая подпрограмма не может работать для специального случая. Следующая форма шаблона обмена (*swapping)* работает для базовых типов.

```
template <class T>

void swap(T &x, T& y) { T temp; temp=x; x=y; y=temp; }
```

Шаблон функции используется, чтобы создавать для любого вызова соответствующую функцию, недвусмысленно соответствующую параметрам.

```
int i.j;

char str1[100],str2[100];

complex c1,c2;

swap(i,j); // i, j -int допустимо

swap(c1,c2); // c1, c2 -complex допустимо

swap(str1[50],str2[33]); // обе -char допустимо

swap(str1,str2); // допустимо

swap(i, chj); // i, -int, ch -char не допустимо
```

Сделаем так, чтобы swap работала для строк, представляемых как символьные массивы; для этого опишем следующий специальный случай.

```
void swap (char *s1, char *s2) {

int len=(strlen(s1) >= strlen(s2)) ? strlen(s1): strlen(s2);

char *temp=new char [len+1]; strcpy(temp,s1); strcpy(s1,s2); strcpy(s2,temp); delete temp;

}
```

С добавлением такого явного случая, при вызове с точным соответствием сигнатуре swap(), имеется преимущество над точным соответствием, найденным с помощью подстановки шаблона.

Алгоритм выбора функции перегрузки следующий:

1. Найти точное соответствие функции не-шаблону.

1. Найти точное соответствие, использующее шаблон функции

1. Обеспечить обычное разрешение параметра для функций не-шаблонов

Шаблоны классов

Дружественность

Шаблоны классов могут содержать друзей. **Friend** функция, не использующая спецификацию шаблона, будет универсальной **friend** для всех экземпляров шаблона класса. **Friend** функция, которая включает шаблоны параметров - особо friend только для того класса, экземпляр которого создается.

```
template <class T>
```

```
class matrix {

friend void foo_bar(); // универсальная

friend vect <T> product(vect <T> v); // создается экземпляр

};
```

Статические члены

Статические члены не универсальны, а специфичны для каждой реализации.

```
template <class T>

class foo {

public:

static int count;

};

template <class T> int foo::count=0;

....

foo <int> a;

foo <double> b;
```

Определены статические переменные foo<int>::count и foo<double>::cout.

Аргументы шаблона класса

Как классы, так и функции могут иметь несколько аргументов шаблона класса. Напишем функцию, которая преобразует значение одного типа к другому типу, при условии, что первый тип, по крайней мере, такой же длины как и второй.

```
template <class T1,class T2 >

boolean coerce(T1& x, T2 y) {

if(sizeof(x) >= sizeof(y)) x=(T1)y;

else false;

return true;

}
```

В этом шаблоне функции есть два, возможно различных, типа, описанных как параметры шаблона.

Другие параметры шаблона включают константные выражения, имена функций и символьные строки.

```
template <int n, class T>

class array {

public:

T a[n];

};

array <50,double> x,y;

x=y; // должно работать эффективно
```

Выгоды от параметризации состоят в распределении из системного стека, в противоположность распределению из свободной памяти. На большинстве систем это более эффективный режим. Тип привязывается к специфической целой константе так, что операции, использующие внутри себя массивы совместимой длины, безопасны по отношению к типу и проверены во время компиляции.

Параметризация класса vect

Класс vect часто подлежит параметризации.

```
template <class T >

class vect {

T *p; // базовый указатель

int size; // число элементов

public:

vect(); // создает массив размерности 10

vect(int n); // создает массив размерности n

vect(const vect &v); // инициализация от vect

vect(const T a[],int n); // инициализация от массива

~vect() { delete p; }

T& operator [](int i); // элемент, проверенный на соответствие границ

vect& operator = (const vect& v);

void print();
```

```
};
```

Определение функций-членов в контексте файла включает метку разрешения *имя класса* <T>.

```
template <class T>

vect<T>::vect() { size=10; p=new T[size]; }

template <class T>

vect <T>::vect(int n) { size=n; p=new T[size]; }

template <class T>

vect<T>::vect(const T a[], int n) { size=n; p=new T [size]; for(int i=0; i<size; i++) p[i]=a[i]; }

template <class T>

vect<T>::vect(const vect<T>&v) {

size=v.size; p=new T[size]; for(int i=0; i<size; i++) p[i]=v.p[i]; }

template <class T>

T& vect<T>::operator[](int i) { return p[i]; }

template <class T>

vect<T>& vect<T>::operator = (const vect<T> &v) {

int s=(size < v.size) ? size : v.size;

for(int i=0; i<s; i++) p[i]=v.p[i];

return *this;

}
```

Код пользователя почти также прост, как и для не параметризованных объявлений. Чтобы использовать эти объявления, в угловых скобках добавляется только специфический тип, который создает экземпляр шаблона. Этот тип может быть встроенным типом, таким как int или типом, определяемом пользователем. Эти шаблоны используются в следующем коде.

```
main()

{

vect<int> v(8);

vect <float> v1(10);
```

}

Наследование

Параметризованные типы могут повторно использоваться, благодаря наследованию. Это сравнимо с использованием наследования при порождении обычных классов. Как шаблоны, так и наследование представляют собой механизм для повторного использования кода и могут включать полиморфизм. Они - отличительные особенности C++ и поэтому могут комбинироваться в различных формах. Класс-шаблон может быть порожден от обычного класса. Обычный класс может быть порожден от класса-шаблона. Каждая из этих возможностей ведет к различным отношениям.

В некоторых ситуациях использованием шаблона ведет к недопустимым затратам, выражающимся в размере объектного модуля. Каждый реализованный класс-шаблон требует собственного откомпилированного объектного модуля.

template <class T>

class vect_safe: private vect {

public:

vect_safe() {}

void set(T *d) { vect::set

};

Шаблоны - резюме

- Шаблоны расширяют понятие функции и класса, предоставляя средства их *параметризации,* то есть объявления функций и классов в терминах ”любого типа”.

Шаблоны функций

- Это объявление функции, предваряемое *спецификацией шаблона*. Спецификация шаблона состоит из ключевого слова template, за которым следует список параметров, заключенный в угловые скобки <>.
- Имеют параметры типа, которые обозначаются ключевым словом *class,* за которым следует идентификатор. Ключевое слово *class* в контексте шаблонов означает *любой тип,* а не только класс. Идентификатор служит для замещения имени типа. Может быть более одного параметра типа.
- Шаблоны функций автоматически расширяются транслятором до полного описания функции так, как это необходимо.

```
// Объявление: максимум из двух значений типа T

template <class T> const T &Max(const T&,const T&);

template <class T> const T & // Определение
```

```
Max(const T &a,const T &b) { return a>b ? a : b; }

int i,j; float a,b;

int k=Max(i,j); // Вызов Max(int,int)

int c=Max(a,b); // Вызов Max(float,float)
```

Если функция-шаблон в рассматриваемом примере должна работать с каким-либо классом, то в этом классе операция должна быть перегружена. В противном случае при связывании программы произойдет ошибка.

- Шаблоны функций могут быть перегружены другими функциями.

```
template <class T> const T & Max(const T &,const T &);

template <class T> const T & Max(const T *,int);

int Max(int (*)(int),int (*)(int));
```

- Для определенных типов шаблоны функций могут быть перекрыты для того, чтобы выполнять (или не выполнять) какие-либо действия, которые функции-шаблоны не выполняют (или выполняют).

```
const char * &Max(const char * &c, const char * &d) {/*

Выполняет что-либо, специфичное для char * }
```

Шаблоны классов

- Шаблоны классов - это объявления классов, предваряемые спецификацией template.
- Шаблоны классов автоматически расширяются компилятором до полных определений классов так, как это необходимо.
- Не могут быть вложены в другие классы (в отличие от других классов).

```
// Объявить класс Stack, который представляет собой стек для любых типов.

template <class T> class Stack {

T *v; // указатель на некоторый тип T

int size,top;

public:

Stack(int s); ~Stack(); // и т.д.

};

Stack <int> i; // Стек для int
```

```
Stack <char*> cp; // Стек для char*
```

- Шаблоны классов могут иметь нетипированные (или только нетипированные) параметры. Значения, указанные для этих параметров, должны быть константными выражениями.

```
// Передать размер как параметр шаблона

template <class T,int size> class Stack {

T v[size]; // Массив элементов типа T.

int top;

public:

Stack():top(-1) {}//...

};

Stack <int,20> tiny;

Stack <int,500> huge;
```

Хотя стеки tiny и huge и хранят тип int, но все же это различные типы, поскольку они имеют разный размер стека. Это можно проиллюстрировать тем, что указатель Stack<int,20> - это не то же самое , что указатель на Stack<int,500>.

```
Stack<int,20> *ps20=&tiny; // Правильно

Stack<int,500> *ps500=&tiny; // Ошибка
```

- Шаблоны классов могут быть порождены как от нешаблонных классов, так и от классов шаблонов. А также могут порождать как нешаблонные классы, так и классы-шаблоны. Когда от класса-шаблона порождается нешаблонный класс, всем параметрам класса-шаблона должны быть присвоены некоторые “реальные” значения. В примере это int.

```
class A { /*...*/ }

template <class T> class B: public A { /*...*/ }

template <class T> class C: public B { /*...*/ }

class D: public C <int> { /*...*/ }
```

- Шаблоны классов для определенных типов могут быть перекрыты для того, чтобы выполнять (или не выполнять) какие-либо действия, которые шаблоны классов не выполняют (или выполняют).

```
// Объявим свой собственный стек для char*

class Stack<char*> {
```

```
char * *v; // указатель на char*

int size,top; //...

};
```

- Шаблоны классов могут также быть классами-структурами или классами-объединениями.

Статические данные-члены

- Статические данные-члены разделяются всеми объектами класса для *каждого конкретного экземпляра* класса-шаблона
- Статические данные-члены определяются в области видимости файла (как и все статические поля), когда определение предваряется спецификацией template

```
template <class T> class C {

static int i; // Обычное статическое поле

static T t; // Параметризованное

};

template<class T> int C<T>::i; // Определить в области видимости файла

template<class T> T C<T>::t;

C<char> c; // Имеет int C::i и char C::t;

C<float> f; // Имеет int C::i и float C::t;
```

Шаблоны функций-членов

- Определяются вне объявлений классов, к которым они принадлежат, при помощи спецификации template

```
template<class T> void Stack<T>::Push(const T &el) {

if(top==size-1) error(“stack overflow”);

else v[++top]=el;

}
```

- Шаблоны функций-членов для определенных типов могут быть перекрыты для того, чтобы выполнять (или не выполнять) какие-либо действия, которые шаблоны функций-членов не выполняют (или выполняют).

```
void Satck<char*>::Push(const char*& cpr) { /* выполнить нечто особенное */}
```

Дружественные функции

- Дружественные функции для каждого типа T могут быть друзьями всех классов типа T. Это обычные дружественные функции.
- Дружественные функции для типа T могут быть друзьями класса типа T.
- Дружественные функции могут предваряться спецификацией template. Для типов T и U, функции-шаблоны типа U являются дружественными функциями каждому классу типа T.

```
template<class T> class Person {

freind void Pet();

friend void Spouse(Person &);

template<class U> freind void Coworker(U&);

};

void Pet() { /*...*/ } // Обычная функция

template<class T>void Spouse(Person &p) {}

template<class U>void Coworker(U &u) {]
```

Здесь Pet() - функция, дружественная Person<T> для каждого типа T.

Person<int>

Pet() Person<char>

....

Person<float>

Для любого типа T, скажем int, Spouse(Person<int>&) - функция, дружественная Person<int>, но не Person<char>, или любому другому типу.

Spouse(Person<int>&) Person<int>

Spouse(Person<char>&) Person<char>

... ...

Spouse(Person<float>&) Person<float>

Coworker(U &) - функция, дружественная Person<T>, для любого типа T и любого типа U.

Coworker(int&) Person<int>

Coworker(char&) Peson<char>

... ...

Coworker(float&) Person<float>

- Дружественные функции могут быть функциями-членами другого класса.

  template<class T> class Perosn {

  freind void Family::Sibilding();

  freind void Acquaintance::Casual(Person &);

  template<class U>friend void Neighbor<U>::Nextdoor(U&);

  };

  Здесь Family::Sibilding() - фукнция, дружественная Person<T>, для каждого типа T. Для любого типа T, скажем для int, Acquaintance<int>::Causal(Person<int>&) - функция, дружественная Person<int>, но не Person<char>, или любому другому типу. Neighbor<U>::NextDoor(U&) -функция, дружественная Person<T>, для любого типа U.

- Дружкственные функции могут быть объявлены для всего класса.

  template<class T>class Person {

  friend class Family;

  freind class Acquaintance;

  template<class U> freind class Neighbor;

  };

  Здесь для каждого типа T все функции-члены класса Family - это функции, дружественные Person<T>. Для любого типа, скажем для int, все функции-члены класса Acquaintance<int> являются друзьями Person<int>, но не Person<char>, или любого другого типа. Для каждого типа T и каждого типа U все фунции-члены Nieghbor<U> являются друзьями Person<T>.

- Дружественные функции могут также быть друзьями нешаблонных классов.

Исключения

Исключения - возникновение непредвиденных ошибочных условий, например, деление на ноль при операциях с плавающей точкой. Обычно эти условия завершают программу пользователя с системным сообщением об ошибке. C++ дает программисту возможность восстанавливать программу из этих условий и продолжать ее выполнение.

Здесь будут рассмотрены *подтверждения*, как средства, предназначенные для того, чтобы справляться с ошибочными условиями и исключениями. Одна из точек зрения на эту проблему заключается в том, что исключение возникает при нарушении договорной гарантии, данной поставщиком кода, относительно того, к чему должен применяться этот код. При такой модели пользователь гарантирует соблюдение условий применение кода, а поставщик гарантирует, что код при этих условиях работает правильно. В этой методологии подтверждения обеспечивают различные гарантии.

Будет также охарактеризован пакет *signal.h* для ANSI C, который обрабатывает асинхронные исключения аппаратных средств. Эти исключения зависят от системы и обычно выдаются аппаратными средствами при обнаружении специального условия. Файл *signal.h* также обрабатывает синхронные исключения с использованием функции raise().

Использование assert.h

То, что вычисление завершилось с правильным результатом, частично можно рассматривать как доказательство того, что программа правильна. Кроме того, правильный результат зависит от правильного ввода, поэтому тот, кто активизировал вычисления, несет ответственность за обеспечение правильного ввода. Это было предусловие. Вычисление, если оно завершилось успешно, удовлетворяет постусловие. Усилие по обеспечению полностью формального доказательства правильности - идеализация, которая обычно не выполняется. Несмотря на это, такие подтверждения могут контролироваться во время выполнения для того, чтобы предоставить весьма полезную диагностику. Действительно, дисциплина продуманных подтверждений часто заставляет программиста избегать ошибок и “ловушек”.

Сообщества пользователей C и C++ все больше и больше подчеркивают важность использования таких подтверждений. Стандартный библиотечный *assert.h* обеспечивает макрокоманду.

```
void assert(int expression);
```

Если expression оценивается как неправильное, тогда выполнение прерывается с выводом диагностики. Подтверждения отбрасываются, если определяется макрокоманда NDEBUG.

Рассмотрим конструктор с одним параметром безопасного массива.

```
vect::vect(int n) {

if(n<1) { cerr << “ illegal vect size “ << n; exit(1); }

size=n; p=new int[size];

}
```

Заменим это на

```
vect::vect(int n) {

assert(n > 0); // оговоренное предусловие

size=n; p=new int[size];

assert(p!=NULL); // оговоренное постусловие

}
```

Использование подтверждений, заменяющих изначальные проверки условий, делает методологию более однородной. Это способствует лучшей практике. Обратной стороной этого будет то, что методология подтверждения не позволяет повторение или другую стратегию восстановления для продолжения выполнения программы.

Использование signal.h

Файл *signal.h* обеспечивает стандартный механизм для непосредственной обработки исключений, определенных системой. Исключения определяются внутри этой библиотеки и представляют собой зависимый от системы значения целых чисел. Например,

```
#define SIGINT 2 /* сигнал прерывания */

#define SIGFPE 8 /* исключение операций с плавающей точкой */

#define SIGABRT 22 /* сигнал аварийного прекращения */
```

Эти исключения может устанавливать система. Например, нажатие control+C на клавиатуре в большинстве систем генерирует прерывание , обычно завершая текущий процесс пользователя.

Для того, чтобы генерировать явное исключение, может использоваться функция raise(), прототип которой находится в signal.h

```
raise(SIGFPE); // установка сигнала исключения, для операций с плавающей точкой.
```

Эти исключения могут обрабатываться с использованием функции signal(). Она связывает функцию-обработчик с сигналом. Также она может использоваться для того, чтобы игнорировать сигнал или повторно устанавливать действие по умолчанию.

```
signal(SIGABRT,my_abort); // вызвать my_abort(), если установлен SIGABRT

signal(SIGABRT,SIG_DFL); // действие по умолчанию, если установлен SIGABRT

signal(SIGFPE,SIG_IGN); // игнорировать SIGFPE
```

Это называется установкой обработчика. Она заменяет обычный системный обработчик, определяемый пользователем. Используем эти идеи при написании цикла, который прерывается от клавиатуры. После прерывания обработчик запрашивает пользователь, должна ли программа продолжать выполнение.

```
//Обработка прерывания с использованием signal.h
#include <signal.h>
#include <time.h>
void cntrl_c_handler(int sig);
main() {
int i=0,j;
cout << “Count to J million, Enter j: “; cin >>j; j*=1000000;
signal(SIGINT, cntrl_c_handler); // Функция обработчик связывается с прерыванием
// SIGINT. После обнаружения следующего прерывания вместо действия системы
// по умолчанию система вызывает cntrl_c_handler
cout << (double)clock()/CLOCKS_PER_SEC << “start time\n”; //
while(1) {
j++;
if(i > j) {
cout << (double)clock()/CLOCKS_PER_SEC <<“end loop\n”;
cout << “HIT “ << j/1000000 << “MILLION” << endl;
raise(SIGINT); // Сигнал прерывания устанавливается явным вызовом.
// Неявно вызывается cntrl_c_handler
cout << “\nENter j: “; cin >> j; j*=1000000; i=0;
cout << (double)clock()/CLOCKS_PER_SEC << “start loop\n”;
}
}
}
// Эта подпрограмма обрабатывает исключение SIGINT
void cntrl_c_handler(int sig) {
char c;
```

```
// Было обнаружено прерывание и пользователя спрашивают, должна ли программа

// продолжаться

cout << " INTERRUPT" << " type y to continue :"; cin >> c;

if(c=='y') signal(SIGINT,cntrl_c_handler); // По этому запросу продолжения выполнения

// повторно устанавливается программа реакции на особую ситуацию. Без повторной

// установки обработчика система будет возвращаться к обработке прерывания по умолчанию.

else exit(0); // Чтобы завершить выполнение

}
```

Применим эту методику к обработке исключений для типа vect. Начнем заменять подтверждения условиями, которые устанавливают сигналы, определяемые пользователем.

```
const int SIGHEAP=SIGUSR1;

const int SIGSIZE = SIGUSR2;

vect::vect(int n) {

if(n<=0) { raise(SIGSIZE); cout << " Enter vect size n: "; cin >> n; }

p= new int[n];

if(p==NULL) { raise(SIGHEAP); p=new int[n]; }

}
```

Теперь можно определять обработчики, которые могут обеспечивать соответствующее действие. Например,

```
void vect_size_handler(int sig) {

char c;

cout << "SIZE ERROR, Enter y to continue with default:"; cin >> c;

if(c=='y') { signal(SIGSIZE,vect_size_handler); }

else exit(0);

}

void vect_heap_handler(int sig) { // Возможные действия:

// вернуть память в кучу или изящно завершиться
```

}

Сигналы и подтверждения часто могут заменяться локальными проверками и вызовами функций, имеющими *большую* гибкость. Однако, подтверждения и сигналы обеспечивают более однородную методологию и смягчают попытки чрезмерного восстановления, которое в большинстве случаев, является предпосылкой неправильного программирования и неаккуратности. Подтверждения проверяют правильность выполнения предварительных соглашений. Сигналы и обработчики позволяют восстановление в глобальном контексте для непредвиденных условий. Сигналы ограничены рядом асинхронных условий, зависимых от системы, но схема легко расширяется установкой глобальных данных, которые сбрасываются во время вызова условия raise.

Исключения в C++

C++ обеспечивает встроенный механизм обработки ошибок, называемый *обработкой исключительных ситуаций*. Благодаря обработке исключительных ситуаций можно упростить управление и реакцию на ошибки времени исполнения. Обработка исключительных ситуаций в C++ строится с помощью трех ключевых слов: **try, catch, throw.**

Операторы программы, во время выполнения которых вы хотите обеспечить обработку исключительных ситуаций, располагаются в блоке **try**. Если исключительная ситуация (т.е. ошибка) имеет место внутри блока **try**, она генерируется (с помощью **throw**). Перехватывается и обрабатывается исключительная ситуация с помощью ключевого слова **catch.**

Любой оператор, который генерирует исключительную ситуацию, должен выполняться внутри блока **try**. (Функции, которые вызываются внутри блока **try** также могут генерировать исключительную ситуацию). Любая исключительная ситуация должна перехватываться оператором **catch**, который следует непосредственно за блоком **try**, генерирующим исключительную ситуацию.

```
try {

// блок try

}

catch (type1 arg) {

// блок catch

}

catch (type2 arg) {

// блок catch

}

.....

catch (typeN arg) {

// блок catch
```

}

Блок **try** должен содержать ту часть программы, в которой вы хотите отслеживать ошибки. Это могут быть несколько операторов внутри одной функции, так и все операторы функции **main()** (что естественно вызывает отслеживания ошибок во всей программе).

Когда исключительная ситуация возникает, она перехватывается соответствующим ей оператором **catch**, который ее обрабатывает. С блоком **try** может быть связано более одного оператора **catch.** То, какой конкретно оператор **catch** используется, зависит от типа исключительной ситуации. Т.е., если тип данных, указанный в операторе **catch**, соответствует типу исключительной ситуации, то выполняется данный оператор **catch**. А все другие операторы блока **try** пропускаются. Если исключительная ситуация перехвачена, то аргумент ***arg*** получает ее значение. Можно перехватить любые типы данных, включая и создаваемые вами типы. Общая формула оператора **throw**

throw *исключительная_ситуация*;

Оператор **throw** должен выполняться либо внутри блока **try,** либо в любой функции, которую этот блок вызывает (прямо или косвенно). Здесь *исключительная_ситуация* - это вызываемая оператором исключительная ситуация.

Замечание. Если вы генерируете исключительную ситуацию, для которой не соответствующего оператора **catch**, может произойти ненормальное завершение программы.

Рассмотрим пример работы исключительной ситуации.

```
void main(){

try

{

throw 10;

}

catch(int i) {

cout << " error " << i << endl;

}

return;

}
```

На экран выведется сообщение:

error 10

C++ имеет чувствительный к контексту механизм обработки особых ситуаций. Исходя из этого, он может быть более информирован, чем обработчик из signal.h и может обеспечить более сложное восстановление. Он не предназначен для обработки асинхронных исключений, определенных в signal.h. Контекст для установки исключения - это блок **try**. Обработчики объявлены в конце блока **try** с использованием ключевого слова **catch**.

Код C++ позволяет непосредственно устанавливать исключения в блоке **try**, используя выражение **throw.** Исключение обрабатывается вызовом соответствующего обработчика, который выбирается из списка обработчиков, находящегося сразу после их блока **try**. Простой пример этой методики.

```
vect::vect(int n) {

if(n <1) throw(n);

p=new int[n];

if(p==NULL) throw(“FREE STORE EXHAUSTED”);

}

void g() {

try {

vect a(n),b(n); ...

}

catch(int n) { ... } // отслеживает все неправильные размеры

catch(char *error) { ... } // отслеживает превышение свободной памяти

}
```

Первый throw() имеет целый аргумент и соответствует сигнатуре catch(int n). При передаче неправильного размера массива в качестве параметра конструктора ожидается, что этот обработчик выполнит соответствующее действие, например, сообщение об ошибке и аварийное прекращение работы. Второй throw() имеет указатель на символьный аргумент и соответствует сигнатуре catch(char *error).

Установленные исключения

Синтаксически выражение throw появляется в двух формах.

```
throw

throw выражение
```

Выражение throw устанавливает исключение. Самый внутренний блок try, в котором устанавливается исключение, используется для выбора оператора catch, который обрабатывает исключение. Выражение

throw без аргумента повторно устанавливает текущее исключение. Обычно оно используется когда, для дальнейшей обработки исключение необходим второй обработчик, вызываемый из первого.

Установленное выражение - статический, временный объект, который хранится до тех пор, пока не производится выход из ветви обработки особых ситуаций. Выражение захватывается обработчиком, который может использовать его значение.

```
void foo() {

int i;

. . .

throw i;

}

main() {

try {

foo();

}

catch(int n) { . . . }

}
```

Значение целого числа, выданное через throw i, хранится до завершения работы обработчика с целочисленной сигнатурой catch (int n). Это значение доступно для использования внутри обработчика в виде аргумента.

Когда вложенная функция вызывает исключение, стек процесса не изменяется до тех пор, пока не найдена программа реакции на особую ситуацию. Это означает, что блокировка выхода из каждого завершенного локального процесса приводит к автоматическому уничтожению объектов.

```
void foo() { int i,j;

. . .

throw i;

. . .

}

void call_foo() { int k; . . .

foo(); . . . }
```

```
main() {

try {

call_foo(); // выход из foo уничтожает i и j

}

catch(int n) { . . . }

}
```

Пример переустановки исключения выглядит следующим образом

```
catch(int n) { . . .

throw; // переустановка

}
```

Поскольку предполагается, что установленное выражение имело целый тип, переустановленное исключение также представляет собой постоянный целый объект, который обрабатывается ближайшим обработчиком, подходящим для этого типа.

Концептуально установленное выражение передает информацию в обработчики. Часто обработчики не нуждаются в этой информации. Например, обработчик, который выводит сообщение и аварийно завершает работу, не нуждается ни в какой информации от окружения. Однако, пользователь может захотеть выводить дополнительную информацию или использовать ее для принятия решения относительно действий обработчика. В таком случае допустимо формирование информации в виде объекта.

```
enum error { bounds, heap, other };

class vect_error {

error e_type;

int ub,index,size;

public:

vect_error(error,int,int); // пакет вне пределов

vect_error(error,int); // пакет вне памяти

}
```

Теперь, используя объект типа vect_error, выражение throw может быть более информативным по отношению к обработчику, чем только установка выражений простых типов.

. . .

throw vect_error(bounds,i,ub);

. . .

Блоки try

Синтаксически, блок **try** имеет такую форму:

try

составной оператор

список обработчиков

Блок try - контекст для принятия решения о том, какие обработчики вызываются для установленного исключения. Порядок, в котором определяются обработчики, определяет тот порядок, в котором проверяются обработчики для установленного исключения соответствующего типа.

try { . . .

throw(“SOS”); . . .

io_condition eof(argv[i]);

throw(eof);

}

catch(const char *) { . . . }

catch(io_condition &x) { . . . }

Выражение throw соответствует аргументу catch, если он:

- Точно соответствует
- Общий базовый класс порожденного типа представляет собой то, что устанавливается
- Объект установленного типа является типом указателя, преобразуемым в тип указателя, являющегося аргументом catch.

Ошибочным будет перечисление обработчиков в порядке, который предотвращает их вызов. Например,

cathc(void *) // будет соответствовать любой char *

catch(char*) //

catch(BaseTypeError&) // будет вызываться всегда для DerivedTypeError

catch(DerivedTypeError&)

В C++ блоки **try** могут быть вложенными. Если в текущем блоке try нет соответствующего обработчика, выбирается обработчик из ближайшего внешнего блока try. Если он не обнаружен и там, тогда используется поведение по умолчанию.

Обработчики

Синтаксически обработчик имеет следующую форму:

catch (*формальный аргумент)*

составной оператор

Catch выглядит подобно объявлению функции с одним параметром без возвращаемого типа.

```
catch (char *message) {

cerr << message << endl; exit(1); }

catch ( . . . ) { // действие, которое нужно приянть по умолчанию.

cerr << " That's all folks" << andl; abort();

}
```

В списке аргументов допускается сигнатура, которая соответствует любому параметру. Кроме того, формальный параметр может быть абстрактным объявлением. Это значит, что он может иметь информацию о типе без имени переменной.

Обработчик вызывается соответствующим выражением throw. При этом, фактически, происходит выход из блока try. Система вызывает функции освобождения, которые включают деструкторы для любых объектов, локальных для блока try. Частично созданный объект будет иметь деструкторы, вызываемые на любые части созданных подобъектов.

Спецификация исключения

Синтаксически спецификация исключения представляет собой части объявления функции и имеет форму

заголовок функции throw (*список типов*)

Список типов - это список типов, которые может иметь выражение throw внутри функции. Если этот список пуст, компилятор может предположить, что непосредственно или косвенно, throw не будет выполняться функцией.

```
void foo() throw(int,over_flow);

void noex(int i) throw();
```

Если спецификация исключения оставлена, тогда возникает допущение, что такой функцией может быть установлено произвольное исключение. Хорошей практикой программирования будет показать с

помощью спецификации, какие ожидаются исключения. Нарушения этих спецификаций приведут к ошибке во время выполнения программы.

Terminate() И unexpected()

Обработчик terminate(), также поставляемый в составе системы, вызывается в том случае, когда для обработки исключения не предоставляется другой обработчик. По умолчанию вызывается функция abort(). В другом случае, чтобы предоставить обработчик, может быть использована set_terminate().

Поставляемый в составе системы обработчик unexpected() вызывается, если функция устанавливает исключение, которого не было в списке спецификаций исключений. По умолчанию вызывается abort(). В другом случае, чтобы предоставить обработчик, может быть использована set_unexpected().

Пример кода, реализующего исключения

```
vect::vect(int n) {

if(n <1)

throw(n);

p=new int [n];

if(p==NULL)

throw(“NOT MEMORY”);

}

void g(int m) {

try {

vect a(m);

}

catch (int n) {

cerr << “SIZE ERROR” << n << endl;

g(10); // повторить g с допустимым размером

}

catch(const char *error) {

cerr << error << endl; abort();

}
```

}

Обработчик заменил запрещенное значение на допустимое значеbние по умолчанию. Это может быть приемлемо на этапе отладки системы, когда большинство подпрограмм объединяются и проверяются. Система пытается продолжать обеспечивать дальнейшую диагностику. Это аналогично компилятору, пытающемуся продолжать анализировать неправильную программу после синтаксической ошибки. Часто компилятор предоставляет дополнительны сообщения об ошибках, которые оказываются полезными.

Вышеупомянутый конструктор проверяет только одну переменную на допустимое значение. Это выглядит искусственно, так как он заменяет код, который мог бы непосредственно заменять запрещенное назначение по умолчанию, устанавливая при этом исключение и позволяя обработчику восстановление значения. Однако, при такой форме записи разделение того, что является ошибкой и того, как она обрабатывается, очевидно. Это иллюстрирует ясную методологию разработки кода.

Более обобщенно, конструктор объекта может выглядеть так:

Object::Object(*аргументы*) {

if(*недопустимый аргумент 1*)

throw *выражение 1*;

if(*недопустимый аргумент 2*)

throw *выражение 2*;

. . . // попытка создания

}

Конструктор Object теперь обеспечивает набор выражений для установки запрещенного состояния. Теперь блок try может использовать информацию для восстановления или прерывания неправильного кода.

try {

// оказоустойчивый код

}

catch(о*бъявление 1*) { /* восстановление этого случая */ }

catch(о*бъявление 2*) { /* восстановление этого случая */ }

catch(о*бъявление K*) { /* восстановление этого случая */ }

// правильные или восстановленные переменные состояния теперь допустимы

Когда существует много определенных ошибочных условий, удобных для состояния данного объекта, может быть использована иерархия классов для создания выбора связанных типов, которые нужно использовать как выражения установки.

```
class Object_Error {

public:

Object_Error(аргументы); // получение полезной информации

члены, содержащие состояние установленного выражения

virtual void repair() {

cerr << “Repair failed in Object “ << endl; abort();

}

};

class Object_Error_S1: public Object_Error {

public:

Object_Error_S1(аргументы); // получение полезной информации

добавленные члены, содержащие состояние установленного выражения

void repair(); // переопределение, для того чтобы обеспечить

// соответствующее восстановление

};

. . . // при необходимости - прочие порожденные классы error
```

Эти иерархии позволяют соответственно упорядочному множеству catch обрабатывать исключения в логической последовательности. Тип базового класса в списке catch должен следовать после типа порожденного класса.

Философия восстановления после ошибок

Восстановление при возникновении ошибок - в основном имеет отношение к правильности написания программы. Обработка исключений близка к восстановлению при возникновении ошибок и к механизму передачи управления. Следуя модели пользователь/изготовитель, изготовитель должен гарантировать, что при приемлемом входном состоянии, его программное обеспечение выполняет правильный вывод. Сколько должно быть встроенного обнаружения и очевидного исправления ошибок - дело изготовителя. Пользователь чаще пользуется библиотеками обнаружения ошибок, потому что так он может принимать решение относительно того, пытаться ли продолжать вычисления.

Восстановление при возникновении ошибок основывается на передачи управления. Недисциплинированная передача управления ведет к хаосу. При восстановлении отказов предполагается, что исключение нарушило вычисления. Продолжать вычисления становится опасно. Полезная обработка исключений влечет за собой упорядочное восстановление при появлении отказа.

В большинстве случаев программирование, которое вызывает исключения, должно выводить диагностическое сообщение и элегантно завершать работу. При специальных формах обработки, типа работы в режиме реального времени и при отказоустойчивом вычислении существует необходимость в том, чтобы система не приостанавливалась. В таких случаях героические попытки восстановления узаконены.

Можно согласовать, для каких классов полезно предусматривать ошибочные условия. В большинстве своем это такие условия, когда объект имеет значения членов в запрещенных состояниях - значения, которые им не позволено иметь. Для таких случаев система устанавливает исключение с действием по умолчанию, являющиеся завершением программы. Это аналогично встроенным типам, устанавливающим исключения, определенные системой, таким как SIGFPE.

Внутренняя противоречивость поднимает следующие вопросы: какой вид вмешательства является приемлемым для продолжения выполнения программы, и указания, куда должен возвращаться поток управления? C++ использует модель завершения, которая вынуждает завершать текущий блок try. При этом режиме пользователь или повторяет код, игнорируя исключение, или подставляет результат по умолчанию и продолжает. При повторении кода более вероятно получить правильный результат.

Опыт показывает, что обычно код слабо комментируется, Трудно вообразить программу, в которой будет слишком много подтверждений. Подтверждения и простые установки обработки исключений, которые завершают вычисления, представляют собой параллельные технологии. Хорошо спланированный набор ошибочных условий, обнаруживаемых пользователями АТД - важная часть проекта. Если при нормальном программировании слишком часто обнаруживаются ошибки и происходит прерывание - это признак того, что программа плохо обдумана и имела слишком большое количество просчетов в первоначальном виде.

Объектно-ориентированное программирование. Лабораторные работы

Лабораторные работы выполняются в ауд. 6407 на компьютерах Sun Ultra Sparc.
Используется компилятор GNU g++.
Чтобы откомпилировать файл, необходимо набрать строку
$gpp имя_файла. Результирующим будет файл с именем a.out.
(При желании можно установить любое имя файла)
Некоторые лабораторные работы выполняются в ауд. 1529 на компьютерах PC Pentium с использованием компилятора Visual C++.

Лабораторная работа №1
Объектно-ориентированное программирование

Потоки

Использование стандартного ввода-вывода (iostream)

Для студентов, готовых ограничиться оценкой "удовлетворительно" за экзамен и курсовую работу, в данном пункте выполнять только 1-3 задание.

Вариант 1

1. Написать программу вычисления ближайшего сверху числа степени 2; Программа должна использовать цикл while. Входные данные поступают с клавиатуры. Результат выводится на экран. Предусмотреть обработку ошибок.
2. Написать программу, печатающую цифры. Использовать цикл while. В строку выводится цифра, в десятичной, шестнадцатеричной и восьмеричной системе.
3. Написать программу преобразования шкалы Цельсия в шкалу Фаренгейта. 0 по Цельсию равен 32 по Фаренгейту. 1 градус по Цельсия равен 1.8 по Фаренгейту. Установить ширину поля 10 символов, установить точность 9 цифры, заполнить вместо пробелов символом : с помощью функций и манипуляторов.
4. Проверить какие флаги потока вывода установлены и заменить попарно на
   левое выравнивание на правое выравнивание (и наоборот)
   убрать вывод основания системы, если установлен и установить, если сброшен
   Проверить результат.
5. Написать программу с двумя своими манипуляторами. Один выводит сообщение, другой устанавливает шестнадцатеричный вывод и ширину поля 10.

Вариант 2

1. Написать программу вычисления ближайшего сверху числа степени 2; Программа должна использовать цикл for. Входные данные поступают с клавиатуры. Результат выводится на экран. Предусмотреть обработку ошибок.
2. Написать программу, печатающую символы от A до Z. Использовать цикл for. В строку выводится номер, символ, шестнадцатеричный и восьмеричный код.
3. Написать программу вычисления частного и остатка от деления двух целых чисел. Установить ширину поля 10 символов, заполнить вместо пробелов символом $ с помощью функций и манипуляторов.
4. Проверить какие флаги потока вывода установлены и заменить попарно на
   десятичные на шестнадцатеричные (и наоборот)
   левое выравнивание на правое выравнивание (и наоборот)
   Проверить результат.
5. Написать программу с двумя своими манипуляторами. Один выводит сообщение, другой устанавливает восьмеричный вывод и точность 15.

Вариант 3

1. Написать программу вычисления ближайшего снизу числа степени 2; Программа должна использовать цикл while. Входные данные поступают с клавиатуры. Результат выводится на экран. Предусмотреть обработку ошибок.

2. Написать программу, печатающую символы от a до z. Использовать цикл for. В строку выводится номер, символ, шестнадцатеричный и восьмеричный код.
3. Написать программу преобразования шкалы Фаренгейта в шкалу Цельсия. 0 по Цельсию равен 32 по Фаренгейту. 1 градус по Цельсия равен 1.8 по Фаренгейту. Установить ширину поля 11 символов, установить точность 8 цифры, заполнить вместо пробелов символом # с помощью функций и манипуляторов.
4. Проверить какие флаги потока вывода установлены и заменить попарно на
   десятичные на шестнадцатеричные (и наоборот)
   левое выравнивание на правое выравнивание (и наоборот)
   Проверить результат.
5. Написать программу с двумя своими манипуляторами. Один выводит сообщение, другой устанавливает шестнадцатеричный вывод и символ заполнения _.

Вариант 4

1. Написать программу вычисления наибольшего общего делителя двух целых чисел. Наибольший общий делитель рекурсивно вычисляется следующим образом;
   GCD(m, n) is:
   if m mod n equals 0 then n;
   else GCD(n, m mod n);
   Программа должна использовать цикл while. Входные данные поступают с клавиатуры. Результат выводится на экран. Предусмотреть обработку ошибок.
2. Написать программу, печатающую символы от A до Z. Использовать цикл while. В строку выводится номер, символ, шестнадцатеричный и восьмеричный код.
3. Написать программу решения квадратного уравнения. Корни только вещественные. Ввод и вывод через стандартные потоки ввода-вывода. Вывод результата в "научном" формате. Установить ширину поля 12 символов, установить точность 4 цифры, заполнить вместо пробелов символом _. с помощью функций и манипуляторов.
4. Проверить какие флаги потока вывода установлены и заменить попарно на
   десятичные на шестнадцатеричные (и наоборот)
   обычную на научную нотацию (и наоборот)
   Проверить результат.
5. Написать программу с двумя своими манипуляторами. Один выводит сообщение, другой устанавливает шестнадцатеричный вывод и переход на новую строку.

Вариант 5

1. Написать программу, вычисляющую $y(n) = \sum_{k=0}^{n} x(n-k), n = 0,...N-1$ Программа должна использовать цикл while. Входные данные поступают с клавиатуры. Результат выводится на экран. Установить точность 4 цифры. Предусмотреть обработку ошибок.
2. Написать программу, печатающую символы. Использовать цикл while. В строку выводится номер, символ, шестнадцатеричный и восьмеричный код.
3. Написать программу проверки является ли число простым, установить ширину поля 10 символов, заполнить вместо пробелов символом ^ с помощью функций и манипуляторов.
4. Проверить какие флаги потока вывода установлены и заменить попарно на
   левое выравнивание на правое выравнивание (и наоборот)
   убрать + перед числом, если установлен и установить, если сброшен
   Проверить результат.

5. Написать программу с двумя своими манипуляторами. Один выводит сообщение, другой устанавливает десятичный вывод и ширину поля 10.

Вариант 6

1. Написать программу вычисления наибольшего общего делителя двух целых чисел. Наибольший общий делитель рекурсивно вычисляется следующим образом;
   GCD(m, n) is:
   if m mod n equals 0 then n;
   else GCD(n, m mod n);
   Программа должна использовать цикл for. Входные данные поступают с клавиатуры. Результат выводится на экран. Предусмотреть обработку ошибок.
2. Написать программу, печатающую символы от a до z. Использовать цикл while. В строку выводится номер, символ, шестнадцатеричный и восьмеричный код.
3. Написать программу решения квадратного уравнения. Корни могут быть комплексными. Установить ширину поля 10 символов, установить точность 4 цифры, заполнить вместо пробелов символом $ с помощью функций и манипуляторов.
4. Проверить какие флаги потока вывода установлены и заменить попарно на
   убрать вывод основания системы, если установлен и установить, если сброшен
   убрать + перед числом, если установлен и установить, если сброшен
   Проверить результат.
5. Написать программу с двумя своими манипуляторами. Один выводит сообщение, другой устанавливает шестнадцатеричный вывод и точность 10.

Вариант 7

1. Написать программу - простой калькулятор с операциями +, -, *, /. Входные данные, включая операции, поступают с клавиатуры. Результат выводится на экран. Предусмотреть обработку ошибок.
2. Написать программу, печатающую символы. Использовать цикл for. В строку выводится номер, символ, шестнадцатеричный и восьмеричный код.
3. Написать программу вычисления корней квадратного уравнения и вывод результата в виде разложения многочлена на множители. Установить ширину поля 10 символов, установить точность 4 цифры, заполнить вместо пробелов символом ~ с помощью функций и манипуляторов.
4. Проверить какие флаги потока вывода установлены и заменить попарно на
   обычную на научную нотацию (и наоборот)
   убрать вывод основания системы, если установлен и установить, если сброшен
   Проверить результат.
5. Написать программу с двумя своими манипуляторами. Один выводит сообщение, другой устанавливает шестнадцатеричный вывод и символ заполнения ?.

Вариант 8

1. Написать программу - возведение числа n в m-ую степень. Входные данные поступают с клавиатуры. Результат выводится на экран. Предусмотреть обработку ошибок.
2. Написать программу, печатающую все символы и цифры. Использовать цикл for. В строку выводится номер, символ, шестнадцатеричный и восьмеричный код.
3. Написать программу вычисления корней квадратного уравнения и вывод результата в виде разложения многочлена на множители. (корни могут быть комплексными). Установить ширину

поля 10 символов, установить точность 4 цифры, заполнить вместо пробелов символом & с помощью функций и манипуляторов.
4. Проверить какие флаги потока вывода установлены и заменить попарно на
   левое выравнивание на правое выравнивание (и наоборот)
   убрать + перед числом, если установлен и установить, если сброшен
   Проверить результат.
5. Написать программу с двумя своими манипуляторами. Один выводит сообщение, другой устанавливает восьмеричный вывод и ширину поля 20.

Вариант 9

1. Написать программу - посчитать длину окружности. Входные данные поступают с клавиатуры. Результат выводится на экран. Предусмотреть обработку ошибок.
2. Написать программу, печатающую все вводимые символы в верхнем регистре. В строку выводится символ, шестнадцатеричный и восьмеричный код.
3. Написать программу, которая получает данные по Цельсию в виде 15C и преобразует их в данные по Фаренгейту 59F. 0 по Цельсию равен 32 по Фаренгейту. 1 градус по Цельсия равен 1.8 по Фаренгейту. Установить ширину поля 10 символов, установить точность 4 цифры, заполнить вместо пробелов символом / с помощью функций и манипуляторов.
4. Проверить какие флаги потока вывода установлены и заменить попарно на
   обычную на научную нотацию (и наоборот)
   убрать вывод основания системы, если установлен и установить, если сброшен
   Проверить результат.
5. Написать программу с двумя своими манипуляторами. Один выводит сообщение, другой устанавливает десятичный вывод и точность 6.

Вариант 10

1. Написать программу - посчитать площадь окружности. Входные данные поступают с клавиатуры. Результат выводится на экран. Предусмотреть обработку ошибок.
2. Написать программу, печатающую все вводимые символы в нижнем регистре. В строку выводится символ, шестнадцатеричный и восьмеричный код.
3. Написать программу, которая получает данные либо по Фаренгейту в виде 59F и преобразует их в данные по Цельсию 15C, либо наоборот. 0 по Цельсию равен 32 по Фаренгейту. 1 градус по Цельсия равен 1.8 по Фаренгейту. Установить ширину поля 10 символов, установить точность 4 цифры, заполнить вместо пробелов символом \ с помощью функций и манипуляторов.
4. Проверить какие флаги потока вывода установлены и заменить попарно на
   левое выравнивание на правое выравнивание (и наоборот)
   убрать + перед числом, если установлен и установить, если сброшен
   Проверить результат.
5. Написать программу с двумя своими манипуляторами. Один выводит сообщение, другой устанавливает шестнадцатеричный вывод и ширину поля 10.

Вариант 11

1. Написать программу - перевода оценки 2,3,4 и 5 в соответствующее слово. Входные данные поступают с клавиатуры. Результат выводится на экран. Предусмотреть обработку ошибок.
2. Написать программу, печатающую все вводимые символы. В строку выводится символ, шестнадцатеричный и восьмеричный код.
3. Написать программу решения линейного уравнения. Установить ширину поля 10 символов. Установить точность 4 цифры. Заполнить вместо пробелов символом %.

4. Проверить какие флаги потока вывода установлены и заменить попарно на
   десятичные на шестнадцатиричные (и наоборот)
   левое выравнивание на правое выравнивание (и наоборот)
   Проверить результат.
5. Написать программу с двумя своими манипуляторами. Один выводит сообщение, другой устанавливает шестнадцатеричный вывод и символ заполнения $.

Вариант 12

1. Написать программу вычисления ближайшего сверху числа степени 2; Программа должна использовать цикл for. Входные данные поступают с клавиатуры. Результат выводится на экран. Предусмотреть обработку ошибок.
2. Написать программу, печатающую цифры. Использовать цикл for. В строку выводится цифра, в десятичной, шестнадцатеричной и восьмеричной системе.
3. Написать программу, которая получает данные по Фаренгейту в виде 59F и преобразует их в данные по Цельсию 15C. 0 по Цельсию равен 32 по Фаренгейту. 1 градус по Цельсия равен 1.8 по Фаренгейту. Установить ширину поля 10 символов, установить точность 4 цифры, заполнить вместо пробелов символом * с помощью функций и манипуляторов.
4. Проверить какие флаги потока вывода установлены и заменить попарно на
   обычную на научную нотацию (и наоборот)
   убрать вывод основания системы, если установлен и установить, если сброшен
   Проверить результат.
5. Написать программу с двумя своими манипуляторами. Один выводит сообщение, другой устанавливает шестнадцатеричный вывод и ширину поля 10.

Задания на лабораторные работы по Объектно-ориентированному программированию

Файлы-Потоки

Использование стандартного ввода-вывода (fstream)

Для студентов, готовых ограничиться оценкой “удовлетворительно” за экзамен и курсовую работу, в данном пункте выполнить только 1 (любое) задание.

Вариант 1

1. Написать программу копирования файлов. Чтение происходит блоками. Обработать ошибки.
2. Написать программу вычисления количества символов пробела в файле. Обработать ошибки.

Вариант 2

1. Написать программу копирования текстового файла с добавлением двойного интервала между строками. Обработать ошибки.
2. Написать программу вычисления количества символов в файле. Чтение происходит блоками. Обработать ошибки.

Вариант 3

1. Написать программу копирования файла в обратном порядке. Чтение происходит блоками. Обработать ошибки.
2. Написать программу вычисления количества цифр в файле. Обработать ошибки.

Вариант 4

1. Написать программу копирования файла первая половина собственно файл, вторая половина он же в обратном порядке. Чтение происходит блоками. Обработать ошибки.
2. Написать программу вычисления количества символов нижнего регистра в файле. Обработать ошибки.

Вариант 5

1. Написать программу копирования файла в двойном экземпляре. Чтение происходит блоками. Обработать ошибки.
2. Написать программу вычисления количества символов верхнего регистра в файле. Обработать ошибки.

Вариант 6

1. Написать программу копирования файла в двойном экземпляре. Чтение происходит блоками. Обработать ошибки.
2. Написать программу вычисления количества печатаемых символов в файле. Обработать ошибки.

Вариант 7

1. Написать программу копирования файла с удалением лишних пробелов. Обработать ошибки.

2. Написать программу вычисления количества символов или цифр в файле. Чтение происходит блоками. Обработать ошибки.

Вариант 8

1. Написать программу копирования файла с удвоением пробелов. Чтение происходит блоками. Обработать ошибки.
2. Написать программу вычисления количества символов пунктуации в файле. Чтение происходит блоками. Обработать ошибки.

Вариант 9

1. Написать программу записи в файл строк, введенных с клавиатуры. Записывать блоком. Обработать ошибки.
2. Написать программу вычисления количества символов нижнего регистра в файле. Обработать ошибки.

Вариант 10

1. Написать программу копирования файла с заменой пробелов на символ |. Чтение происходит блоками. Обработать ошибки.
2. Написать программу вычисления количества символов верхнего регистра в файле. Обработать ошибки.

Вариант 11

1. Написать программу записи заголовка в файл данных (*.dat). Запись блоками, но поэлементно. Обработать ошибки.
2. Написать программу вычисления количества символа ':' в файле. Обработать ошибки.

Вариант 12

3. Написать программу копирования файла с удалением пробелов. Чтение происходит блоками. Обработать ошибки.
4. Написать программу вычисления количества символов перевода строки в файле. Чтение происходит блоками. Обработать ошибки.

Задания на лабораторные работы по Объектно-ориентированному программированию

Отличия C и C++

Материал для выполнения лабораторной работы можно найти здесь

Для студентов, готовых ограничиться оценкой “удовлетворительно” за экзамен и курсовую работу, в данном пункте достаточно выполнить любые 3 задания.

Вариант 1

1. Написать программу вычисления среднего массивов типа int,float и complex. Функции, вычисляющие среднее имеют одно и тоже имя.
2. Написать родовую функцию, которая меняет местами первый и второй аргумент. Проверить для данных типа char, int, float, double.
3. Создать абстрактный тип данных для комплексных чисел (структура). Определить функции, которые устанавливает значения реальной и мнимой части, . Мнимая часть параметр по умолчанию.
4. Создать абстрактный тип данных (структура)- вектор, который имеет указатель на int и число элементов. Определить функцию, которая выделяет память для массива и инициализирует данные. Определить 2 функции, отличающиеся только спецификатором inline, которые устанавливают элемент массива в некоторое значение и использовать их при инициализации. Освободить память. Сравнить время вычисления.
5. Написать функцию инкремента единственного параметра. Написать функцию возвращающую ссылку на передаваемый параметр. Изменить его при вызове функции. Что, если все ссылки сделать const?

Вариант 2

1. Написать программу вычисления дисперсии массивов типа int,float и complex. Функции, вычисляющие дисперсию имеют одно и тоже имя.
2. Написать родовую функцию, которая копирует n байт со второго аргумента по адресу первого аргумента, для которого выделила память. Проверить для данных типа char, int, float, double.
3. Создать абстрактный тип данных - человек, у которого есть имя и возраст (структура). Определить функцию, которая установит имя и возраст по умолчанию. Для задания имени использовать оператор new.
4. Создать абстрактный тип данных (структура) - вектор, который имеет указатель на float и число элементов. Определить функцию, которая выделяет память для массива и инициализирует данные. Определить 2 функции, отличающиеся только спецификатором inline, которые устанавливают элемент массива в некоторое значение, использовать их при инициализации. Освободить память. Сравнить время вычисления.
5. Написать функцию декремента единственного параметра. Написать функцию возвращающую ссылку на элемент глобального массива. Изменить его при вызове функции. Что, если все ссылки сделать const?

Вариант 3

1. Написать программу вычисления среднеквадратичного отклонения массивов типа int, float и complex. Функции, вычисляющие среднеквадратичное отклонение имеют одно и тоже имя.

2. Написать родовую функцию, которая копирует n байт со второго аргумента по адресу первого аргумента, для которого в функции выделяется память. Записать по адресу второго аргумента нули. Проверить для данных типа char, int, float, double.
3. Создать абстрактный тип данных - стул , у которого есть фирма изготовитель и число ножек (структура). Определить функцию, которая устанавливает число ножек по умолчанию. Для задания названия фирмы использовать оператор new.
4. Создать абстрактный тип данных (структура) - вектор, который имеет указатель на double и число элементов. Определить функцию, которая выделяет память для массива и инициализирует данные. Определить 2 функции, отличающиеся только спецификатором inline, которые получают элемент массива, использовать их при вычислении среднего. Освободить память. Сравнить время вычисления.
5. Написать функцию ввода в единственный параметр значения с клавиатуры. Написать функцию возвращающую ссылку на глобальную переменную. Изменить его при вызове функции. Что, если все ссылки сделать const?

Вариант 4

1. Написать программу упорядочивание массивов типа int,float и complex по возрастанию. Функции, упорядочивающие массивы имеют одно и тоже имя.
2. Написать родовую функцию, которая копирует n байт из файла в память, которая выделяется в родовой функции. Копирование происходит по байтам. Проверить для данных типа char, int, float, double.
3. Создать абстрактный тип данных (структура) - город , у которого есть название и количество жителей. Определить функцию, которая устанавливает название “Владивосток” и 700 тыс. по умолчанию. Для задания названия города использовать оператор new.
4. Создать абстрактный тип данных (структура) - вектор, который имеет указатель на double и число элементов. Определить функцию, которая выделяет память для массива и инициализирует данные. Определить 2 функции, отличающиеся только спецификатором inline, которые получают элемент массива, использовать их при вычислении дисперсии. Освободить память. Сравнить время вычисления.
5. Написать функцию изменения знака единственного параметра. Написать функцию возвращающую ссылку на передаваемый параметр. Изменить его при вызове функции. Что, если все ссылки сделать const?

Вариант 5

1. Написать программу упорядочивание массивов типа int,float и complex по убыванию. Функции, упорядочивающие массивы имеют одно и тоже имя.
2. Написать родовую функцию, которая считывает n байт из файла в память, которая выделяется в родовой функции, считывание блоком. Проверить для данных типа char, int, float, double.
3. Создать абстрактный тип данных (структура) - институт , у которого есть название и число студентов. Определить функцию, которая устанавливает название “ВГУЭС” и 20 тыс. по умолчанию. Для задания названия института использовать оператор new.
4. Создать абстрактный тип данных (структура) - вектор, который имеет указатель на double и число элементов. Определить функцию, которая выделяет память для массива и инициализирует данные. Определить 2 функции, отличающиеся только спецификатором inline, которые получают элемент массива, использовать их при вычислении средне квадратичного отклонения. Освободить память. Сравнить время вычисления.
5. Написать функцию абсолютного значения единственного параметра. Написать функцию возвращающую ссылку на глобальную переменную. Изменить его при вызове функции. Что, если все ссылки сделать const?

Вариант 6

1. Написать программу вычисления нормы массивов типа int,float и complex. Функции, вычисляющие нормы массивов имеют одно и тоже имя.
2. Написать родовую функцию, которая сравнивает первые n байт содержимого первого и второго аргумента и помещает в третий только отличающиеся симвоы. Место под третий аргумент выделяется в родовой функции. Проверить для данных типа char, int, float, double.
3. Создать абстрактный тип данных (структура)- компьютер , у которого есть фирма изготовитель, процессор, объем жесткого диска и ОЗУ в Мбайтах. Определить функцию, которая устанавливает стандартную конфигурацию по умолчанию (стандартная конфигурация - это компьютер, который вас устроит). Для задания названия фирмы использовать оператор new.
4. Создать абстрактный тип данных (структура) - вектор, который имеет указатель на double и число элементов. Определить функцию, которая выделяет память для массива и инициализирует данные. Определить 2 функции, отличающиеся только спецификатором inline, которые получают элемент массива, использовать их при упорядочивании массива по возрастанию. Освободить память. Сравнить время вычисления.
5. Написать функцию увеличения в 2 раза единственного параметра. Написать функцию возвращающую ссылку на глобальную переменную. Изменить его при вызове функции. Что, если все ссылки сделать const?

Вариант 7

1. Написать программу, записывающую массивы типа int,float и complex в обратном порядке. Функции, переупорядочивания массивов имеют одно и тоже имя.
2. Написать родовую функцию, которая копирует файл в память, которая выделяется в родовой функции. Проверить для данных типа int, float, double.
3. Создать абстрактный тип данных (структура) - принтер , у которого есть фирма изготовитель, тип - матричный, струйный или лазерный, разрешающая способность. Определить функцию, которая устанавливает по умолчанию тип струйный с 300x300. Для задания названия фирмы использовать оператор new.
4. Создать абстрактный тип данных (структура) - вектор, который имеет указатель на double и число элементов. Определить функцию, которая выделяет память для массива и инициализирует данные. Определить 2 функции, отличающиеся только спецификатором inline, которые получают элемент массива, использовать их при упорядочивании массива по убыванию. Освободить память. Сравнить время вычисления.
5. Написать функцию уменьшения в 2 раза единственного параметра. Написать функцию возвращающую ссылку на передаваемый параметр. Изменить его при вызове функции. Что, если все ссылки сделать const?

Вариант 8

1. Написать программу вычисления моды массивов типа int,float и complex. Функции, вычисления моды имеют одно и тоже имя.
2. Написать родовую функцию, которая копирует файл в память по адресу первого аргумента, если файл больше, то память должна быть перевыделена в родовой функции. Проверить для данных типа int, float, double.
3. Создать абстрактный тип данных (структура) - студент , у которого есть имя, номер зачетки, средняя успеваемость. Определить функцию, которая устанавливает по умолчанию среднюю успеваемость - 4. Для задания имени использовать оператор new.
4. Создать абстрактный тип данных (структура) - вектор, который имеет указатель на double и число элементов. Определить функцию, которая выделяет память для массива и инициализирует

данные. Определить 2 функции, отличающиеся только спецификатором inline, которые получают элемент массива, использовать их при вычислении максимального значения. Освободить память. Сравнить время вычисления.

5. Написать функцию увеличения в 10 раза единственного параметра. Написать функцию изменения знака единственного параметра. Написать функцию возвращающую ссылку на передаваемый параметр. Изменить его при вызове функции. Что, если все ссылки сделать const?

Вариант 9

1. Написать программу вычисления абсолютного значения для типа int,float и complex. Функции, вычисления абсолютного значения имеют одно и тоже имя.
2. Написать родовую функцию, которая копирует файл в память по адресу первого аргумента, если файл больше, то память должна быть перевыделена в родовой функции. Проверить для данных типа int, float, double.
3. Создать абстрактный тип данных (структура) - машина, у которого есть марка, объем двигателя, средняя скорость. Определить функцию, которая устанавливает по умолчанию среднюю скорость - 100 км/ч. Для задания марки использовать оператор new.
4. Создать абстрактный тип данных (структура) - вектор, который имеет указатель на double и число элементов. Определить функцию, которая выделяет память для массива и инициализирует данные. Определить 2 функции, отличающиеся только спецификатором inline, которые определяют положительный или отрицательный элемент массива, использовать их при вычислении числа отрицательных элементов Освободить память. Сравнить время вычисления.
5. Написать функцию уменьшения в 10 раза единственного параметра. Написать функцию возвращающую ссылку на элемент глобального массива. Изменить его при вызове функции. Что, если все ссылки сделать const?

Вариант 10

1. Написать программу вывода даты как строки и три параметра - месяц, день и год. Функции, вывода даты на экран имеют одно и тоже имя.
2. Написать родовую функцию, которая копирует файл в память по адресу первого аргумента, если файл больше, то память должна быть перевыделена в родовой функции. Проверить для данных типа int, float, double.
3. Создать абстрактный тип данных (структура) - окно, у которого есть Заголовок, максимальный размер, рамки (или нет). Определить функцию, которая устанавливает по умолчанию максимальный размер - весь экран. Для задания имени использовать оператор new.
4. Создать абстрактный тип данных (структура) - вектор, который имеет указатель на double и число элементов. Определить функцию, которая выделяет память для массива и инициализирует данные. Определить 2 функции, отличающиеся только спецификатором inline, которые определяют положительный или отрицательный элемент массива, использовать их при вычислении числа положительных элементов Освободить память. Сравнить время вычисления.
5. Написать функцию преобразующую единственный параметр в ближайшее четное. Написать функцию возвращающую ссылку на глобальную переменную. Изменить его при вызове функции. Что, если все ссылки сделать const?

Вариант 11

1. Написать программу вычисления минимального значения в массивах типа int,float и complex. Функции, вычисления минимума имеют одно и тоже имя.

2. Написать родовую функцию, которая копирует файл в память по адресу первого аргумента, если файл больше, то память должна быть перевыделена в родовой функции. Проверить для данных типа int, float, double.
3. Создать абстрактный тип данных (структура) - студент , у которого есть имя, номер зачетки, средняя успеваемость. Определить функцию, которая устанавливает по умолчанию среднюю успеваемость - 4. Для задания имени использовать оператор new.
4. Создать абстрактный тип данных (структура) - вектор, который имеет указатель на int и число элементов. Определить функцию, которая выделяет память для массива и инициализирует данные. Определить 2 функции, отличающиеся только спецификатором inline, которые проверяют элемент массива на нечетность, использовать их при вычислении числа нечетных элементов. Освободить память. Сравнить время вычисления.
5. Написать функцию преобразующую единственный параметр в ближайшее нечетное. Написать функцию изменения знака единственного параметра. Написать функцию возвращающую ссылку на передаваемый параметр. Изменить его при вызове функции. Что, если все ссылки сделать const?

Вариант 12

1. Написать программу вычисления максимального значения в массивах типа int,float и complex. Функции, вычисления максимума имеют одно и тоже имя.
2. Написать родовую функцию, которая копирует файл в память по адресу первого аргумента, если файл больше, то память должна быть перевыделена в родовой функции. Проверить для данных типа int, float, double.
3. Создать абстрактный тип данных (структура) - картина, у которой есть название, автор, стоимость. Определить функцию, которая устанавливает по умолчанию стоимость - 4 тыс. долларов. Для задания имени использовать оператор new.
4. Создать абстрактный тип данных (структура) - вектор, который имеет указатель на int и число элементов. Определить функцию, которая выделяет память для массива и инициализирует данные. Определить 2 функции, отличающиеся только спецификатором inline, которые проверяют элемент массива на четность, использовать их при вычислении числа четных элементов. Освободить память. Сравнить время вычисления.
5. Написать функцию, преобразующую единственный параметр в ближайшее сверху степени 2. Написать функцию возвращающую ссылку на элемент глобального массива. Изменить его при вызове функции. Что, если все ссылки сделать const?

Классы. Протокол класса. Конструкторы и деструкторы

Материал для выполнения лабораторной работы можно найти здесь

Для студентов, готовых ограничиться оценкой “удовлетворительно” за экзамен и курсовую работу, в данном пункте достаточно выполнить любые 2 задания.

Вариант 1. Классы. Протокол класса. Конструкторы и деструкторы
Задание 1

Создать абстрактный тип данных - класс вектор, который имеет указатель на int, число элементов и переменную состояния. Определить конструктор без параметров, конструктор с параметром, конструктор с двумя параметрами. Конструктор без параметров выделяет место для одного элемента и инициализирует его в ноль. Конструктор с одним параметром, - размер вектора, - выделяет место и инициализирует номером в массиве, конструктор с двумя параметрами выделяет место (первый аргумент) и инициализирует вторым аргументом. Деструктор освобождает память. Определить функцию, которая присваивает элементу массива некоторое значение (параметр по умолчанию), функцию которая получает некоторый элемент массива. В переменную состояния устанавливать код ошибки, когда не хватает памяти, выходит за пределы массива. Определить функцию печати. Определить функции сложения, умножения, вычитания, которые производят эти арифметические операции с данными этого класса и встроенного int. Определить методы сравнения: больше, меньше или равно. Предусмотреть возможность подсчета числа объектов данного типа. Проверить работу этого класса.

Задание 2

Создать класс матрица Данный класс содержит указатель на int, размер строк и столбцов и состояние ошибки. Определить конструктор без параметров, конструктор с одним параметром и конструктор с двумя параметрами, деструктор. Определить методы доступа: возвращать значение элемента (i,j) и адрес этого элемента. Определить функцию печати. Определить функции сложения и вычитания (матрицы с матрицей), умножение матрицы на матрицу. Определить умножение матрицы на число. Проверить работу этого класса. В случае нехватки памяти, несоответствия размерностей, выхода за пределы устанавливать код ошибки.

Задание 3

Создать класс типа - дата с полями: день (1-31), месяц (1-12), год (целое число). Класс имеет конструктор. Функции-члены установки дня, месяца и года. функции-члены получения дня, месяца и года, а также две функции-члены печати: печать по шаблону: “5 января 1997 года” и “05.01.1997”. Функции-члены установки полей класса должны проверять корректность задаваемых параметров.

Вариант 2. Классы. Протокол класса. Конструкторы и деструкторы
Задание 1

Создать абстрактный тип данных - класс вектор, который имеет указатель на float, число элементов и переменную состояния. Определить конструктор без параметров, конструктор с параметром, конструктор с двумя параметрами. Конструктор без параметров выделяет место для одного элемента и инициализирует его в ноль. Конструктор с одним параметром, - размер вектора, - выделяет место и инициализирует номером в массиве, конструктор с двумя параметрами выделяет место (первый аргумент) и инициализирует вторым аргументом. Деструктор освобождает память. Определить функцию, которая присваивает элементу массива некоторое значение (параметр по умолчанию), функцию которая получает некоторый элемент массива. В переменную состояния устанавливать код ошибки, когда не хватает памяти, выходит за пределы массива. Определить функцию печати. Определить функции сложения, умножения, вычитания, которые производят эти арифметические операции с данными этого класса и встроенного float. Определить методы сравнения: больше, меньше или равно. Предусмотреть возможность подсчета числа объектов данного типа. Проверить работу этого класса.

Задание 2

Создать класс матрица Данный класс содержит указатель на float, размер строк и столбцов и состояние ошибки. Определить конструктор без параметров, конструктор с одним параметром и конструктор с двумя параметрами, деструктор. Определить методы доступа: возвращать значение элемента (i,j) и адрес этого элемента. Определить функцию печати. Определить функции сложения и вычитания (матрицы с матрицей), умножение матрицы на матрицу. Определить умножение матрицы на число. Проверить работу этого класса. В случае нехватки памяти, несоответствия размерностей, выхода за пределы устанавливать код ошибки.

Задание 3

Создать класс типа - время с полями: час (0-23), минуты (0-59), секунды (0-59). Класс имеет конструктор. Функции-члены установки времени,. функции-члены получения часа, минуты и секунды, а также две функции-члены печати: печать по шаблону: “16 часов 18 минут 3 секунды” и “4 p.m. 18 минут 3 секунды”. Функции-члены установки полей класса должны проверять корректность задаваемых параметров.

Вариант 3. Классы. Протокол класса. Конструкторы и деструкторы

Задание 1

Создать абстрактный тип данных - класс вектор, который имеет указатель на double, число элементов и переменную состояния. Определить конструктор без параметров, конструктор с параметром, конструктор с двумя параметрами. Конструктор без параметров выделяет место для одного элемента и инициализирует его в ноль. Конструктор с одним

параметром, - размер вектора, - выделяет место и инициализирует номером в массиве, конструктор с двумя параметрами выделяет место (первый аргумент) и инициализирует вторым аргументом. Деструктор освобождает память. Определить функцию, которая присваивает элементу массива некоторое значение (параметр по умолчанию), функцию которая получает некоторый элемент массива. В переменную состояния устанавливать код ошибки, когда не хватает памяти, выходит за пределы массива. Определить функцию печати. Определить функции сложения, умножения, вычитания, которые производят эти арифметические операции с данными этого класса и встроенного double. Определить методы сравнения: больше, меньше или равно. Предусмотреть возможность подсчета числа объектов данного типа. Проверить работу этого класса.

Задание 2

Создать класс матрица Данный класс содержит указатель на double, размер строк и столбцов и состояние ошибки. Определить конструктор без параметров, конструктор с одним параметром и конструктор с двумя параметрами, деструктор. Определить методы доступа: возвращать значение элемента (i,j) и адрес этого элемента. Определить функцию печати. Определить функции сложения и вычитания (матрицы с матрицей), умножение матрицы на матрицу. Определить умножение матрицы на число. Проверить работу этого класса. В случае нехватки памяти, несоответствия размерностей, выхода за пределы устанавливать код ошибки.

Задание 3

Создать класс типа - прямоугольник. Поля - высота и ширина. Функции-члены вычисляют площадь, периметр, устанавливает поля и возвращают значения. Функции-члены установки полей класса должны проверять корректность задаваемых параметров. Функция печати.

Вариант 4. Классы. Протокол класса. Конструкторы и деструкторы
Задание 1

Создать абстрактный тип данных - класс вектор, который имеет указатель на long, число элементов и переменную состояния. Определить конструктор без параметров, конструктор с параметром, конструктор с двумя параметрами. Конструктор без параметров выделяет место для одного элемента и инициализирует его в ноль. Конструктор с одним параметром, - размер вектора, - выделяет место и инициализирует номером в массиве, конструктор с двумя параметрами выделяет место (первый аргумент) и инициализирует вторым аргументом. Деструктор освобождает память. Определить функцию, которая присваивает элементу массива некоторое значение (параметр по умолчанию), функцию которая получает некоторый элемент массива. В переменную состояния устанавливать код ошибки, когда не хватает памяти, выходит за пределы массива. Определить функцию печати. Определить функции сложения, умножения, вычитания, которые производят эти арифметические операции с данными этого класса и встроенного long. Определить методы сравнения: больше, меньше или

равно. Предусмотреть возможность подсчета числа объектов данного типа. Проверить работу этого класса.

Задание 2

Создать класс матрица Данный класс содержит указатель на long, размер строк и столбцов и состояние ошибки. Определить конструктор без параметров, конструктор с одним параметром и конструктор с двумя параметрами, деструктор. Определить методы доступа: возвращать значение элемента (i,j) и адрес этого элемента. Определить функцию печати. Определить функции сложения и вычитания (матрицы с матрицей), умножение матрицы на матрицу. Определить умножение матрицы на число. Проверить работу этого класса. В случае нехватки памяти, несоответствия размерностей, выхода за пределы устанавливать код ошибки.

Задание 3

Создать класс типа - циклическая очередь. Функции-члены получают элемент и вставляют элемент.

Вариант 5. Классы. Протокол класса. Конструкторы и деструкторы
Задание 1

Создать абстрактный тип данных - класс вектор, который имеет указатель на int, число элементов и переменную состояния. Определить конструктор без параметров, конструктор с параметром, конструктор с двумя параметрами. Конструктор без параметров выделяет место для одного элемента и инициализирует его в ноль. Конструктор с одним параметром, - размер вектора, - выделяет место и инициализирует номером в массиве, конструктор с двумя параметрами выделяет место (первый аргумент) и инициализирует вторым аргументом. Деструктор освобождает память. Определить функцию, которая присваивает элементу массива некоторое значение (параметр по умолчанию), функцию которая получает некоторый элемент массива. В переменную состояния устанавливать код ошибки, когда не хватает памяти, выходит за пределы массива. Определить функцию печати. Определить функции сложения, умножения, вычитания, которые производят эти арифметические операции с данными этого класса и встроенного int. Определить методы сравнения: больше, меньше или равно. Предусмотреть возможность подсчета числа объектов данного типа. Проверить работу этого класса.

Задание 2

Создать класс матрица Данный класс содержит указатель на int, размер строк и столбцов и состояние ошибки. Определить конструктор без параметров, конструктор с одним параметром и конструктор с двумя параметрами, деструктор. Определить методы доступа: возвращать значение элемента (i,j) и адрес этого элемента. Определить функцию печати. Определить функции сложения и вычитания (матрицы с матрицей),

умножение матрицы на матрицу. Определить умножение матрицы на число. Проверить работу этого класса. В случае нехватки памяти, несоответствия размерностей, выхода за пределы устанавливать код ошибки.

Задание 3

Создать класс типа - двухсвязный список. Функции-члены добавляют элемент к списку, удаляют элемент из списка. Отображают элементы списка от начала и от конца. Найти элемент в списке.

Вариант 6. Классы. Протокол класса. Конструкторы и деструкторы
Задание 1

Создать абстрактный тип данных - класс вектор, который имеет указатель на float, число элементов и переменную состояния. Определить конструктор без параметров, конструктор с параметром, конструктор с двумя параметрами. Конструктор без параметров выделяет место для одного элемента и инициализирует его в ноль. Конструктор с одним параметром, - размер вектора, - выделяет место и инициализирует номером в массиве, конструктор с двумя параметрами выделяет место (первый аргумент) и инициализирует вторым аргументом. Деструктор освобождает память. Определить функцию, которая присваивает элементу массива некоторое значение (параметр по умолчанию), функцию которая получает некоторый элемент массива. В переменную состояния устанавливать код ошибки, когда не хватает памяти, выходит за пределы массива. Определить функцию печати. Определить функции сложения, умножения, вычитания, которые производят эти арифметические операции с данными этого класса и встроенного float. Определить методы сравнения: больше, меньше или равно. Предусмотреть возможность подсчета числа объектов данного типа. Проверить работу этого класса.

Задание 2

Создать класс матрица Данный класс содержит указатель на float, размер строк и столбцов и состояние ошибки. Определить конструктор без параметров, конструктор с одним параметром и конструктор с двумя параметрами, деструктор. Определить методы доступа: возвращать значение элемента (i,j) и адрес этого элемента. Определить функцию печати. Определить функции сложения и вычитания (матрицы с матрицей), умножение матрицы на матрицу. Определить умножение матрицы на число. Проверить работу этого класса. В случае нехватки памяти, несоответствия размерностей, выхода за пределы устанавливать код ошибки.

Задание 3

Создать класс типа игра в крестики-нолики. Поле класса - массив из (3x3). Ставить можно только на свободные.

Вариант 7. Классы. Протокол класса. Конструкторы и деструкторы
Задание 1

Создать абстрактный тип данных - класс вектор, который имеет указатель на double, число элементов и переменную состояния. Определить конструктор без параметров, конструктор с параметром, конструктор с двумя параметрами. Конструктор без параметров выделяет место для одного элемента и инициализирует его в ноль. Конструктор с одним параметром, - размер вектора, - выделяет место и инициализирует номером в массиве, конструктор с двумя параметрами выделяет место (первый аргумент) и инициализирует вторым аргументом. Деструктор освобождает память. Определить функцию, которая присваивает элементу массива некоторое значение (параметр по умолчанию), функцию которая получает некоторый элемент массива. В переменную состояния устанавливать код ошибки, когда не хватает памяти, выходит за пределы массива. Определить функцию печати. Определить функции сложения, умножения, вычитания, которые производят эти арифметические операции с данными этого класса и встроенного double. Определить методы сравнения: больше, меньше или равно. Предусмотреть возможность подсчета числа объектов данного типа. Проверить работу этого класса.

Задание 2

Создать класс матрица Данный класс содержит указатель на double, размер строк и столбцов и состояние ошибки. Определить конструктор без параметров, конструктор с одним параметром и конструктор с двумя параметрами, деструктор. Определить методы доступа: возвращать значение элемента (i,j) и адрес этого элемента. Определить функцию печати. Определить функции сложения и вычитания (матрицы с матрицей), умножение матрицы на матрицу. Определить умножение матрицы на число. Проверить работу этого класса. В случае нехватки памяти, несоответствия размерностей, выхода за пределы устанавливать код ошибки.

Задание 3

Создать класс типа - односвязный список. Функции-члены добавляют элемент к списку, удаляют элемент из списка. Отображают элементы списка от начала. Найти элемент в списке.

Вариант 8. Классы. Протокол класса. Конструкторы и деструкторы
Задание 1

Создать абстрактный тип данных - класс вектор, который имеет указатель на long, число элементов и переменную состояния. Определить конструктор без параметров, конструктор с параметром, конструктор с двумя параметрами. Конструктор без параметров выделяет место для одного элемента и инициализирует его в ноль. Конструктор с одним параметром, - размер вектора, - выделяет место и инициализирует номером в массиве, конструктор с двумя параметрами выделяет место (первый аргумент) и

инициализирует вторым аргументом. Деструктор освобождает память. Определить функцию, которая присваивает элементу массива некоторое значение (параметр по умолчанию), функцию которая получает некоторый элемент массива. В переменную состояния устанавливать код ошибки, когда не хватает памяти, выходит за пределы массива. Определить функцию печати. Определить функции сложения, умножения, вычитания, которые производят эти арифметические операции с данными этого класса и встроенного long. Определить методы сравнения: больше, меньше или равно. Предусмотреть возможность подсчета числа объектов данного типа. Проверить работу этого класса.

Задание 2

Создать класс матрица Данный класс содержит указатель на long, размер строк и столбцов и состояние ошибки. Определить конструктор без параметров, конструктор с одним параметром и конструктор с двумя параметрами, деструктор. Определить методы доступа: возвращать значение элемента (i,j) и адрес этого элемента. Определить функцию печати. Определить функции сложения и вычитания (матрицы с матрицей), умножение матрицы на матрицу. Определить умножение матрицы на число. Проверить работу этого класса. В случае нехватки памяти, несоответствия размерностей, выхода за пределы устанавливать код ошибки.

Задание 3

Создать класс типа - окружность. Поля - радиус. Функции-члены вычисляют площадь, длину окружности, устанавливает поля и возвращают значения. Функции-члены установки полей класса должны проверять корректность задаваемых параметров. Функция печати.

Вариант 9. Классы. Протокол класса. Конструкторы и деструкторы
Задание 1

Создать абстрактный тип данных - класс вектор, который имеет указатель на int, число элементов и переменную состояния. Определить конструктор без параметров, конструктор с параметром, конструктор с двумя параметрами. Конструктор без параметров выделяет место для одного элемента и инициализирует его в ноль. Конструктор с одним параметром, - размер вектора, - выделяет место и инициализирует номером в массиве, конструктор с двумя параметрами выделяет место (первый аргумент) и инициализирует вторым аргументом. Деструктор освобождает память. Определить функцию, которая присваивает элементу массива некоторое значение (параметр по умолчанию), функцию которая получает некоторый элемент массива. В переменную состояния устанавливать код ошибки, когда не хватает памяти, выходит за пределы массива. Определить функцию печати. Определить функции сложения, умножения, вычитания, которые производят эти арифметические операции с данными этого класса и встроенного int. Определить методы сравнения: больше, меньше или равно.

Предусмотреть возможность подсчета числа объектов данного типа. Проверить работу этого класса.

Задание 2

Создать класс матрица Данный класс содержит указатель на int, размер строк и столбцов и состояние ошибки. Определить конструктор без параметров, конструктор с одним параметром и конструктор с двумя параметрами, деструктор. Определить методы доступа: возвращать значение элемента (i,j) и адрес этого элемента. Определить функцию печати. Определить функции сложения и вычитания (матрицы с матрицей), умножение матрицы на матрицу. Определить умножение матрицы на число. Проверить работу этого класса. В случае нехватки памяти, несоответствия размерностей, выхода за пределы устанавливать код ошибки.

Задание 3

Создать класс типа - дата с полями: день (1-31), месяц (1-12), год (целое число). Класс имеет конструктор. Функции-члены установки дня, месяца и года. функции-члены получения дня, месяца и года, а также две функции-члены печати: печать по шаблону: “5 января 1997 года” и “05.01.1997”. Функции-члены установки полей класса должны проверять корректность задаваемых параметров. Функция-член дает приращение на 1 день.

Вариант 10. Классы. Протокол класса. Конструкторы и деструкторы
Задание 1

Создать абстрактный тип данных - класс вектор, который имеет указатель на float, число элементов и переменную состояния. Определить конструктор без параметров, конструктор с параметром, конструктор с двумя параметрами. Конструктор без параметров выделяет место для одного элемента и инициализирует его в ноль. Конструктор с одним параметром, - размер вектора, - выделяет место и инициализирует номером в массиве, конструктор с двумя параметрами выделяет место (первый аргумент) и инициализирует вторым аргументом. Деструктор освобождает память. Определить функцию, которая присваивает элементу массива некоторое значение (параметр по умолчанию), функцию которая получает некоторый элемент массива. В переменную состояния устанавливать код ошибки, когда не хватает памяти, выходит за пределы массива. Определить функцию печати. Определить функции сложения, умножения, вычитания, которые производят эти арифметические операции с данными этого класса и встроенного float. Определить методы сравнения: больше, меньше или равно. Предусмотреть возможность подсчета числа объектов данного типа. Проверить работу этого класса.

Задание 2

Создать класс матрица Данный класс содержит указатель на float, размер строк и столбцов и состояние ошибки. Определить конструктор без

параметров, конструктор с одним параметром и конструктор с двумя параметрами, деструктор. Определить методы доступа: возвращать значение элемента (i,j) и адрес этого элемента. Определить функцию печати. Определить функции сложения и вычитания (матрицы с матрицей), умножение матрицы на матрицу. Определить умножение матрицы на число. Проверить работу этого класса. В случае нехватки памяти, несоответствия размерностей, выхода за пределы устанавливать код ошибки.

Задание 3

Создать класс типа - время с полями: час (0-23), минуты (0-59), секунды (0-59). Класс имеет конструктор. Функции-члены установки времени,. функции-члены получения часа, минуты и секунды, а также две функции-члены печати: печать по шаблону: “16 часов 18 минут 3 секунды” и “4 p.m. 18 минут 3 секунды”. Функции-члены установки полей класса должны проверять корректность задаваемых параметров.

Вариант 11. Классы. Протокол класса. Конструкторы и деструкторы
Задание 1

Создать абстрактный тип данных - класс вектор, который имеет указатель на double, число элементов и переменную состояния. Определить конструктор без параметров, конструктор с параметром, конструктор с двумя параметрами. Конструктор без параметров выделяет место для одного элемента и инициализирует его в ноль. Конструктор с одним параметром, - размер вектора, - выделяет место и инициализирует номером в массиве, конструктор с двумя параметрами выделяет место (первый аргумент) и инициализирует вторым аргументом. Деструктор освобождает память. Определить функцию, которая присваивает элементу массива некоторое значение (параметр по умолчанию), функцию которая получает некоторый элемент массива. В переменную состояния устанавливать код ошибки, когда не хватает памяти, выходит за пределы массива. Определить функцию печати. Определить функции сложения, умножения, вычитания, которые производят эти арифметические операции с данными этого класса и встроенного double. Определить методы сравнения: больше, меньше или равно. Предусмотреть возможность подсчета числа объектов данного типа. Проверить работу этого класса.

Задание 2

Создать класс матрица Данный класс содержит указатель на double, размер строк и столбцов и состояние ошибки. Определить конструктор без параметров, конструктор с одним параметром и конструктор с двумя параметрами, деструктор. Определить методы доступа: возвращать значение элемента (i,j) и адрес этого элемента. Определить функцию печати. Определить функции сложения и вычитания (матрицы с матрицей), умножение матрицы на матрицу. Определить умножение матрицы на число. Проверить работу этого класса. В случае нехватки памяти, несоответствия размерностей, выхода за пределы устанавливать код ошибки.

Задание 3

Создать класс типа - время с полями: час (0-23), минуты (0-59), секунды (0-59). Класс имеет конструктор. Функции-члены установки времени,. функции-члены получения часа, минуты и секунды, а также две функции-члены печати: печать по шаблону: “16 часов 18 минут 3 секунды” и “4 p.m. 18 минут 3 секунды”. Функции-члены установки полей класса должны проверять корректность задаваемых параметров. Функция-член дает приращение на 1 сек. 1 мин, и 1 час.

Вариант 12. Классы. Протокол класса. Конструкторы и деструкторы
Задание 1

Создать абстрактный тип данных - класс вектор, который имеет указатель на int, число элементов и переменную состояния. Определить конструктор без параметров, конструктор с параметром, конструктор с двумя параметрами. Конструктор без параметров выделяет место для одного элемента и инициализирует его в ноль. Конструктор с одним параметром, - размер вектора, - выделяет место и инициализирует номером в массиве, конструктор с двумя параметрами выделяет место (первый аргумент) и инициализирует вторым аргументом. Деструктор освобождает память. Определить функцию, которая присваивает элементу массива некоторое значение (параметр по умолчанию), функцию которая получает некоторый элемент массива. В переменную состояния устанавливать код ошибки, когда не хватает памяти, выходит за пределы массива. Определить функцию печати. Определить функции сложения, умножения, вычитания, которые производят эти арифметические операции с данными этого класса и встроенного int. Определить методы сравнения: больше, меньше или равно. Предусмотреть возможность подсчета числа объектов данного типа. Проверить работу этого класса.

Задание 2

Создать класс матрица Данный класс содержит указатель на int, размер строк и столбцов и состояние ошибки. Определить конструктор без параметров, конструктор с одним параметром и конструктор с двумя параметрами, деструктор. Определить методы доступа: возвращать значение элемента (i,j) и адрес этого элемента. Определить функцию печати. Определить функции сложения и вычитания (матрицы с матрицей), умножение матрицы на матрицу. Определить умножение матрицы на число. Проверить работу этого класса. В случае нехватки памяти, несоответствия размерностей, выхода за пределы устанавливать код ошибки.

Задание 3

Создать класс типа - квадрат. Поля - сторона. Функции-члены вычисляют площадь, периметр, устанавливает поля и возвращают значения. Функции-члены установки полей класса должны проверять корректность задаваемых параметров. Функция печати.

Вариант 13. Классы. Протокол класса. Конструкторы и деструкторы
Задание 1

Создать абстрактный тип данных - класс вектор, который имеет указатель на long, число элементов и переменную состояния. Определить конструктор без параметров, конструктор с параметром, конструктор с двумя параметрами. Конструктор без параметров выделяет место для одного элемента и инициализирует его в ноль. Конструктор с одним параметром, - размер вектора, - выделяет место и инициализирует номером в массиве, конструктор с двумя параметрами выделяет место (первый аргумент) и инициализирует вторым аргументом. Деструктор освобождает память. Определить функцию, которая присваивает элементу массива некоторое значение (параметр по умолчанию), функцию которая получает некоторый элемент массива. В переменную состояния устанавливать код ошибки, когда не хватает памяти, выходит за пределы массива. Определить функцию печати. Определить функции сложения, умножения, вычитания, которые производят эти арифметические операции с данными этого класса и встроенного long. Определить методы сравнения: больше, меньше или равно. Предусмотреть возможность подсчета числа объектов данного типа. Проверить работу этого класса.

Задание 2

Создать класс матрица Данный класс содержит указатель на long, размер строк и столбцов и состояние ошибки. Определить конструктор без параметров, конструктор с одним параметром и конструктор с двумя параметрами, деструктор. Определить методы доступа: возвращать значение элемента (i,j) и адрес этого элемента. Определить функцию печати. Определить функции сложения и вычитания (матрицы с матрицей), умножение матрицы на матрицу. Определить умножение матрицы на число. Проверить работу этого класса. В случае нехватки памяти, несоответствия размерностей, выхода за пределы устанавливать код ошибки.

Задание 3

Создайте класс, который использует массив из 40 элементов для хранения целых чисел вплоть до больших целых, содержащих по 40 цифр. Функции-члены должны вводить, выводить, складывать и вычитать эти большие, целые. Сравнение больших целых чисел. Деление и умножение целых больших чисел.

Вариант 14. Классы. Протокол класса. Конструкторы и деструкторы
Задание 1

Создать абстрактный тип данных - класс вектор, который имеет указатель на float, число элементов и переменную состояния. Определить конструктор без параметров, конструктор с параметром, конструктор с двумя параметрами. Конструктор без параметров выделяет место для одного элемента и инициализирует его в ноль. Конструктор с одним параметром, - размер вектора, - выделяет место и инициализирует номером в массиве, конструктор с двумя параметрами выделяет место (первый аргумент) и

инициализирует вторым аргументом. Деструктор освобождает память. Определить функцию, которая присваивает элементу массива некоторое значение (параметр по умолчанию), функцию которая получает некоторый элемент массива. В переменную состояния устанавливать код ошибки, когда не хватает памяти, выходит за пределы массива. Определить функцию печати. Определить функции сложения, умножения, вычитания, которые производят эти арифметические операции с данными этого класса и встроенного float. Определить методы сравнения: больше, меньше или равно. Предусмотреть возможность подсчета числа объектов данного типа. Проверить работу этого класса.

Задание 2

Создать класс матрица Данный класс содержит указатель на float, размер строк и столбцов и состояние ошибки. Определить конструктор без параметров, конструктор с одним параметром и конструктор с двумя параметрами, деструктор. Определить методы доступа: возвращать значение элемента (i,j) и адрес этого элемента. Определить функцию печати. Определить функции сложения и вычитания (матрицы с матрицей), умножение матрицы на матрицу. Определить умножение матрицы на число. Проверить работу этого класса. В случае нехватки памяти, несоответствия размерностей, выхода за пределы устанавливать код ошибки.

Задание 3

Создать класс типа - стек. Функции-члены вставляют элемент в стек, вытаскивают элемент из стека. Проверяют вершину стека.

Преобразование типов. Дружественные функции

Материал для выполнения лабораторной работы можно найти здесь

Для студентов, готовых ограничиться оценкой “удовлетворительно” за экзамен и курсовую работу, в данном пункте достаточно выполнить любое 1 задание.

Вариант 1. Преобразование типов. Дружественные функции. Конструктор копирования.
Задание 1

Создать класс комплексных чисел, члены класса - реальная и мнимая части. Класс имеет конструктор по умолчанию, конструктор - преобразующий float в объект класса. Определить оператор преобразования объекта типа комплексных чисел в число типа float. Создать класс вещественных чисел. Определить взаимное преобразование с классом комплексных чисел.

Задание 2

Создать класс комплексных чисел. Определить перегруженную функцию, возвращающую максимальный из двух аргументов. Функция не является членом класса комплексных чисел. Перегруженные функции имеют аргументы типа int, double, complex. Тело перегруженных функций должны быть одинаковыми.

Задание 3

Создать два класса вектор (int *) и матрица (int **). Определить конструкторы - по умолчанию, с параметром, для класса матрица с двумя параметрами, копирования, деструкторы. Определить функцию умножения матрицу на вектор как дружественную.

Вариант 2. Преобразование типов. Дружественные функции. Конструктор копирования.
Задание 1

Создать класс вещественных чисел. Класс имеет конструктор по умолчанию, конструктор - преобразующий float в объект класса. Определить оператор преобразования объекта типа вещественных чисел в число типа float. Создать класс целых чисел. Определить взаимное преобразование с классом вещественных чисел.

Задание 2

Создать класс комплексных чисел. Определить перегруженную функцию, возвращающую минимальный из двух аргументов. Функция не является членом класса комплексных чисел. Перегруженные функции имеют аргументы типа int, double, complex. Тело перегруженных функций должны быть одинаковыми.

Задание 3

Создать два класса: целые(Integer) и матрица (int *). Определить конструкторы - по умолчанию, с параметром, для класса матрица с двумя параметрами, копирования, деструкторы. Определить функцию умножения матрицу на целое как дружественную.

Вариант 3. Преобразование типов. Дружественные функции. Конструктор копирования.

Задание 1

Создать класс целых чисел. Класс имеет конструктор по умолчанию, конструктор - преобразующий int в объект класса. Определить оператор преобразования объекта типа целых чисел в число типа int. Создать класс вещественных чисел. Определить взаимное преобразование с классом целых чисел.

Задание 2

Создать класс целых чисел Integer. Определить перегруженную функцию, возвращающую максимальное из двух аргументов. Функция не является членом класса целых чисел. Перегруженные функции имеют аргументы типа int, double, Integer. Тело перегруженных функций должны быть одинаковыми.

Задание 3

Создать два класса: вещественные (Float) и матрица (float **). Определить конструкторы - по умолчанию, с параметром, для класса матрица с двумя параметрами, копирования, деструкторы. Определить функцию умножения матрицу на вещественное (Float) как дружественную.

Вариант 4. Преобразование типов. Дружественные функции. Конструктор копирования.
Задание 1

Определить два класс, строку с преобразование из char * в строку и обратно и Целое Int с преобразованием из int и обратно, а также взаимное преобразование String и Int.

Задание 2

Создать класс вещественных чисел Double. Определить перегруженную функцию, возвращающую максимальное из двух аргументов. Функция не является членом класса Double. Перегруженные функции имеют аргументы типа int, double, Double. Тело перегруженных функций должны быть одинаковыми.

Задание 3

Создать два класса вектор (double *) и матрица (double **). Определить конструкторы - по умолчанию, с параметром, для класса матрица с двумя параметрами, копирования, деструкторы. Определить функцию умножения матрицу на вектор как дружественную.

Вариант 5. Преобразование типов. Дружественные функции. Конструктор копирования.
Задание 1

Определить два класс, строку с преобразование из char * в строку и обратно и вещественное Float с преобразованием из float и обратно, а также взаимное преобразование String и Float.

Задание 2

Создать класс вещественных чисел Double. Определить перегруженную функцию, возвращающую максимальное из двух аргументов. Функция не является членом класса Double. Перегруженные функции имеют аргументы типа int, double, Double. Тело перегруженных функций должны быть одинаковыми.

Задание 3

Создать два класса вектор (long *) и матрица (long **). Определить конструкторы - по умолчанию, с параметром, для класса матрица с двумя параметрами, копирования, деструкторы. Определить функцию умножения матрицу на вектор как дружественную.

Вариант 6. Преобразование типов. Дружественные функции. Конструктор копирования.
Задание 1

Определить два класс, строку с преобразование из char * в строку и обратно и Double с преобразованием из double и обратно, а также взаимное преобразование String и Double.

Задание 2

Создать класс комплексных чисел. Определить перегруженную функцию, возвращающую максимальный из двух аргументов. Функция не является членом класса комплексных чисел. Перегруженные функции имеют аргументы типа int, double, complex. Тело перегруженных функций должны быть одинаковыми.

Задание 3

Создать два класса вектор (int *) и матрица (int **). Определить конструкторы - по умолчанию, с параметром, для класса матрица с двумя параметрами, копирования, деструкторы. Определить функцию умножения матрицу на вектор как дружественную.

Вариант 7. Преобразование типов. Дружественные функции. Конструктор копирования.
Задание 1

Определить два класс, строку с преобразование из char * в строку и обратно и Целое Long с преобразованием из long и обратно, а также взаимное преобразование String и Long.

Задание 2

Создать класс комплексных чисел. Определить перегруженную функцию, возвращающую максимальный из двух аргументов. Функция не является членом класса комплексных чисел. Перегруженные функции имеют аргументы типа int, double, complex. Тело перегруженных функций должны быть одинаковыми.

Задание 3

Создать два класса вектор (float *) и матрица (float **). Определить конструкторы - по умолчанию, с параметром, для класса матрица с двумя параметрами, копирования, деструкторы. Определить функцию умножения матрицу на вектор.

Вариант 8. Преобразование типов. Дружественные функции. Конструктор копирования.
Задание 1

Определить два класс, строку с преобразование из char * в строку и обратно и Целое Int с преобразованием из int и обратно, а также взаимное преобразование String и Int.

Задание 2

Создать класс целых чисел Integer. Определить перегруженную функцию, возвращающую максимальное из двух аргументов. Функция не является членом класса целых чисел. Перегруженные функции имеют аргументы типа int, double, Integer. Тело перегруженных функций должны быть одинаковыми.

Задание 3

Создать два класса вектор (long *) и матрица (long **). Определить конструкторы - по умолчанию, с параметром, для класса матрица с двумя параметрами, копирования, деструкторы. Определить функцию умножения матрицу на вектор.

Вариант 9. Преобразование типов. Дружественные функции. Конструктор копирования.
Задание 1

Создать класс коротких целых чисел. Класс имеет конструктор по умолчанию, конструктор - преобразующий short int в объект класса. Определить оператор преобразования объекта типа короткое целое в число типа short int. Создать класс вещественных чисел. Определить взаимное преобразование с классом коротких целых чисел.

Задание 2

Создать класс вещественных чисел Double. Определить перегруженную функцию, возвращающую максимальное из двух аргументов. Функция не является членом класса Double. Перегруженные функции имеют аргументы

типа int, double, Double. Тело перегруженных функций должны быть одинаковыми.

Задание 3

Создать два класса: целые (Integer) и вектор (int *). Определить конструкторы - по умолчанию, с параметром, для класса вектор с двумя параметрами, копирования, деструкторы. Определить функцию умножения вектора на целое как дружественную.

Вариант 10. Преобразование типов. Дружественные функции. Конструктор копирования.
Задание 1

Создать класс длинных целых чисел. Класс имеет конструктор по умолчанию, конструктор - преобразующий long в объект класса. Определить оператор преобразования объекта типа длинных целых чисел в число типа long. Создать класс целых чисел. Определить взаимное преобразование с классом длинных целых чисел.

Задание 2

Создать класс целых чисел Integer. Определить перегруженную функцию, возвращающую максимальное из двух аргументов. Функция не является членом класса целых чисел. Перегруженные функции имеют аргументы типа int, double, Integer. Тело перегруженных функций должны быть одинаковыми.

Задание 3

Создать два класса: вещественные (Float) и вектор (float *). Определить конструкторы - по умолчанию, с параметром, для класса вектор с двумя параметрами, копирования, деструкторы. Определить функцию умножения вектора на Float как дружественную.

Вариант 11. Преобразование типов. Дружественные функции. Конструктор копирования.
Задание 1

Создать класс без знаковых целых чисел. Класс имеет конструктор по умолчанию, конструктор - преобразующий unsigned int в объект класса. Определить оператор преобразования объекта типа без знаковых целых чисел в число типа unsigned int. Создать класс вещественных чисел. Определить взаимное преобразование с классом без знаковых целых чисел.

Задание 2

Создать класс целых чисел Integer. Определить перегруженную функцию, возвращающую минимальное из двух аргументов. Функция не является членом класса целых чисел. Перегруженные функции имеют аргументы типа int, double, Integer. Тело перегруженных функций должны быть одинаковыми.

Задание 3

Создать два класса вектор (float *) и матрица (float **). Определить конструкторы - по умолчанию, с параметром, для класса матрица с двумя параметрами, копирования, деструкторы. Определить функцию умножения матрицу на вектор как дружественную.

Вариант 12. Преобразование типов. Дружественные функции. Конструктор копирования.
Задание 1

Создать класс вещественных с двойной точностью чисел. Класс имеет конструктор по умолчанию, конструктор - преобразующий double в объект класса. Определить оператор преобразования объекта типа вещественных чисел с двойной точностью в число типа double. Создать класс целых чисел. Определить взаимное преобразование с классом вещественных чисел.

Задание 2

Создать класс вещественных чисел Float. Определить перегруженную функцию, возвращающую минимальное из двух аргументов. Функция не является членом класса Float. Перегруженные функции имеют аргументы типа int, float, Float. Тело перегруженных функций должны быть одинаковыми.

Задание 3

Создать два класса вектор (double *) и матрица (double **). Определить конструкторы - по умолчанию, с параметром, для класса матрица с двумя параметрами, копирования, деструкторы. Определить функцию умножения матрицу на вектор как дружественную.

Вариант 13. Преобразование типов. Дружественные функции . Конструктор копирования.
Задание 1

Определить два класс, строку с преобразование из char * в строку и обратно и Double с преобразованием из double и обратно, а также взаимное преобразование String и Double.

Задание 2

Создать класс вещественных чисел Float. Определить перегруженную функцию, возвращающую минимальное из двух аргументов. Функция не является членом класса Float. Перегруженные функции имеют аргументы типа int, float, Float. Тело перегруженных функций должны быть одинаковыми.

Задание 3

Создать два класса вектор (float *) и матрица (float **). Определить конструкторы - по умолчанию, с параметром, для класса матрица с двумя

параметрами, копирования, деструкторы. Определить функцию умножения матрицу на вектор как дружественную.

Вариант 14. Преобразование типов. Дружественные функции. Конструктор копирования.
Задание 1

Определить два класс, строку с преобразование из char * в строку и обратно и Double с преобразованием из double и обратно, а также взаимное преобразование String и Double.

Задание 2

Создать класс целых чисел Integer. Определить перегруженную функцию, возвращающую минимальное из двух аргументов. Функция не является членом класса целых чисел. Перегруженные функции имеют аргументы типа int, double, Integer. Тело перегруженных функций должны быть одинаковыми.

Задание 3

Создать два класса вектор (double *) и матрица (double **). Определить конструкторы - по умолчанию, с параметром, для класса матрица с двумя параметрами, копирования, деструкторы. Определить функцию умножения матрицу на вектор как дружественную.

Перегрузка операторов

Материал для выполнения лабораторной работы можно найти здесь

Для студентов, готовых ограничиться оценкой “удовлетворительно” за экзамен и курсовую работу, в данном пункте достаточно выполнить первые 2 задания.

Вариант 1. Перегрузка операторов
Задание 1 Унарная операция

Создать класс целых чисел. Определить оператор ++, как функцию-член и -- как дружественную функцию.

Задание 2. Бинарная операция

Создать класс целых чисел. Определить оператор +, как функцию-член и - как дружественную функцию.

Задание 3.

Создать класс вектор, содержащий ссылку на int, размерность вектора и переменную ошибки. Класс имеет конструкторы по умолчанию, конструктор с одним и двумя параметрами, конструктор копирования и деструктор. Определить оператор +, - , * , - как дружественные функции, =, +=, -=, *=,[] - как функции-члены. Определить операторы =, +, - , *, +=, -=, *= с целым числом операторы ++ и --. Определить функцию печати. Сравнить время работы созданного класса и встроенного массива типа int. Перегрузить операторы вывода и ввода в поток.

Задание 4

Создать класс матриц, содержащий ссылку на int, число строк и столбцов и состояние ошибки. Определить конструкторы по умолчанию, конструктор с одним и с двумя параметрами, конструктор копирования, деструктор. Определить операторы =, +, -, +=, -=, *., *= с объектами этого класс, с целым и с вектором, определенном в задании 10. Определить оператор [] так, чтобы обращение [][] к элементам имело смысл, аналогичный встроенному. Перегрузить операторы вывода и ввода в поток.

Вариант 2. Перегрузка операторов
Задание 1 Унарная операция

Создать класс целых чисел. Определить оператор --, как функцию-член и ++ как дружественную функцию.

Задание 2. Бинарная операция

Создать класс координат. Определить оператор +, как функцию-член и - как дружественную функцию. Сложить и вычесть координаты с друг другом и с числом. Присвоить координаты, сравнить координаты (==, !=).

Задание 3

Создать класс вектор, содержащий ссылку на long, размерность вектора и переменную ошибки. Класс имеет конструкторы по умолчанию, конструктор с одним и двумя параметрами, конструктор копирования и деструктор. Определить оператор +, - , * , - как дружественные функции, =, +=, -=, *=,[] - как функции-члены. Определить операторы =, +, - , *, +=, -=, *= с числом типа long, операторы ++ и --. Определить функцию печати. Сравнить время работы созданного класса и встроенного массива типа long. Перегрузить операторы вывода и ввода в поток.

Задание 4

Создать класс матриц, содержащий ссылку на long, число строк и столбцов и состояние ошибки. Определить конструкторы по умолчанию, конструктор с одним и с двумя параметрами, конструктор копирования, деструктор. Определить операторы =, +, -, +=, -=, *., *= с объектами этого класс, с long и с вектором, определенном в задании 10. Определить оператор [] так, чтобы обращение [][] к элементам имело смысл, аналогичный встроенному. Перегрузить операторы вывода и ввода в поток.

Вариант 3. Перегрузка операторов

Задание 1 Унарная операция

Создать класс вещественных чисел. Определить оператор ++, как функцию-член и -- как дружественную функцию.

Задание 2. Бинарная операция

Создать класс целых чисел. Определить оператор -, как функцию-член и + как дружественную функцию.

Задание 3

Создать класс вектор, содержащий ссылку на float, размерность вектора и переменную ошибки. Класс имеет конструкторы по умолчанию, конструктор с одним и двумя параметрами, конструктор копирования и деструктор. Определить оператор +, - , * , - как дружественные функции, =, +=, -=, *=,[] - как функции-члены. Определить операторы =, +, - , *, +=, -=, *= с целым числом операторы ++ и --. Определить операторы =, +, - , *, +=, -=, *= с вещественным числом, операторы ++ и --. Определить функцию печати. Сравнить время работы созданного класса и встроенного массива типа float. Перегрузить операторы вывода и ввода в поток.

Задание 4

Создать класс матриц, содержащий ссылку на float, число строк и столбцов и состояние ошибки. Определить конструкторы по умолчанию, конструктор с одним и с двумя параметрами, конструктор копирования, деструктор. Определить операторы =, +, -, +=, -=, *., *= с объектами этого класс, с float и

с вектором, определенном в задании 10. Определить оператор [] так, чтобы обращение [][] к элементам имело смысл, аналогичный встроенному. Перегрузить операторы вывода и ввода в поток.

Вариант 4. Перегрузка операторов
Задание 1. Унарная операция

Создать объект типа очередь. Перегрузить оператор ++ как функцию член и -- как дружественную функцию. (Как постфиксными так префиксными). ++ добавляет элемент в очередь (пустой элемент, например int i=0), -- вытаскивает элемент из очереди. Оператор ! проверяет очередь на пустоту.

Задание 2. Бинарная операция

Создать объект типа стек. Перегрузить оператор + как функцию член и * как дружественную функцию. + складывает элемент в новый стек, * умножает верхушку стека на параметр. Стеки можно присваивать, проверять на равенство == или !=, вводить и выводить в поток, добавлять += элемент в стек.

Задание 3

Создать объект - двунаправленный список, в котором определены операции, + - добавляет в конец списка, += добавляет в этот же список в конец списка. - удаляет указанный элемент из списка (номер элемента через параметр), = - присвоение списков, сравнение списков ==, !=, >, <, >=, <=, [] получение элемента списка, ++ - устанавливает указатель на следующий элемент, -- устанавливает указатель на предыдущий элемент. () выдать подсписок от первого до второго элемента.

Задание 4

Создать класс матриц и вектор, содержащие ссылку на float, число строк и столбцов (для вектора длину) и состояние ошибки. Определить конструкторы по умолчанию, конструктор с одним и с двумя параметрами, конструктор копирования, деструктор. Определить операторы =, +, -, +=, -=, *., *= с объектами этого класс, с float и с вектором. Определить оператор [] так, чтобы обращение [][] к элементам имело смысл, аналогичный встроенному. Перегрузить операторы вывода и ввода в поток. Для вектора должны быть определены кроме перечисленных ++, --, - унарный, [], /=, /. Проверки. Операторы * И *= должны быть определены для умножения вектора и матрицы.

Вариант 5. Перегрузка операторов
Задание 1 Унарная операция

Создать объект типа стек. Перегрузить оператор ++ как функцию член и -- как дружественную функцию. (Как постфиксными так префиксными). ++ добавляет элемент новый в стек, -- удаляет верхушку стека. Оператор ! проверяет стек на пустоту.

Задание 2. Бинарная операция

Создать объект типа очередь. Перегрузить оператор + как функцию член и * как дружественную функцию. + добавляет элемент в очередь, * умножает элемент в очереди. Вытаскивает элемент из очереди --. Очереди можно присваивать, проверять на равенство == или !=, вводить и выводить в поток, добавлять += элемент в очередь.

Задание 3

Создать объект - однонаправленный список, в котором определены операции, + - добавляет в конец списка, += добавляет в этот же список в конец списка. - удаляет указанный элемент из списка (номер элемента через параметр), = - присвоение списков, сравнение списков ==, !=, >, <, >=, <=, [] получение элемента списка, ++ - устанавливает указатель на следующий элемент. () выдать подсписок от первого до второго элемента.

Задание 4

Создать класс матриц и вектор, содержащие ссылку на double, число строк и столбцов (для вектора длину) и состояние ошибки. Определить конструкторы по умолчанию, конструктор с одним и с двумя параметрами, конструктор копирования, деструктор. Определить операторы =, +, -, +=, -=, *., *= с объектами этого класс, с float и с вектором. Определить оператор [] так, чтобы обращение [][] к элементам имело смысл, аналогичный встроенному. Перегрузить операторы вывода и ввода в поток. Для вектора должны быть определены кроме перечисленных ++, --, - унарный, [], /=, /. Проверки. Операторы * И *= должны быть определены для умножения вектора и матрицы.

Вариант 6. Перегрузка операторов
Задание 1 Унарная операция

Создать объект - связный двунаправленный список, с перегруженными унарными операциями ++, --, как движение по списку. (Как постфиксными так префиксными).

Задание 2 Бинарная операция

Создать объект динамический стек. Перегрузить операции +, +=, -= (с извлечением элемента).

Задание 3

Создать объект динамическая очередь. Перегрузив операции +, --, +=, -=, =, !=, ==, >=, <=, >, <, ввода, вывода в поток, получить под-очередь ().

Задание 4

Создать класс матриц и вектор, содержащие ссылку на long, число строк и столбцов (для вектора длину) и состояние ошибки. Определить

конструкторы по умолчанию, конструктор с одним и с двумя параметрами, конструктор копирования, деструктор. Определить операторы =, +, -, +=, -=, *., *= с объектами этого класс, с float и с вектором. Определить оператор [] так, чтобы обращение [][] к элементам имело смысл, аналогичный встроенному. Перегрузить операторы вывода и ввода в поток. Для вектора должны быть определены кроме перечисленных ++, --, - унарный, [], /=, /. Проверки. Операторы * И *= должны быть определены для умножения вектора и матрицы.

Вариант 7. Перегрузка операторов
Задание 1. Унарная операция

Создать объект стек, перегрузив ++ и --. (Как постфиксными так префиксными). ++ Добавляет элемент в стек. -- извлекает элемент из стека.

Задание 2. Бинарная операция

Создать объект очередь с перегруженными +, +=, добавление элемента в очередь и сложение очередей, -- для извлечения из очереди, - для вычитания очередей.

Задание 3

Определить класс список однонаправленный с перегруженными операциями ++ вперед по списку, -- удалить элемент, на котором стоит указатель, += с другим списком и с новым элементом, - унарный удаляет с конца списка, =, ==, !=, >, <, <=, >=. Ввод, вывод в поток. () - выдает подсписок.

Задание 4

Создать класс матриц и вектор, содержащие ссылку на int, число строк и столбцов (для вектора длину) и состояние ошибки. Определить конструкторы по умолчанию, конструктор с одним и с двумя параметрами, конструктор копирования, деструктор. Определить операторы =, +, -, +=, -=, *., *= с объектами этого класс, с float и с вектором. Определить оператор [] так, чтобы обращение [][] к элементам имело смысл, аналогичный встроенному. Перегрузить операторы вывода и ввода в поток. Для вектора должны быть определены кроме перечисленных ++, --, - унарный, [], /=, /. Проверки. Операторы * И *= должны быть определены для умножения вектора и матрицы.

Вариант 8. Перегрузка операторов
Задание 1. Унарная операция

Создать класс - координаты с унарным ++ и --, -. ++ и -- постфиксная и префиксная. - меняет знак у обеих координат. ++ как функция-член, -- как дружественная функция.

Задание 2. Бинарная операция

Создать класс целых чисел (long). Определить оператор -, как функцию-член и + как дружественную функцию. Оператор присвоения, и сравнений.

Задание 3

Создать класс вектор, содержащий ссылку на double, размерность вектора и переменную ошибки. Класс имеет конструкторы по умолчанию, конструктор с одним и двумя параметрами, конструктор копирования и деструктор. Определить оператор +, - , * , - как дружественные функции, =, +=, -=, *=,[] - как функции-члены. Определить функцию печати. Сравнить время работы созданного класса и встроенного массива типа double. Перегрузить операторы вывода и ввода в поток.

Задание 4

Создать класс матриц, содержащий ссылку на double, число строк и столбцов и состояние ошибки. Определить конструкторы по умолчанию, конструктор с одним и с двумя параметрами, конструктор копирования, деструктор. Определить операторы =, +, -, +=, -=, *., *= с объектами этого класс, с double и с вектором, определенном в задании 10. Определить оператор [] так, чтобы обращение [][] к элементам имело смысл, аналогичный встроенному. Перегрузить операторы вывода и ввода в поток.

Вариант 9. Перегрузка операторов
Задание 1. Унарная операция

Создать класс вещественных чисел (double). Определить оператор ++, как функцию-член и -- как дружественную функцию.

Задание 2. Бинарная операция

Создать класс целых чисел (long). Определить оператор +, как функцию-член и - как дружественную функцию.

Задание 3

Создать класс вектор, содержащий ссылку на unsigned , размерность вектора и переменную ошибки. Класс имеет конструкторы по умолчанию, конструктор с одним и двумя параметрами, конструктор копирования и деструктор. Определить оператор +, - , * , - как дружественные функции, =, +=, -=, *=,[] - как функции-члены. Определить операторы =, +, - , *, +=, -=, *= с целым без знаковым числом, операторы ++ и --. Определить функцию печати. Сравнить время работы созданного класса и встроенного массива типа unsigned. Перегрузить операторы вывода и ввода в поток.

Задание 4

Создать класс матриц, содержащий ссылку на unsigned, число строк и столбцов и состояние ошибки. Определить конструкторы по умолчанию, конструктор с одним и с двумя параметрами, конструктор копирования, деструктор. Определить операторы =, +, -, +=, -=, *., *= с объектами этого

класс, с unsigned и с вектором, определенном в задании 10. Определить оператор [] так, чтобы обращение [][] к элементам имело смысл, аналогичный встроенному. Перегрузить операторы вывода и ввода в поток.

Вариант 10. Перегрузка операторов
Задание 1. Унарная операция

Создать класс вещественных чисел (double). Определить оператор --, как функцию-член и ++ как дружественную функцию.

Задание 2. Бинарная операция

Создать класс вещественных чисел (double). Определить оператор -, как функцию-член и + как дружественную функцию.

Задание 3

Создать класс вектор, содержащий ссылку на unsigned long, размерность вектора и переменную ошибки. Класс имеет конструкторы по умолчанию, конструктор с одним и двумя параметрами, конструктор копирования и деструктор. Определить оператор +, - , * , - как дружественные функции, =, +=, -=, *=,[] - как функции-члены. Определить операторы =, +, - , *, +=, -=, *= с типа unsigned long, операторы ++ и --. Определить функцию печати. Сравнить время работы созданного класса и встроенного массива типа unsigned long. Перегрузить операторы вывода и ввода в поток.

Задание 4

Создать класс матриц, содержащий ссылку на unsigned long, число строк и столбцов и состояние ошибки. Определить конструкторы по умолчанию, конструктор с одним и с двумя параметрами, конструктор копирования, деструктор. Определить операторы =, +, -, +=, -=, *., *= с объектами этого класс, с unsigned long и с вектором, определенном в задании 10. Определить оператор [] так, чтобы обращение [][] к элементам имело смысл, аналогичный встроенному. Перегрузить операторы вывода и ввода в поток.

Вариант 11. Перегрузка операторов
Задание 1. Унарная операция

Создать класс целых чисел (long). Определить оператор ++, как функцию-член и -- как дружественную функцию.

Задание 2. Бинарная операция

Создать класс вещественных чисел (double). Определить оператор +, как функцию-член и - как дружественную функцию.

Задание 3

Создать класс вектор, содержащий ссылку на unsigned char, размерность вектора и переменную ошибки. Класс имеет конструкторы по умолчанию,

конструктор с одним и двумя параметрами, конструктор копирования и деструктор. Определить оператор +, - , * , - как дружественные функции, =, +=, -=, *=,[] - как функции-члены. Определить операторы =, +, - , *, +=, -=, *= с числом типа unsigned char, операторы ++ и --. Определить функцию печати. Сравнить время работы созданного класса и встроенного массива типа unsigned char. Перегрузить операторы вывода и ввода в поток.

Задание 4

Создать класс матриц, содержащий ссылку на unsigned char, число строк и столбцов и состояние ошибки. Определить конструкторы по умолчанию, конструктор с одним и с двумя параметрами, конструктор копирования, деструктор. Определить операторы =, +, -, +=, -=, *., *= с объектами этого класс, с unsigned char и с вектором, определенном в задании 10. Определить оператор [] так, чтобы обращение [][] к элементам имело смысл, аналогичный встроенному. Перегрузить операторы вывода и ввода в поток.

Вариант 12. Перегрузка операторов
Задание 1. Унарная операция

Создать класс целых чисел (long). Определить оператор --, как функцию-член и ++ как дружественную функцию.

Задание 2. Бинарная операция

Создать класс вещественных чисел. Определить оператор -, как функцию-член и + как дружественную функцию.

Задание 3

Создать класс вектор, содержащий ссылку на long double, размерность вектора и переменную ошибки. Класс имеет конструкторы по умолчанию, конструктор с одним и двумя параметрами, конструктор копирования и деструктор. Определить оператор +, - , * , - как дружественные функции, =, +=, -=, *=,[] - как функции-члены. Определить операторы =, +, - , *, +=, -=, *= с целым числом операторы ++ и --. Определить операторы =, +, - , *, +=, -=, *= с числом типа long double, операторы ++ и --. Определить функцию печати. Сравнить время работы созданного класса и встроенного массива типа long double. Перегрузить операторы вывода и ввода в поток.

Задание 4

Создать класс матриц, содержащий ссылку на long double, число строк и столбцов и состояние ошибки. Определить конструкторы по умолчанию, конструктор с одним и с двумя параметрами, конструктор копирования, деструктор. Определить операторы =, +, -, +=, -=, *., *= с объектами этого класс, с long double и с вектором, определенном в задании 10. Определить оператор [] так, чтобы обращение [][] к элементам имело смысл, аналогичный встроенному. Перегрузить операторы вывода и ввода в поток.

Вариант 13. Перегрузка операторов
Задание 1. Унарная операция

Создать объект - очередь с перегруженными операциями ++ как функциями-членами, -- как дружественными функциями. (Как постфиксными так префиксными).

Задание 2. Бинарная операция

Создать объект - однонаправленный список, в котором определены операции, + - добавляет в конец списка, += добавляет в этот же список в конец списка. - удаляет указанный элемент из списка (номер элемента через параметр), = - присвоение списков, сравнение списков ==, !=, >, <, >=, <=, [] получение элемента списка, ++ - устанавливает указатель на следующий элемент. () выдать подсписок от первого до второго элемента.

Задание 3

Создать объект типа стек. Перегрузить оператор ++, --, !, !=, ==, >, <, >=, <=, +,. Ввод, вывод в поток.

Задание 4

Создать класс матриц и вектор, содержащие ссылку на float, число строк и столбцов (для вектора длину) и состояние ошибки. Определить конструкторы по умолчанию, конструктор с одним и с двумя параметрами, конструктор копирования, деструктор. Определить операторы =, +, -, +=, -=, *., *= с объектами этого класс, с float и с вектором. Определить оператор [] так, чтобы обращение [][] к элементам имело смысл, аналогичный встроенному. Перегрузить операторы вывода и ввода в поток. Для вектора должны быть определены кроме перечисленных ++, --, - унарный, [], /=, /. Проверки. Операторы * И *= должны быть определены для умножения вектора и матрицы.

Вариант 14. Перегрузка операторов
Задание 1. Унарная операция

Создать объект - однонаправленный список, в котором определены операции, ++ - добавляет в конец списка, -- удаляет элемент из списка. (Как постфиксными так префиксными).

Задание 2. Бинарная операция

Создать объект стек, с перегруженными операциями +, *, =, +=, и для вытаскивания из стека --. () - выдает под-стек.

Задание 3

Определить класс - комплексные числа, перегрузив различные операторы, +, -, ++, --, +=, -=, *, ./, *=, /=, !, !=, ==, >,<, >=, <=. Ввода, вывода в поток.

Сложение и вычитание должно производиться как с элементами данного класса так и со встроенными float.

Задание 4

Создать класс матриц и вектор, содержащие ссылку на double, число строк и столбцов (для вектора длину) и состояние ошибки. Определить конструкторы по умолчанию, конструктор с одним и с двумя параметрами, конструктор копирования, деструктор. Определить операторы =, +, -, +=, -=, *., *= с объектами этого класс, с float и с вектором. Определить оператор [] так, чтобы обращение [][] к элементам имело смысл, аналогичный встроенному. Перегрузить операторы вывода и ввода в поток. Для вектора должны быть определены кроме перечисленных ++, --, - унарный, [], /=, /. Проверки. Операторы * И *= должны быть определены для умножения вектора и матрицы.

Наследование. Иерархия и контейнерные классы

Материал для выполнения лабораторной работы можно найти здесь

Для студентов, готовых ограничиться оценкой “удовлетворительно” за экзамен и курсовую работу, в данном пункте достаточно выполнить одно задания.

Вариант 1. Одиночное наследование
Задание 1 . Иерархия

Создать класс студент, имеющий имя (указатель на строку), курс и идентификационный номер. Определить конструкторы, деструктор и функцию печати. Создать public-производный класс - студент-дипломник, имеющий тему диплома. Определить конструкторы по умолчанию и с разным числом параметров, деструкторы, функцию печати. Определить функции переназначения названия диплома и идентификационного номера.

Задание 2. Композиция и иерархия

Создать класс комната, имеющая площадь. Определить конструктор и метод доступа. Создать класс однокомнатных квартира, содержащий комнату и кухню (ее площадь), этаж (комната содержится в классе однокомнатная квартира). Определить конструкторы, методы доступа. Определить public-производный класс однокомнатных квартир разных городов (дополнительный параметр - название города). Определить конструкторы, деструктор и функцию печати.

Задание 3. Конструктор копирования и оператор присваивания

Создать иерархию классов спортивная игра и футбол. Переопределить вывод в поток и ввод из потока, конструктор копирования, оператор присваивания через соответствующие функции базового класса.

Вариант 2. Одиночное наследование
Задание 1 Иерархия.

Создать класс животное, имеющий классификацию (указатель на строку), число конечностей, число потомков. Определить конструкторы, деструктор и функцию печати. Создать public-производный класс - домашнее животное, имеющий кличку. Определить конструкторы по умолчанию и с разным числом параметров, деструкторы, функцию печати. Определить функции переназначения названия клички и числа потомков.

Задание 2 Композиция и иерархия.

Создать класс хвост, имеющий длину. Определить конструкторы и метод доступа. Создать класс собачьих, содержащий класс хвост. Дополнительно есть цвет (указатель на строку), возраст. Определить конструкторы и деструктор. Определить public- производный класс собака, имеющий дополнительно кличку(указатель на строку). Определить конструкторы, деструкторы и функцию печати.

Задание 3. Конструктор копирования и оператор присваивания

Создать иерархию классов образование и высшее образование. Переопределить вывод в поток и ввод из потока, конструктор копирования, оператор присваивания через соответствующие функции базового класса.

Вариант 3. Одиночное наследование

Задание 1. Иерархия

Создать класс машина, имеющий марку (указатель на строку), число цилиндров, мощность. Определить конструкторы, деструктор и функцию печати. Создать public-производный класс - грузовики, имеющий грузоподъемность кузова. Определить конструкторы по умолчанию и с разным числом параметров, деструкторы, функцию печати. Определить функции переназначения марки и грузоподъемности.

Задание 2. Композиция и иерархия.

Создать класс двигатель, имеющий мощность. Определить конструкторы и метод доступа Создать класс машин, содержащий класс двигатель. Дополнительно есть марка (указатель на строку), цена. Определить конструкторы и деструктор. Определить public- производный класс грузовик, имеющий дополнительно грузоподъемность. Определить конструкторы, деструкторы и функцию печати.

Задание 3. Конструктор копирования и оператор присваивания

Создать иерархию классов мебель и стол. Переопределить вывод в поток и ввод из потока, конструктор копирования, оператор присваивания через соответствующие функции базового класса.

Вариант 4. Одиночное наследование
Задание 1. Иерархия

Создать класс карта, имеющая ранг и масть. Карту можно перевернуть и открыть. Создать класс - колода карт, содержащий карты. Создать два производных класса от колоды карт, в одном карты могут доставаться только по порядку, в другом - вытаскиваться произвольно.

Задание 2 Композиция и иерархия.

Используя иерархию и композицию классов, создать бинарное дерево. У бинарного дерева есть корневой узел. Мы можем вставлять узел. Мы можем обходить вначале левое поддерево, потом правое (последовательный обход) и обратный обход. Узел может быть помещен в дерево двоичного поиска только в качестве концевого узла. Если дерево является пустым, то создается новый экземпляр класса узел дерева и узел помещается в дерево. Если дерево не является пустым, то программа сравнивает вставляемое в дерево значение со значением в корневом узле и если меньше, то помещает

в левые поддеревья, а если больше, то в правые. Если значения равны, то выводится сообщение, что повтор и не вставляется.

Задание 3. Конструктор копирования и оператор присваивания

Создать класс строк и производный от него - класс строк из цифр. Определить конструкторы и деструкторы, переопределить вывод и ввод в поток. Перегрузить оператор присваивания и конструктор копирования в базовом и производном классе.

Вариант 5. Одиночное наследование
Задание 1. Иерархия

Создайте класс точка, которая имеет координаты. Класс эллипсов, и класс окружностей. Определить иерархию типов. Определить функции печати, конструкторы, деструкторы, вычисление площади.

Задание 2. Композиция и иерархия

Используя иерархию и композицию классов, создать бинарное дерево. У бинарного дерева есть корневой узел. Мы можем вставлять узел. Мы можем обходить в ширину и обратный обход. Узел может быть помещен в дерево двоичного поиска только в качестве концевого узла. Если дерево является пустым, то создается новый экземпляр класса узел дерева и узел помещается в дерево. Если дерево не является пустым, то программа сравнивает вставляемое в дерево значение со значением в корневом узле и если меньше, то помещает в левые поддеревья, а если больше, то в правые. Если значения равны, то выводится сообщение, что повтор и не вставляется.

Задание 3. Конструктор копирования и оператор присваивания

Создать класс строк и производный от него - класс строк из цифр. Определить конструкторы и деструкторы, переопределить вывод и ввод в поток. Перегрузить оператор присваивания и конструктор копирования в базовом и производном классе.

Вариант 6. Одиночное наследование
Задание 1. Иерархия

Создать класс четырехугольников, квадратов и прямоугольников. Создать из них иерархию. Определить функции печати, конструкторы и деструкторы, вычисление площади и периметра.

Задание 2. Композиция и иерархия

Создать карту и использую композицию - колоду карт. Конструкторы колоды должны инициализировать колоду упорядочено и случайным образом. Создать производный класс от колоды - пасьянс, в котором выбираются по три карты и, если две крайние одного цвета, то их выбрасывают. Всю колоду проходят три раза.

Задание 3. Конструктор копирования и оператор присваивания

Создать иерархию классов вектор и безопасный вектор с проверкой выхода за пределы. Безопасный вектор определяет переменные нижний и верхний предел Переопределить вывод в поток и ввод из потока, конструктор копирования, оператор присваивания через соответствующие функции базового класса.

Вариант 7. Одиночное наследование
Задание 1. Иерархия

Создать класс транспортные средства, автомобиль, грузовик, пароход и самолет. Создать из них иерархию. Определить функции печати, конструкторы и деструкторы.

Задание 2. Композиция и иерархия

Создать классы колесо, велосипед и автомобиль. Составить из них иерархию или композицию.

Задание 3. Конструктор копирования и оператор присваивания

Создать иерархию классов человек и преподаватель. Переопределить вывод в поток и ввод из потока, конструктор копирования, оператор присваивания через соответствующие функции базового класса.

Вариант 8. Одиночное наследование
Задание 1. Иерархия

Создать класс жидкость, имеющий название (указатель на строку), плотность. Определить конструкторы, деструктор и функцию печати. Создать public-производный класс - спиртные напитки, имеющий крепость. Определить конструкторы по умолчанию и с разным числом параметров, деструкторы, функцию печати. Определить функции переназначения плотности и крепости.

Задание 2. Композиция и иерархия

Используя иерархию и наследование, создать классы окна, окна с заголовком и окна с кнопкой.

Задание 3. Конструктор копирования и оператор присваивания

Создать иерархию классов студент и студент дипломник. Переопределить вывод в поток и ввод из потока, конструктор копирования, оператор присваивания через соответствующие функции базового класса.

Вариант 9. Одиночное наследование
Задание 1. Иерархия

Создать класс человек, имеющий имя (указатель на строку), возраст, вес. Определить конструкторы, деструктор и функцию печати. Создать public-производный класс - школьник, имеющий класс (год обучения). Определить конструкторы по умолчанию и с разным числом параметров, деструкторы, функцию печати. Определить функции переназначения возраста и класса.

Задание 2. Композиция и иерархия

Создать класс жесткий диск, имеющий объем (Мбайт). Определить конструкторы и метод доступа. Создать класс компьютер, содержащий класс жесткий диск. Дополнительно есть марка (указатель на строку), цена. Определить конструкторы и деструктор. Определить private-, public-производный класс компьютеров с монитором, имеющий дополнительно размер монитора. Определит конструкторы, деструкторы и функцию печати.

Задание 3. Конструктор копирования и оператор присваивания

Создать иерархию классов четырехугольник и квадрат. Переопределить вывод в поток и ввод из потока, конструктор копирования, оператор присваивания через соответствующие функции базового класса.

Вариант 10. Одиночное наследование
Задание 1. Иерархия

Создать класс окно, имеющий координаты верхнего левого и нижнего правого угла, цвет фона (указатель на строку). Определить конструкторы, деструктор и функцию печати. Создать public-производный класс - окно с меню, имеющий строку меню. Определить конструкторы по умолчанию и с разным числом параметров, деструкторы, функцию печати. Определить функции переназначения цвета фона и строки меню.

Задание 2. Композиция и иерархия

Создать класс процессор, имеющий мощность (МГц). Определить конструкторы и метод доступа. Создать класс компьютер, содержащий класс процессор. Дополнительно есть марка (указатель на строку), цена. Определить конструкторы и деструктор. Определить private-, public-производный класс компьютеров с монитором, имеющий дополнительно размер монитора. Определит конструкторы, деструкторы и функцию печати.

Задание 3. Конструктор копирования и оператор присваивания

Создать иерархию классов человек и служащий. Переопределить вывод в поток и ввод из потока, конструктор копирования, оператор присваивания через соответствующие функции базового класса.

Вариант 11. Одиночное наследование
Задание 1. Иерархия

Создать класс точка, имеющая координаты. Определить конструкторы, деструктор и функцию печати. Создать public-производный класс - цветная точка, имеющий цвет точки . Определить конструкторы по умолчанию и с разным числом параметров, деструкторы, функцию печати. Определить функции переназначения цвета и координат точки, вывода точки на экран.

Задание 2. Композиция и иерархия

Создать класс кнопка, имеющий размер. Определить конструкторы и метод доступа. Создать класс окно, содержащий класс кнопка. Дополнительно есть координаты окна. Определить конструкторы и деструктор. Определить public- производный класс окно с кнопкой и имеющее меню (указатель на строку). Определить конструкторы, деструкторы и функцию печати.

Задание 3. Конструктор копирования и оператор присваивания

Создать иерархию классов человек и студент. Переопределить вывод в поток и ввод из потока, конструктор копирования, оператор присваивания через соответствующие функции базового класса.

Вариант 12. Одиночное наследование
Задание 1. Иерархия

Создать класс человек, имеющий имя (указатель на строку), возраст, вес. Определить конструкторы, деструктор и функцию печати. Создать public-производный класс - совершеннолетний, имеющий номер паспорта. Определить конструкторы по умолчанию и с разным числом параметров, деструкторы, функцию печати. Определить функции переназначения возраста и номера паспорта.

Задание 2. Композиция и иерархия

Создать класс колесо, имеющий радиус. Определить конструкторы и метод доступа. Создать класс машин, содержащий класс колесо. Дополнительно есть марка (указатель на строку), цена. Определить конструкторы и деструктор. Определить public- производный класс грузовик, имеющий дополнительно грузоподъемность. Определить конструкторы, деструкторы и функцию печати.

Задание 3. Конструктор копирования и оператор присваивания

Создать иерархию классов окно и окно с заголовком. Переопределить вывод в поток и ввод из потока, конструктор копирования, оператор присваивания через соответствующие функции базового класса.

Вариант 13. Одиночное наследование
Задание 1. Иерархия

Создать колоду карт. Конструкторы колоды должны инициализировать колоду упорядочено и случайным образом. Создать производный класс от

колоды - пасьянс, в котором выбираются по три карты и, если две крайние одного цвета, то их выбрасывают. Всю колоду проходят три раза.

Задание 2. Композиция и иерархия

Используя иерархию и композицию классов, создать бинарное дерево. У бинарного дерева есть корневой узел. Мы можем вставлять узел. Мы можем обходить в ширину .

Задание 3. Конструктор копирования и оператор присваивания

Создать иерархию классов точка и цветная точка. Переопределить вывод в поток и ввод из потока, конструктор копирования, оператор присваивания через соответствующие функции базового класса.

Вариант 14. Одиночное наследование.
Задание 1. Иерархия

Создать класс точка и производные от него - окружность и эллипс. Определить конструкторы, деструктор и функцию печати. Определить функции переустановки центра окружности и эллипса.

Задание 2. Композиция и иерархия

Создать класс окно, используя композицию и иерархию, определить окно с заголовком и кнопкой. Класс заголовок и кнопка тоже должны быть определены. Навесить на кнопку функцию обратного вызова - (метод класса окна). Эта функция вызывается, когда нажимается любая клавиша.

Задание 3. Конструктор копирования и оператор присваивания

Создать иерархию классов строк и строк из прописных букв. Переопределить вывод в поток и ввод из потока, конструктор копирования, оператор присваивания через соответствующие функции базового класса.

Виртуальные функции

Материал для выполнения лабораторной работы можно найти здесь

Для студентов, готовых ограничиться оценкой "удовлетворительно" за экзамен и курсовую работу, в данном пункте достаточно выполнить одно задания.

Вариант 1.
Задание 1 .

Создать абстрактный базовый класс с виртуальной функцией - площадь. Создать производные классы: прямоугольник, круг, прямоугольный треугольник, трапеция со своими функциями площади. Для проверки определить массив ссылок на абстрактный класс, которым присваиваются адреса различных объектов. Площадь трапеции:$S=(a+b)h/2$

Задание 2.

Создать класс - данные - абстрактный базовый класс. Создать производные классы - данные типа сигнал, данные типа результат обработки и вспомогательные данные. Все данные имеют функции отображения, сохранения и обработки.

Вариант 2.
Задание 1 .

Создать абстрактный класс с виртуальной функцией: норма. Создать производные классы: комплексные числа, вектор из 10 элементов, матрица (2x2). Определить функцию нормы - для комплексных чисел - модуль в квадрате, для вектора - корень квадратный из суммы элементов по модулю, для матрицы - максимальное значение по модулю.

Задание 2

Создать класс - данные - абстрактный базовый класс. Создать производные классы - данные типа сигнал, данные типа результат обработки и вспомогательные данные. Все данные имеют функции отображения, сохранения и обработки**..**

Вариант 3.
Задание 1 .

Создать абстрактный класс (кривые) вычисления координаты y для некоторой x. Создать производные классы: прямая, эллипс, гипербола со своими функциями вычисления y в зависимости от входного параметра x.

Уравнение прямой: $y=ax+b$, эллипса: $x^2/a^2+y^2/b^2=1$, гиперболы: $x^2/a^2-y^2/b^2=1$

Задание 2.

Создать класс - данные - абстрактный базовый класс. Создать производные классы - данные типа сигнал, данные типа результат обработки и вспомогательные данные. Все данные имеют функции отображения, сохранения и обработки.

Вариант 4.
Задание 1 .

Создать абстрактный базовый класс с виртуальной функцией - сумма прогрессии. Создать производные классы: арифметическая прогрессия и геометрическая прогрессия. Каждый класс имеет два поля типа double. Первое - первый член прогрессии, второе (double) - постоянная разность (для арифметической) и постоянное отношение (для геометрической). Определить функцию вычисления суммы, где параметром является количество элементов прогрессии.

Арифметическая прогрессия $a_j=a_0+jd$, $j=0,1,2,\ldots$

Сумма арифметической прогрессии: $s_n=(n+1)(a_0+a_n)/2$

Геометрическая прогрессия: $a_j=a_0r^j$, $j=0,1,2,\ldots$

Сумма геометрической прогрессии: $s_n=(a_0-a_nr)/(1-r)$

Задание 2.

Создать класс - данные - абстрактный базовый класс. Создать производные классы - данные типа сигнал, данные типа результат обработки и вспомогательные данные. Все данные имеют функции отображения, сохранения и обработки.

Вариант 5.
Задание 1 .

Создать базовый класс список. Реализовать на базе списка стек и очередь с виртуальными функциями вставки и вытаскивания.

Задание 2.

Создать класс - данные - абстрактный базовый класс. Создать производные классы - данные типа сигнал, данные типа результат обработки и вспомогательные данные. Все данные имеют функции отображения, сохранения и обработки.

Вариант 6.
Задание 1 .

Создать базовый класс - фигура, и производные класс - круг, прямоугольник, трапеция. Определить виртуальные функции площадь, периметр и вывод на печать.

Задание 2.

Создать класс - данные - абстрактный базовый класс. Создать производные классы - данные типа сигнал, данные типа результат обработки и вспомогательные данные. Все данные имеют функции отображения, сохранения и обработки.

Вариант 7.
Задание 1 .

Создать базовый класс - работник и производные классы - служащий с почасовой оплатой, служащий в штате и служащий с процентной ставкой. Определить функцию начисления зарплаты.

Задание 2.

Создать класс - данные - абстрактный базовый класс. Создать производные классы - данные типа сигнал, данные типа результат обработки и вспомогательные данные. Все данные имеют функции отображения, сохранения и обработки.

Вариант 8.
Задание 1 .

Создать абстрактный базовый класс с виртуальной функцией - площадь поверхности. Создать производные классы: параллелепипед, тетраэдр, шар со своими функциями площади поверхности. Для проверки определить массив ссылок на абстрактный класс, которым присваиваются адреса различных объектов.

Площадь поверхности параллелепипеда: S=6*xy*. Площадь поверхности шара: $S=4\pi r^2$. Площадь поверхности тетраэдра: $S=a^2\sqrt{3}$

Задание 2.

Создать класс - данные - абстрактный базовый класс. Создать производные классы - данные типа сигнал, данные типа результат обработки и вспомогательные данные. Все данные имеют функции отображения, сохранения и обработки.

Вариант 9.
Задание 1 .

Создать класс человек, производные от которого девушки и молодые люди. Определить виртуальную функцию реакции человека на вновь увиденного другого человека.

Задание 2.

Создать класс - данные - абстрактный базовый класс. Создать производные классы - данные типа сигнал, данные типа результат обработки и вспомогательные данные. Все данные имеют функции отображения, сохранения и обработки.

Вариант 10.
Задание 1 .

Создать абстрактный базовый класс с виртуальной функцией - объем. Создать производные классы: параллелепипед, пирамида, тетраэдр, шар со своими функциями объема. Для проверки определить массив ссылок на абстрактный класс, которым присваиваются адреса различных объектов.

Объем параллелепипеда - $V=xyz$ (*x,y,z - стороны* , пирамиды: $V=xyh$ (*x,y,* - стороны, *h - высота)*, тетраэдра: $V=a^3\sqrt{2}/12$, шара: $V=4\pi r^3/3$.

Задание 2.

Создать класс - данные - абстрактный базовый класс. Создать производные классы - данные типа сигнал, данные типа результат обработки и вспомогательные данные. Все данные имеют функции отображения, сохранения и обработки.

Вариант 11.
Задание 1

Создать абстрактный класс - млекопитающие. Определить производные классы - животные и люди. У животных определить производные классы собак и коров. Определить виртуальные функции описания человека, собаки и коровы.

Задание 2.

Создать класс - данные - абстрактный базовый класс. Создать производные классы - данные типа сигнал, данные типа результат обработки и вспомогательные данные. Все данные имеют функции отображения, сохранения и обработки.

Вариант 12.
Задание 1 .

Создать базовый класс - Предок, у которого есть имя. определить виртуальную функцию печати. Создать производный класс Ребенок, у которого функция печати дополнительно выводит имя. Создать производный класс от последнего класса - Внук, у которого есть отчество. Написать свою функцию печати.

Задание 2.

Создать класс - данные - абстрактный базовый класс. Создать производные классы - данные типа сигнал, данные типа результат обработки и вспомогательные данные. Все данные имеют функции отображения, сохранения и обработки.

Вариант 13.
Задание 1 .

Создать класс живущих с местоположением. Определить наследуемые классы - лиса, кролик и трава. Лиса ест кролика. Кролик ест траву. Лиса может умереть - определен возраст. Кролик тоже может умереть. Кроме этого определен класс - отсутствие жизни. Если в окрестности имеется больше травы, чем кроликов, то трава остается, иначе трава съедена. Если лис слишком старый он может умереть. Если лис слишком много (больше 5 в окрестности), лисы больше не появляются. Если кроликов меньше лис, то лис ест кролика.

Задание 2.

Создать класс - данные - абстрактный базовый класс. Создать производные классы - данные типа сигнал, данные типа результат обработки и вспомогательные данные. Все данные имеют функции отображения, сохранения и обработки.

Вариант 14.
Задание 1 .

Создать абстрактный базовый класс с виртуальной функцией - корни уравнения. Создать производные классы: класс линейных уравнений и класс квадратных уравнений. Определить функцию вычисления корней уравнений.

Задание 2.

Создать класс - данные - абстрактный базовый класс. Создать производные классы - данные типа сигнал, данные типа результат обработки и вспомогательные данные. Все данные имеют функции отображения, сохранения и обработки.

Множественное наследование

Материал для выполнения лабораторной работы можно найти здесь

Для студентов, готовых ограничиться оценкой "удовлетворительно" за экзамен и курсовую работу, в данном пункте достаточно выполнить одно задания.

Вариант 1.
Задание 1 .

Создать иерархию типов, описывающую - студента, отца семейства и студента-отца семейства. Классы должны конструкторы, включая конструктор копирования, виртуальные деструкторы, перегруженные функции вывода в поток и ввода в поток.

Задание 2.

Создать иерархию типов, описывающую - человека, студента, отца семейства и студента-отца семейства. Использовать виртуальные базовые классы.

Вариант 2.
Задание 1 .

Создать иерархию типов, описывающую работника и отца-семейства, и работника-отца семейства. Классы должны конструкторы, включая конструктор копирования, виртуальные деструкторы, перегруженные функции вывода в поток и ввода в поток.

Задание 2

Создать иерархию типов, описывающую - человека, работника, отца семейства и работника-отца семейства. Использовать виртуальные базовые классы.

Вариант 3.
Задание 1 .

Создать иерархию типов - файл для чтения, файл для записи и файл для чтения и записи. Классы должны конструкторы, включая конструктор копирования, виртуальные деструкторы, перегруженные функции вывода в поток и ввода в поток.

Задание 2.

Создать иерархию типов, описывающую файл, файл для чтения, файл для записи и файл для чтения и записи. Использовать виртуальные базовые классы.

Вариант 4.
Задание 1 .

Создать иерархию типов, описывающую работника и женщину, и работника-женщину семейства. Классы должны конструкторы, включая конструктор копирования, виртуальные деструкторы, перегруженные функции вывода в поток и ввода в поток.

Задание 2.

Создать иерархию типов, описывающую человека, работника и женщину, и работника-женщину семейства. Использовать виртуальные базовые классы.

Вариант 5.
Задание 1 .

Создать иерархию типов, описывающую операционную систему и прикладное программное обеспечение, и Windows NT как операционную систему и прикладное программное обеспечение. Классы должны конструкторы, включая конструктор копирования, виртуальные деструкторы, перегруженные функции вывода в поток и ввода в поток.

Задание 2.

Создать иерархию типов, описывающую программное обеспечение, операционную систему и прикладное программное обеспечение, и Windows NT как операционную систему и прикладное программное обеспечение. Использовать виртуальные базовые классы.

Вариант 6.
Задание 1 .

Создать иерархию типов, описывающую данные - сигнал, данные результат обработки и данные, как результат обработки сигнала и представляющие собой сигнал. Классы должны конструкторы, включая конструктор копирования, виртуальные деструкторы, перегруженные функции вывода в поток и ввода в поток.

Задание 2.

Создать иерархию типов, описывающую данные - сигнал, данные результат обработки и данные, как результат обработки сигнала и представляющие собой сигнал. Использовать виртуальные базовые классы.

Вариант 7.
Задание 1 .

Создать иерархию типов - море, залив и бухта. . Классы должны конструкторы, включая конструктор копирования, виртуальные деструкторы, перегруженные функции вывода в поток и ввода в поток.

Задание 2.

Создать иерархию типов - соленная вода, море, залив и бухта. Использовать виртуальные базовые классы.

Вариант 8.
Задание 1 .

Создать иерархию типов - корабль, пассажирский транспорт и пассажирский корабль. Классы должны конструкторы, включая конструктор копирования, виртуальные деструкторы, перегруженные функции вывода в поток и ввода в поток.

Задание 2.

Создать иерархию типов - транспорт, корабль, пассажирский транспорт и пассажирский корабль. Использовать виртуальные базовые классы.

Вариант 9.
Задание 1 .

Создать иерархию типов - машина, пассажирский транспорт и автобус. Классы должны конструкторы, включая конструктор копирования, виртуальные деструкторы, перегруженные функции вывода в поток и ввода в поток.

Задание 2.

Создать иерархию типов - машина, пассажирский транспорт и автобус. Использовать виртуальные базовые классы.

Вариант 10.
Задание 1 .

Создать иерархию типов, описывающую - студента, отца семейства и студента-отца семейства. Классы должны конструкторы, включая конструктор копирования, виртуальные деструкторы, перегруженные функции вывода в поток и ввода в поток.

Задание 2.

Создать иерархию типов, описывающую - человека, студента, отца семейства и студента-отца семейства. Использовать виртуальные базовые классы.

Вариант 11.
Задание 1 .

Создать иерархию типов, описывающую работника и отца-семейства, и работника-отца семейства. Классы должны конструкторы, включая конструктор копирования, виртуальные деструкторы, перегруженные функции вывода в поток и ввода в поток.

Задание 2

Создать иерархию типов, описывающую - человека, работника, отца семейства и работника-отца семейства. Использовать виртуальные базовые классы.

Вариант 12.
Задание 1 .

Создать иерархию типов - файл для чтения, файл для записи и файл для чтения и записи. Классы должны конструкторы, включая конструктор копирования, виртуальные деструкторы, перегруженные функции вывода в поток и ввода в поток.

Задание 2.

Создать иерархию типов, описывающую файл, файл для чтения, файл для записи и файл для чтения и записи. Использовать виртуальные базовые классы.

Вариант 13.
Задание 1 .

Создать иерархию типов - корабль, пассажирский транспорт и пассажирский корабль. Классы должны конструкторы, включая конструктор копирования, виртуальные деструкторы, перегруженные функции вывода в поток и ввода в поток.

Задание 2.

Создать иерархию типов - транспорт, корабль, пассажирский транспорт и пассажирский корабль. Использовать виртуальные базовые классы.

Вариант 14.
Задание 1 .

Создать иерархию типов, описывающую операционную систему и прикладное программное обеспечение, и Unix как операционную систему и прикладное программное обеспечение. Классы должны конструкторы, включая конструктор копирования, виртуальные деструкторы, перегруженные функции вывода в поток и ввода в поток.

Задание 2.

Создать иерархию типов, описывающую программное обеспечение, операционную систему и прикладное программное обеспечение, и Unix как операционную систему и прикладное программное обеспечение. Использовать виртуальные базовые классы.

Лабораторная работа №10
Объектно-ориентированное программирование

Шаблоны

Материал для выполнения лабораторной работы можно найти здесь

Для студентов, готовых ограничиться оценкой "удовлетворительно" за экзамен и курсовую работу, в данном пункте достаточно выполнить одно задание.

Вариант 1.
Задание 1 .

Написать функцию-шаблон последовательного поиска в массиве по ключу. Функция возвращает индекс первого, найденного элемента в массиве, равного ключу.

Задание 2.

Создать параметризованный массив с конструкторами, деструктором и перегруженными операторами [], =.

Вариант 2.
Задание 1 .

Написать функцию-шаблон, вычисляющую максимальное значение в массиве.

Задание 2

Создать параметризованный стек.

Вариант 3.
Задание 1 .

Написать функцию-шаблон, переставляющую элементы в массиве.

Задание 2.

Создать параметризованный массив с конструкторами, деструктором и перегруженными операторами [], =, вывода и ввода в поток.

Вариант 4.
Задание 1 .

Написать параметризованную функцию сортировки методом отбора.
Сортировка методом отбора заключается в следующем: алгоритм находит элемент с наименьшим значением и выполняет перестановку, меняя его местами с первым элементом массива. После этого из оставшихся n-1 элементов ищется наименьший, после чего осуществляется его перестановка со вторым элементом и так далее. Перестановка будет осуществляться до тех пор, пока не поменяются местами последние два элемента. Например, если бы строка "dcab" сортировалась методом отбора, то каждый из проходов давал бы результат:

Исходный массив	Dcab
Проход 1	Acdb
Проход 2	Abdc
Проход 3	Abcd

Задание 2.

Создать параметризованную очередь.

Вариант 5.
Задание 1 .

Написать параметризованную функцию сортировки методом быстрой сортировки. Алгоритм быстрой сортировки построен на основе идеи разбиения массива на разделы. Общая процедура заключается в выборе пограничного значения, называемого *компарандом*, которое разбивает сортируемый массив на две части. Все элементы, значения которых больше пограничного значения, переносятся в один раздел, а все элементы с меньшими значениями - в другой. Затем это процесс повторяется для каждой из частей и так до тех пор, пока массив не будет отсортирован. Например, допустим, что необходимо отсортировать следующий массив: "fedacb". Если в качестве компарадера использовать "d", то после первого прохода алгоритм быстрой сортировки упорядочит массив следующим образом

Исходный массив	Fedacb
Проход 1	Bcadef

Затем этот подход повторяется для каждого раздела, а именно "bca" И "def".

Проход 2	Acb	Def
Проход 3	Abc	Def

Процесс рекурсивен по своей природе и наилучшие решения получаются при рекурсивном подходе. Пограничное значение можно выбирать двумя путями. Во-первых, его можно делать случайным образом, или путем осреднения небольшого набора значений, принадлежащих к разделу. Для того, чтобы сортировка была оптимальной, надо выбирать значение, расположенное точно в середине диапазона значений. Поскольку на практике это не всегда осуществимо, можно выбирать срединный элемент каждого из разделов.

Задание 2.

Создать параметризованный стек.

Вариант 6.
Задание 1 .

Создать класс типа сигнал, как шаблон, чтобы на его основе реализовать и двухбайтовые данные, собранные с платы сбора данных, так и данные типа float, смоделированные программно. С сигналом определить конструктор по умолчанию, конструктор с параметром, конструктор копирования, деструктор. Переопределить операторы присваивания, [], +=, -=, +, -, *, сохранения в файле.

Задание 2.

Создать параметризованную циклическую очередь.

Вариант 7.
Задание 1 .

Написать параметризованную функцию сортировки методом вставки.
В методе вставки на первом шаге выполняется сортировка первых двух элементов массива. Далее алгоритм ставит третий элемент в порядковую позицию, соответствующую его положению относительно первых двух элементов. Затем в этот список вставляется четвертый элемент и т.д. Процесс продолжается до тех пор, пока все элементы не будут отсортированы. Например, если бы строка "dcab" сортировалась методом вставки, то каждый из проходов давал бы результат:

Исходный массив	Dcab
Проход 1	Cdab
Проход 2	Acdb
Проход 3	Abcd

Задание 2.

Создать параметризованный связный список с двойными связями.

Вариант 8.
Задание 1 .

Создать шаблон функции прореживания: три параметра: откуда выбирать, куда выбирать и через сколько выбирать

Задание 2.

Создать параметризованный стек.

Вариант 9.
Задание 1 .

Написать функцию-шаблон, вычисляющую среднее значение в массиве.

Задание 2.

Создать параметризованный класс - матрица. Определены конструкторы, деструктор и перегружены операторы =, [].

Вариант 10.
Задание 1 .

Написать функцию-шаблон, вычисляющую максимальное значение в массиве.

Задание 2.

Создать параметризованный стек.

Вариант 11.
Задание 1 .

Написать функцию-шаблон, вычисляющую минимальное значение в массиве.

Задание 2

Создать параметризованный массив с конструкторами, деструктором и перегруженными операторами =, -.

Вариант 12.
Задание 1 .

Написать родовую функцию в виде функции-шаблон. Функция меняет местами два аргумента.

Задание 2.

Создать параметризованный массив с конструкторами, деструктором и перегруженными операторами + , =.

Вариант 13.
Задание 1 .

Написать параметризованную функцию - сортировка методом Шелла. Метод построен на основе метода вставки с минимизацией промежуточных шагов. Рассмотрим таблицу

Проход1	F	D	A	C	B	E
Проход1	C	B	A	F	D	E
Проход1	A	B	C	F	D	E
Результат	A	B	C	D	E	F

Сначала выполняется сортировка элементов, отстоящих друг от друга на три позиции. После этого сортируются элементы, отстоящие друг от друга на две позиции. Наконец, выполняется сортировка смежных элементов. Точная последовательность изменения приращений может изменяться. Единственным требованием остается равенство

последнего приращения 1. Например, хорошо себя зарекомендовала последовательность 9, 5, 3, 2, 1, которую и предлагается использовать.

Задание 2.

Создать параметризованный класс бинарного дерева. С методами - добавить элемент в дерево, прохождение по дереву в нисходящем и в восходящем порядке. Осуществить поиск по дереву.

Вариант 14.
Задание 1 .

Написать функцию-шаблон бинарного поиска. Если данные, по которым требуется провести поиск, отсортированы, то можно использовать бинарный поиск. Поэтому, первоначально надо написать функцию сортировки (или использовать ту функцию, которую написали в задании №5.1 по практическим занятиям). При использовании бинарного метода на первом шаге проверяется срединный элемент. Если он больше ключа поиска, то проверяется срединный элемент второй половины массива. Эта процедура повторяется до тех пор, пока не будет найдено совпадение, или до тех пор, пока больше не останется элементов, которые можно было бы проверять.

Задание 2.

Создать параметризованный класс односвязного списка.

Обработка исключительных ситуаций

Материал для выполнения лабораторной работы можно найти здесь

Для студентов, готовых ограничиться оценкой "удовлетворительно" за экзамен и курсовую работу, в данном пункте достаточно выполнить одно задание.

Задание 1 .

Написать программу, в которой перехватываются исключения типа int, char *. Сгенерировать исключительную ситуацию.

Задание 2.

Добавить к программе в задании 1 перехват любой исключительной ситуации.

Задание 3.

Создайте тип ошибка - error. Добавить к программе в задании 1 перехват исключительной ситуации типа созданного Вами.

Задание 4.

Создайте тип ошибок памяти и тип ошибок с файлами, наследуемые от error. Добавить к программе в задании 1 перехват исключительной ситуации ваших типов.

Задания на курсовую работу

Три темы приписаны Майорову, Паутову и Семенову. Остальные могут быть выбраны произвольно. В некоторых указана максимально возможная оценка.

1. Программа должна шифровать файлы различными методами: шифры замены, шифры перестановок, шифры битовых манипуляций.
2. Программа рисует двумерный график различными способами: точками, линиями, точками и линиями, гистограммой, с сеткой, вертикальной и горизонтальной.
3. Создать элемент управления ActiveX – рисование 3D графика различными способами. (Паутов И.)
4. Создать COM-объект рисование двумерного графика различными способами, точками, линиями, гистограммой, с сеткой вертикальной и горизонтальной..
5. Создать граф проекта – течение данных. Узлы графа – объекты с именем машины, алгоритмами обработки, параметрами обработки и т.д. В граф узлы можно вставлять, удалять, изменять течение данных. (Майоров).
6. Модифицировать курсовую работу по ЦОС “Моделирование сигналов” в полноценную объектно-ориентированную программу. Разработать проект, с классом моделей, для которых определены операции суммирования, вычитания и т.д., с классом сигнал, который содержит несколько моделей и т.п. Необходимо добавить “помощь”. (Семенов Г.)
7. Создать дерево разбора выражений (двоичное дерево). (Оценка зависит от того, самостоятельно ли написан данный класс, если что-то использовалось автоматически “хорошо” или “удовлетворительно”, если нет, то оценка может быть и “отлично”).
8. Создать шаблон класса массив с методом сортировки, отсортировать свой класс “Адрес” по его полям город, улица, индекс. (максимальная оценка “удовлетворительно”).
9. Создать шаблон циклической очереди. С помощью него обработать ввод с клавиатуры, заполнение информацией вашей памяти. (максимальная оценка “удовлетворительно”).
10. Создать шаблон приоритетной очереди. При добавлении элемента в такую очередь порядковый номер нового элемента определяется его приоритетом. (максимальная оценка “удовлетворительно”).
11. С помощью шаблона класса стек (разработать самим) написать программу калькулятор с операциями сложения, вычитания, деления, умножения, возведения в степень. (максимальная оценка “удовлетворительно”).
12. Создать шаблон связный список с двойными ссылками, в котором просмотр списка в любом направлении, поиск в списке конкретного элемента, удаление элемента из списка, ввод нового элемента в список, получение указателя на начало и конец списка, ввод нового элемента в список. (Оценка зависит от того, самостоятельно ли написан данный класс, если что-то использовалось автоматически “удовлетворительно”, если нет, то оценка может быть и “хорошо” и “отлично”).
13. Создать шаблон бинарное дерево, в котором можно проходить по дереву последовательно, нисходяще и восходяще. Добавлять элементы в дерево, поиск по дереву, удалять элементы из дерева.

Последовательный	Abcdefg
Нисходящий	Dbacfeg
Восходящий	Acegbfd

14. Базовый класс будет определять природу объектов, формирующих дерево, а от него наследовать класс, определяющий операции на деревом.
15. (Оценка зависит от того, самостоятельно ли написан данный класс, если что-то использовалось автоматически “удовлетворительно”, если нет, то оценка может быть и “хорошо” и “отлично”).
16. Построить мультивариантное дерево. Остальное аналогично предыдущей теме.
17. Разработать систему сбора данных для мониторинга погоды. Система состоит из датчиков скорости и направления ветра, температуры, давления, влажности воздуха. Система должна снимать показания каждые 5 сек. Выводить результаты на экран, сохранять в файле, выводить данные за последние 24 часа, усредненные по каждому часу, вычислять относительное изменение температуры, давления и влажности, а также скорости ветра.
18. Написать игру - пасьянс - косынка.
19. Написать карточную игру - в "дурака".