Введение. Предпосылки появления объектно-ориентированного программирования

ООП — это технология, возникающая как реакция на очередную фазу кризиса программного обеспечения, когда методы структурного программирования уже не позволяли справляться с растущей сложностью промышленного программного продукта. Следствия — срыв сроков проектов, перерасход бюджета, урезанная фукциональность и множество ошибок. Последствия ошибок — от материального урона до угрозы жизни людей.

Существенная черта промышленной программы — ее *сложность*: один разработчик не в состоянии охватить все аспекты системы, поэтому в создании участвует целый коллектив. Таким образом, к первичной сложности самой задачи добавляется управление процессом разработки с учетом необходимости координации действий в команде разработчиков.

Сложные системы разрабатываются в расчёте на длительную эксплуатацию, поэтому появляются ещё две проблемы: *сопровождение* системы (настройка параметров, устранение обнаруженных ошибок) и её *модификация*. Затраты на сопровождение и модификацию, как правило, сопоставимы с затратами на разработку системы.

Способ управления сложными системами известен с древности — *divide et impera* (разделяй и властвуй). Решение проблемы — в *декомпозиции* системы на всё меньшие и меньшие подсистемы, каждую из которых можно совершенствовать независимо. Структурный подход, характерный для C, так же использует этот принцип, в его рамках декомпозиция понимается, как разбиение алгоритма на модули, модуль выполняет один из этапов общего процесса.

ООП — другой подход, его суть в том, что в качестве декомпозиции понимается разделение элементов задачи по различным *абстракциям проблемной области*. Абстракции возникают в голове программиста, который анализирует проблемную область и вычленяет из неё отдельные объекты. Для каждого объекта определяются свойства, существенные для решения поставленной задачи. Каждому реальному объекту ставится в соответствие программный объект.

Пример. Человек, машина, дорога.

Критерии качества декомпозиции проекта

Сложность приложения не контролируется, определяется целью создания программы. *Сложность реализации* можно попытаться оптимизировать.

Первый вопрос при декомпозиции — на какие компоненты (модули, функции, объекты, классы) нужно разбить программу? С ростом числа компоненитов сложность программы растет, возникает необходимость в кооперации, координации, коммуникации между ними. Особенно опасно неоправданное разбиение на компоненты, когда разделяются действия, тесно связанные между собой.

Второй вопрос — организация *взаимодействия* между компонентами. Взаимодействие упрощается, когда каждый компонент рассматривается как черный ящик, внутреннее устройство, которого неизвестно, но известны выполняемые им функции, так же его входы и выходы. Вход компонента позволяет ввести в него значения некоторой входной переменной, а выход — получить в качестве ответа значение некоторой выходной. В программировании совокупность таких входов и выходов определяет *интерфейс* компонента. Общепринятая в программировании терминология компонент, которому понадобились услуги другого компонента, называет *клиентом*, второй компонент — *сервером*. (В структурном подходе функция–клиент пользуется услугами функции-сервера путём её вызова.) Интерфейс реализуется как набор запросов или некоторых функций, вызывая которые клиент либо получает какую–то информацию, либо меняет состояние компонента.

Основные показатели для учета качества программного проекта:

Сцепление (cohesion) внутри компонента — показатель, характеризующий степень взаимосвязи отдельных его частей. Если внутри компонента решаются две подзадачи, которые легко можно разделить, то компонент обладает слабым (плохим) сцеплением.

Связанность (coupling) между компонентами — показатель, опирающийся на интерфейс между клиентом и сервером. Общее число входов и выходов сервера есть мера связанности. Чем меньше связанность между двумя компонентами, тем проще понять и отслеживать в будущем их взаимодействие.

Описанные показатели имеют относительный характер. Сильное сцепление может привести к излишнему дроблению проекта на большое число мелких компонент.

На основании этих критериев, можно заметить, что при структурном подходе к программированию особенно трудно достичь слабой связанности между компонентами. Интерфейс в этом случае реализуется либо через глобальные переменные, либо через некоторые параметры. В сложной программной системе нельзя обойтись без глобальных структур данных. Значит любая функция в случае ошибки может испортить эти данные. Как правило эти ошибки очень трудно найти. Кроме того, программист постоянно

должен помнить десятки, а скорее сотни, обращений к общим данным из разных частей проекта. Соответственно модификация потребует очень большой работы, так как каждый раз надо проверять все части проекта.

Ещё одна проблема — общее глобальное простанство имён. Все программисты, участвующие в проекте должны согласовывать имена для своих функций, чтобы не было пересечений.

В результате борьбы с этими недостатками были выработаны три новые концепции программирования:
ООП
унифицированный язык моделирования (UML)
специализированные среды разработки программного обеспечения
все три пункта можно назвать частями объектно–ориентированного проектирования.

Погружение в мир ООП

Любой язык программирования основан на абстрактных представлениях. Сложность решаемых задач зависит от того, что представляет данная абстракция и её качества. Можно выделить в области программирования два подхода к решению поставленной задачи: первый отталкивается от модели средства решения задачи (компьютера), пример — язык ассемблер, следующий уровень — Fortran, BASIC, C и др. Они совершеннее, но программисту приходится мыслить в контексте структуры компьютера, а не структуры решаемой задачи. Приходится подгонять модель задачи под машинную модель, это требует больших усилий, чужеродно по отношению к языку программирования. Возникла большая отрасль — методология программирования. Другой недостаток подхода — малопригодность для отражения реального мира. Реальные объекты нельзя отнести ни к функциям ни к данным, они являются совокупностью свойств и поведения.

Второй подход — моделирование решаемой задачи. Первые языки LISP и APL опирались каждый на своё особое представление мира — любая задача сводится к обработке списков или любая задача алгоритмизируется. Эти и другие подобные языки программирования (PROLOG) хорошо подходили только к конкретному узкому кругу задач.

ООПодход стал так популярен, потому что не ограничивается узкой специализацией (позволил преодолеть эту проблему). Компоненты пространства задачи и пространства решений называются *объектами*. Идея в том, чтобы программа адаптировалась к формулировке задачи за счёт добавления новых типов объектов.

Основополагающая идея ООП — объединение данных и действий производимых этими данными в единое целое, называемое объектом. Поэтому в ООП в отличие от структурного программирования введено понятие *класса*. Класс — это тип данных, определяемый программистом, в котором объединяются структуры данных и функции их обработки. Первый объектно-ориентированный язык — Simula-67. Именно в нём появилось основополагающее ключесое слово — class. Этот имитационный язык создавался для решения знаменитой «задачи кассира». В этой задаче идёт речь о работе и взаимодействии кассиров, клиентов, счетов, операций, денежных средств, т.е. множества объектов. Объекты, обладающие общими параметрами, но различающиеся их значениями объединялись в классы (все кассиры выполняют одинаковые операции, но обладают разными именим, все счета различаются только количеством денег на счёте). Создание абстрактных типов данных (классов) является одной из основополагающих концепций ООП. Абстрактные типы данных работают почти также. как и встроенные: программист по ходу программы определяет переменные этого типа. Конкретные переменные типа данных «класс» называются *экземплярами класса* или *объектами*. После определения класса можно создать сколько угодно объектов этого класса и работать с ними, как с элементами в пространстве решения задачи. Работа с объектами осуществляется при помощи *сообщений* или *запросов*. Программист отправляет запрос, а объект сам решает как этот запрос будет обработан.

Фактически программист создаёт новые типы данных. как бы создавая новый язык программирования, свой для каждой задачи. Встроенные типы данных могут быть рассмотрены как классы, обладающие набором характеристик и поведением. Система программирования принимает новые классы, обеспечивает всю обработку наравне со встроенными.

Программы, разрабатываемые на основе концепций ООП реализуют алгоритмы, описывающие взаимодействие между объектами. Одна из трудностей ООП заключается в создании взаимно–однозначного соответствия между элементами пространства задачи и объектами в пространстве решений.

Чтобы объект сделал что-то полезное необходимо направить к нему запрос. В типе с каждым возможным запросом ассоциируется некоторая функция, которая вызывается при получении запроса. Интерфейс объекта определяет какие запросы им обрабатываются. Где-то внутри объекта находится программный код, который эти запросы обрабатывает. Этот код вместе со скрытыми данными образует *реализацию*. Программист обращается с запросом к объекту, а объект решает какой фрагмент кода выполнить.

Класс содержит константы и переменные, которые называют *данными* или *полями* и выполняемые над ними операции и функции. Функции класса называются *методами* или *функциями–членами*. Поля и методы являются элементами или членами класса.

В любых отношениях важно установить границы, которые соблюдаются всеми участниками. Если все члены класса будут доступны всем желающим, то с ними можно сделать что угодно и обеспечить соблюдение правил невозможно. Для удобства разделим программистов на создателей библиотек классов и прикладных программистов, использующих эти библиотеки для разработки своих приложений. Создатель класса стремится построить класс, который предоставляет прикладному прграммисту доступ лишь к тому, что необходимо для его работы и скрывает всё остальное. При этом создатель класса может изменять скрытые аспекты класса, не заботясь как это отразится на других. Скрывается то, что может быть испорчено прикладным программистом по неосторожности или из-за нехватки информации. Таким образом, уменьшается количество ошибок в программе. Скрытие реализации — одна из важнейших концепций ООП. Ограничение доступа помогает прикладному программисту выделить то, что действительно существенно для его работы.

Предполагается, что доступ к полям класса возможен только через вызов соответствующих методов. Такой подход позволяет ослабить связанность между компонентами, это — инкапсуляция. *Инкапсуляция* — это ограничение доступа к данным и объединение их с методами, обрабатывающими эти данные. Доступ отдельным частям классарегулируется с помощью разделения членов класса на три группы: открытая часть (public), закрытая часть (private), защищенная часть (protected).

Методы, расположенные в открытой части образуют *интерфейс* класса и могут свободно вызываться через любой экземпляр класса, т.е. объект, относящийся к этому классу. Доступ к закрытой части возможен только из методов самого класса, к защищенной могут обращаться свои собственные методы, а также методы классов–потомков.

Инкапсуляция предотвращает непреднамеренный ошибочный доступ к полям объекта. Можно изменять внутреннюю реализацию методов, не затрагивая интерфейс, при этом программа–клиент изменяться не будет.

Другой положительный момент в таком сокрытии информации, то что в разных классах похожие по сути методы могут иметь одинаковые имена, не надо тратить время на согласование имен функций, структур данных и т.п.

Созданный и протестированный класс должен в идеале быть достаточно самостоятельным, чтобы использоваться в других программах, для этого требуется опыт и интуиция. Возможность повторного использования кода — ещё одно преимущество ООП. Простейший случай — создание объектов имеющегося класса. Другой способ — включение объекта имеющегося класса в новый класс. Методика создания нового класса из старых называется *композицией*.

С ООП связаны ещё два инструмента, грамотное использование которых повышает качество проектов: наследование и полиморфизм.

Наследование — механизм получения нового класса из существующего. Производный класс (потомок, дочерний, субкласс) получается путём дополнения или изменения существующего класса (базового или родительского или суперкласса). При этом реализуется концепция повторного использования кода. Например, базовый тип геометрическая фигура определяет свойства, общие для всех фигур: размер, цвет, местоположение и т.п. С фигурами выполняются различные операции: выод, стирание, перемещение, закраска. Из этого типа производятся типы конкретных фигур: круг, квадрат, треугольник, обладающие дополнительными характеристиками и поведением.

Таким образом, иерархия типов становится основной моделью задачи. Эта модель совпадает с описанием задачи на концептуальном уровне и позволяет переходить к программированию без промежуточных моделей. Круг — частный случай геометрической фигуры и класс круг — производный от класс геометрическая фигура.

Дочерний класс содержит интерфейс родительского, все сообщения, которые принимал базовый класс будет принимать и производный, т. е. дочерний класс является частным случаем родительского.

Чтобы поведение производного класса отличалось от базового его модифицируют. Первый способ тривиален: добавление новых функций, не входящих в интерфейс родительского. Это, можно сказать, идеальное решение проблемы. Второй способ: переопределение, то есть изменение поведения, существующих функций базового класса.

Полиморфизм даёт возможность создавать множественные определения для операций и функций. При этом, какое именно определение будет использоваться зависти от контекста программы. Когда существующая операция, например + или =, наделяется возможностью совершать действия над операндами нового типа, говорят, что такая операция является перегруженной. Перегрузка — частный случай полиморфизма.

Алан Кей (Alan Kay), создатель первого успешного объектно–ориентированного языка SmallTalk, кратко сформулировал пять основных концепций чистого ООП:

Любая сущность (компонент задачи) является объектом. Объект можно рассматривать, как усовершенствованную переменную: он содержит данные, но ему можно направлять запросы на выполнение операций «с самим собой».

Программа представляет собой совокупность объектов, которые общаютя друг с другом посредством отправки сообщений. Чтобы обратиться с запросом к объекту, следует отправить ему сообщение. Сообщение может рассматриваться как запрос на вызов функции, принадлежащей конкретному объекту.

Каждый объект владеет собственным блоком памяти, состоящим из других объектов. Новые объекты создаются как оболочки для уже существующих объектов. Это позволяет усложнить программу, скрываясь за простотой отдельного объекта.

Каждый объект обладает типом. В терминологии ООП каждый объект является экземпляром класса, где термин «класс» является синонимом слова «тип».

Все объекты одного типа могут получать одни и те же сообщения. Например, объект «круг» является одновременно объектом типа «геометрическая фигура», следовательно круг заведомо принимает сообщения, предназначенные для объектов–фигур и программа для работы с геометрическими фигурами может автоматически работать с кругами. Эта особенность — одна из самых сильных сторон ООП.

Проектирование:

Какие объекты нужны для решения задачи? Физические объекты(автомобили, страны, элементы электрической схемы), элементы интерфейса (окна,меню, графические объекты — линии, круги, прямоугольники, клавиатура, мышь), структуры данных (массивы,стеки, списки, деревья), группы людей, хранилища данных (словари, описи).

Какими интерфейсами они должны обладать?

История C++

Язык C++ развился из языка C, автором которого в 1972 году стал Деннис Ритчи. Первоначально C стал известен, как язык разработки системы UNIX. К концу 70-х C развился в то, что теперь называют классическим C или C Кернигана и Ритчи. В 1989 году был принят стандарт C (ANSI C), т.к. появилось большое количество несовместимых версий.

Язык C++ — расширение C — был разработан в начале 80–х Бъёрном Страуструпом в Bell Laboratories. Самое важное, что C++ обеспечивает возможность ООП, хотя на нём можно программировать в стиле C или смешивать стили. С середины 90–х он стал доминирующим системо–образующим языком. Сейчас почти все операционные системы написаны на C или C++.

Основы разработки приложений

Рассмотрим основные процессы и инструменты для разработки приложений.

Процесс трансляции

Любой язык программирования транслируется (переводится) из формы, удобной для человеческого восприятия (исходного текста), в форму , воспринимаемую компьютером (машинный код). Традиционно принято делить трансляторы на интерпретаторы и компиляторы.

Интерпретатот транслирует исходный текст (часть) в несколько машинных команд и сразу их выполняет (BASIC — по строкам). Такие языки работают очень медленно, т.к. после трансляции очередной строки интерпретатор о ней «забывает» и в случае возвращения (например в цикле) должен переводить её заново. Особенно это важно в больших проектах, когда интерпретатор требует полностью загрузить в него текст программы. Некоторые новые интерпретаторы преодолевают эту проблему. Достоинством является тесная связь выдаваемых ошибок с текстом, при этом разработка ускоряется , ошибки исправляются интерактивно.

Компилятор транслирует исходный код на язык ассемблер или машинные команды. конечным результатом является файл с машинным кодом.Компиляция обычно состоит из нескольких этапов. Некоторые языки поддерживают независимую компиляцию отдельных компонентов программы (C). Затем части соединяются с помощью специальной программы компоновщика. Это — раздельная компиляция. Раздельная клмпиляция позволяет решать много проблем связанных с размером больших пректов. Строить и тестировать программу можно частями. Наборы протестированных, заведомо работоспособных компонентов объединяются в библиотеки, используемые другими программистами. Раньше в случае использования компиляторов была сложна отладка программ, сейчас разработаны отладчики, решающие эту проблему.

В языке C++ (как и в C) компиляция начинается с обработки исходного текста препроцессором. Директивы препроцессора уменьшают объём вводимого текста и делают программу более наглядной, но могут служить источником нетривиальных ошибок, поэтому C++ разработан так, чтобы свести применение препроцессора к минимуму. Далее компилятор проходит по тексту дважды. На первом проходе осуществляется лексический разбор программы. Компилятор разбивает исходный текст на составляющие и организует их в структуру дерево, каждая составляющая — узел дерева. На втором проходе анализируется дерево и генерируется машинный код. Получается объектный файл.

Компоновщик объединяет объектные модули в исполняемую программу. Если функции одного объектного модуля ссылаются на функции других объектных модулей, компоновщик производит поиск по специальным файлам — библиотекам (набор объектных модулей, объединённых в один файл).

Объявления и определения

Раздельная компиляция важна в крупных проектах. Создаваемая программа состоит из нескольких файлов, функции из одного файла обращаются к функциям из других. Компилятор должен обладать информацией об этих функциях, знать их имена и способ правильного использования. Для этого используют *объявления*.

Объявление просто сообщает компилятору некоторое имя (функции или переменной), т.е. говорит, что эта функция где-то существует и выглядит так. *Описание* означает создание переменной или функции, т.е. выделение памяти под переменную или генерацию кода функции. Объявлений может быть много, описание — только одно.

Объявление функции:

```
int f1(int,int);
int f1(int x,int y);
```

Во втором случае компилятор имена переменных проигнорирует.

Определение функции:

```
int f1(int x,int y){...};
```

Определение переменной

```
int a;
```

выглядит как и объявление, поэтому было введено ключевое слово `extern`. Оно может указывать, что переменная определена в другом файле или в этом же, но в другом месте.

Определение может одновременно быть объявлением.

```
extern  int i;//объявление
double f (double); //объявление
double b; //объявление и определение
double f(double x){    //определение
return x+1.0;
}
int i; //определение
int h(int k){   //объявление и определение
return k+2;
}

int main() {

  b=2.0;
  i=2;
  f(b);
  h(i);
}
```

Библиотеки содержат большое количество функций и переменных. В C и C++ используется механизм заголовочных файлов. Заголовочным называется файл, внешние объявления элементов библиотеки. По общепринятой схеме ему присваивается имя с расширением .h. Создатель библиотеки предоставляет заголовочный файл. Пользователь включает в программу заголовочный файл с помощью директивы

include. Директива указывает препроцессору, что нужно открыть соответствующий файл и вставить его содержимое на место директивы. Имена файлов задаются двумя способами: 1) в угловых скобках, тогда препроцессор будет искать файл в специально отведённой директории, где зависит от конкретной реализации языка и операционной системы; 2) в кавычках, тогда файл должен находиться в текущем каталоге. Помере развития C++ производители компиляторов выбирали разные расширения для заголовочных файлов, кроме того устанавливались ограничения на длину имени. Это создавало проблемы с переносимостью исходных текстов. Чтобы решить возникающие проблемы в стандарте была использована схема, когда длина имени не ограничивается восемью символами, а расширение не указывается. (include <iostream>) Библиотеки, унаследованные от C, сохранили расширение .h, но их можно включать и без расширения, если снабдить префиксом c (<cstdio>).

Последняя стадия компиляции — это компоновка. Программа компоновщик собирает объектные модули, сгенерированные компилятором в исполняемую программу. В большинстве пакетов C++ компановщик запускается незаметно для программиста при помощи компилятора. Как работает компоновщик, обнаружив внешнюю ссылку на функцию или переменную, он выясняет, если определение уже известно, то компоновщик ссылку разрешает. Если определение ранее не встречало, то компоновщик включает его в список нерарешённых ссылок. Затем ищет его сначала среди объектных модулей пректа, затем в библиотеках. В библиотеках для оптимизации поиска поддердиваются средства индексирования, так что нужно просматривать только индексы. Если определение находится, то компоновщик подсоединяет к исполняемой программе отдельный модуль. Если строить свою библиотеку, необходимо позаботиться, чтобы выделить каждую функцию в отдельный объектный файл.

Элементы языка C++

Функции

Любая программа состоит из функций. В отличие от C в C++ в определении функции могут содержаться аргументы без указания имён. Такие аргументы в теле функции не могут использоваться, при вызове функции они должны присутствовать. Разработчик функции может задействовать их в будущем, не изменяя прикладных программ. Пустой список аргументов определяется двумя способами: func() или func(void). Прототип функции должен явно определять её тип (в C по умолчанию int), примеры:

int f1(void); int f1(); float f2(float, int, char, double);

При программировании доступны библиотеки функций, прежде, чем определять свою функцию следует поискать аналогичную в библиотеке, скорее всего это решение лучше спроектировано и отлажено. Многие компиляторы содержат кроме стандартной ещё и дополнительные библиотеки, их можно использовать только в том случае, если Вы уверены, что не будете переносить свой проект на другую платформу. Если без этого не обойтись в идеале необходимо расположить соответствующий код в одном месте, например, инкапсулировать его в одном классе, тогда при переносе переделка будет минимальной.

Можно разрабатывать собственные библиотеки, в любой среде для этого есть специальные средства. Общая идея такая: создаётся заголовочный файл с прототипами всех функций и помещается в каталог (обычно либо стандартный либо текущий), затем собираются объектные модули и передаются библиотекарю, сформированная библиотека должна быть размещена так, чтобы компановщик мог её найти.

Типы данных.

Четыре основных встроенных типа: int, float, double,char. В стандарте C++ есть тип bool, имеющий два состояния, выраженных встроенными константами true (целое число 1) и false (0). Элементы && || <><=>= == != , операторы выбора адаптированы. В следствие традиции компилятор производит автоматическое приведение типа int к типу bool (ненулевое значение true, нулевое false). Спецификаторы long и short позволяют определить ещё несколько типов:

long double
double
float
long (long int)
int
short (short int)
char

Переменные.

Синтаксис определения переменных не изменился по сравнению с C. Область видимости (где переменная создаётся, существует, уничтожается) продолжается от места определения до ближай шей закры-

вающей фигурной скобки (фактически определяется парой фигурных скобок — блоком, внутри которых расположено определение переменной). Язык C++ позволяет определить переменную в любой точке блока (в C можно только в начале блока). Удобнее всего определять переменные непосредственно перед началом их использования и совмещать определение и инициализацию (for(int i=0;i<N;i++)). В точке определения переменные должны инициализироваться.

Глобальные переменные определяются вне тел функций и доступны всем, в том числе и во внешних файлах. Чтобы глобальная переменная из одного файла была доступна в другом, то в последнем должно быть её объявление со словом extern.

Локальные переменные доступны только в своём блоке, их иногда называют автоматическими, т. к. они автоматически уничтожаются при выходе из области видимости.

Обычно переменные локальные для некоторой функции исчезают при её завершении, а при повторном вызове создаются заново и их значение теряется. Если значение переменной должно сохраняться на протяжении всей жизни программы, то можно объявить локальную переменную со словом static и присвоить ей начальное значение, если переменная изменится, то это изменеие сохранится между вызовами функции. Глобальные переменные не используются в силу доступности.

```
void f(){
static int i=0;
cout<<"i="<<++i<<endl;
}

void main(){
for(int j=0;j<10;j++)
  f();
}
```

Если слово static применяется к глобальной переменной, она не будет доступна в других файлах.

Константы в C++ определяются с помощью ключевого слова const и отличаются от переменных только запретом изменения значения. В остальном компилятор выполняет все проверки, например типа, как и у переменных, константа обладает видимостью и т.п. Кроме того, компилятор выполняет дополнительную оптимизацию кода, которая основывается на информации о неизменности этого значения, при обращении к константе.

Наоборот, ключевое слово volatile (волэтэйл) означает, что значение переменной может измениться в любой момент и выполнять оптимизацию при обращениях нельзя, это необходимо при работе с потоками.

Преобразование типов.

Компилятор может неявно приводить один тип данных к другому. Арифметические операции выполняются следующим образом: операнд с более низким типом, в смысле указанной иерархии, будет приведён к более высокому. Например:

```
int i;
float x;
double rez;
rez=i*x;
```

Явное преобразование типов осуществляется самим программистом, обычно, когда компилятор не может сделать это автоматически. Явное приведение типов должно быть сведено к минимуму, т.к. при этом отключается автоматическая проверка безопасности и соответствия типов. Для того, чтобы приведение типов бросалось в глаза в тесте программы, синтаксис C++ отличается от C.

Операция `static_cast`. Используется для так называемого «безопасного» чётко определённого приведения типов. Например:

```
void func(){};

void main () {
int i=1;
long l;
float f;

l=i;// неявное преобразование типов
f=i;// без потери информации
```

```
// можно так
l=static_cast<long>(i);
f=static_cast<float>(i);

// преобразование с потерей информации
i=l;// вызовет
i=f;// предупреждение
// подавляет предупреждение, означает <<я понимаю, что делаю>>
i=static_cast<int>(f);
i=static_cast<int>(i);

void *u;
float *v=static_cast<float*>(u);

double d;
func(static_cast<int>(d));
}
```

Операция `const_cast` позволяет лишить переменную статуса const или volatile. Это приведение типов запрещается выполнять вместе с другими подобными операциями.

```
const int i=0;
int *j;
j=const_cast<int*>(&i);
long *l=const_cast<long*>(&i); // Ошибка!!!
```

При получении адреса константного объекта результат представляет собой указатель на const, который нельзя присвоить указателю на неконстантный объект без приведения типов.

Указатели.

Все элементы программы находятся в том или ином месте памяти. Каждая ячейка памяти однозначно определяется её адресом. Во время работы программа находится в памяти, поэтому каждый элемент обладает собственным адресом. В C и C++ имеется операция `&`, предназначенная для получения адреса элемента. Пример: `int a;cout<<&a;`. Адрес переменной можно сохранить в другой переменной для последующего использования. В C++ существует специальный тип переменных для хранения адресов — указатели. Операци определения указателя `*`. Пример: `int* u;`. Согласно рекомедациям, все переменные должны инициализироваться, поэтому:

```
int a=47;
int* u=&a;
```

Чтобы обратиться к значению через указатель следует разыменовать указатель операцией `*`.

```
*u=100;
```

Допускается сложение и вычитание указателя и целого числа.

Адрес, который помещается в указатель, должен быть одного с ним типа. Нельзя присвоить указателю на int указатель на float. Но есть исключение. Существует тип указателя, который может указывать на любой тип данных. Это `void*`.

```
int *ptrint;
float* ptrfloat;
void *ptrvoid;

ptrfloat=ptrint;//нельзя!!!
ptrvoid=ptrint;//можно
```

После такого присвоения вся информация о типе теряется. В дальнейшем для использования такого указателя необходимо приведение типа, при котором могут возникнуть ошибки, например, неоправданно изменится объём памяти, занимаемой переменной, если вместо int мы приведём указатель к типу double. Поэтому использование пустых указателей не рекомендуется за исключением некоторых случаев.

Основные области применения указателей:

указатели могут использоваться для изменения «внешних объектов» внутри функций;

указатели используются для работы с нетривиальными структурами данных;

в некоторых приёмах программирования.

Модификация внешних объектов.

Функции, получающие аргументы при вызове обычно создают внутренние копии этих аргументов. Этот механизм называется передачей данных по значению.

```
#include <iostream>
using namespace std;

void func(int a){
   cout<<"a="<<a<<"\n";
   a=3;
   cout<<"a="<<a<<"\n";
};

void main () {
   int x=54;
   cout<<"x="<<x<<"\n";
   f(x);
   cout<<"x="<<x<<"\n";
}
```

Переменная `a` является локальной для функции `f()`, то есть для неё выделяется память при вызове функции и уничтожается при её завершении. При вызове функции значение переменной `x`, равное `54` копируется в `a`. Результат:

```
x=54
a=54
a=3
x=54
```

Вне функции `f()` переменная `x` является внешним объектом и изменение локальной переменной на ней не отражается.

Если необходимо изменить внешний объект, вместо обычного объекта передаётся указатель на него:

```
#include <iostream>
using namespace std;

void func(int *p){
   cout<<"p="<<p<<"\n";
   cout<<"*p="<<*p<<"\n";
   *p=3;
   cout<<"p="<<p<<"\n";
};

void main () {
   int x=54;
   cout<<"x="<<x<<"\n";
   cout<<"&x="<<&x<<"\n";
   f(&x);
   cout<<"x="<<x<<"\n";
}
```

Результат:

```
x=54
&x=0065FE00
p=0065FE00
*p=54
p=0065FE00
x=3
```

Ссылки.

В C++ появился новый способ передачи адресов функциям. Это механизм передачи по ссылке. Ссылка определяется операцией `& int &a`. Хранящийся в ссылке адрес можно получить только с помощью `&`. Общий принцип их работы напоминает механизм работы указателей. Отличие заключается в синтаксисе, который выглядит более удобно.

```
#include <iostream>
using namespace std;

void func(int &r){
   cout<<"r="<<r<<"\n";
   cout<<"&r="<<&r<<"\n";
   r=3;
   cout<<"r="<<r<<"\n";
};

void main () {
   int x=54;
   cout<<"x="<<x<<"\n";
   cout<<"&x="<<&x<<"\n";
   f(x);// выглядит как передача по значению, но передаётся аргумент по ссылке
   cout<<"x="<<x<<"\n";
}
```

Результат:

```
x=54
&x=0065FE00
r=54
&r=0065FE00
r=3
x=3
```

В списке аргументов вместо указателя (int*) передаётся ссылка (int&). Внутри функции достаточно использовать `r`, чтобы получить значение переменной на которую она ссылается. Новое значение, присваиваемое `r` в действительности присваивается переменной `x`, на которую она ссылается.

Операции.

Все операции в языке C++ можно рассматривать как особую разновидность функций. При перегрузке они именно так и интерпретируются. Операция получает один или несколько аргументов и генерирует новое значение. Синтаксис не похож на функции, но суть та же. Операции арифметические +,-,/,*, ++,–, присваивания, отношения <=,>=,<,>,==.!=, логические //, &&, получение адреса &, разименование ->, *, операция приведения типов.

Управляющие команды.

Все управляющие команды языка C поддерживаются (if–else while do–while for switch break continue). Отличие в том, что результат логического выражения логическая величина true или false.

Класс

Определение класса.

Класс=данные + функции для работы с ними.

Описание класса вводит новый тип и содержит поля (члены–данные), определяющие состояние объекта и методы (функции–члены), которые обуславливают поведение объекта класса.

Класс можно определить с помощью конструкции:

```
тип_класса имя_класса {компоненты класса};
```

Точка с запятой обязательна!!! Синтаксис методов совпадает с синтаксисом функций.

- тип класса — одно из служебных слов class struct union;
- имя класса — идентификатор;
- компоненты класса — определения и объявления данных и принадлежащих классу функций.

Можно определить класс:

- с полями и методами;
- только с полями;
- только с методами;
- без полей и методов (пустой класс).

Имея определение класса, можно создавать объекты (экземпляры класса), указатели и массивы.

```
имя_класса имя_объекта // объект

имя_класса *имя_объекта // указатель

имя_класса имя_объекта[количество] // массив объектов
```

Пример:

```
class TTochka {
private:
  double x,y;
public:
  TTochka(){x=0; y=0;}
  TTochka(double _x,double _y){x=_x; y=_y;};
  double GetX() const {return x;}
  double GetY() const {return y;}
  void SetX(double _x){x=_x;}
  void SetY(double _y){y=_y;}
  void PrintTochka(){cout << "(" << x << "," << y << ")\n"; };
};
```

Объекты класса разрешается определять в качестве поля другого класса. Это называется композицией. Их можно передавать в качестве аргументов любыми способами (по значению, по ссылке, по указателю) и получать как результат из функции.

```
void f1(TTochka t)
void f2(TTochka &t)
void f3(TTochka *t)

TTochka p;
//Вызов функций
f1(p);
f2(p);
f2(&p);
};
```

Для каждого элемента класса определена область действия класс. Это означает, что при написании кода класса (внутри класса) к членам класса можно обращаться по имени. Вне класса, в программе–приложении, обратиться к членам класса можно только через объекты. Доступ к элементам класса осуществляется с помощью операции «.» для объектовили ссылок и «->» для указателей на объект.

```
class TTochka {
private:
  double x,y;
public:
  TTochka(){x=0; y=0;}
  TTochka(double _x,double _y){x=_x; y=_y;};
  double GetX() const {return x;}
  double GetY() const {return y;}
  void SetX(double _x){x=_x;}
  void SetY(double _y){y=_y;}
  void PrintTochka(){cout << "(" << x << "," << y << ")\n"; };
```

```
};
void main(){
 TTochka p,
         &ref=p,
         *ptr=&p;
 p.SetX(1);
 p.SetY(2);
 p.PrintTochka();
 ref.SetX(2);
 ref.SetY(3);
 ref.PrintTochka();
 ptr->SetX(5);
 ptr->SetY(6);
 ptr->PrintTochka();

}
```

Методы класса могут быть описаны (или определены) внутри класса. Это рекомендуется делать, если они короткие (умещаются в одну строчку), их называют встроенными. Если описание метода размещыется в другом месте (при хорошем стиле программирования — это соответствующий файл с расширением cpp), то в начале заголовка метода после указания типа возвращаемого значения следует поместить имя класса и операцию указания принадлежности к этому классу ::.

```
  void TTochka::PrintTochka(){cout << "(" << x << "," << y << ")\n"; };
```

Все методы класса могут узнать и использовать адрес своего родного (текущего) объекта. Этот адрес находится в указателе со стандартным названием this. Когда из внешнего приложения вызывается метод, то в указатель this записывается адрес того объекта, для которого вызван этот метод.

```
class Test{
private:
   int a;
public:
   void metod(void){
      cout<<this->a;  //то же, что и cout<<a;
   }
};
```

```
Указатель this используется для возврата значений из методов и перегружаемых
операций.
```

Доступ к членам класса.

Все элементы класса делятся на три группы по ограничению доступа к ним. Группы определяются с помощью ключевых слов private (по умолчанию), public, protected. После каждого спецификатора, режим доступа определённый им, действует до следующего спецификатора или до конца класса. Хорошим стилем программирования считается однократное использование каждого спецификатора внутри одного класса.

Основная задача закрытых элементов — защита информации от ошибочного изменения. Основная задача открытых элементов — дать клиентам класса представление о возможностях, которые обеспечивает класс. Все данные класса следует делать закрытыми, создавая открытые методы, позволяющие изменить значение данных (часто их названия включают слово Set) или просмотреть их значение (часто их названия включают слово Get). Функции, изменяющие данные могут содержать проверку вводимых значений или перевод данных из формы клиента во внутреннее представление данных класса. Функции, позволяющие просмотреть данные, не обязательно должны возвращать их во внутреннем виде и даже полностью, они могут ограничивать доступ к значениям полей. Не все методы класса входят в его интерфейс, часто необходимы внутренние служебные функции, обеспечивающие работу других членов класса, их тоже следует помещать в закрытую часть класса. Если метод класса не должен изменять поля класса хорошимстилем программирования считается употребление ключевого слова const:

```
  double GetX() const {return x;}
```

Вне класса доступны только члены public. В некоторых ситуациях желательно, чтобы функция, не являющаяся членом класса тем не менее имела доступ к закрытым членам этого класса. Язык C++

позволяет это сделать, если объявить функцию другом класса или объявить класс, членом которого она является дружественным. Это позволяет служебное слово friend. Если класс объявлен дружественным другому, то все его методы являются для того дружественными функциями. Дружественные функции имеют доступ ко всем членам этого класса независимо от спецификаторов private и public. Но чтобы внутри дружественной функции обратиться к соответствующему члену класса, необходимо создать объект этого класса.

```
// Предварительное объявление, необходимое для одного из классов
class MyClass1;
// Класс, использующий дружественную функцию и дружественный класс
class MyClass2{
private:
   int j;
public:
   MyClass2(){j=0;}
   int GetJ() const {return j;}
   friend void Vn_func(MyClass2 &x);
   friend MyClass1;
};
//Дружественный класс
class MyClass1 {
private:
   int i;
public:
   MyClass1(){i=0;}
   void Metod1(void){ MyClass2 a; a.j=i;}
   void Metod2(MyClass2 &x){i=x.j;}
   friend void main(void);
};

void Vn_func (MyClass2 &y){
 y.j++;
};

void main (void){
 MyClass1 ob1;
 cout <<ob1.i<<"\n";
 MyClass2 ob2;
 cout <<ob2.GetJ()<<"\n";
//функция членом класса не является
 Vn_func(ob2);
 cout <<ob2.GetJ()<<"\n";
 ob1.Metod2(ob2);
 cout <<ob1.i<<"\n";
}
```

Использовать дружественные классы и функции следует только в случае крайней необходимости.

Конструктор

В C++ при создании объекта (в момент его определения и только в этот момент) обязательно вызывается специальный метод инициализирующий объект. Это конструктор. Компилятор может его найти среди остальных методов, т.к. его имя совпадает с именем класса. Конструктор не имеет возвращаемого значения, но в отличие от остальных функции слово `void` перед ним не пишется. Конструктор может иметь аргументы, можно определять несколько конструкторов, отличающихся друг от друга набором и количеством аргументов. Это называется перегрузкой конструктора. Вызвать конструктор как обычный метод класса нельзя. Если конструктор не имеет параметров, то его называют конструктором по умолчанию. Определять конструктор по умолчанию следует практически всегда. Области применения этого конструктора:

- используется компилятором при создании массивов объектов, при создании каждого элемента массива вызывается конструктор по умолчанию, если он не определён массив создать нельзя;
- в случае наследования.

Если в классе нет явно определённых конструкторов, то будет автоматически создан конструктор по умолчанию.

```
class TTochka {
private:
  double x,y;
public:
  TTochka(){x=0; y=0;}
}
```

В хорошем стиле программирования нельзя инициализировать поля класса там, где они определяются. Инициализация всех полей должна происходить в конструкторе.

```
class TTochka {
private:
  double x,y;
public:
  TTochka(double _x,double _y){x=_x; y=_y;};
}
```

Существует альтернативный способ инициализации отдельных полей объекта в конструкторе — с помощью списка инициализаторов, расположенного между заголовком и телом конструктора, при этом сразу после заголовка ставится двоеточие. Каждыи инициализатор имеет вид:

имяполя (выражение, которым оно инициализируется)

Если инициализаторов несколько, они разделяются запятыми. Если полем является объект другого класса, для его инициализации будет вызван конструктор.

```
class TTochka {
private:
  double x,y;
public:
  TTochka(double _x,double _y):x(_x),y(_y){};
}
class TOtr {
private:
  TTochka a,b;
public:
  TOtr():a(0,0),b(1,1){}
}
```

В последнем случае без инициализатора не обойтись, т.к. в классе TTochka нет конструктора по умолчанию. Конструктор с инициализаторами работает быстрее, если объявлен объект TTochka p(1,2), а конструктор без инициализаторов, то полям сначала будут присвоены значения 0 (по стандарту C++), а уже следующим действием 1 и 2, при использовании инициализаторов эти значения будут присвоены сразу.

Одна из важнейших форм перегружаемого конструктора — конструктор копирования. Он вызывается в следующих случаях:

- при объявлении объекта, если он инициализируется другим объектом
- при передаче объекта в функцию по значению
- при возврате объекта из функции по значению.

Если в классе нет конструктора копирования, компилятор создаст его по умолчанию, просто побайтно копируя его поля в новый объект. Если полем является указатель, то при этом он будет ссылаться на ту же область памяти, что и скопированное поле другого объекта. При различных действиях с объектами можно вместо желаемого результата испортить другой объект, например, при выходе из области видимости объект будет уничтожен, а память освобождена, последствия очевидны.

Конструктор копирования имеет следующую форму:

имякласса (const имякласса & имяобъекта)

Операции которые его вызовут:

```
TTochka x=y;
f1(x);
y=f2();
```

Деструктор

Парным конструктору является другой метод, называемый деструктором, который автоматически вызывается перед уничтожением объекта. Именем деструктора является имя класса, перед которым поставлен символ тильда.

```
~TTochka(){}
```

В классе может быть только один деструктор. Так же как и у конструктора у него нет возвращаемого значения и не должно быть ключевого слова viod. Деструктор не имеет аргументов. Объект, как и локальная переменная, уничтожается, когда достигнут конец соответствующего блока, при этом автоматически вызывается деструктор. Если программист не создаст деструктор, компилятор создаст его сам, но если объект был создан с использованием динамического выделения памяти (операция new), то в деструкторе обязательно должна применяться операция освобождения памяти (delete).

```
class TMatrix {
private:
  // Размеры матрицы
  int rows, cols;
  // Хранилище данных
  double** a;
public:
  // Конструктор
  TMatrix(int _rows, int _cols);
  TMatrix(const TMatrix& m);
  // Деструктор
  ~TMatrix();
};

TMatrix::TMatrix(int _rows, int _cols) {
  rows = _rows;
  cols = _cols;
  a = new double*[rows];
  for(int i = 0; i < rows; ++i)
    a[i] = new double[cols];
}
// Конструктор копий
TMatrix::TMatrix(const TMatrix& m) {
  rows = m.rows;
  cols = m.cols;
  a = new double*[rows];
  for(int i = 0; i < rows; ++i) {
    a[i] = new double[cols];
    for(int j = 0; j < cols; ++j)
      a[i][j] = m.a[i][j];
  }
}
// Деструктор
TMatrix::~TMatrix() {
  for(int i = 0; i < rows; ++i)
    delete[] a[i];
  delete[] a;
}
```

В отличие от конструктора, деструктор можно вызывать как обычный метод класса. И в деструкторе и в конструкторе не следует использовать оператор return.

Перегрузка операций

Любая операция в C++ рассматривается как функция с аргументами и возвращаемым значением. Она может быть перегружена (переопределена) для объектов вашего класса. Такие функции называют операторными или функциями–операциями.

возвращаемыйтип operator знакоперации(список параметров)тело функции

Функция–операция может быть реализована как функция класса (при этом текущий объект класса становится левосторонним или единственным аргументом) или как внешняя дружественная функция (если операция унарная, то у неё будет один аргумент, если бинарная — два).

Как функция класса:

```
class TTochka {
private:
  double x,y;
public:
  TTochka(double _x,double _y):x(_x),y(_y){};
  TTochka operator +(TTochka& p){TTochka tmp(x+p.x, y+p.y);return tmp;}
//  return TTochka(x+p.x, y+p.y);
}
```

Как дружественная функция:

```
class TTochka {
private:
  double x,y;
public:
  TTochka(double _x,double _y):x(_x),y(_y){};
  friend TTochka operator +(TTochka& p1, TTochka& p2);
}
  TTochka operator +(TTochka& p1, TTochka& p2){
   return TTochka(p1.x+p2.x, p1.y+p2.y);
  };
```

В обоих случаях использование операции будет выглядеть так:

```
TTochka a1(1,2),a2(3,-4);
TTochka a3=a1+a2;
```

В первом случае будет вызван метод класса (a1.operator+(a2)), а во втором глобальная внешняя функция (operator+(a1,a2)). Заметим так же, что для инициализации a3 будет вызван конструктор копирования, т.к. в класса мы его не описали, то он будет создан по умолчанию. Если нет никаких препятствий, то используется переопределение операции как метода класса (препятствием может служить, например, то, что левый операнд операции другого типа (м.б. базового)), тогда используется внешняя функция.

Правила перегрузки операций:

- нельзя перегружать операции определения размера памяти (sizeof()), точка (или селектор члена объекта), условная операция (?:), указание области видимости (::), выбор компонента класса через указатель (.*), операции препроцессора (#);
- нельзя перегружать операции для встроенных типов данных;
- нельзя изменить приореитет прегружаемой операции и количество операндов. Унарная операция должна иметь один операнд (кроме инкремента и декремента), бинарная — два.
- операции присваивания (=), вызова функции (()), индексирования ([]), доступа по указателю (->) можно перегружать только как методы класса;
- если операция является методом класса, то текущий объект будет её левым операндом, в случае унарной операции единственным объектом будет объект для которого она вызывается. Таким образом, унарная операция не будет иметь аргументов, а бинарная только один (ссылку на некоторый внешний объект класса);
- при перегрузке арифметических операций перегружающие функции возвращают обычно объект, но могут возвращать и ссылку. Остальные бинарные операции возвращают ссылку на объект, таким образом можно создавать цепочки операций;
- если левый операнд перегружаемой бинарной операции относится не к пользовательскому классу, а к другому типу, например встроенному, то операторная функция не может быть членом класса, а должна быть внешней дружественной функцией, у которой первый аргумент соответствующий другой тип, а второй — объект пользовательского класса.

  ```
  class TTochka {
  private:
    double x,y;
  public:
    TTochka(double _x,double _y):x(_x),y(_y){};
  ```

```
  friend ostream& operator << (ostream& s, TTochka& t);
}

ostream& operator << (ostream& s, TTochka& t){
 cout << "(" << t.x << "," << t.y << ")\n";
 return s;
};
```

Операции инкремента и декремента имеют две формы записи: префиксную и постфиксную. Отличия в работе этих операций возникают, когда они используются в выражениях совместно с другими операциями. В первом случае вначале изменяется значение переменной, а затем это изменённое значение используется в выражении, во втором случае в выражении используется первоначальное значение переменной, а уже после оно изменяется.

Чтобы компилятор мог различить две этих формы операции, для них используются разные заголовки:
тип& operator ++();
тип operator ++(int);

Во втором случае дополнительный параметр int является фиктивний константой, внутри функции никак не используется, а нужен только для того, чтобы отличить эти функции.

```
TTochka& TTochka::operator ++(){
  x=x+1;
  y=y+1;
  return *this;
}
TTochka TTochka::operator ++(int){
  TTochka tmp=*this;
  x=x+1;
  y=y+1;
  return tmp;
}
```

В префиксной форме используется возврат результата по ссылке, таким образом предотвращается вызов конструктора копирования для создания возвращаемого значение и, соответственно, не требуется вызывать деструктор для уничтожения копии при завершении работы метода. В постфиксной операции это не подходит, т.к. необходимо вернуть первоначальное значение объекта, которое приходится сохранять в его копии. Таким образом префиксный инкремент работает быстрее, чем постфиксный. Для переменных встроенного типа (int, double и т.п.) это безразлично. Но если мы, например, в цикле, используем инкремент для увеличения значения некоторого объекта, следует пользоваться префиксной формой, как наиболее эффективной (for(TTochka i(0,0);...;++i)).

Если в классе не определена операция присваивания, то компилятор создаст её по умолчанию простым копированием всех полей объекта. В этом случае можно получить не то, что хочется, особенно, если среди полей есть указатели. Как правило, если требуется определить конструктор копирования, то необходимо определить и операцию присваивания.

Операция присваивания может быть определена только как метод класса. Правила по которым она определяется:

- обязательна проверка на самоприсваивание, т. к. иначе следующий шаг уничтожит объект, который мы копируем;
- если память выделяется динамически, то необходимо её удалить, а затем снова выделить под новые значения полей;
- скопировать новые значения полей в поля класса;
- возвратить объект, на который указывает this (*this).

```
class TMatrix {
private:
  int rows, cols;
  double** a;
public:
  TMatrix& TMatrix::operator=(const TMatrix& m);
};

TMatrix& TMatrix::operator=(const TMatrix& m) {
  if(this==&m) return *this;
```

```
  for(int i = 0; i < rows; ++i)
    delete[] a[i];
  delete[] a;
  rows = m.rows;
  cols = m.cols;
  a = new double*[rows];
  for(int i = 0; i < rows; ++i) {
    a[i] = new double[cols];
    for(int j = 0; j < cols; ++j)
      a[i][j] = m.a[i][j];
  }
  return *this;
}
```

Обработка исключительных ситуаций

Многие методы не могут иметь лишние параметры для сигнализации об ошибках или возвращать код ошибки. В C++ встроен механизм обработки исключений, с помощью которого можно сообщить программе–клиенту об ошибочной ситуации. При возникновении исключительной или ошибочной ситуации программа должна генерировать объект–исключение. Генерация такого объекта и создаёт исключительную ситуацию.

Генерируется исключение оператором throw:

throw выражение;

Выражение должно быть переменной или константой какого–нибудь типа или даже непосредственно выражением. Тип исключения может быть встроенным, а может задаваться самим программистом. Часто для этого используют пустые классы:

class ZeroDevide ;

Сгенерировать исключение можно так:

throw 1; //будет сгенерировано исключение типа int

throw "a"; //будет сгенерировано исключение типа char* или string

throw ZeroDevide(); //будет сгенерировано исключение класса ZeroDivide

В последнем случае для генерации исключения фактически в явном виде вызывается конструктор по умолчанию пустого класса — это наиболее распространённый способ генерации исключений.

Исключения надо перехватить и обработать. Перехват исключений осуществляется с помощью оператора tryконтролируемый блок, который выделяет контролируемый блок (часть текста программы, где может генерироваться исключение). После контролируемого блока обязательно должны следовать один или несколько блоков — обработчиков исключений, определяемых оператором catch(спецификация исключения)действия по обработке исключений. Спецификация исключений может иметь три формы:

catch(тип)обработка любого исключения заданного типа или класса

```
try{
   ...
   x.metod1();// в методе throw "test.txt";
   ...
}
catch(string){
cout<<"Ошибка в файле";
return;
}
```

catch(тип переменная)обработка исключения заданного типа или класса с использованием значения переменной

```
try{
   ...
   x.metod1();// в методе throw "test.txt";
   ...
}
catch(string f){
cout<<"Ошибка в файле "<<f;
return;
}
```

catch(...)обработка исключений всех типов

```
try{
   ...
   x.metod1();// в методе throw "test.txt";
   ...
}
catch(...){
cout<<"Ошибка";
return;
}
```

Общая схема обработки исключений состоит в следующем:

выделить контролируемый блок try

предусмотреть генерацию одного или нескольких исключений внутри контролируемого блока с помощью throw, исключения могут быть в вызываемых функциях или методах, а может быть в явном виде

разместить сразу после блока try один или несколько блоков catch (блок–ловушка) для обработки ошибочных ситуаций.

Блоки try могут быть вложенными.

Обработка исключений происходит в следующем порядке. Когда в контролируемом блоке try генерируется исключение, начинается сравнениетипа исключения и типов параметров в блоках–ловушках, выполняется только тот блок, где типы совпадут. Если такого блока нет, но есть catch(...), то выполняется этот блок. Если блок не найден в текущем списке обработчиков, то ищется другой список обработчиков следующего уровня вложенности, возможно расположенный в вызывающей функции. Этот процесс продолжается до функции main(). Если и там обработчик не найден, то вызывается стандартная функция завершения работы программы с сообщением об ошибке.

Если catch–блок найден, то выполняются операторы этого блока, тогда

если нет операторов перехода или новой генерации исключений(return, exit(), throw), то после завершающего оператора блока–ловушки осуществляется переход к операторам, расположенным после всей конструкции try–catch;

если встречен оператор return, осуществляется нормальный выход из функции;

в секции–ловушке можно расположить оператор throw без выражения, тогда он сгенерирует исключение того же самого типа, а обработчик ищется в блоках, расположенных выше по иерархии вложенности;

может быть сгенерировано новое исключение.

Если происходит выход из функции (при любой ситуации в обработке исключений), то происходит нормальное завершение её работы с уничтожением всех локальных объектов, не уничтожаются только динамические объекты, созданные операцией new, для них деструкторы вызываются только при явном использовании операции delete.

Пример использования исключений в конструкторе:

```
class Vect{
 private:
   int *p;
   int size;
 public:
   Vect(int n);
   ~Vect(){ delete[] p;}
   int& operator [] (int i);
}

Vect::Vect(int n){
   size=n;
   p=new int[size];
   if(!p)
      throw "Ошибка в конструкторе вектора";
   for(int i=0;i<size;i++)
      p[i]=0;
}

int& Vect::operator [] (int i){
    if(i<0||i>=size) throw i;
    return p[i];
```

```
}

void main(){
 try{
   Vect a(3);
   a[0]=0;
   a[1]=1;
   a[2]=2;
 }
catch(string s){
   cout <<s;
}
catch(int x){
   cout<<"нет элемента с индексом "<<x;
}
```

Ни одно исключение, которое могло бы появиться в деструкторе не должно покинуть его пределы.

Ошибки ввода лучше проверять старым способом и просить пользователя снова ввести данные.

Строки

В языке C нет специального строкового типа. Вместо этого используются массивы символов, завершаемые нулевым байтом. В C++ есть новый класс, называемый `string`, с помощью которого работа со строками символов становится намного удобнее.

Одним из важнейших удобств является то, что класс `string` берёт на себя управление памятью при первоначальном определении строки и при всех её модификациях, увеличивающих или уменьшающих длину строки. При этом изменение размера строки происходит динамически и независимо от программиста. Строки класса `string` защищены от ошибочных обращений к памяти, в отличие от строк в стиле C или C–строк.

Класс `string`, как стандартный библиотечный класс позволяет работать со строками, как с обычными типами данных, т.е. копировать, присваивать, складывать, сравнивать, используя привычный синтаксис соответствующих операций. Большое количество методов класса позволяют легко решать многочисленные задачи, связанные с обработкой текста.

Платой за удобство является некоторое понижение быстродействия по сравнению с C–строками, но в большинстве программ на C++ строки типа `string` обеспечивают необходимую скорость обработки, поэтому для работы с текстом их применение является более предпочтительным.

Чтобы использовать объекты класса `string`, необходимо подключить заголовочный файл `<string>`. Для понимания определений методов класса следует знать назначение некоторых имён из пространства `std`, например, идентификатор `size_type` является синонимом типа `unsigned int`.

В классе `string` определено несколько конструкторов:

- конструктор по умолчанию, инициализирующий объект пустой строкой (`string()`):
  `string s1;`
- конструктор, инициализирующий объект C–строкой (`string(const char* str)`):
  `string s2("Hello!");`
- конструктор, инициализирующий объект заданным количеством символов (или не более чем заданным количеством символов) из C–строки, символы отсчитываются от начала (`string(const char* str,size_type len)`):
  `string s3("Hello!",3);// в строку запишется последовательность \verb'Hel'`
- конструктор копирования (`string(const string &str)`):
  `string s4(s2);`
- конструктор, инициализирующий объект заданным количеством символов (или не более чем заданным количеством символов) из внешней строки, начиная с заданной позиции `stridx` (`string(const string &str, size_type stridx, size_type len)`):
  `string s5(s2,4,2);// в строку запишется последовательность 'o!'`
- конструктор, инициализирующий объект повторением заданного количества определённого символа (`string(size_type num, char c)`):
  `string s6(5,'t');// в строку запишется последовательность 'ttttt'`

Класс `string` содержит три операции присваивания:

- присвоение объекта класса `string` (`string& operator=(const string &str)`)
  `s2=s6;`
- присвоение значения C–строки (`string& operator=(const char *s)`)

```
s2="Привет!";
```

- присвоение символа (`string& operator=(char c)`)
  ```
  s2='A';
  ```

Для объектов класса `string` определены следующие операции:

Операция	Действие	Операция	Действие
=	Присваивание	>	Больше
+	Сливание (конкатенация)	>=	Больше или равно
==	Равенство	[]	Индексация
!=	Неравенство	«	Вывод в поток
<	Меньше	»	Ввод из потока
<=	Меньше или равно	+=	Добавление

Сравнение строк происходит в лексикографическом порядке.

В классе `string` есть два метода, позволяющих определить длину строки:

```
size_type size(void) const;
size_type lenght(void) const;
```

Методы эквивалентны и возвращают количество элементов строки. Можно узнать максимально возможную длину строки:

```
size_type max_size(void) const;
```

Следующий метод возвращает true,если строка является пустой и false в противном случае:

```
bool empty(void) const;
```

Операция индексации осуществляет доступ к отдельному символу строки, находящемуся на соответсвующей позиции. Начинается индексация с нуля. Но при использовании этой операции не проверяется выход за размеры строки, поэтому для доступа к отдельному элементу лучше использовать метод at() (s.at(i)), тогда, если индекс превысит длину строки будет сгенерировано исключение `out_of_rang`.

Кроме операции присваивания есть метод, позволяющий присвоить строке только часть другой, у него есть несколько вариантов использования, различающихся аргументами:

- `string& assign(const string &str, size_type pos, size_type n)`
  `s2.assign(s1,5,7);`
  строка s2 получит значение подстроки s1, начиная с пятой позиции , длиной семь символов, если второй аргумент при вызове метода выйдет за пределы строки s1. то будет сгенерировано исключение `out_of_rang`, если к выходу за размер строки приведёт третий аргумент, то произойдёт присвоение меньшего количества символов.
- `string& assign(const string &str, size_type n)`
  `s2.assign(s1,5);`
  строке s2 присваивается пять первых символов строки s1.
- `string& assign(const string &str)`
  `s2.assign(s1);`
  строке s2 присваивается строка s1.

Добавить к строке другую строку или её часть можно с помощью метода append:

- `string& append(const string &str, size_type pos, size_type n)`
  `s2.append(s1,3,10);`
  к строке s2 будет добавлено десять символов строки s1, начиная с третьей позиции, если второй аргумент при вызове метода выйдет за пределы строки s1. то будет сгенерировано исключение `out_of_rang`, если третий аргумент в вызове метода приведёт к выходу за размер строки s1, то будет добавлена вся строка до конца, если длина результата получится больше максимально допустимой длины строки, порождается исключение `length_error`;
- `string& append(const char *str, size_type n)`
  `s2.append(s1,3);`
  строке s2 добавляется три первых символов с–строки s1.
- `string& append(const string &str)`
  `s2.append(s1);`
  к строке s2 добавляется строка s1.

Чтобы вставить в строку другую строку или её часть используется метод insert:

- `string& insert(size_type pos1, const string &str, size_type pos2, size_type n)`
  `s2.insert(2,s1,5,7);`
  в строку s2 с позиции номер два будет вставлено семь символов строки s1, начиная с пятой позиции в строке s1;

- `string& insert(size_type pos1, const string &str)`
  `s2.insert(3,s1);`
  в строку s2 с позиции номер три будет вставлена строка s1;

Можно заменить часть строки другой строкой — replace:

- `string& replace(size_type pos1, size_type n1, const string &str, size_type pos2, size_type n2)`
  `s2.replace(2,3,s1,5,7);`
  в строке s2 три символа, начиная с позиции номер два, будут заменены на семь символов строки s1, начиная с пятой позиции в строке s1;
- `string& replace(size_type pos1, size_type n1, const string &str)`
  `s2.replace(3,4,s1);`
  в строке s2 с позиции номер три четыре символа будут заменены на строку s1;

Удалить часть строки можно, используя метод erase:

- `string& replace(size_type pos, size_type n)`
  `s2.erase(2,3);`
  в строке s2 три символа, начиная с позиции номер два, будут удалены;

Полностью очищает строку метод clear:

```
s2.clear();
```

Содержимое двух строк можно обменять — swap:

```
s2.swap(s1);
```

Есть метод, выделяющий часть строки в другую строку — substr:

```
string& substr(size_type pos, size_type n) const
s3=s2.substr(2,3);
```

В строку s3 будет скопировано три символа строки s2, начиная со второй позиции.

Большое разнообразие методов предусмотрено для поиска подстрок или отдельных символов. Каждый метод возвращает позицию найденной подстроки или -1, когда подстрока не найдена.

Рассмотрим подробно метод поиска самого левого вхождения заданной строки, подстроки или символа — find.

- `size_type find(const string &str, size_type pos) const`
  `s2.find(s1,5);`
  ищет самое левое вхождение строки s1 в строку s2, начиная с пятой позиции строки s2, если позиция не указана, то с начала строки s2;
- `size_type find(char c, size_type pos) const`
  `s2.find('a',5);`
  ищет самое левое вхождение символа `a` в строку s2, начиная с пятой позиции строки s2, если позиция не указана, то с начала строки s2;
  `size_type find(const char *s, size_type pos, size_type len) const`
  `s2.find(s4,2,7);`
  ищет самое левое вхождение подстроки с–строки длиной не более семи символов в строку s2, начиная со второй позиции строки s2, если позиция не указана, то с начала строки s2;

Перечислим другие методы поиска, их аргументы аналогичны аргументам метода find:

- `rfind()`
  ищет самое правое вхождение заданной строки, подстроки или символа в строку, для которой вызван метод;
- `find_first_of()`
  ищет самое левое вхождение любого из символов заданной строки в строку, для которой вызван метод;
- `find_last_of()`
  ищет самое правое вхождение любого из символов заданной строки в строку, для которой вызван метод;
- `find_first_not_of()`
  ищет самый левый символ строки, для которой вызван метод, не входящий в заданную строку;
- `find_last_not_of()`
  ищет самый правый символ строки, для которой вызван метод, не входящий в заданную строку;

В языке C++ поддерживается автоматическое преобразование типа `const char*` в строку, но не наоборот. Преобразование содержимого строки в массив символов или с–строку осуществляется тремя методами класса `string`:

- `c_str()` — возвращает содержимое строки в формате с–строки, т.е. с символом `\0` в конце;

- `data()` — возвращает содержимое строки в виде массива символов, но не являющегося с–строкой;
- `copy(char* str,size_type n,size_type pos)` — копирует содержимое в символьный массив `str`.

Шаблоны классов

Шаблоны классов поддерживают в C++ парадигму программирования с использованием типов в качестве параметров, *обобщённого программирования*. Механизм шаблонов допускает применение абстрактного типа в качестве параметра при определении класса или функции. Шаблон задаёт структуру целого семейства классов. Преимущество его использования состоит в том, что как только разработан и отлажен алгоритм работы с данными, он может применяться к любым типам данных без переписывания кода. После того, как шаблон класса определён, он может использоваться для определения конкретных классов. Процесс генерации компилятором определения конкретного класса по шаблону и значению аргументов шаблона называется *инстанцированием шаблона*.

Определение шаблонного (обобщённого, родового) класса имеет вид:

```
template<параметры шаблона> class имя_класса {};
```

Например, если мы хотим создать вектора из элементов разных типов, задавая при этом длину вектора, то шаблонный класс можно описать так:

```
template<class T, int n> class TVector{};
```

В этом случае параметрами шаблона являются некоторый абстрактный класс T и переменная целого типа, внутри класса имя T может использоваться также, как и имена любых других типов. Параметрами шаблонов могут быть абстрактные типы или переменные встроенных типов. При инстанцировании на место абстрактного класса подставляется аргумент встроенного типа или класса, разработанного прогрмистом. Синтаксис определения полей и встроенных методов шаблонного класса не отличается от записи в обычном классе. Только в том случае, если определение метода выносится за пределы класса, то в заголовок класса добавляется префикс `template<class T, int n>`, а при объявлении области видимости после имени класса в угловых скобках записывается имя параметра шаблона: `TVector<T,n>::`.

При включении шаблона класса в программу класс будет сгенерирован только тогда, когда, при создании экземпляра класса, вместо абстрактного типа T будет указан конкретный тип данных:

```
TVector<int> A;
```

В многофайловом проекте всё определение шаблонного класса, включая определения методов, вынесенные за пределы класса, необходимо помещать в один файл, как правило, с расширением `.h`, и подключать его с помощью директивы `include`.

Рассмотрим пример шаблона класса.

```
template<class T, int n> class TVector{
private:
   T *a;
public:
   TVector();
   ~TVector(){
      if(a)
        delete[] a;
   }
   T& operator[](int i){
      if(i<0||i>n)
         throw "Incorrect value of index";
      return a[i];
   }
   void print(void);
};
template<class T, int n> TVector<T,n>::TVector(void){
   a=new T [n];
   if(!a)
     throw "No place";
   for(int i=0;i<n;i++)
      a[i]=(T)0;
}
```

```
template<class T, int n> void TVector<T,n>::print(void){
   cout<<"vector:"<<endl;
   for(int i=0;i<n;i++){
      if(i%5==0)
        cout<<endl;
      cout<<a[i];
   }
   cout<<endl;
}

void main(void){
   TVector<int,100> i;
   TVector<double,5> d;
   TVector<char,10> c;
   i.printf();
   d.printf();
   int j;
   for(j=0;j<10;j++)
      c[j]=j+'a';
   c.printf();
}
```

Стандартная библиотека шаблонов

Шаблонами классов настолько удобно пользоваться, что была разработана стандартная библиотека шаблонов (standard template library) и включена в стандарт языка C++. Стандартная библиотека шаблонов состоит из двух основных частей: набора контейнерных классов и набора обобщённых алгоритмов.

Контейнеры — это объекты, содержащие другие однотипные объекты, организованные в одну из типовых структур данных: список, стек, очередь и т.п.. Объекты, хранимые в контейнерах, могут быть как встроенных, так и пользовательских типов. Так как контейнер — это шаблонный класс, то необходимо, чтобы помещаемые в него объекты допускали копирование и присваивание, поэтому в пользовательском классе должны быть в открытой части конструктор копирования и операция присваивания.

Можно выделить два типа контейнеров: последовательные, хранящие данные в виде непрерывной последовательности, и ассоциативные, построенные на основе сбалансированных деревьев. К базовым последовательным контейнерам относятся векторы (`vector`), списки (`list`) и двусторонние очереди (`deque`), кроме того есть специализированные контейнеры, реализованные на основе базовых: стеки (`steck`), очереди (queue) и очереди с приоритетами (`priority_queue`). Ассоцитивные контейнеры — это словари (`map`), словари с дубликатами (`multimap`), множества (`set`), множества с дубликатами (`multiset`) и битовые множества (`bitset`).

Чтобы использовать контейнер в программе необходимо включить в неё соответствующий заголовочный файл и задать тип сохраняемых объектов в аргументах шаблона:

```
#include<list>
list <string> sentence;
```

Практически во всех контейнерах используются унифицированные типы:

- `value_type` — тип элемента контейнера;
- `size_type` — тип индексов или счётчиков;
- `iterator` — итератор, являющийся аналогом указателя на объект, хранимый в контейнере; итератор представляет собой основной интерфейс для работы со стандартной библиотекой шаблонов;
- `reverse_iterator` — обратный итератор, просматривающий элементы контейнера в обратном порядке;
- `reference` — ссылка на элемент контейнера;
- `key_type` — тип ключа, используемый только для ассоциативных контейнеров;
- `key_compare` — тип критерия сравнения, используемый только для ассоциативных контейнеров.

Все контейнеры удовлетворяют трём основным требованиям:

- при вставке элемента в контейнер, должна создаваться его внутренняя копия, а не ссылка на внешний объект;
- элементы в контейнере должны располагаться в определённом порядке, т.е. при повторном переборе элементов с помощью итератора, порядок перебора остаётся прежним;

- в общем случае, программист самостоятельно должен следить за корректностью параметров методов и функций, работающих с контейнером, например, использование недействительного индекса может привести к непредсказуемым последствиям.

Для всех контейнерных классов определён тип итератор — это более абстрактная сущность, чем указатель, но обладающая похожим поведением. При объявлении объекта типа итератор всегда указывается область видимости, т.е. в явном виде определяется к какому шаблонному классу он относится:

```
list <string>::iterator i;
```

Основные операции с итераторами:

- разыменование итератора: если `p`— итератор, то `*p` — значение объекта, на который он ссылается;
- обращение к полям или методам объекта, на который ссылается итератор: `->`, например `p->metod1();`
- присваивание одного итератора другому;
- сравнение итераторов на равенство `==` и неравенство `!=`;
- перемещение итератора по всем элементам контейнера с помощью префиксного или постфиксного инкремента.

При помощи итераторов можно просматривать содержимое контейнера независимо от фактического типа объектов, хранящихся в нём. Для этого в каждом контейнере определены соответствующие методы.

Во всех контейнерах есть конструктор по умолчанию, клнструктор копирования и деструктор, кроме того есть набор методов, определённых для каждого контейнера.

- метод, возвращающий итератор на начало контейнера, при этом перемещение по контейнеру (итерации) будет производиться в прямом направлении:
  `iterator begin();`
  приведём пример использования:
  ```
  vector<int> V;
  vector<int>::iterator i=V.begin();
  ```
- метод, возвращающий итератор на конец контейнера, при этом итерации в прямом направлении будут закончены:
  `iterator end();`
- метод, возвращающий итератор на последний элемент контейнера, при этом итерации будут производиться в обратном направлении:
  `reverse_iterator rbegin();`
- метод, возвращающий итератор на первый элемент контейнера, при этом итерации в обратном направлении будут закончены:
  `reverse_iterator rend();`
  приведём пример использования:
  ```
  vector<int> V;
   for(vector<int>::reverse_iterator i=V.rbegin();i!=V.rend();i++) {/*...*/}
  ```
- метод, возвращающий `true`, если контейнер пуст и `false` в противном случае:
  `bool empty();`
- метод, возвращающий количество элементов в контейнере:
  `size_type size() const;`
- метод, возвращающий максимально допустимый размер контейнера:
  `size_type max_size() const;`
- метод, удаляющий все элементы контейнера:
  `void clear();`
- операция присваивания одного объекта–контейнера другому (`=`) и метод, меняющий местами два контейнера одинакового типа:
  `swap(контейнер);`
- операции сравнения объектов–контейнеров (`==, !=, <, >, <=, >=`), при этом сравниваемые контейнер должны быть одинакового типа, два контейнера считаются равными, если их элементы совпадают и следуют в одинаковом порядке, отношение «меньше–больше» проверяется по лексикографическому критерию.

Кроме общих методов, в каждом контейнере предусмотрены собственные методы, в соответствии с их назначением. Последовательные контейнеры (векторы, двусторонние очереди и списки) поддерживают разные методы, среди которых есть совпадающие по смыслу:

- вставка заданного элемента в конец — `push_back(элемент)`;
- удаление заданного элемента из конеца — `pop_back(элемент)`;
- вставка элемента в произвольное место, задаваемое с помощью итератора — `insert(итератор,элемент)`;

- удаление элемента из произвольного места, задаваемого с помощью итератора — `erase(итератор)`;
- доступ к произвольному элементу (отсутствует для списка) — `at(позиция)`;
- вставка заданного элемента в начало (отсутствует для вектора)— `push_front(элемент)`;
- удаление заданного элемента из начала (отсутствует для вектора)— `pop_front(элемент)`;

Контейнеры списки поддерживают много специальных методов для перемещения элементов:

- метод `splice()` служит для перемещения элементов списка в другой, можно вставить в список, вызывающий метод, перед элементом, на который указывает заданный итератор, все элементы другого списка (этот список будет вторым аргументом метода и после выполненения метода окажется пустым):
  `sp1.splice(iter,sp2);`
  можно перенести один элемент, при этом необходимо добавить третий аргумент — итератор, определяющий его положение в списке, заметим, что так можно перемещать элементы внутри одного списка:
  `sp1.splice(iter1,sp1,iter2);`
  можно перенести несколько элементов, расположенных подряд, указав в третьем и четвёртом аргументах итераторы, определяющей диапазон перемещаемых элементов:
  `sp1.splice(iter1,sp2,iter_first,iter_last);`
  этот метод можно использовать для изменения одного списка, но если элемент, на который указывает первый итератор, окажется в заданном далее диапазоне, то результат операции непредсказуем;
- метод `remove()` удаляет из списка все элементы с заданным значением:
  `sp1.remove(elem);`
- метод `sort()` сортирует элементы списка по возрастанию:
  `sp1.sort();`
- метод `reverse()` изменяет порядок следования элементов на обратный:
  `sp1.reverse();`
- метод `unique()` оставляет в списке только первый элемент из каждой серии идущих подряд одинаковых элементов:
  `sp1.unique();`
- метод `merge()` сливает два списка упорядоченных по возрастанию в один тоже упорядоченный по возрастанию:
  `sp1.merge();`

Алгоритм — это функция, выполняющая некоторые действия с элементами контейнера. Стандартная библиотека предоставляет алгоритмы для выполнения большинства распространённых операций. Общее число таких алгоритмов около шестидесяти. Объявления стандартных алгоритмов находятся в заголовочном файле `<algorithm>`. В качестве первых двух параметров алгоритмов почти всегда выступают итераторы, определяющие диапазон элементов контейнера, к которым будет применён алгоритм. Тип итераторов определяется типом контейнера, для которого может быть использован алгоритм. Алгоритмы не проверяют выход за пределы последовательности!

Рассмотрим в качестве примера несколько алгоритмов.

Алгоритм `find(начало, конец, значение_объекта)` выполняет поиск в последовательности заданного значения и возвращает итератор на самое левое найденное значение в случае успешного поиска или на конец последовательности в ином случае.

```
vector<int>::iterator i=find(v.begin(),v.end(),value);
if(i==v.end())
   cout<<"Нет элемента со значением"<<value;
```

Алгоритм `count(начало, конец, значение_объекта)`подсчитывает количество элементов в последовательности, равных заданному.

Алгоритм `search(начало1, конец1, начало2, конец2)` находит первое вхождение в первую подпоследовательность второй подпоследовательности.

Алгоритм `copy(начало, конец, итератор)` копирует подпоследовательность от начала до конца и вставляет её во вторую последовательность, начиная с того места, на которое указывает итератор.

Алгоритм `remove(начало, конец, значение_объекта)` удаляет из заданного диапазона последовательности элемент с заданным значением.

Алгоритм `replace(начало, конец, старое_значение, новое_значение)` заменяет старое значение элемента из заданного диапазона последовательности на новое.

Алгоритм `sort(начало, конец)` сортирует элементы последовательности по возрастанию, заметим, что для объектов, хранящихся в контейнере, должны быть определены операции сравнения, кроме того, алгоритм нельзя применять к спискам, так как он требует произвольного доступа к элементам, который в списке невозможен.

Алгоритмы `max_element(начало, конец)` и `min_element(начало, конец)` возвращают итератор на максимальный или минимальный соответственно элементы контейнера, при этом для объектов, хранящихся в контейнере, должны быть определены операции сравнения, соответственно меньше или больше.

Очень часто в качестве агументов алгоритма используются так называемые функциональные объекты. Это объекты некоторого класса, для которого определена одна единственная операция вызова функции `()`.

```
class My_Less{
public:
   bool operator()(const int x, const int y){return x<y;}
};

void main(){
   My_Less t;
   int a=5, b=3;
   if(t(a,b)) cout<<a<<"less"<<b<<endl;
   if(t(b,a)) cout<<b<<"less"<<a<<endl;
}
```

В стандартной библиотеке определены шаблоны функциональных объектов для операций сравнения, встроенных в язык C++.

Операция	Функциональный объект
==	`equal_to`
!=	`not_equal_to`
<	`less`
<=	`less_equal`
>	`greater`
>=	`greater_equal`

Чтобы использовать шаблонные функциональные объекты необходимио их инстанцировать, например, если необходимо отсортировать вектор целых чисел по убыванию надо добавить в алгоритм `sort` третий аргумент:

```
sort(v.begin(),v.end(),greater<int>());
```

Алгоритмы `cout_if` и `find_if` требуют в качестве третьего аргумента функциональный объект, который возвращает значение типа `bool`. Тогда с помощью этих алгоритмов можно соответственно подсчитать количество элементов последовательноти, удовлетворяющих заданному условию или найти первое вхождение такого элемента в контейнер. Рассмотрим пример, когда условием является попадание целого числа в заданный интервал.

```
class My_Int{
public:
   bool operator()(const int x){return ((x>-3)&&(x<3));}
};

void main(){
   vector<int> v;
   /* заполнение вектора */
   int k=count_if(v.begin(),v.end(),My_Int());
   vector<int>::iterator p=find_if(v.begin(),v.end(),My_Int());
}
```

В ассоциативных контейнерах объекты не выстроены в линейную последовательность, они организованы в более сложные структуры, как правило, это сбалансированные деревья, что даёт большой выигрыш в скорости поиска. Поиск производится с помощью ключей, обычно представляющих собой одно числовое или строковое значение.

В множестве `set` хранятся объекты, упорядоченые по некоторому ключу, являющемуся полем самого объекта, если в множестве хранятся элементы одного из встроенных типов, например `int`, то ключом будет сам элемент.

Словарь (map) можно представить как набор элементов, каждый из которых состоит из пары ключ+значение.

И в словарях и в множествах все ключи являются уникальными, т.е. одному ключу соответствует только один элемент множества. Мультисловари (`multimap`) и мультимножества (`multiset`) аналогичны родственным контейнерам, но в них одному ключу может соответствовать несколько значений.

Шаблон множества имеет два параметра: тип ключа и тип функционального объекта, определяющего отношение меньше. Второй параметр шаблона можно опустить, если в множестве должны храниться объекты встроенных типов, тогда по умолчанию будет использоваться критерий сортировки `less`, например, если объявлено множество целых чисел `set<int>`, то по умолчанию будет использован функциональный объект `less<int>`. Точно так же можно поустить второй параметр, если элементы множества принадлежат классу в котором определена операция <.

```
set<string> s1;
s1.insert("Маша");
s1.insert("Даша");
s1.insert("Саша");
set<string>::iterator i;
for(i=s1.begin();i!=s1.end();i++)
   cout<<i<<endl;
```

Шаблон словаря имеет три параметра: тип ключа, тип объекта и тип функционального объекта, определяющего отношение меньше. Второй параметр шаблона можно опустить аналогично предыдущему случаю. Доступ к элементу по ключу осуществляется при помощи операции индексации.

```
map<string,int> dic1;
dic1["Иванов Иван Иванович"]=555555;
dic1["Петров Пётр Петрович"]=333333;
map<string,int>::iterator i;
for(i=dic1.begin();i!=dic1.end();i++)
   cout<<i->first<<i->second<<endl;
cout<<dicl1["Петров Пётр Петрович"];
string fam="Иванов Иван Иванович";
if(dic1.find(fam)!=dic1.end())
   cout<<"Нашли"<<fam<<endl;
```

Наследование

Наследование — один из главных механизмов ООП. С его помощью можно разрабатывать очень сложные классы, продвигаясь от общего к частному, а также наращивая уже созданные классы, получая из них новые. Проектируя новый класс, прежде всего необходимо уяснить, какими общими свойствами должны обладать его объекты и нет ли похожего готового класса.

При наследовании обязательно имеются класс–предок (родитель, порождающий класс) и класс–наследник (потомок, порождённый класс) В C++ класс–родитель называется базовым, а класс–наследник — производным. Отношения между родительским классом и его потомками называется иерархией наследования. Глубина наследования не ограничивается. Иерархия чаще всего имеет вид дерева, но в общем случае может иметь структуру графа.

Производный класс может использовать элементы базового класса как свои собственные. Фактически, повторно используется уже написанный и отлаженный код, что естественно, повышает эффективность разработки программы и её надёжность.

Простым или одиночным называется наследование, при котором производный класс имеет только одного родителя. Наследование задаётся следующей конструкцией:

```
class имя_класса_потомка: [ключ_доступа] имя_класса_родителя {
компоненты класса
};
```

Ключ доступа и модификаторы доступа внутри классов регулируют защиту элементов классов и определяют кто из них и каким образом будет доступен.

Существует три ключа доступа или модификатора наследования:

- `public` — в этом случае наследование называется открытым;
- `protected` — в этом случае наследование называется защищённым;
- `private`— в этом случае наследование называется закрытым.

Следует напомнить, что модификаторы доступа разрешают использование членов класса таким образом:

- `public` — члены класса в этом разделе доступны в любом месте программы;
- `protected` — члены класса в этом разделе доступны только в классах–потомках;
- `private`— члены класса в этом разделе доступны только внутри самого класса.

Доступ к элементам базового класса из класса потомка можно определить с помощью следующей таблицы:

Модификатор доступа внутри базового класса	Ключ доступа	Доступ в производном классе
public	public	public
protected	public	protected
private	public	недоступны
public	protected	protected
protected	protected	protected
private	protected	недоступны
public	private	private
protected	private	private
private	private	недоступны

Заметим, что члены базового класса из раздела `private` недоступны в любом случае. Если ключ доступа отсутствует, то по умолчанию доступ к членам базового класса будет осуществляться аналогично случаю `private`.

Существует множественное наследование, когда класс–потомок наследует методы и свойства от нескольких классов–родителей. В этом случае наследование задаётся следующей конструкцией:

```
class имя_класса_потомка: [ключ_доступа] имя_класса_родителя_1,
[ключ_доступа] имя_класса_родителя_2, ... {
компоненты класса
};
```

Все базовые классы со своими ключами доступа перечисляются после двоеточия через запятую.

Рассмотрим пример наследования:

```
class A {
private:
   int x,y;
protected:
   double z;
   int sum (){return x+y;};
public:
   A():x(0),y(0),z(0){};
   void SetX(int x1){x=x1;}
   void SetY(int y1){y=y1;}
   void SetZ(double z1){z=z1;}
};

class B: public A{
private:
   int s;
   double v;
public:
   B(){s=sum(); v=z*sum();}//v=V();
   double V() {return z*sum();};
};

void main (){
   B ob1;
   ob1.SetX(1);
   ob1.SetY(2);
   ob1.SetZ(3);
   double c=ob1.V();
}
```

Конструкторы не наследуются — они создаются в производном классе самим программистом или компилятором по умолчанию. При этом выполняются следующие правила:

- если в базовом классе нет конструктора или есть конструктор без аргументов, то в производном классе конструктор можно не писать — компилятор создаст конструктор по умолчанию (без аргументов) и конструктор копирования;

- если в базовом классе все конструкторы с аргументами, в производном классе обязательно надо определить конструктор, в котором явно вызывается конструктор базового класса;
- при создании объекта производного класса сначала вызывается конструктор базового класса, а затем производного.

Рассмотрим пример, где класс–наследник может быть создан без явного определения конструктора:

```
class Basis{
private:
   double sum;
public:
   Basis(){sum=0;}
}
class Inherit: public Basis{};
```

Следующий пример показывает, каким образом может быть вызван конструктор базового класса в классе–наследнике:

```
class Basis{
private:
   int a,b;
public:
   Basis(int _a, int _b){a=_a;b=_b;}
};
class Inherit: public Basis{
private:
   int sum;
public:
   Inherit():Basis(1,1){sum=0;}
   Inherit(int x, int y,int s):Basis(x,y){sum=s;}
};
```

В последнем примере при создании объекта `Inherit i;` сначала будет вызван конструктор `Basis(1,1)`, а затем `Inherit()`.

Деструктор, как и конструктор, не наследуется, а создаётся:

- если в производном классе нет деструктора, то компилятор создаст деструктор по умолчанию не зависимо от того какой деструктор в базовом классе;
- деструктор базового класса автоматически вызывается в деструкторе производного класса в любом случае (определены деструкторы явно или по умолчанию); обязательно надо определить конструктор, в котором явно вызывается конструктор базового класса;
- при уничтожении объекта производного класса сначала вызывается деструктор производного класса, а затем базового.

Порождённый класс может добавить собственные поля, эти поля должны инициализироваться в конструкторе наследника.

Если дополнительное поле производного класса по типу и по имени совпадает с полем базового класса, то новое поле скрывает поле класса–родителя. Чтобы получить доступ к этому полю необходимо использовать перед его именем префикс, указывающий на принадлежность к базовому классу, например:

```
class Basis{
public:
   int a,b;
   Basis(){a=0;b=0;}
};
class Inherit: public Basis{
private:
   int sum,a;
public:
   Inherit(){sum=0;a=1;}
   void Metod(){sum=a+Basis::a;}
};
```

Аналогично, если имя метода производного класса совпадает с именем метода из базового класса, то первый скрывает (замещает или переопределяет) метод класса родителя и, чтобы обратиться к последнему, необходимо указать соответствующий префикс.

```
class Basis{
public:
   void Print(){cout<<"Hello, world!"<<endl;}
};
class Inherit: public Basis{
public:
   void Print(){
      Basis::Print();
      cout<<"How ara you?"<<endl;
   }
};
void main(){
Inherit a;
a.Print();
}
// на экране будет выведен текст
Hello, world!
How ara you?
```

В общем случае любое переопределение имени функции из базового класса автоматически скрывает все остальные версии этой функции и делает их недоступными в проиводном классе.

```
class Base{
public:
   int f(){cout<<"Base::f"<<endl; return 1;}
   int f(string str){cout<<str<<endl; return 1;}
   void g(){}
};
class In1: public Base{
public:
   void g(){}
};
class In2: public Base{
public:
   int f(){cout<<"In2::f"<<endl; return 2;}
};
class In3: public Base{
public:
   void f(){cout<<"In3::f"<<endl;}
};
class In4: public Base{
public:
   int f(int){cout<<"In4::f"<<endl; return 4;}
};
void main(){
   string s("abc");
   In1 a;
   int x=a.f();
   a.f(s);
   x=a.f(s);
   In2 b;
   x=b.f(); // нельзя! x=b.f(s);
   In3 c;
   c.f(); // нельзя! x=c.f(); x=c.f(s);
   In4 d;
   x=d.f(1); // нельзя! x=d.f(); x=c.f(s);
}
```

Кроме конструктора не наследуется два вида функций: операция присваивания и дружественные функции, т.к. последние не являются методами класса.

Если наследование является открытым (ключ доступа — public), то говорят, что класс–потомок является разновидностью класса–родителя. Это означает, что везде, где может быть использован объект

базового класса (при присваивании, передаче параметров в функцию, возврате результата из функции), вместо него можно подставлять объект производного класса. Таким образом, осуществляется принцип подстановки. Он работает и для ссылок и для указателей (вместо ссылки или указателя на объект базового класса можно использовать соответственно ссылку или указатель на объект производного класса). Важно помнить, что при работе с таким указателем оказываются доступны только элементы базового класса!

Если определены два класса — базовый класс `Basis` и производный `Inherit`, то можно выполнять следующее присваивание:

```
Basis b1,b2,*b3;
Inherit i1, i2,*i3;
b1=b2;
i1=i2;
b1=i1; //нельзя! i1=b1;
b3=&i1; //нельзя! i3=&b1;
```

В последнем случае работает принцип подстановки: справа от знака присваивания задан объект производного класса, который подставляется на место аргумента базового класса в операции присваивания класса–родителя. Преобразование типов при этом происходит автоматически. Следует обратить внимание, что копируются только те поля, которые унаследованы от базового класса, происходит *срезка*.

Для того, чтобы было возможно присваивать нооборот объект базового класса объекту производного, в классе–потомке необходимо явно переопределить соответствующую операцию присваивания, её прототип может выглядеть так:

```
Inherit& Inherit::operator=(const Basis &t){
   this->Base::operator=(t);
   return *this;
}
```

При открытом наследовании можно не дублировать дружественные функции, так как по принципу подстановки можно помещать аргументы производного класса на место параметров базового класса. например, если в родительском классе переопределена операция вывода в поток, то её можно применять к объектам класса–потомка.

При закрытом наследовании все (ключ доступа — private) члены базового класса становятся закрытыми, таким образом, программа–пользователь производного класса не имеет доступа к функциональной части класса–родителя, и класс–потомок не может быть интерпретирован как разновидность базового класса. Чтобы получить доступ к методам родительского класса их надо продублировать:

```
class Basis{
public:
   void metod1(){}
   void metod2(){}
};
class Inherit: private Basis{
public:
   void metod1(){ Basis::metod1();}
   void metod2(){ Basis::metod21();}
};
```

Можно в явном виде открыть доступ к методам базового класса, используя внутри производного класса директиву `using`:

```
class Basis{
public:
   void metod1(){}
   void metod2(){}
};
class Inherit: private Basis{
public:
   using Basis::metod21();
};
```

На практике, вместо закрытого наследования чаще всего используют композицию классов (когда объект одного класса становится полем другого).

Защищённое наследование (ключ доступа — protected) используется, в том случае, если необходимо скрыть функцию или переменную от внешних пользователей, но сохранить доступ к ней из производных классов.

Одним из самых важных механизмов наследования является возможность изменять свойства базового класса. Очень часто при создании программ требуется, чтобы поведение некоторых методов класса–предка отличалось для классов–потомков. Изменить содержание метода базового класса можно, если соответствующий метод объявлен в нём как виртуальный. В C++ это делается при помощи ключевого слова `virtual` перед заголовком метода.

Эта возможность обеспечивается особенностями механизма сопоставления вызова функции с телом этой функции — связывания. Для обычных методов связывание просходит на этапе трансляции до запуска программы и называется ранним или статическим. При наследовании обычного метода его поведение в классе–наследнике не меняется.

Рассмотрим пример статического связывания:

```
class Base{
public:
   void metod(){cout<<"Hello!";}
};
class Inherit: public Base{
public:
   void metod(){cout<<"Bye-bye!"}
};
void main(){
   int x;
   Base *b;
   cin>>x;
   if(x==1) b=new Base;
      else b=new Inherit;
   b->metod();
}
```

Так как на этапе компиляции нельзя предсказать, какой выбор будет произведён в ходе работы программы, то компилятор для связывания выберет метод базового класса в соответствии с типом указателя, и через указатель `b` метод производного класса окажется недоступен. В итоге, во время работы программы, независимо от значения переменной `x`, всегда будет выведено слово `Hello!`.

Ещё один пример:

```
class Base{
protected:
   int x;
public:
   Base(int _x){x=_x;}
   void print(){cout<<x<<"  ";}
   void metod(){x*=2;}
};
class Inherit: public Base{
public:
   Inherit(int _x):Base(_x){}
   void metod(){x/=2;}
};
void main(){
   Base b(10);
   Inherit i(10);
   b.metod();
   b.print();
   i.metod();
   i.print();
   Base *pointB;
   pointB=&b;
   pointB->metod();
   pointB->print();
   pointB=&i;
```

```
   pointB->metod();
   pointB->print();
}
```

Так как вызов метода `pointB->metod();` происходит из базового класса, то программа выведет на экран

```
20  5  40  10
```

Для виртуальных методов связывание происходит в процессе выполнения программы, т.е. решение о том, какой из методов будет вызван принимается после компиляции, это называется поздним или динамическим связыванием. Если в предыдущем примере сделать `metod()` виртуальным:

```
class Base{
protected:
   int x;
public:
   Base(int _x){x=_x;}
   void print(){cout<<x<<"  ";}
   virtual void metod(){x*=2;}
};
```

то на экран будет выведено:

```
20  5  40  2
```

Если в класса определяется виртуальный метод, то в такой класс автоматически добавляется скрытый член–указатель на таблицу виртуальных функций, а также генерируется специальный код, позволяющий осуществить выбор виртуального метода, подходящего для объекта данного типа, во время работы программы.

Если некоторый метод объявлен в базовом классе как виртуальный, то метод с тем же именем, с таким же списком параметров и тем же возвращаемым значением (метод с такой же сигнатурой), переопределённый в производном классе, автоматически становится виртуальным. Ключевое слово `virtual` при его описании в классе–потомке уже можно не использовать.

Существует ряд важных правил работы с виртуальными методами:

- вируальная функция может быть только методом класса;
- виртуальным может быть любой метод класса кроме конструктора;
- виртуальные методы наследуются, т.е. переопределять их в производном классе требуется только при необходимости задать отличающиеся действия; права доступа изменять нельзя;
- если виртуальный метод каким–либо образом переопределён в классе–потомке, объекты этого класса могут получить доступ к виртуальному методу базового класса с помощью префикса, указывающего на принадлежность соответствующему классу (иначе этот префикс называется операцией доступа к области видимости);
- если виртуальный метод не был переопределён в производном классе, то при его вызове для объекта этого класса будет происходить обращение к соответствующему виртуальному методу из ближайшего по иерархической лестнице базового класса, в котором он определён;
- в производном классе нельзя переопределять метод, отличающийся от виртуального метода базового класса только типом возвращаемого значения (при попытке описать такой метод, компилятор выдаёт ошибку);
- членом производного класса может быть метод с именем, совпадающим с именем виртуального метода базового класса, но с другой сигнатурой; в этом случае он будет другим, не виртуальным методом, соответствующий метод класса–предка будет скрыт.

Рассмотрим пример использования виртуальных методов родительского класса:

```
class Base{
public:
    void f(){cout<<"Base::f()"<<endl;}
    virtual void f(int i){cout<<"virtual Base::f(int)"<<endl;}
};
class In1: public Base{
public:
void f(){cout<<"In1::f()"<<endl;}
void f(int i){cout<<"virtual In1::f(int)"<<endl;}
};
```

```
class In11: public In1{
};
void main(){
   Base b;
   In1 i1;
   Base *b_ptr1,*b_ptr2;
   b_ptr1=&b;
   b_ptr2=&i1;
   b.f();
   i1.f();
   b.f(1);
   i1.f(1);
   //на экране:
   //Base::f()
   //In1::f()
   //virtual Base::f(int)
   //virtual In1::f(int)
   b_ptr1->f();
   b_ptr2->f();
   //на экране:
   //Base::f()
   b_ptr1->f(1);
   b_ptr2->f(1);
   //на экране:
   //virtual Base::f(int)
   //virtual In1::f(int)
}
```

Рассмотренный пример позволяет сделать ряд важных выводов:

- если вызов обычного или виртуального методов выполняется с помощью операции ., то вызывается метод соответствующего класса и использование виртуальных методов ничем не отличается от обычных;
- если вызов обычных, не виртуальных методов выполняется через указатель с использованием операции ->, то вызывается метод из класса, соответствующего типу указателя;
- если вызов виртуальных методов выполняется через указатель с использованием операции ->, то вызов метода зависит от типа объекта, на который указывается, а не от типа самого указателя; таким образом, виртуальные методы нужны, если ведётся работа с указателями.

Конструктор не может быть виртуальным, так как он вызывается только при создании и при этом, конечно же известен тип создаваемого объекта.

В отличие от конструктора, деструктор может быть виртуальным. Более того, если в классе объявлены виртуальные методы, то и деструктор должен быть виртуальным. Это необходимо для корректного уничтожения объектов через указатели. Если указатель имеет тип базового класса, а на самом деле указывает на объект класса–потомка, то при уничтожении такого объекта должен быть вызван сначала деструктор производного класса, а затем родительского. Это возможно только в том случае, если деструктор будет виртуальным.

Если при определении базового класса некоторый метод объявлен как виртуальный, то он должен быть определён подобно обычным методам, либо объявлен как абстрактный или чистый метод при помощи спецификатора `=0`:

```
virtual тип_возвращаемого_значения имя_метода (аргументы)=0;
```

Класс, содержащий хотя бы один абстрактный метод, называется абстрактным классом. Абстрактный класс может служить только в качестве базового для других классов. Объект абстрактного класса создать невозможно, но можно определить указатель или ссылку на такой класс. В классах производных от абстрактного класса чистые методы должны быть определены либо вновь объявлены как абстрактные, если явно определены не все методы, то класс–потомок тоже будет абстрактным. Абстрактный класс не может быть типом аргумента в методе и типом возвращаемого значения. Все методы абстрактного класса, кроме деструктора, могут вызывать чистые методы своего класса. В этом случае в производном классе будут вызваны соответствующие переопределённые методы. Абстрактные классы предназначены для представления общих понятий, которые предстоит конкретизировать в производных классах.

Потоки

Программа на C++ представляет ввод и вывод как поток байтов. Понятие потока позволяет абстрагироваться от того с каким устройством ввода/вывода идёт работа. Обычно ввод/вывод информации идёт через буфер — область оперативной памяти, накапливающей некоторое количество байтов, прежде чем они отправляются по назначению.

Язык C++ предоставляет возможность ввода/вывода как на низком уровне (неформатированный ввод/вывод), когда передача информации осуществляется одинаковыми блоками без преобразования, так и на высоком уровне (форматированный ввод/вывод), когда байты группируются по разному для представления различных элементов данных: целе числа, числа с плавающей запятой, строки символов и т. д.

Для поддержки потоков библиотека C++ содержит иерархию классов, построенную на основе двух базовых классов `ios` и `streambuf`. Класс `ios` содержит базовые средства для управления потоками, являясь родительским для других классов ввода/вывода. Класс `streambuf` обеспечивает общие средства управления буферами потоков и их взаимодействие с физическими устройствами, являясь родительским для других буферных классов. В классе `ios` есть поле `bp`, являющееся указателем на `streambuf`.

Все потоки могут быть однонаправленными (только для ввода или только для вывода) и двунаправленными (одновременно входными и выходными).

Потоки, связанные с консольным вводом/выводом (клавиатура/экран), называются стандартными. Стандартному потоку ввода соответствует класс `istream`, стандартному потоку вывода — класс `ostream`. Оба класса являются наследниками класса `ios`. Следующим в иерархии является класс `iostream`, наследующий классы `istream` и `ostream`.

При наличии директивы `#include<iostream>` в программе автоматически становятся доступны объекты

- `cin` — объект класса `istream`, соответствующий стандартному потоку ввода;
- `cout` — объект класса `ostream`, соответствующий стандартному потоку вывода.

Кроме стандартных есть ещё файловые и строковые потоки. Для их использования в программе следует объявлять объекты соответствующих классов.

Для использования файловых потоков необходимо подключить заголовочный файл `fstream`, тогда можно объявлять объекты–потоки трёх видов: входной `ifstream`, выходной `ofstream`, двунаправленный `fstream`.

Благодаря этим объектам становятся доступны методы соответствующих классов. Например, операция помещения (включения, вставки) в поток `<<` и операция извлечения из потока `>>`.

С помощью методов можно изменять формат данных в потоке, этой же цели служат специальные функции, называемые манипуляторами, которые можно включать в поток.

- манипулятор `endl` включает в поток символ новой строки и очищает буфер (`cout<<endl;`)
- манипулятор `setprecision(n)` или метод `precision(n)` устанавливают количество отображаемых знаков (`cout<<setprecision(6)<<x;` или `cout.precision(6); cout<<x;`)
- манипулятор `setw(n)` или метод `width(n)` устанавливают ширину поля вывода;
- манипулятор `hex` или метод `flags(ios::hex)` позволяют перейти в шестнадцатеричную систему счисления.

Большинство параметров форматирования сохраняют своё состояние вплоть до следующего вызова функции, изменяющей это состояние. Исключением является только манипулятор `setw(n)` исоответствующий ему метод `width(n)`, которые распространяются только на ближайшую операцию вывода.

Для отслеживания ошибок в базовом классе `ios` определено полеstate, отдельные биты (флаги), которого отображают состояние потока. Чаще всего более удобны для анализа состояния потока методы.

- флаг `ios::goodbit` — нет ошибок;
- флаг `ios::eofbit` — достигнут конец файла, соответствующий метод —`int eof()`, возвращающий ненулевое значение в этом случае;
- флаг `ios::failbit` — ошибка форматирования или преобразования, соответствующий метод — `int fail()`, возвращающий ненулевое значение в случае ошибки.

Сбросить все флаги в ноль можно с помощью метода `clear()`. Проще всего проверить состояние потока можно с помощью обычного логического выражения `if(!cin)`.

-

История

Впервые средства разработки QT стали известны общественности в мае 1995 года. QT была разработана норвежцами Хаарвардом Нордом и Айриком Чеймб–Ингом — исполнительным директором и президентом компании «Trolltech».

QT представляет собой комплексную рабочую среду, предназначенную для создания на C++ межплатформенных приложений с графическим пользовательским интерфейсом (graphical user interface GUI). Главным достоинством является то, то эта среда разработки даёт возможность построения и компиляции приложений в любой операционной системе.

Первое приложение

Рассмотрим пример простейшего приложения с использованием библиотеки QT:

```
#include<QApplication>
#include<QLabel>
int main(int argc,char *argv[]){
   QApplication app(argc,argv);
   QLabel label("Hello!");
   label.show();
   return app.exec();
}
```

В первых двух строках в программу включаются заголовочные файлы, которые позволяют использовать соответствующие классы — `QApplication` и `QLabel`. Для применения в программе класса библиотеки QT необходимо подключить заголовочный файл, содержащий определение этого класса, имя файла совпадает с именем класса.

В первой строке функции `main` создаётся объект `app` класса `QApplication`. В его конструкторе необходимо указать в качестве параметров аргументы командной строки. В следующей строке создаётся текстовая метка — объект класса `QLabel`. Любой визуальный объект графического интерфейса называется виджетом (widget от window gadget). В терминологии Windows виджеты называются элементами управления. Кроме метки, виджетами являются, например, кнопки, меню, панель инструментов, полосы прокрутки, бегунки. Виджетами являются также окно приложения и окно диалога. Кроме того, QT устроена так, что любой виджет может быть окном приложения, в рассматриваемом примере окном приложения служит метка `label`.

В следующей строке вызывается метод `show()`, который делает метку видимой. Все виджеты создаются невидимыми и таким образом их можно настроить не допуская мерцания экрана.

И в последней строке программы происходит передача управления приложению `app`. Метод `exec()` работает так, что при этом программа переходит в цикл обработки событий, т.е. ждёт от пользователя нажатия кнопки мыши или клавиатуры.

Чтобы получить испоняемый файл.exe необходимо в любом текстовом редакторе создать файл с программой с расширением `cpp`, например `hello.cpp`, поместить его в отдельную директорию, например, с тем же именем `hello`, а затем выполнить команды:

- `qmake -project` — для создания проекта, имя проекта будет совпадать с именем директории, если необходимо дать проекту другое имя, то следует указать ключ `-o` и имя проекта — `qmake -project -o app.pro hello.cpp`
- `qmake -makefile hello.pro` — для создания `make`–файла;
- `mingw32-make -f Makefile.Debug` — для отладки программы, информация об ошибках будет выведена на экран в текстовом режиме;
- `mingw32-make -f Makefile.Release` — после исправления всех ошибок для создания исполняемого файла, который будет помещён в директорию `release`.

Для упрощения работы можно создать командный файл, например `build.cmd`:

```
qmake -project
qmake -makefile hello.pro
mingw32-make -f Makefile.Debug
mingw32-make -f Makefile.Release
pause
```

При запуске некоторые строки можно закомментарить:

```
rem mingw32-make -f Makefile.Debug
rem mingw32-make -f Makefile.Release
```

Рассмотрим простейший пример взаимодействия с пользователем:

```
#include<QApplication>
#include<QPushButton>
int main(int argc,char *argv[]){
   QApplication app(argc,argv);
   QPushButton button("Quit");
   QObject::connect(&button,SIGNAL(clicked()),&app,SLOT(quit()));
   button.show();
   return app.exec();
}
```

Приложение представляет собой кнопку, которую пользователь может нажать, и тогда приложение закончит свою работу. Вместо объекта класса `QLabel` в качестве главного виджета выступает объект класса `QPushButton` — кнопка. В программу добавляется код, обеспечивающий реакцию приложения на действие пользователя (нажатие этой кнопки).

При GUI-программировании, часто необходимо сообщать одним элементам об изменении других элементов управления. Более обобщенно можно сказать, что нужно обеспечить связь между объектами любых видов.

Для связывания событий, происходящий с объектами, и функций, предназначенных для обработки этих событий, в библиотеке Qt используется механизм сигналов и слотов. Сигнал – это сообщение о том, что произошло какое-либо событие, например, нажатие на кнопку или выбор пункта меню. Вся информация о событии сохраняется в полях экземпляра соответствующего класса. У сигнала есть источник (например, кнопка) и приёмник (объект, метод которого будет обрабатывать это событие). Слот – это сама функция-обработчик события. Связь между всеми четырьмя перечисленными элементами задаётся с помощью метода connect (соединить): Для определения сигнала и слота используются макросы SIGNAL и SLOT.

В рассматриваемом примере источником сигнала является кнопка, при её нажатии генерируется предопределённый сигнал `clicked()`, описанный в классе `QPushButton`. Приёмником является объект `app` — приложение, метод `quit()` которого будет вызван в ответ на сигнал нажатия кнопки, а именно, работа приложения будет завершена.

Виджеты Qt имеют множество предопределенных сигналов, но всегда можно создать их подклассы, чтобы добавить свои сигналы. Виджеты Qt имеют также множество предопределенных слотов, но можно создавать подклассы виджетов и добавлять свои слоты для того, чтобы обрабатывать поступающие сигналы.

Иерархия классов

Основной класс — `Qt`. Он является базовым для многих классов. Например, `QObject`(основа для создания объектных моделей), `QWidget` (базовый класс для всех объектов интерфейса), `QPainter`(для рисования на виджетах и других устройствах рисования).

Прямыми наследниками `QObject` являются `QApplication` (класс, служащий для создания и настройки GUI–приложения), `QLayout` (базовый класс для менеджеров компоновки).

Класс `QObject` — это основной класс с которого всё начинается. Во–первых, многие классы библиотеки `QT` являются его потомками, во–вторых, очень часто в приложении создаются классы–наследники, т.к., например, если класс должен иметь сигналы и слоты, то он обязательно должен быть потомком `QObject`. Следует заметить, что при множественном наследовании, только один из базовых классов может быть потомком или самим `QObject`, а при перечислении родителей этот класс необходимо записывать первым.

Класс `QApplication` является основой любого GUI–приложения. Объект этого класса создаётся в приложении обязательно в единственном экземпляре. Задачей этого объекта является получение событий клавитуры, таймера, мыши, операционной системы вообще, высылка событий к элементам управления, анализ аргументов командной строки, инициализация некоторых настроек работы приложения (стиля, цвета и т.п.). Объект доступен в любой момент работы приложения, а срок жизи этого объекта соответствует концу работы всего приложения.

Прямыми наследниками `QWidget` являются `QButton` (базовый класс для создания различных кнопок, например, его наследник — `QPushButton`), `QDialog` (базовый класс для диалоговых окон, его наследником, в частности являются `QFountDialog` — стандартный диалог для выбора параметров текста, `QMessageBox` — диалог с коротким сообщением, пиктограммой и некоторыми кнопками), `QMainWindow` (класс главного окна приложения).

Иерархия объектов

В QT существует возможность создавать иерархию объектов программы, если они являются наследниками класса QObject. Один из конструкторов этого класса имеет два параметра: QObject(QObject* pobj=0,const char* pszName=0). Первый параметр — указатель на объект–предок, если он отсутствует или равен нулю, то объект является вершиной иерархии. Объект–предок задаётся один раз при создании объекта и больше не меняется. Второй параметр — имя объекта, которое может быть полезно при отладке программы. Создание иерархии объектов полезно для упрощения работы с динамическим выделением памяти, так как при уничтоженииии объекта–предка, все объекты–потомки уничтожаются автоматически, не требуя от программиста написания соответствующего кода. По этой причине, большинство объектов в программах, написанных на QT, создаются динамически с помощью указателей.

Механизм сигналов и слотов

Элементы графического интерфейса определённым образом реагируют на действия пользователя и посылают сообщения. В приложении необходимо обеспечить связь между его элементами. Более старые инструментарии обеспечивают подобную связь с помощью отзывов. Это концепция функций обратного вызова, при которой отзыв осуществляется с помощью обратного вызова — указателя на функцию. Если необходимо, чтобы функция обработки уведомляла о некотором событии, то ей передаётся указатель на другую функцию (отзыв). Функция обработки вызовет функцию отзыва, когда это будет уместно. Отзывы имеют два фундаментальных недостатка: во-первых, нельзя проверить, что функция обработки вызывает отзыв с правильными аргументами, во-вторых, отзыв жестко связан с функцией обработки, так как функция обработки должна знать, какой отзыв вызывать.

Недостатком языка C++ является то, что он изначально не был предназначен для написания пользовательского интерфейса, поэтому в нём нет удобных средст для создания связей между элементами. Поэтому любая библиотека, решающая эту проблему должна создавать дополнительные средства.

В библиотеке Qt для связывания событий, происходящих с объектами, и функций, предназначенных для обработки этих событий, используется механизм сигналов и слотов, специальных методов классов. Проблема расширения языка C++ решена в QT с помощью специального препроцессора MOC (Meta Object Compiler) — метаобъектного компилятора. В каждом классе, где есть сигналы и слоты должен быть указан специальный макрос Q_OBJECT. Препроцессор MOC анализирует классы на наличие этого макроса в их определении и внедряет в отдельный файл всю необходимую дополнительную информацию. Процесс происходит автоматически, не требую вмешательства разработчика приложения, необходимо лишь знать несколько правил для эффективного использования сигналов и слотов:

- каждый класс, в котором должны быть сигналы и слоты , должен быть наследником класса QObject;
- макрос Q_OBJECT должен располагаться в самом начале определения класса сразу после открывающей скобки (обычно в отдельной строке);
- каждый класс, унаследованный от QObject может содержать любое количество сигналов и слотов;
- сигнал может иметь множество аргументов любого типа;
- один сигнал можно соединять с любым количеством слотов, при генерации сигнал поступит ко всем, соединённым с ним слотам;
- слот может принимать сообщения от многих сигналов разных объектов;
- один сигнал может соединяться с другим сигналом;
- для успешного соединения сигнала со слотом, их параметры должны быть одинаковы и задаваться в одном и том же порядке, но у сигнала может быть больше параметров, вэтом случае дополнительные параметры игнорируются;
- при уничтожении объекта все связи автоматически уничтожаются;
- связь между сигналом и слотом можно аннулировать принудительно с помощью функции disconnect.

Существенным недостатком механизма связи является то, что не производится проверки на совместимость соединяемых сигнала и слота и о наличии их в соответствующих классах. Такого рода ошибки выявляются только после запуска приложения.

Как уже упоминалось ранее, у сигнала есть источник (например, кнопка) и приёмник (объект, метод которого будет обрабатывать это событие). Слот – это сама функция-обработчик события. Связь между всеми четырьмя перечисленными элементами задаётся с помощью метода connect (соединить):

```
bool QObject::connect (
    const QObject *sender,     // Источник события.
    const char *signal,        // Сигнал.
    const QObject *receiver,   // Объект-приёмник сигнала.
    const char *method,        // Функция-обработчик.
```

```
    Qt::ConnectionType type = Qt::AutoConnection
    ) const
```

Первый параметр `sender` — указатель на объект, высылающий сигнал. Второй параметр `signal` — сигнал, с которым осуществляется соединение. Прототип (имя и аргументы) метода–сигнала должен быть заключён в специальном макросе `SIGNAL`. Параметр `receiver`— указатель на объект, в котором есть слот для обработки сигнала. А `method` — этот слот, его прототип должен быть заключён в макросе `SLOT`. Последний параметр определяет режим обработки: `Qt::DirectConnection` – событие обрабатывается сразу; `Qt::QueuedConnection` – событие ставится в общую очередь и будет обработано только после всех сообщений, уже имеющихся в этой очереди; `Qt::AutoConnection` – если источник события находится в том же потоке, что и приёмник, то будет использован режим `Qt::DirectConnection`, в противном случае – `Qt::QueuedConnection`. Этот параметр можно не указывать, компилятор автоматически определит очерёдность обработки сигналов.

Чтобы создать соединение, необходимое для решения поставленной задачи, надо знать какие сигналы и слоты уже есть в используемых объектах. Например, если необходимо, чтобы текстовая метка `label` (экземпляр класса `QLabel`) отображала позицию полосы прокрутки `scrollBar` (экземпляр класса `QScrollBar`), то как создать связь можно узнать обратившись к документации. В документации на библиотеку `Qt` (открыв `doc/html/index.html` или программу `assistant`) находим, что в классе `QAbstractSlider`, потомком которого является `QScrollBar`, определён сигнал `void QAbstractSlider::valueChanged ( int value )`, оповещающий об изменении положения ползунка полосы прокрутки. Далее, в описании класса `QLabel` находим, что изменение текста надписи производится с помощью функции `setText(строка)` или `setNum(число)`. Тогда вызов метода connect должен выглядеть следующим образом:

```
QObject::connect(
    scrollBar,                 // Источник события.
    SIGNAL(valueChanged(int)), // Сигнал.
    label,                     // Объект-приёмник сигнала.
    SLOT( setNum(int) ) );     // Функция-обработчик.
```

Заметим, что параметрами сигнала и слота являются типы, а не переменные. При выполнении программы значением i-го параметра слота становится значение i-го параметра сигнала.

Если вызов происходит из класса–потомка `QObject`, то соответствующий префикс перед `connect` можно не указывать. В том случае, если слот содержится в классе, в котором происходит соединение, то третьим параметром будет указатель на текущий объект `this`, при чём, этот параметр можно опустить.

Очень важным является то, что соединяемые объекты не зависят друг от друга, и, соответственно , могут быть реализованы отдельно. Объект, высылающий сигналы, может не заботиться, кто и как их принимает и обрабатывает и наоборот. Таким образом, программу легко разбить на независимые части, которые разрабатываются отдельно, а потом очень просто соединяются с помощью механизма сигналов и слотов. Если какой–нибудь компонент в проекте устареет, то его замена произойдёт незаметно для других элементов программы, так как имена сигналов и слотов, а также связь между ними сохранятся неизменными.

Сигналы определяются в классе, как и обычные методы, только без реализации. С точки зрения программиста, они выглядят, как прототип метода, содержащийся в заголовочном файле определения класса. Всю дальнейшую работу о реализации кода этих методов берёт на себя `MOC`. Сигналы не возвращают никаких значений, поэтому перед именем сигнала всегда ставится 'void'. Сигнал не обязательно соединять со слотом, если соединения не произошло, то он просто не будет обработан.

Обычно в программе используются стандартные сигналы, но если требуется определить собственные, то их надо объявить в разделе `signals`. Выслать сигнал можно с помощью ключевого слова `emit`. Сигналы высылаются из класса, кототорый их содержит. Чаще всего для это происходит в методах этого класса:

```
class Myclass : public QObject {
    Q_OBJECT
signals:
    void mySignal();
public:
    void sendSignal()(
       emit mySignal();
    }
}
```

Сигнала могут передавать информацию в параметре:

```
class Myclass : public QObject {
    Q_OBJECT
signals:
    void mySignal(const QString&);
public:
    void sendSignal()(
       emit mySignal("Information");
    }
}
```

Слоты — это методы, которые присоединяются к сигналам. Синтаксически, они являются обычными методами класса, но их надо определять в отдельном разделе `slots`, указывая способ доступа, например, `public slots` или `private slots`. Слоты могут быть виртуальными. Но в слотах нельзя задавать параметры по умолчанию. Внутри слота с помощью метода `sender()` можно определить на сигнал какого объекта вызван слот. Этот метод возвращает указатель на объект типа `QObject`, например:

```
class MyClass : public QObject {
    Q_OBJECT
public slots:
    void mySlot(){
      cout<<sender()->name()<<endl;
   }
}
```

В этом примере будет выведено на экран имя объекта, который послал обрабатываемый сигнал.

Приведём также пример, в котором соединяются два сигнала:

```
class MyClass : public QObject {
    Q_OBJECT
signals:
    void mySignal();
public:
    void myMetod(){
      connect(pSender,SIGNAL(signalSender()),SIGNAL(mySignal()));
   }
}
```

В примере сигнал `signalSender()` должен содержаться в классе, на объект которого указывает `pSender`.

Рассмотрим пример использования сигналов и слотов. Создадим приложение, в котором создаются кнопка нажатия и надпись, отражающая текущее состояние счётчика, при нажатии на кнопку счётчик увеличивается на единицу. Если состояние счётчика достигает десяти, приложение завершает работу. Для этого опишем класс «счётчик», объект которого будет генерировать один сигнал при увеличении значения счётчика и другой сигнал при достижении им значения 10. Чтобы обеспечить решение задачи, в основной функции программы `main` создадим необходимые связи между объектами приложения, счётчика, кнопки и надписи.

Сначала создадим класс для организации работы счётчика `Counter`. Определение этого класса содержится в заголовочном файле `counter.h`, а описание метода–слота в соответствующем файле с расширением `cpp`:

```
#include<qobject>
class Counter : public QObject {
    Q_OBJECT
private:
int Value;
public:
Counter():QObject(),Value(0){}
public slots:
    void Inc();
signals:
    void EndWork();
void Change(int);
};
```

В классе определено поле Value для хранения состояния счётчика, конструктор, слот Inc(), который будет описывать изменение счётчика при нажатии на клавишу и два сигнала, соответствующих завершению работы приложения, когда поле Value получит значение десять (EndWork()) и изменению значения этого поля (Change(int)).

Файл counter.cpp содержит описание слота:

```
void Counter::Inc(){
emit Change(++Value);
if(Value==10)
emit EndWork();
}
```

В главной функции main создаются четыре объекта: приложение, надпись, кнопка и счётчик.

```
#include<QApplication>
#include<QLabel>
#include<QPushButton>
#include"counter.h"
int main(int argc,char *argv[]){
   QApplication app(argc,argv);
   QLabel label("0");
   QPushButton button ("ADD");
   Counter i;
   label.show();
   button.show();
   QObject::connect(&button,SIGNAL(clicked()),&i,SLOT(Inc()));
   QObject::connect(&i,SIGNAL(Change(int)),&label,SLOT(setNum(int)));
   QObject::connect(&i,SIGNAL(EndWork()),&app,SLOT(quit()));
   QObject::connect(&app,SIGNAL(lastWindowClosed()),&app,SLOT(quit()));
   return app.exec();
}
```

Первый вызов функции connect обеспечивает соединение сигнала нажатия кнопки clicked() с методом счётчика Inc(). Этот метод, при каждом нажатии клавиши, увеличивает значение счётчика, а также генерирует сигнал изменения значения счётчика (Change(int)) или сигнал достижения значения 10 (EndWork()). Чтобы элемент надписи всегда отображал актуальное значение счётчика, в следующей строке создаётся связь между сигналом Change(int) и методом надписи setNum(int), позволяющим вывести на надпись число. Далее сигнал EndWork() связывается с методом, завершающим работу приложения app — quit(). Приложение состоит из двух окон (каждому виджету соответствует своё окно) и завершить его работу необходимо после закрытия последнего окна. Поэтому в последнем вызове функции connect происходит соединение сигнала lastWindowClosed() и метода quit().

Высылку сигналов объектом можно заблокировать методом blockSignals(false) и разблокировать blockSignals(true). Текущее состояние блокировки можно получить с помощью метода signalsBlocked().

В QT при уничтожении объекта, все связи с ним автоматически уничтожаются, но существует возможность отменить связь между объектами «вручную». Чтобы разорвать связь между сигналом и слотом, используется метод disconnect:

```
bool QObject::disconnect ( const QObject *sender,
                           const char *signal,
                           const QObject *receiver,
                           const char *method )
```

Виджеты

Виджеты — это строительный материал для создания пользовательского интерфейса. QT предоставляет широкий набор элементов управления о кнопок до различных диалоговых окон. Если для решения поставленной задачи недостаточно свойств имеющихся виджетов, то программист может создать свой, наследуя классы уже существующих виджетов.

Базовым для всех виджетов является класс QWidget. Он содержит массу методов, необходимых каждому виджету для изменения размера, местоположения, обработки событий и т.д. Сам класс QWidget является наследником класса QObject, поэтому может использовать механизм сигналов и слотов, а также

механизм объектной иерархии. Последнее очень важно, так как позволяет одному виджету служить контейнером для других, причём вложенность контейнеров может быть сколь угодно глубокой. Например, диалоговое окно содержит кнопки, то есть является контейнером.

Класс `QWidget` и большинство унаследованных от него классов имеют конструктор с двумя основными параметрами:

```
QWidget ( QWidget * parent = 0, Qt::WindowFlags f = 0 )
```

Первый параметр — указатель на объект–предок, по умолчанию предок отсутствует. Виджеты без предка называются виджетами верхнего уровня. Каждый такой элемент имеет отдельное окно для визуального отображения. Виджетом верхнего уровня может быть любой виджет без исключения. Если у виджета указан предок, то виджет является дочерним и располагается внутри родителя. При этом, если уничтожается виджет–предок, то то все дочерние уничтожаются автоматически, если виджет–предок невидим, то невидимыми будут и все дочерние виджеты.

Второй параметр используется для задания свойств окна виджета. С его помощью можно установить внешний вид окна (стиль отображения) и режим отображения (например, режим, при котором окно не может перекрываться другими окнами). Чтобы задать этот параметр используют специальные константы, определённые в классе `QT`, несколько констант обычно соединяются с помощью побитовой операции. Обычно окно ограничено рамкой, а вверху располагается строка с заголовком и тремя конпками (title bar).

Для виджетов верхнего уровня можно установить заголовок с помощью метода–слота `void setWindowTitle ( const QString & c)` .

Каждый виждет создаётся невидимым, чтобы процесс настройки его параметров не создавал на экране мерцание. Чтобы сделать виджет видимым, необходимо вызвать метод–слот `void show ()`, `void setHidden ( bool hidden )` (если аргумент — `true`, то виджет будет невидим, `false` наоборот) или `void setVisible ( bool visible )`(`true` — видим, `false` наоборот). Определить видимость виджета можно с помощью метода `bool isVisible () const`.

Виджет представляет из себя прямоугольную область включающую рамку, заголовок и клиентскую область. Существует целый ряд методов, позволяющих определить местоположение виджета и его размеры. для дочерних виджетов расположение определяется относительно клиентской области родителя. Например, методы `int x()` и `int y()` возвращают координаты левого верхнего угла виджета, `int frameGeometry().width()` и `int frameGeometry().heigth()` соответственно ширину и высоту, а методы `int geometry().x()` и `int geometry().y()` возвращают координаты левого верхнего угла клиентской части виджета, `int width()` и `int heigth()` соответственно ширину и высоту клиентской части. Изменить расположение виджета можно с помощью метода `void move(int x, int y)`, ширину и высоту — `void resize ( int w, int h )` или все параметры одновременно — `void setGeometry ( int x, int y, int w, int h )`. В этих методах параметрами `x` и `y` считаются координаты верхнего левого угла клиентской части виджета, а `w` и `h` — ширина и высота клиентской части.

Расстояние от рамки до содержимого виджета можно задать методом `void setContentsMargins (int left, int top, int right, int bottom)`.

Во всех виджетах есть возможность обработки событий клавиатуры и мыши. Эти события — ещё один способ организации связи между объектами и действиями пользователя.

Когда виджет используется как контейнер для группы дочерних виджетов, то это называется композицией виджетов. Чаще всего в качестве контейнеров используются виджеты классов `QMainWindow` (главное окно), `QFrame` (рамка), `QDialog` (диалог). Для размещения виджетов внутри контейнера в `QT` разработаны специальные классы двух типов:

- менеджеры компоновки, базирующиеся на классе `QLayout`,
- менеджеры, базирующиеся на классе `QFrame`.

Менеджер — это контейнер виджетов, который автоматически определяет оптимальный размер, содержащихся в нём элементов, и их расположение, кроме того, при изменении размера окна он автоматически приводит в соответствии размеры и расположение находящихся в нём виджетов.

Самый легкий способ задания правильного расположения виджетов состоит в использовании встроенных менеджеров компоновки, иногда их называют лейаут–менеджерами. Все классы лейаут–менеджеров определены в заголовочном файле `qlayout.h`. Эти классы являются наследниками абстрактного класса `QLayout`, который, в свою очередь, происходит от `QObject`. Они берут на себя заботы об управлении геометрией и расположении виджетов. Для создания более сложных компоновок, можно помещать менеджеры компоновок друг в друга.

Общая иерархия менеджеров компановок выглядит следующим образом:

Чтобы добавить виджет в менеджер используется метод `void addWidget ( QWidget * w )`. После помещения в менеджер всех элементов, сам менеджер должен быть размещён в дру-

гом менеджере или виджете. Для добавления менеджера в другой менеджер используется метод `addLayout ( QLayout * layout)`. Для помещения менеджера в виджет используется метод `void QWidget::setLayout ( QLayout * layout )`, важно знать, что таким образом в виджет может быть помещён только один менеджер (но он может служить контейнером для других менеджеров).

Чаще всего используются менеджеры:

- `QHBoxLayout` — менджер горизонтальной компановки, все компоненты которого располагаются в один ряд по горизонтали слева направо:

```
#include<QApplication>
#include<QWidget>
#include<QPushButton>
#include<qlayout>
int main(int argc,char *argv[]){
   QApplication app(argc,argv);
   QWidget *window = new QWidget;
   QPushButton *button1 = new QPushButton("One");
   QPushButton *button2 = new QPushButton("Two");
   QPushButton *button3 = new QPushButton("Three");
   QPushButton *button4 = new QPushButton("Four");
   QPushButton *button5 = new QPushButton("Five");

   QHBoxLayout *layout = new QHBoxLayout;
   layout->addWidget(button1);
   layout->addWidget(button2);
   layout->addWidget(button3);
   layout->addWidget(button4);
   layout->addWidget(button5);
   window->setLayout(layout);
   window->show();
   return app.exec();
}
```

- `QVBoxLayout` — менджер вертикальной компановки, все компоненты которого располагаются в один столбик по вертикали сверху вниз, код программы, демонстрирующей работу менеджера , аналогичен предыдущему примеру, за исключением строки создания менеджера:

```
QVBoxLayout *layout = new QVBoxLayout;
```

- `QGridLayout` — располагает виджеты в двумерной таблице, при этом виджеты могут занимать несколько ячеек, для того чтобы определить положение виджета в таблице, при его помещении в менеджер надо указать соответствующий номер строки и столбца (`void addWidget (QWidget * widget, int row, int column)`) или номера строк и столбцов, определяющих диапазон ячеек таблицы, который займёт виджет (`void addWidget (QWidget * widget, int fromRow, int fromColumn, int rowSpan, int columnSpan)`):

```
QGridLayout *layout = new QGridLayout;
layout->addWidget(button1, 0, 0);
layout->addWidget(button2, 0, 1);
layout->addWidget(button3, 1, 0, 1, 2);
layout->addWidget(button4, 2, 0);
layout->addWidget(button5, 2, 1);
```

Для горизонтального или вертикального размещения виджетов можно воспользоваться и классом `QBoxLayout`, передав в его конструктор вторым параметром одно из следующих значений:

- `LeftToRight` — горизонтальное размещение слева направо;
- `RightToLeft` — горизонтальное размещение справа налево;
- `TopToBottom` — верткальное размещение сверху вниз;
- `BottomToTop` — верткальное размещение снизу вверх;

Ещё одним важным элементом размещения является возможность добавления в менеджеры горизонтального и вертикального размещения между виджетами невидимой «пружинки» (метод `void addStretch (int stretch = 0 )`). Иначе эта «пружинка» называется фактором растяжения. Добавление такого элемента позволяет сдвинуть или раздвинуть виджеты в нужную сторону, например, чтобы сдвинуть кнопки вверх можно добавить «пружинку» в качестве последнего элемента:

```
QVBoxLayout *layout = new QVBoxLayout;
layout->addWidget(button1);
layout->addWidget(button2);
```

```
layout->addStretch();
```

Фактор растяжения может быть вторым параметром метода `addWidget (QWidget *widget, int stretch=0)`, если он установлен, то элементы располагаются в менеджере пропорционально значению этого фактора.

Важно знать, что если дочерние виджеты помещаются в менеджер компоновки,то нельзя передавать указатель на родительский виджет в их конструктор. Менеджер автоматически установит родителя виджетов (используя метод `QWidget::setParent()`), так, чтобы они стали дочерними виджемами по отношению к виджету, на котором расположен этот менеджер. Таким образом, компонуемые виджеты являются дочерними виджетами по отношению к виджету, на котором расположен менеджер, а не по отношению к самому менеджеру.

Рассмотрим пример использования менеджеров компоновки для размещения в окне приложения виджетов, с помощью которых можно указать возраст человека.

```
#include<QApplication>
#include<QWidget>
#include<QLabel>
#include<QSlider>
#include<QSpinBox>
#include<qlayout>

int main(int argc,char *argv[]){
   QApplication app(argc,argv);
   QWidget *window=new QWidget;
   window->setWindowTitle("Enter your age");
   QSpinBox *spinbox=new QSpinBox;
   spinbox->setRange (0,130);
   spinbox->setValue(35);
   QSlider *slider=new QSlider;
   slider->setRange (0,130);
   QObject::connect(spinbox,SIGNAL(valueChanged(int)),slider,SLOT(setValue(int)));
   QObject::connect(slider,SIGNAL(valueChanged(int)),spinbox,SLOT(setValue(int)));
   QLabel label1("You age:");
   QLabel label2("35");
   QObject::connect(slider,SIGNAL(valueChanged(int)),&label2,SLOT(setNum(int)));
   QVBoxLayout *leftlayout = new QVBoxLayout;
   leftlayout->addWidget(spinbox);
   leftlayout->addWidget(slider);
   QVBoxLayout *rightlayout = new QVBoxLayout;
   rightlayout->addWidget(&label1);
   rightlayout->addWidget(&label2);
   rightlayout->addStretch();
   QHBoxLayout *mainlayout = new QHBoxLayout;
   mainlayout->addLayout(leftlayout);
   mainlayout->addLayout(rightlayout);
   window->setLayout(mainlayout);
   window->show();
   return app.exec();
}
```

В этом примере сначала создаются и настраиваются виджеты: окошко счётчика `spinbox` и ползунок `slider`. Затем они связываются друг с другом, чтобы их значения соответствовали друг другу при изменении одного из виджетов. Далее создаются две метки, вторая связывается с ползунком таким образом, чтобы актуально отображать вводимый возраст. Счётчик и ползунок помещаются в менеджер вертикальной компановки (`leftlayout`), метки в другой такой же менеджер (`rightlayout`). Оба менеджера, в свою очередь, помещаются в менеджер горизонтальной компановки (`mainlayout`).

Класс `QFrame` унаследован от класса `QWidget` и, в свою очередь, является препдком многих виджетов. Методы этого класса позволяют изменять стиль и внешний вид рамки (метод `void setFrameStyle (int style)`, его параметр задаётся с помощью специальных констант, соединяемых побитовой операцией или |), толщину рамки (методы `void setLineWidth (int)` и `void setMidLineWidth (int)`), например:

```
QFrame *frame=new QFrame;
```

```
frame->setFrameStyle(QFrame::Panel|QFrame::Sunken);
frame->setLineWidth(3);
```

Потомком класса QFrame является класс QSplitter. Этот класс–контейнер используется для одновременного порсмотра текстовых или графических объектов, так как он позволяет изменять размер, помещённых в него виджетов, во время работы приложения. Между виджетами располагается разделитель, горизонтальный или вертикальный, в зависимости от параметра, переданного в конструктор объекта. Разделитель можно перемещать во время работы программы кнопками мыши, при этом размеры виджетов, соответственно, уменьшаются или увеличиваются. Рассмотрим пример, в котором два окна для редактирования текста размещаются друг над другом, отделяемые горизонтальным разделителем:

```
QSplitter spl(Qt::Horizontal);
QTextEdit txt1;
QTextEdit txt2;
spl.addWidget(&txt1);
spl.addWidget(&txt2);
```

Другим потомком класса QFrame является класс QStackedWidget — стек виджетов, показывающий в текущий момент времени только одного из помещённых в него виджетов. Метод int addWidget (QWidget * widget) добавляет виджет в стек и возвращает его идентификационный номер. Слоты void setCurrentWidget (QWidget * widget) и void setCurrentIndex (int index) делают соответствующий виджет видимым.

Виджеты, являющиеся элементами отображения используются для демонстрации какой–либо информации. Эта информация обычно представляется в текстовом или графическом (например, картинки) виде.

Виджет надписи (класс QLabel) представляет собой текстовое поле, содержимое которого может изменять только само приложение, но не его пользователь. Изменить информацию, отображаемую на метке можно с помощью нескольких слотов, например, void setText(const QString &) меняет текст, void setPixmap(const QPixmap &) помещает на метку объект растрового изображения. В последнем случае, в приложении нужно создать объект класса QPixmap и проинициализировать его картинкой из соответствующего файла.

В приложении с графическим пользовательским интерфейсом обычно используется много изображений. Для работы с файлами, в том числе с графическими, в QT предлагается несколько механизмов:

- хранение информации в файлах и загрузка их во время выполнения приложения с использованием потоков данных;
- включение файлов специального формата XPM, совместимого с C++ в код программы;
- использование механизма определения ресурсов, предусмотренного в QT.

В качестве примера, рассмотрим работу механизма определения ресурсов для помещения изображения на метку. Для хранения используемых в приложении картинок обычно определяют отдельный подкаталог там же, где располагаются файлы с текстом программы. Назовём его, например, image. Поместим в него используемый файл, пусть он называется pic.jpg. Для применения системы ресурсов QT необходимо создать дополнительный файл ресурсов с расширением qrc (например, res.qrc). Он имеет формат XML:

```
<!DOCTYPE RCC><RCC version="1.0">
<qresource>
   <file>image/pic.jpg</file>
</qresource>
</RCC>
```

Затем, необходимо создать проект, напомним, что это можно сделать с помощью команды qmake -project. А затем, в получившийся проектный файл с расширением pro, надо добавить строку, задающую ресурсный файл:

```
RESOURCES=res.qrc
```

Тогда, в коде программы можно на метку поместить изображение из файла pic.jpg:

```
QPixmap pix;
pix.load(":/image/pic.jpg");
QLabel label;
label.resize(pix.size());
label.setPixmap(pix);
```

Обратите внимание, что при ссылке на ресурс (в методе load) указывается путь к файлу с префиксом пути :/. Ресурсами могут быть любые файлы, если в параметрах методов необходимо указывать имя файла, используемого для работы приложения.

Класс QLabel унаследован от класса QFrame, поэтому можно использовать соответствующие методы для изменения внешнего вида метки. Кроме того, класс QLabel позволяет размещать на метке текст в формате языка разметки HTML.

При помощи метода void setBuddy(QWidget* buddy) метка может быть связана с другим виджетом. Тогда. если текст метки содержит &, то символ, перед которым он поставлен будет подчёркнут, и при нажатии на клавиатуре этого символа в комбинации с клавишей Alt фокус перейдёт к виджету, с которым связана метка.

Другие полезные виджеты отображения — индикатор прогресса (класс QProgressBar) и и электронный индикатор (класс QLCDNumber). Индикатор прогресса незамения для отображения результатов работы длительных процессов, чтобы пользователю было понятно работает программа или она зависла. Электронный индикатор отображает числа с помощью сегментных указателей аналогично электронным часам, можно изменить стиль отображения и даже систему счисления.

Одним из самых важных элементов пользовательского интерфейса — кнопки. Предком всех кнопок является абстрактный класс QAbstractButton. В пользовательских приложениях применяются три вида кнопок: кнопки нажатия (класс QPushButton), кнопки–флажки (класс QCheckBox) и радиокнопки или кнопки–переключатели (класс QRadioButton).

Все кнопки могут содержать текст, который задаётся либо в конструкторе либо методом void setText(const QString& text) или изображение, устанавливаемое с помощью метода void setIcon(const QIcon& icon). Изменить размер картинки можно слотом void setIconSize(const QSize & size).

```
QPushButton button;
button.setIcon(QIcon(":/image/list.png"));
button.setIconSize(QSize(500,100));
```

Для взаимодействия с пользователем в классе QAbstractButton определены четыре сигнала:

- void clicked(bool checked = false) — генерируется при щелчке на кнопку;
- void pressed () — генерируется при нажатии на кнопку;
- void released () — генерируется при отпускании кнопки;
- void toggled(bool checked) — генерируется при изменении состояния кнопки, имеющей специальный статус выключателя, чтобы установить этот статус, надо вызвать метод void setCheckable(bool) с параметром true.

Для определения состояния кнопки и его изменения используются такие методы:

- bool isDown () const — возвращает true, если кнопка находится в нажатом состоянии, изменяется это состояние при нажатии на кнопку пользователем или при вызове метода void setDown(bool);
- bool isChecked()const — возвращает true, если кнопка находится во включённом состоянии, изменяется это состояние либо пользователь либо слот void setChecked(bool).

Кнопки класса QPushButton представляют из себя прямоугольник, содержащий надпись или (и) рисунок. Этот элемент используется, как правило, для выполнения определённой операции при нажатии на него.

Кнопки класса QCheckBox называются флажками и используются для выбора параметров работы приложения (если параметров больше пяти, то лучше воспользоваться виджетом списка QListWidget). Флажок состоит из маленького прямоугольника, при щелчке на виджете в прямоугольнике появляется отметка. Существует два типа флажков: обычные с двумя состояниями и флажки с неопределённым состоянием, которые могут находится в трёх состояниях: включённом выключенном и неопределённом. Обычный флажок переводится в режим поддержки третьего состояния вызовом метода void setTristate(bool y = true) с параметром true. Неопределённое состояния устанавливается методом void setCheckState(Qt::CheckState state), в который передаётся специальный параметр Qt::PartiallyChecked.

Переключатель или радио–кнопка (класс QRadioButton) представляет собой виджет, который может находиться в одном из двух состояний. Эти состояния может устанавливать пользователь во время работы приложения. Виджеты переключателей не могут использоваться отдельно, и обязательно должны быть сгруппированы вместе, так как по смыслу их работы, нажатие на одну из группы радио–кнопок приводит к отключению другой, нажатой ранее. Сгруппировать переключатели можно с помощью менеджеров или с помощью виджетов–контейнеров, например, объектов класса QGroupBox.

Группа виджетов, включающая в себя элементы ввода, обеспечивает пользователю возможность ввода и редактирования данных. Большая часть элементов ввода может работать с буфером обмена, поддерживает технологию перетаскивания (drag&drop — перетащить и оставить). Текст можно выделять с помощью мыши, клавиатуры и контекстного меню.

Наиболее простой элемент ввода — однострочное текстовое поле QLineEdit. Виджет представляет собой прямоугольную область для ввода одной строки текста, поэтому его следует использовать только

для ввода небольшой по объёму информации. Рассмотрим основные методы, слоты и сигналы, доступные объектам этого класса.

Методы:

- `QString text () const` возвращает строку, содержащуюся в виджете;
- `void setMaxLength (int)` ограничивает количество вводимых символов заданным значением;
- `QString selectedText () const` возвращает строку, содержащую выделенный текст, если текст не выделен, то пустую;
- `void setReadOnly (bool)` устанавливает для виджета режим «только для чтения», если параметром является значение `true`;
- `void setEchoMode (EchoMode)` задаёт различные режимы ввода символов, в зависимости от флага, указанного в качестве параметра:
  - флаг `QLineEdit::Password`, устанавливает режим ввода пароля, при котором вместо вводимых символов отображаются `*`,
  - флаг `QLineEdit::NoEcho`, определяет режим, при котором вводимые символы не отображаются, этот режим применяется тогда, когда длина пароля является секретом,
  - флаг `QLineEdit::PasswordEchoOnEdit` задаёт режим, при котором вводимые символы отображаются только во время ввода и редактирования, в остальное время вместо них отображаются звёздочки,
  - флаг `QLineEdit::Normal` установлен по умолчанию и задаёт обычный режим ввода символов с их отображением в виджете.

Слоты:

- `void setText ( const QString & )` изменяет строку в виджете;
- `void clear ()` очищает строку;
- `void copy () const` копирует выделенную часть строки в буфер обмена;
- `void cut ()` вырезает выделенную часть строки и копирует её в буфер обмена;
- `void paste ()` вставляет текст из буфера обмена;
- `void undo ()` отменяет последнее изменение;
- `void redo ()` повторяет последнее действие.

Сигналы:

- `void textChanged ( const QString & text )` высылается, если содержимое виджета изменилось;
- `void returnPressed ()` высылается, если пользователь нажал кнопку `Enter`;
- `void selectionChanged ()` высылается, если в строке выделен текст;

Для ввода многострочного текста используются виджеты класса `QTextEdit`. Этот класс позволяет осуществлять просмотр и редактирование обычного текста и текста в формате языка разметки `HTML`. Для установки текста в формате `HTML` используется слот `void setHtml ( const QString & text )`. Ряд элементов класса совпадает с аналогичными методами класса `QLineEdit`, но другие методы существенно обогащают функциональность класса `QTextEdit`.

Виджет этого класса может быть использован, как текстовый редактор, так как у него есть методы, позволяющие изменить шрифт, копировать, удалять, вставлять текст и т.п. Выделенный текст может быть обработан с помощью класса `QTextCursor`. Объект этого класса возвращает метод `QTextCursor textCursor () const`. Для работы со сложным текстом используются объекты класса `QTextDocument`. Каждый виджет класса `QTextEdit` содержит такой объект, получить на него ссылку можно методом `QTextDocument * document () const`. Изменить содержимое виджета можно, передав в него другой объект класса `QTextDocument` с помощью метода `void setDocument ( QTextDocument * document )`.

Другими элементами ввода являются счётчики:

- `QSpinBox` — виджет для ввода целых чисел;
- `QDoubleSpinBox` — виджет для ввода действительных чисел;
- `QDateTimeEdit` — виджет для ввода даты и времени.

Одной из важнейших задач, решаемых программистом, является проверка данных вводимых пользователем. `QT` предоставляет возможность такой проверки с помощью механизма, использующего специальные классы. Их предком является абстрактный класс `QValidator`. Для проверки ввода чисел используются его потомки классы `QIntValidator` и `QDoubleValidator`. Для проверки ввода целых чисел в конструктор объекта класса `QIntValidator` передаются границы диапазона допустимых значений и обязательно указатель на родительский объект:

```
QIntValidator ( int minimum, int maximum, QObject * parent )
```

У объектов класса `QDoubleValidator` в конструктор, кроме границ диапазона допустимых значений и указателя на родительский объект, передаётся число, определяющее количество цифр после запятой:

`QDoubleValidator::QDoubleValidator (double bottom, double top, int decimals, QObject * parent )`

Кроме проверки чисел в `QT` реализован класс, проверяющий правильность регулярных выражений `QRegExpValidator`.

Регулярное выражение — это некоторый шаблон, описывающий структуру строки символов по определённым правилам. При этом используются обычные символы и, так называемые, метасимволы, позволяющие определить различные сложные свойства строки.

Метасимвол(ы)	Значение
.	любой символ, кроме ограничителей строки (например «\n»)
^	начало строки
$	конец строки
[ ]	один из заданного набора символов
[^]	любой символ, не входящий в заданный набор
–	определяет диапазон допустимых символов
*	предыдущий символ должен встретиться в строке 0 или более раз
+	предыдущий символ должен встретиться в строке 1 или более раз
?	предыдущий символ должен встретиться в строке 0 или 1 раз
{n}	предыдущий символ должен встретиться в строке ровно n раз
{n,}	предыдущий символ должен встретиться в строке не менее n раз
{,n}	предыдущий символ должен встретиться в строке не более n раз
( )	группирует и сохраняет символы, соответствующие некоторому регулярному выражению
|	выбор одного из двух регулярных выражений

Приведём несколько примеров регулярных выражений:

Регулярное выражение	Значение
ab1	последовательность символов «ab1»
ab|cde	последовательность символов «ab» или «cde»
[ab]	символ «a» или «b»
[A-Z]	любая латинская буква в верхнем регистре
[A-Za-z0-9]	любая латинская буква или цифра
[^ 0-9]	любой символ кроме цифр
a{3}.$	строка из трёх букв «a», заканчивающаяся любым символом

В `QT` есть специальный класс для работы с регулярными выражениями `QRegExp`.

Чтобы проверить данные вводимые пользователем в строку `QLineEdit`, необходимо для соответствующего виджета установить нужный объект класса `QValidator` или его потомка. Сделать это можно с помощью метода `void setValidator ( const QValidator * v )`. Рассмотрим пример, где в виджет можно вводить только один символ — латинскую букву или цифру. Для этого создаётся соответствующее регулярное выражение, которое затем используется при создании объекта проверки ввода класса `QRegExpValidator`.

```
   QLineEdit *edit=new QLineEdit;
 //  QRegExp rx("^[A-Z][a-z]*");
    QRegExp rx("[(0-9)|(A-F)|(a-f)]");
   QRegExpValidator *v=new QRegExpValidator(rx,0);
   edit->setValidator(v);
```

Если необходимо создать свой класс проверки, то необходимо переопределить виртуальный метод `virtual State validate ( QString & input, int & pos ) const`. Аргументами метода является вводимая строка и позиция курсора. Метод должен возвращать одно из трёх значений:

- `QValidator::Invalid` — если строка не может быть принята;
- `QValidator::Acceptable` — если строка допустима;
- `QValidator::Intermediate` — если строка является частью допустимой строки.

Рассмотрим пример создания шестнадцатеричного счётчика. Определим сначала класс для проверки ввода шестнадцатеричных чисел. Кроме регулярного выражения, для проверки вводимой строки используется метод `bool contains ( const QRegExp & rx ) const` класса `QString`, который возвращает `true`, если строка содержит заданное регулярное выражение:

```
#include<QValidator>
#include<QRegExp>
#include<QString>

class Validator16 : public QValidator{
public:
```

```
   Validator16(QObject *p):QValidator(p){}
   virtual State validate ( QString & str, int & pos ) const {
       QRegExp rx("[^0-9A-Fa-f]");
       if(str.contains(rx)||str.size()>3)
            return Invalid;
       return Acceptable;
   }
};
```

Класс счетчика является наследником обычного виджета–счётчика. В нём переопределяется метод `validate()`, который автоматически вызывается при изменении значения счётчика во время работы приложения. Второй переопределённый виртуальный метод `virtual int valueFromText ( const QString & text ) const` выполняет преобразование строки, введённой пользователем, в число. Третий переопределённый метод `virtual QString textFromValue ( int value ) const` осуществляет обратное преобразование числа в строку для вывода на экран.

```
#include<QSpinBox>
#include"v.h"

class Counter : public QSpinBox{
    Q_OBJECT
private:
Validator16 *val;
public:
Counter(QWidget *p=0):QSpinBox(p){
setRange(0,4095);
val=new Validator16(this);
        }
protected:
QValidator::State validate ( QString & str, int & pos ) const{
return val->validate(str,pos);
}
int valueFromText ( const QString & str ) const{
return str.toInt(0,16);
}
QString textFromValue ( int value ) const{
return QString::number(value,16).toUpper();
}
};
```

В основной функции `main` просто создаётся экземпляр класса шестнадцатеричного счётчика:

```
#include<QApplication>
#include"s.h"

int main(int argc,char *argv[]){
   QApplication app(argc,argv);
   Counter x;
   x.show();
   return app.exec();
}
```

-