**1.Классы. Отличия классов от структур. Модификаторы доступа. Поля и методы, понятие инкапсуляции.**

Отиличия классов от структур

Классы - это абстракция описывающая методы, свойства, ещё не существующих объектов. В С++ *классом* называются типы объявленные с помощью class , struct или union .

Отличий class от struct всего два:

Member access control [class.access]

Члены класса, определенного с помощью ключевого слова class , по умолчанию являются private .

Члены класса, определенного с помощью ключевого слова struct или union , по умолчанию являются

public .

Accessibility of base classes and base class members [class.access.base]

При отсутствии спецификатора доступа (т.е. private/protected/public) у базового класса, базовый класс будет public если класс определен с помощью struct и private если класс определен с помощью class .

Модификаторы доступа

public - доступ открыт всем другим классам, кто видит определение данного класса.

private - доступ открыт самому классу (т.е. функциям-членам данного класса) и друзьям (friend) данного класса - как функциям, так и классам. Однако производные классы не получают доступа к этим данным совсем. И все другие классы такого доступа не имеют.

protected - оступ открыт классам, производным от данного. То есть, производные классы получают свободный доступ к таким свойствам или метода. Все другие классы такого доступа не имеют.

Поля и методы, понятие инкапсуляции

Поля — это любые данные, которыми можно характеризовать объект класса.

Методы — это функции, которые могут выполнять какие-либо действия над данными (полями) класса.

Инкапсуляция — это свойство, позволяющее объединить в классе и данные, и методы, работающие с ними и скрыть детали реализации от пользователя.

**2.Конструкторы. Конструктор по умолчанию. Конструктор копирования, его сигнатура и схема реализации.**

Конструктор — это особый тип метода класса, который автоматически вызывается при создании

объекта этого же класса. Конструкторы обычно используются для инициализации переменных-членов класса значениями, которые предоставлены по умолчанию/пользователем.

Конструкторы имеют определенные правила их именования: конструкторы всегда должны иметь то же имя, что и класс; конструкторы не имеют типа возврата (даже void).

Конструкторы по умолчанию. Конструктор, который не имеет параметров (или содержит параметры, которые все имеют значения по умолчанию. Вызывается, если пользователем не указаны значения для инициализации.

Конструктор копирования — это особый тип конструктора, который имеет 1 параметр - ссылка на объект данного типа класса/структуры. Вызывается авто компилятором при необход. создать копию

имеющегося объекта (может вызываться и вручную), использует почленную инициализацию. Существует четыре случая вызова конструктора копирования:

Когда объект является возвращаемым значением

Когда объект передается (функции) по значению в качестве аргумента

Когда объект конструируется на основе другого объекта (того же класса)

Если в классе не объявлен конструктор копирования, то используется конструктор копирования, который автоматически генерируется компилятором. Этот конструктор копирования реализует

побитовое копирование для получения копии объекта - это приемлемым для классов, в которых нет динамического выделения памяти. Однако, если в классе есть динамическое выделение памяти (класс

использует указатели), то побитовое копирование приведет к тому, что указатели обоих объектов будут указывать на один и тот же участок памяти. А это ошибка.

**3.Правила генерации компилятором конструкторов.**

**Деструкторы. Ключевое слово this и пример, когда оно нужно. Списки инициализации и пример, когда они необходимы.**

**Явные конструкторы, ключевое слово explicit, пример.**

Правила генерации компилятором конструкторов.

Конструктор копирования генерируется, если все базовые классы и члены допускают копирование

Конструктор по умолчанию без параметров создаётся, если класс не определяет никакого конструктора

Оператор присваивания T &T::operator=(const T &) генерируется, если все базовые классы и члены допускают копирование

Деструктор всегда автоматически генерируется, если это возможно

Деструкторы.

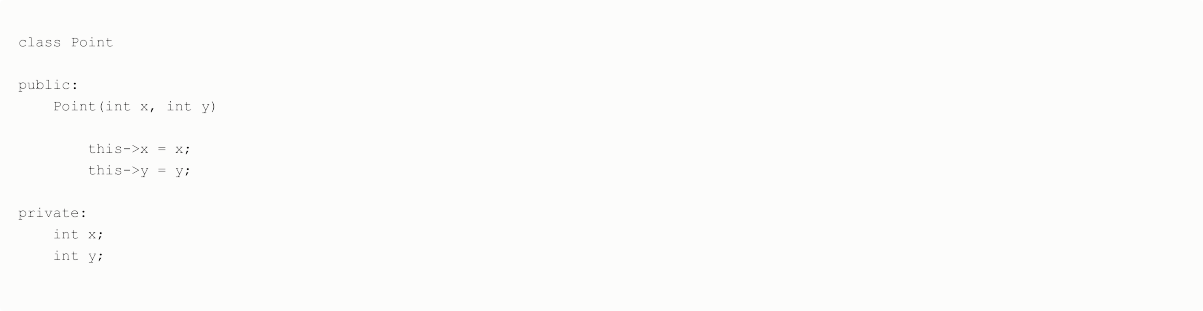
Деструктор также является специальной функцией-членом, такой как конструктор. Деструктор уничтожает объекты класса, созданные конструктором. Деструктор имеет то же имя, что и имя их класса, которому предшествует тильда (~).

Невозможно определить более одного деструктора. Деструктор — это единственный с, созданный конструктором. Следовательно, деструктор не может быть перегружен. Деструктор не требует никаких

аргументов и не возвращает никакого значения. Он вызывается автоматически, когда объект выходит за пределы области видимости. Деструктор освобождает место в памяти, занятое объектами, созданными конструктором. В деструкторе объекты уничтожаются в обратном порядке создания объекта.

Ключевое слово this и пример, когда оно нужно.

Ключевое слово this представляет указатель на текущий объект данного класса. Соответственно через this мы можем обращаться внутри класса к любым его членам.

Пример:

{

{

}

};

Списки инициализации и пример, когда они необходимы.

Список инициализации членов находится сразу же после параметров конструктора. Он начинается с двоеточия ( : ), а затем значение для каждой переменной указывается в круглых скобках. Больше не нужно выполнять операции присваивания в теле конструктора.

Списки инициализации членов позволяют инициализировать члены, а не присваивать им значения. Это единственный способ инициализации констант и ссылок, которые являются переменными-членами вашего класса. Во многих случаях использование списка инициализации может быть более

результативным, чем присваивание значений переменным членам в теле конструктора. Списки инициализации работают как с переменными фундаментальных типов данных, так и с членами, которые сами являются классами.



{

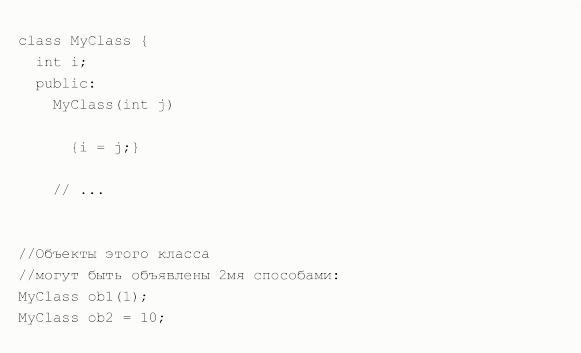
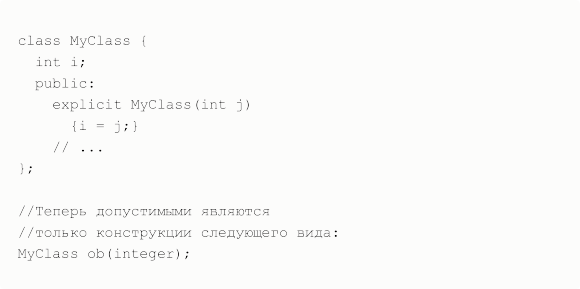
{

}

};

Явные конструкторы, ключевое слово explicit, пример.

По умолчанию язык C++ обрабатывает любой конструктор, как оператор неявного преобразования. Явные конструкторы (с ключевым словом explicit) не используются для неявных конвертаций.



};

**4.Статические поля и методы, пример. Локальные статические переменные.**

Статические поля и локальные статические переменные.

Статические поля - представляют собой обычную переменную, которая создаётся в момент запуска программы.

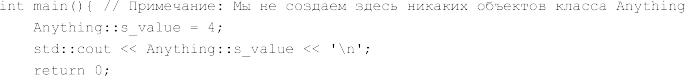
Правила видимости такие же, как и для обычных полей.

Переменные-члены класса можно сделать статическими, используя ключевое слово static. В отличие от обычных переменных-членов, статические переменные-члены являются общими для всех объектов класса. Они создаются при запуске программы и уничтожаются, когда программа завершает свое выполнение. Следовательно, статические члены принадлежат классу, а не объектам этого класса. Также можно получить доступ к статическим членам через разные объекты класса, а также осуществляется напрямую через имя класса и оператор разрешения области видимости, но статические члены

существуют, даже если объекты класса не созданы!

Определение и инициализация статических переменных-членов класса. Когда объявляется static переменная член внутри тела класса, то компилятору сообщается о существовании статической

переменной-члене, но не о ее определении. Поскольку статические переменные-члены не являются частью отдельных объектов класса, то должны явно определить статический член вне тела класса — в глобальной области видимости.



{

};

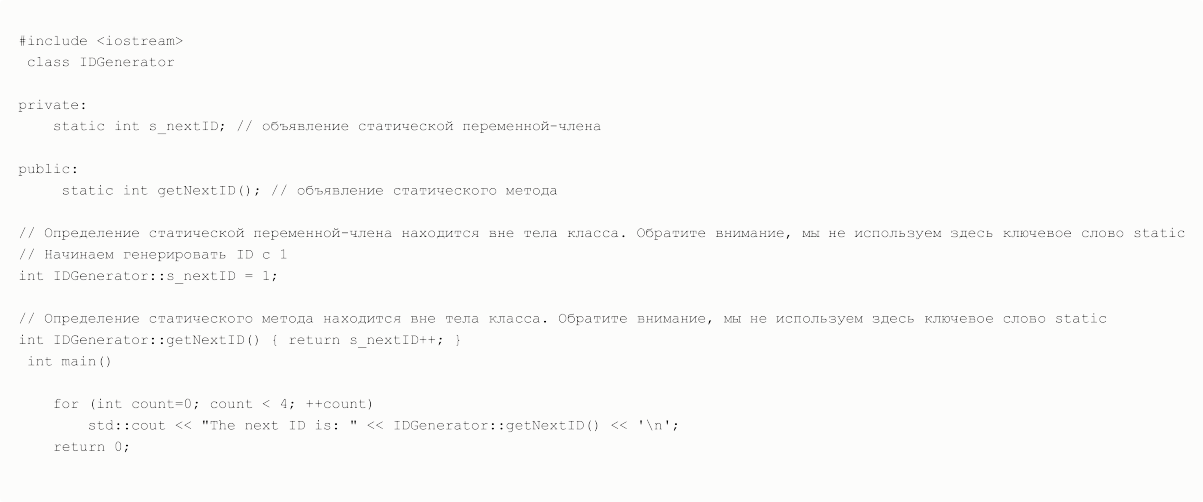
}

Зачем использовать статические переменные-члены внутри классов? Для присвоения уникального идентификатора каждому объекту класса (как вариант).

Статические методы. Не привязаны к какому-либо одному объекту класса, а значит они не имеют скрытого указателя \*this (т.к. он указывает всегда на объект, с которым работает метод, а статич-ие

методы могут не работать через объекты) и + могут напрямую обращаться к другим статическим членам (переменным или функциям), но не могут напрямую обращаться к не статическим членам

(нестатические члены принадлежат объекту класса, а статические методы — нет!). Можно определять вне тела класса. Могут принимать объект, в качестве параметра или создавать объект данного класса.



{

};

{

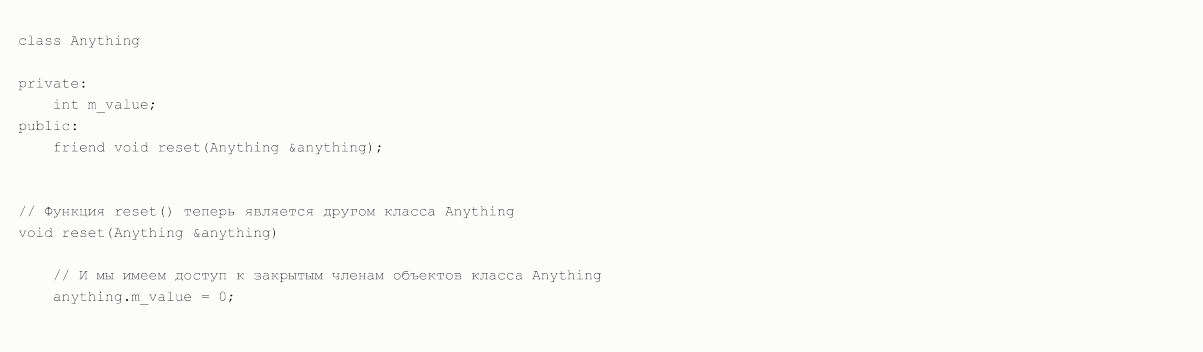
}

**5.Дружественные методы и классы, пример. Константные и не константные методы.**

Дружественные методы и классы, пример.

Дружественная функция — это функция, которая имеет доступ к закрытым членам класса, как если бы она сама была членом этого класса. Во всех других отношениях дружественная функция является

обычной функцией. Ею может быть, как обычная функция, так и метод другого класса. Для объявления дружественной функции используется ключевое слово friend перед прототипом функции, которую вы хотите сделать дружественной классу.



{

};

{

}

Один класс может быть дружественным другому классу. Это откроет всем членам первого класса доступ к закрытым членам второго класса.



{

};

{

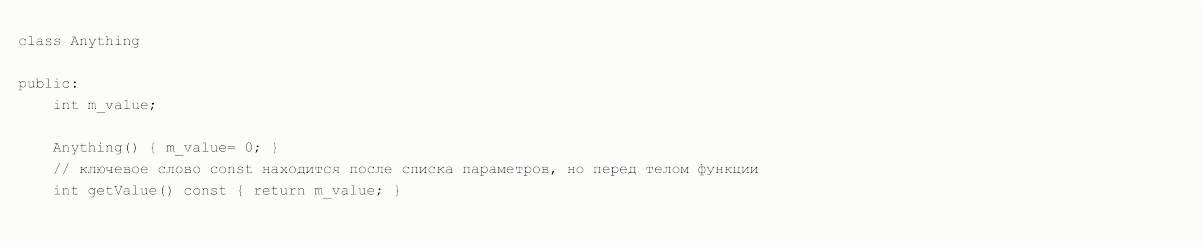
{

}

};

Константные и не константные методы.

Константный метод — это метод, который гарантирует, что не будет изменять объект или вызывать неконстантные методы класса (поскольку они могут изменить объект).



{

};

**6.Перегрузка операторов. Какие операторы нельзя перегружать. Чего можно и чего нельзя добиться перегрузкой операторов.**

**Синтаксис перегрузки арифметических операторов.**

Перегрузка операторов.

Перегрузка операторов позволяет определить действия, которые будет

выполнять оператор. Перегрузка подразумевает создание функции, название которой содержит слово operator и символ перегружаемого оператора. Функция оператора может быть определена как член класса, либо вне класса.

Какие операторы нельзя перегружать.

Следующие операторы перегружать нельзя:

?: (тернарный оператор);

:: (доступ к вложенным именам);

. (доступ к полям);



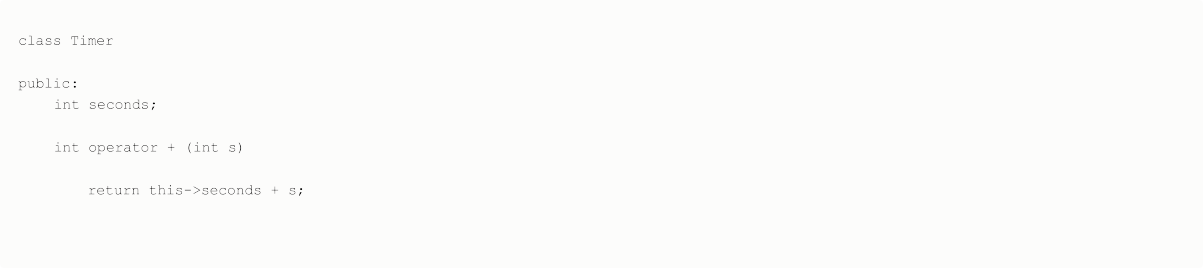
.\* (доступ к полям по указателю);

sizeof , typeid и операторы каста.

Чего можно и чего нельзя добиться перегрузкой операторов.  
Запрещено определять свои операторы (возможны проблемы с определением приоритетов).

Нельзя изменять приоритеты операторов

Перегружать операторы стоит тогда и только тогда, когда это имеет смысл. То есть если смысл перегрузки очевиден и не несет в себе скрытых сюрпризов.

Синтаксис перегрузки арифметических операторов.

**7.Перегрузка префиксного и постфиксного инкремента.**

**Перегрузка оператора "квадратные скобки", ее особенности.**

**Перегрузка оператора "круглые скобки", пример использования.**

{

{

}

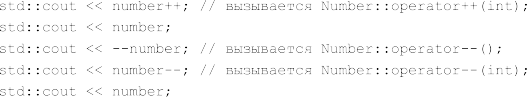
}

Перегрузка операторов инкремента и декремента. Поскольку операторы инкремента и декремента являются унарными и изменяют свои операнды, то перегрузку следует выполнять через методы класса. Версии префикс: аналогична перегрузке любых других унарных

операторов

Версии постфикс: что язык C++ использует фиктивную переменную (или *«фиктивный параметр»*) для операторов версии постфикс. Этот фиктивный целочисленный параметр используется только с одной

целью: отличить версию постфикс операторов инкремента/декремента от версии префикс.

{

};

{

}

{

}

{

}

{

}

{

}

{

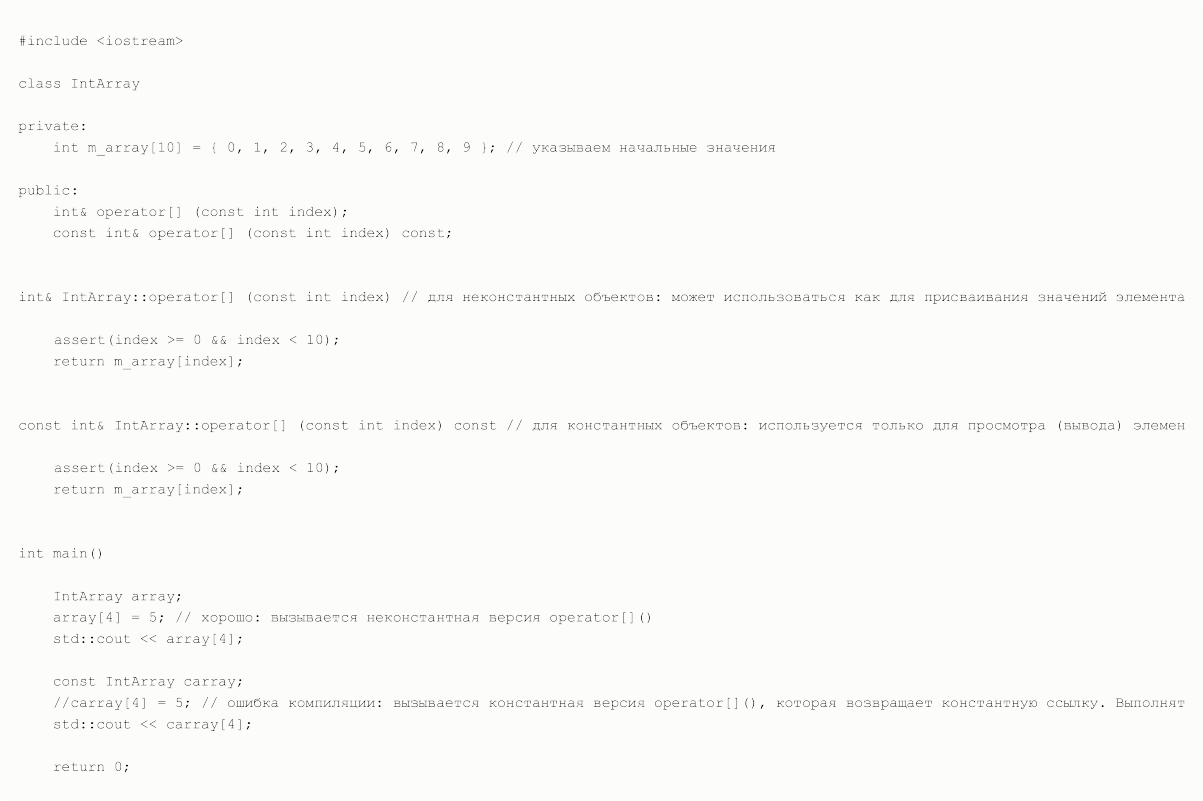
}



Перегрузка оператора индексации [], которого должна выполняться через метод класса и всегда будет принимать один параметр: значение индекса (элемент массива, к которому требуется доступ). Можно определить отдельно неконстантную и константную версии operator. Неконстантная версия будет

использоваться с неконстантными объектами, а версия const — с объектами const + преимуществом

ещё в том, что мы можем выполнять проверку передаваемых значений индекс. Передаваемый аргумент не обязательно должен быть целым числом.



{

};

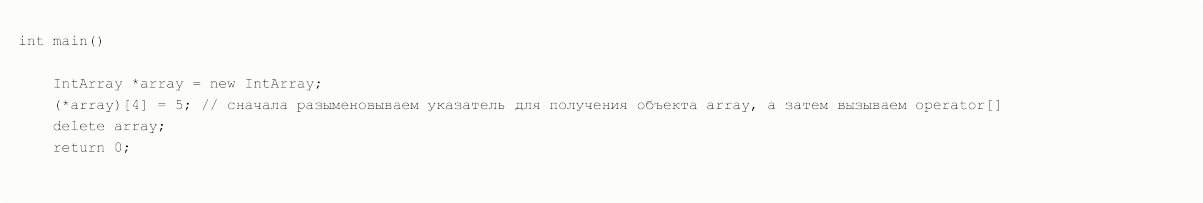
{

}

{

}

{

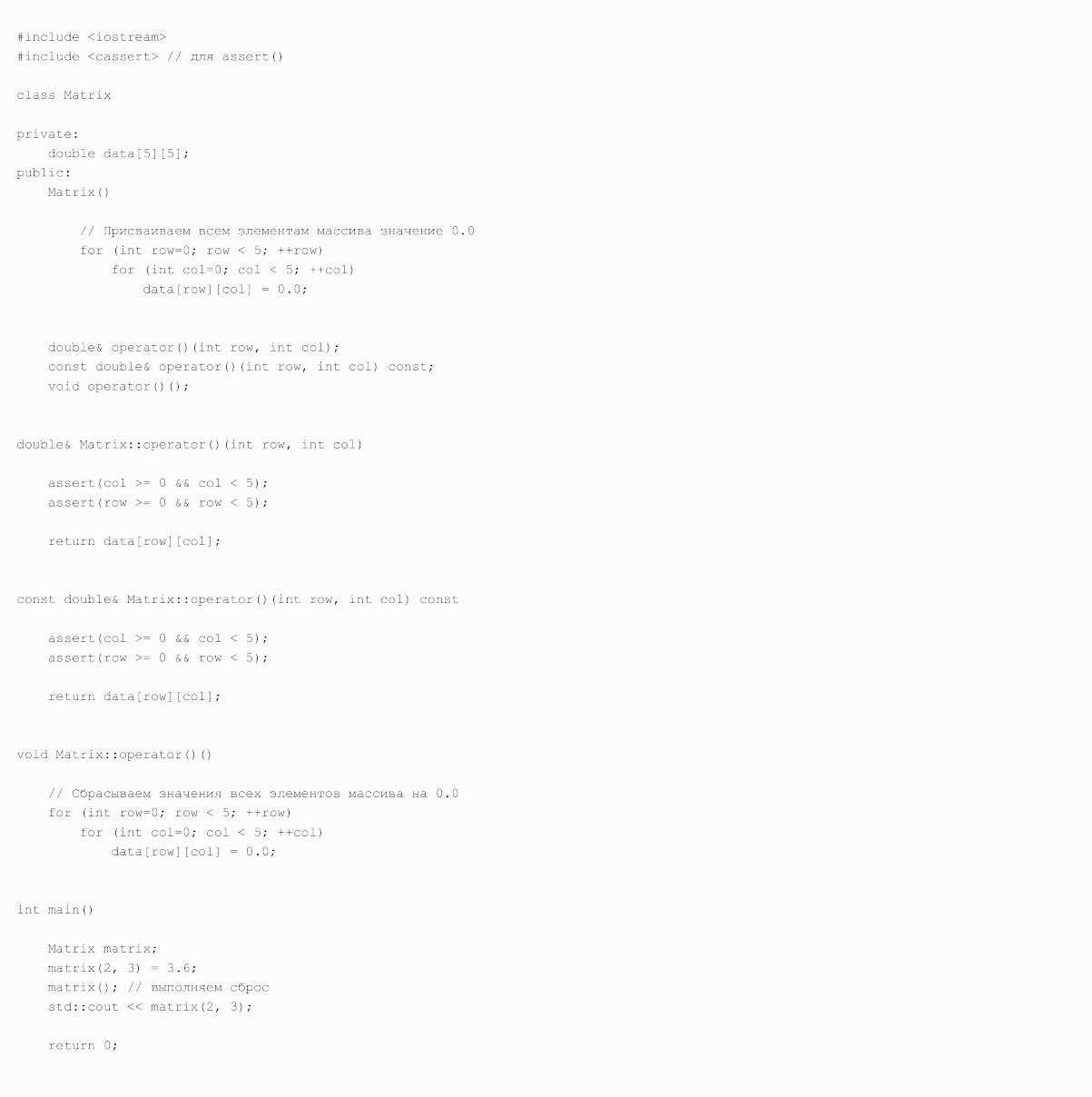
Указатели на объекты:

{

}

должна осуществляться через метод класса.

Перегрузка оператора () с двумя параметрами используется для получения доступа к двумерным массивам или для возврата подмножеств одномерного массива.



{

{

}

};

{

}

{

}

{

}

{

}



Перегрузка оператора () используется в реализации функторов (или «функциональных объектов») — классы, которые работают как функции ( + в том, что могут хранить данные в переменных-членах (поскольку они сами являются классами))



{

{

}

};

{



}

8.Перегрузка операторов приведения типа, ее особенности и недостатки, пример.  
Особенности перегрузки логических операторов и оператора "запятая".

Операторы преобразования (conversion operator) определяют преобразование объекта одного типа в





{

{

}

};

другой. Они имеют следующий общий синтаксис: operator тип() const ;

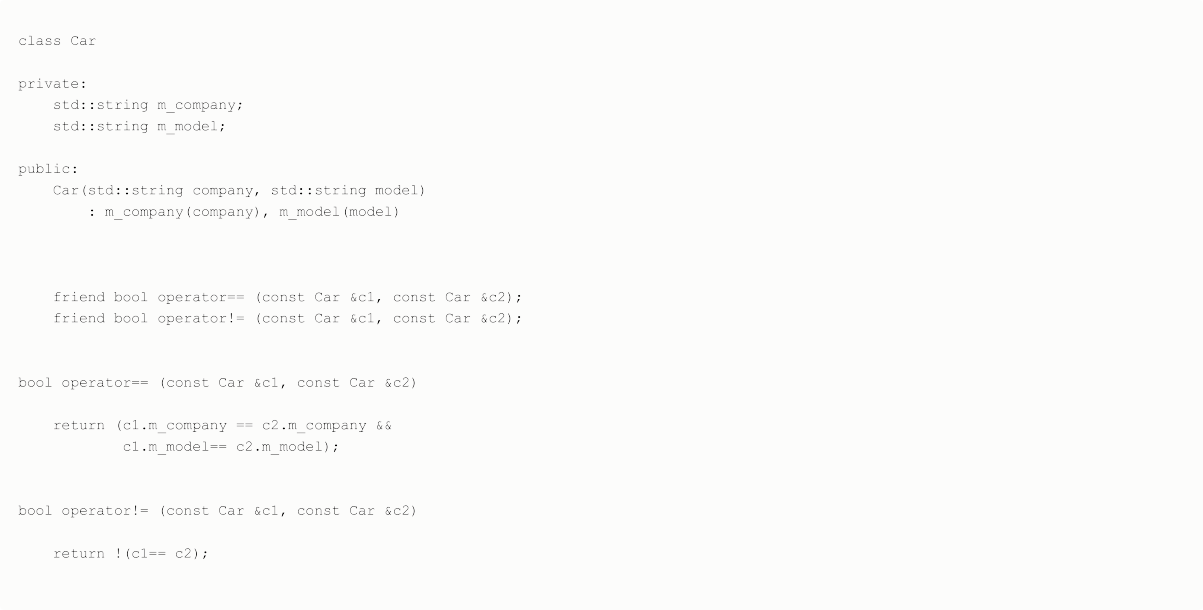
В качестве функции перегрузки используется метод operator int() . Обратите внимание, между

словом operator и типом, в который мы хотим выполнить конвертацию (в данном случае, тип int), находится пробел.

Функция перегрузки не имеет типа возврата. Язык C++ предполагает, что вы будете возвращать корректный тип.

Особенности перегрузки логических операторов и оператора "запятая".

Принципы перегрузки операторов сравнения те же, что и в перегрузке других операторов, которые мы рассматривали на предыдущих уроках. Поскольку все операторы сравнения являются бинарными и не изменяют свои левые операнды, то выполнять перегрузку следует через дружественные функции.



{

{

}

};

{

}

{

}



В случае стандартного использования оператора ‘,’ нужно учесть следующие особенности:

оператор ‘ , ‘ считается бинарным. Поэтому операторная функция operator,() получает один параметр;



**9.Понятие наследования. Синтаксис наследования, правила сокрытия имен при наследовании. Схема использования**

**объектов производного класса в местах, где требуются объекты базового класса.**

{

};

Понятие наследования.

Наследование — это инструмент, позволяющий описать новый класс на основе уже существующего

Синтаксис наследования, правила сокрытия имен при наследовании.

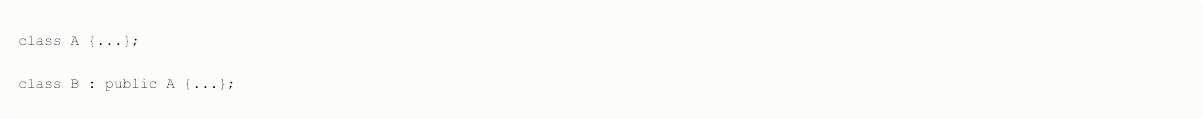
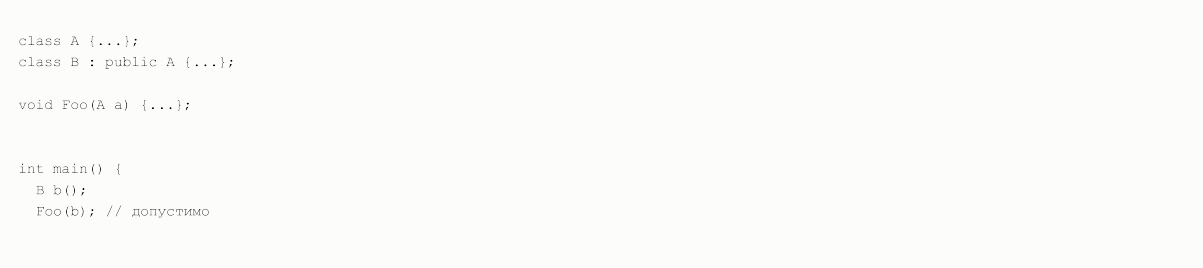




Схема использования объектов производного класса в местах, где требуются объекты базового класса.

В метод, принимающий экземпляр базового класса можно передавать экземпляры производного. В обратную сторону это не работает



}

**10.Порядок выполнения конструкторов и деструкторов при**

**наследовании, вызов конструктора родителя из конструктора потомка. Виды наследования: публичное, приватное,**

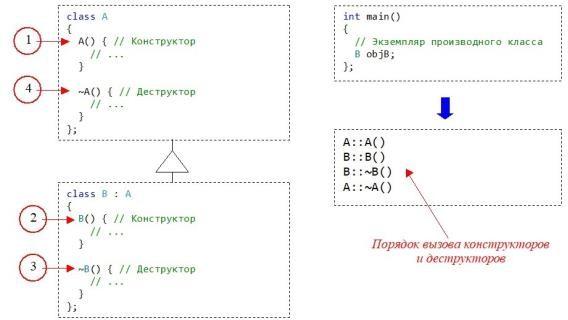
**защищенное. Пример использования приватного наследования.**

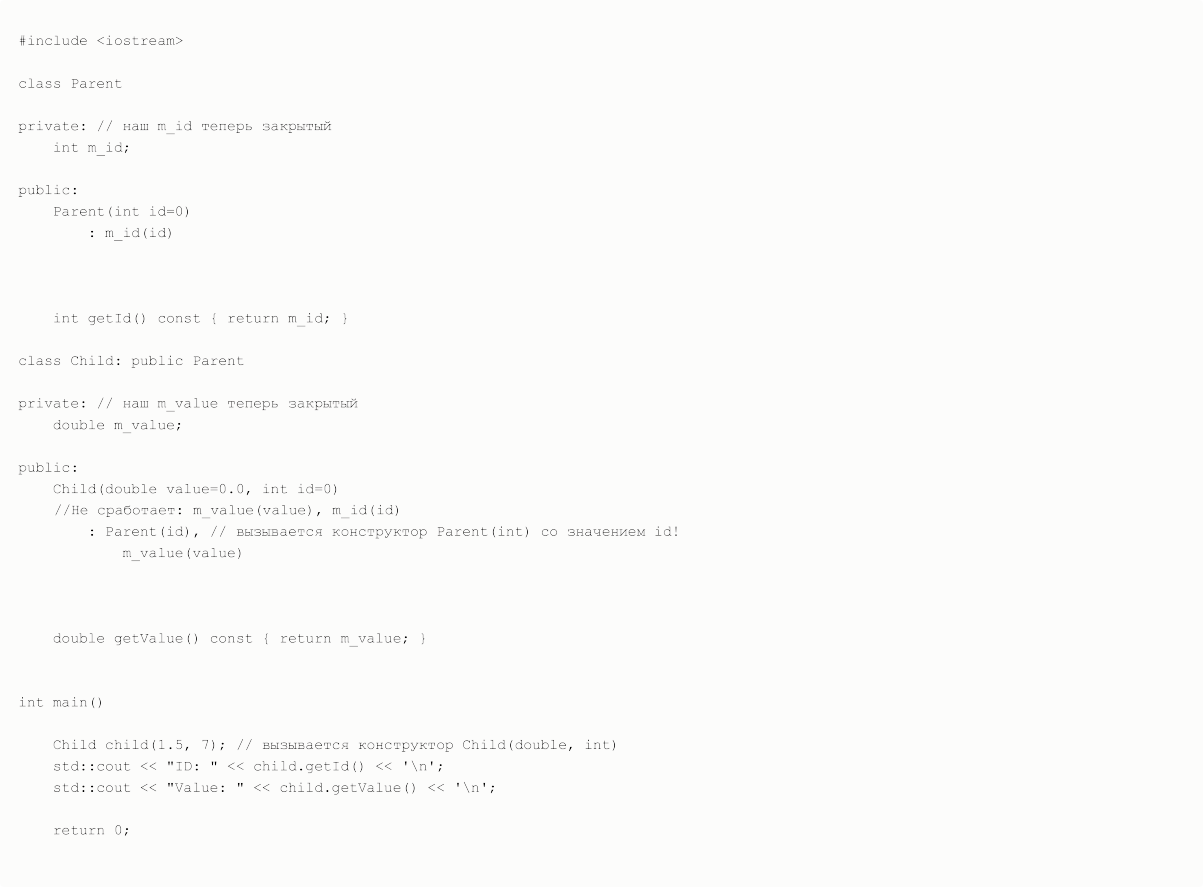
Порядок выполнения конструкторов и деструкторов при наследовании.

Если два класса образуют иерархию наследования, то при создании экземпляра производного класса сначала вызывается конструктор базового класса конструирующий объект производного класса. Затем этот конструктор становится недоступен и дополняется кодом конструктора производного класса (сначала происходит инициализация данных базового класса, затем инициализация данных

производного класса)

Если классы образуют иерархию, деструкторы этих классов вызываются в обратном порядке по отношению к вызову конструкторов. Сначала вызывается деструктор производного класса, затем вызывается деструктор базового класса.



Вызов конструктора родителя из конструктора потомка.

{

{

}

};

{

{

}

};

{

}

Виды наследования: публичное (открытое), приватное (закрытое), защищенное. Для определения типа наследования нужно просто указать нужное ключевое слово возле наследуемого класса:



{

};

{

};

{

};

{

};

Нужно помнить следующие правила: класс всегда имеет доступ к своим (не наследуемым)

членам;доступ к члену класса основывается на его спецификаторе доступа; дочерний класс имеет

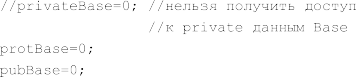
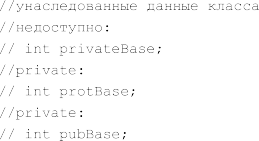
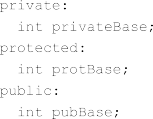
доступ к унаследованным членам родительского класса на основе спецификатора доступа этих членов в родительском классе.

Публичное (открытое): когда вы открыто наследуете родительский класс, то унаследованные public-

члены остаются public, унаследованные protected-члены остаются protected, а унаследованные private- члены остаются недоступными для дочернего класса. Ничего не меняется.

Приватное (закртытое): все члены родительского класса наследуются как закрытые. Это означает, что private-члены остаются недоступными, а protected- и public-члены становятся private в дочернем классе.

Внимание: это не влияет на то, как дочерний класс получает доступ к членам родительского класса! Это влияет только на то, как другими объектами осуществляется доступ к этим членам через дочерний класс.

Пример использования приватного наследования:

{

};

{

{

}

};

{

{



}

};

{

}

Закрытое наследование может быть полезно, когда дочерний класс не имеет очевидной связи с родительским классом, но использует его в своей реализации. В таком случае мы не хотим, чтобы

открытый интерфейс родительского класса был доступен через объекты дочернего класса (как это было, когда мы использовали открытый тип наследования).

Защищенное: с защищенным наследованием, public- и protected-члены становятся protected, а private- члены остаются недоступными.

**11.Множественное наследование. Проблема неоднозначности вызова метода из базовых классов и ее решение. Проблема ромбовидного наследования и ее решение.**

Множественное наследование.

Множественное наследование позволяет одному дочернему классу иметь несколько родителей.

Проблема неоднозначности вызова метода из базовых классов и ее решение.

Суть проблемы: Несколько родительских классов имеют метод с одним и тем же именем

Решение: Явно указывать какую версию метода мы хотим вызвать

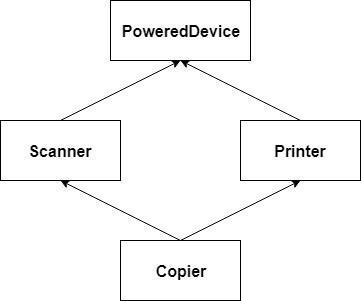
}

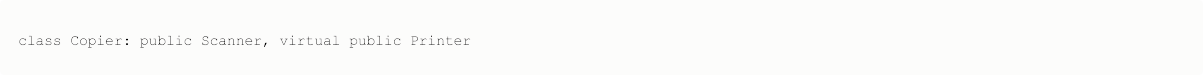
}

}

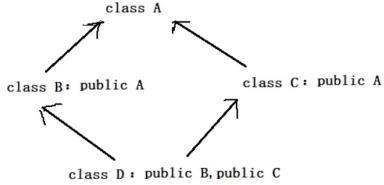
Проблема ромбовидного наследования и ее решение.

Суть пробемы: Это ситуация, когда один класс имеет 2 родительских класса, каждый из которых, в свою очередь, наследует свойства одного и того же родительского класса. Иллюстративно мы получаем форму ромба.



Решение: Использование виртуального наследования

12.Виртуальное наследование, его особенности.



Виртуальное наследование решает проблему избыточности данных и проблему неоднозначности.

Почему существует избыточность данных? - Прежде всего, класс B и класс C наследуют класс A,

поэтому класс B и класс C должны содержать переменную-член класса A, то есть есть одна в классе B int a и еще одну в классе C int a. Затем, когда класс D наследует класс B и класс C, тогда класс D должен

содержать две int a . Это так называемая избыточность данных. Почему это вызывает

двусмысленность? Существует два типа int a . Итак, когда мы хотим использовать объект класса D для доступа int a ,тогда компилятор не знает, к какому из них осуществляется доступ a . То, к чему

осуществляется доступ, унаследовано от класса B a или доступ наследуется от класса C a это так называемая двусмысленность. (Конечно, для двусмысленности мы можем использовать " Название класса

:: "Чтобы решить проблему, но этот метод не может решить проблему избыточности данных, поэтому существует виртуальное наследование)

С++ по умолчанию не создает ромбовидного наследования: компилятор обрабатывает каждый путь наследования отдельно, в результате чего объект D будет на самом деле содержать два разных подобъекта A , и при использовании членов A потребуется указать путь наследования ( B::A или C::A ). Чтобы сгенерировать ромбовидную  структуру наследования, необходимо воспользоваться виртуальным наследованием класса A на нескольких путях наследования: если оба наследования от A к B и от A к C помечаются спецификатором virtual (например, class B : virtual public A ), C++ специальным образом проследит за созданием только одного подобъекта A , и использование членов A будет работать корректно. Если виртуальное и невиртуальное наследования смешиваются, то получается один виртуальный подобъект A и по одному невиртуальному подобъекту A для каждого пути невиртуального наследования к A . При виртуальном вызове метода виртуального базового класса используется так называемое правило доминирования: компилятор запрещает виртуальный вызов метода, который был перегружен на нескольких путях наследования.

**13.Виртуальные функции, их применение. Понятие**

**полиморфизма. Разница в поведении виртуальных и не виртуальных функций при наследовании.**

Виртуальные функции, их применение.

Виртуальная функция в языке — это особый тип функции, которая, при её вызове, выполняет

«наиболее» дочерний метод, который существует между родительским и дочерними классами.

{

};

{

};

{

}

Понятие полиморфизма.

Полиморфизм — это способность объекта использовать методы производного класса, который не существует на момент создания базового.

Разница в поведении виртуальных и не виртуальных функций при наследовании.

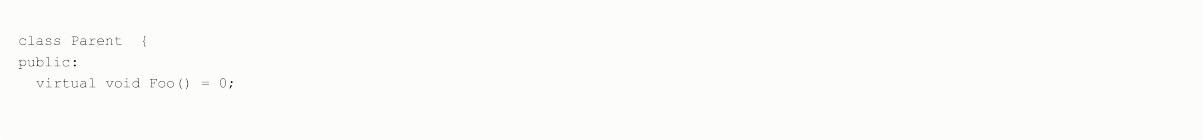
**14.Чисто виртуальные функции. Понятие абстрактного класса, особенность абстрактных классов.**

Чисто виртуальные функции.

Часто в самом базовом классе сами виртуальные функции фиктивны и имеют пустое тело. Определенное значение им придается лишь в порожденных классах. Такие функции называются *чистыми виртуальными функциями*.

*Чистая виртуальная функция* — это метод класса, тело которого не определено.

При создании чистой виртуальной функции, вместо определения (написания тела) виртуальной функции, мы просто присваиваем ей значение 0.



}

Виртуальная таблица в языке С++ — это таблица поиска функций для выполнения вызовов функций в режиме позднего (динамического) связывания. Виртуальную таблицу еще называют *«vtable»*,

«таблицей виртуальных функций» или «таблицей виртуальных методов».

Во-первых, любой класс, который использует виртуальные функции (или дочерний класс, родительский класс которого использует виртуальные функции), имеет свою собственную виртуальную таблицу. Это обычный статический массив, который создается компилятором во время компиляции. Виртуальная таблица содержит по одной записи на каждую виртуальную функцию, которая может быть вызвана

объектами класса. Каждая запись в этой таблице — это указатель на функцию, указывающий на наиболее дочерний метод, доступный объекту этого класса.

Во-вторых, компилятор также добавляет скрытый указатель на родительский класс, который мы

будем называть \* vptr . Этот указатель автоматически создается при создании объекта класса и

указывает на виртуальную таблицу этого класса.

Понятие абстрактного класса, особенность абстрактных классов.

Любой класс с одной и более чистыми виртуальными функциями становится абстрактным классом, объекты которого создавать нельзя.

**15.Виртуальные деструкторы, чисто виртуальные деструкторы, их назначение. Проблема с вызовами виртуальных функций в**

**конструкторах.**

Виртуальные деструкторы. Основное правило: если в классе присутствует хотя бы одна виртуальная функция, деструктор также следует сделать виртуальным. Деструктор по умолчанию виртуальным не будует, поэтому следует объявить его явно.



};

};

{

}

Если базовый деструктор указать как не vitrual, то при разрушении происходит утечка памяти, потому как деструктор производного класса не вызывается: A() B() ~A(). Происходит это потому, что удаление производится через указатель на базовый класс и для вызова деструктора компилятор использует

раннее связывание. Деструктор базового класса не может вызвать деструктор производного, потому что он о нем ничего не знает. В итоге часть памяти, выделенная под производный класс, безвозвратно теряется. Чтобы этого избежать, деструктор в базовом классе должен быть объявлен как

виртуальный.Теперь порядок вызовов: A() B() ~В()~A(). Происходит так потому, что для вызова

деструктора используется позднее связывание, то есть при разрушении объекта берется указатель на класс, затем из таблицы виртуальных функций определяется адрес нужного нам деструктора, а это деструктор производного класса, который после своей работы, как и полагается, вызывает деструктор базового. Итог: объект разрушен, память освобождена.

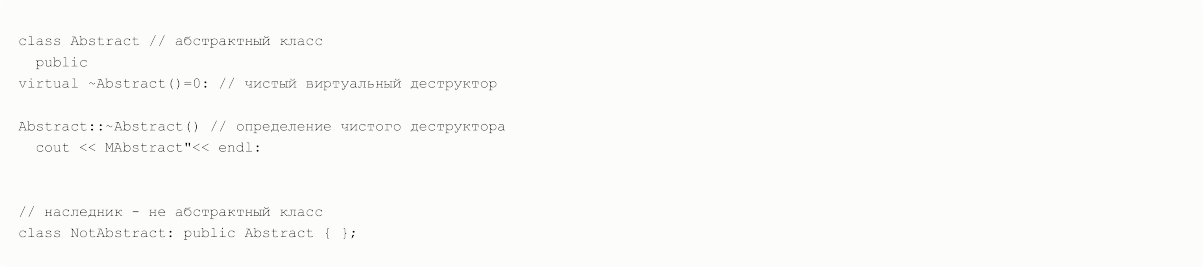
Чисто виртуальные деструкторы. Деструкторам разрешено быть «чистыми». Объявляются такие деструкторы точно так же, как чистые виртуальные функции, например:

virtual ~VBase()

= 0;

Класс, в котором определен чистый виртуальный деструктор является абстрактным, и создавать

объекты этого класса запрещено. Класс-наследник не является абстрактным классом! При объявлении чисто виртуального деструктора нужно написать и его определение. Объясняется тем, что деструктор наследника обязательно вызывает деструктор базового класса, поэтому чистый деструктор все же должен быть определен.



{

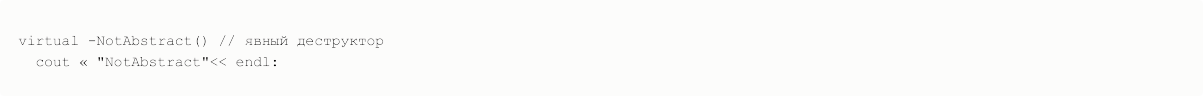
:

};

{

}

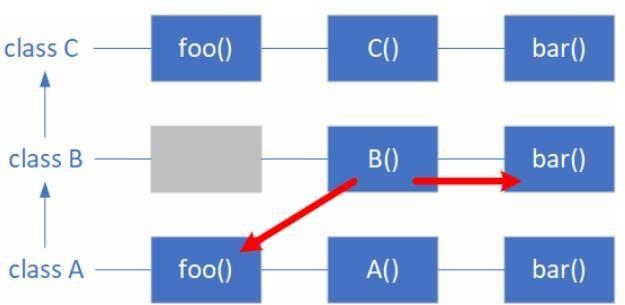
Для объекта класса Abstract компилятор немедленно выдает сообщение о том, что нельзя создавать объекты абстрактного класса.

Объект класса-наследника создается без проблем.Определим в наследнике явный деструктор:

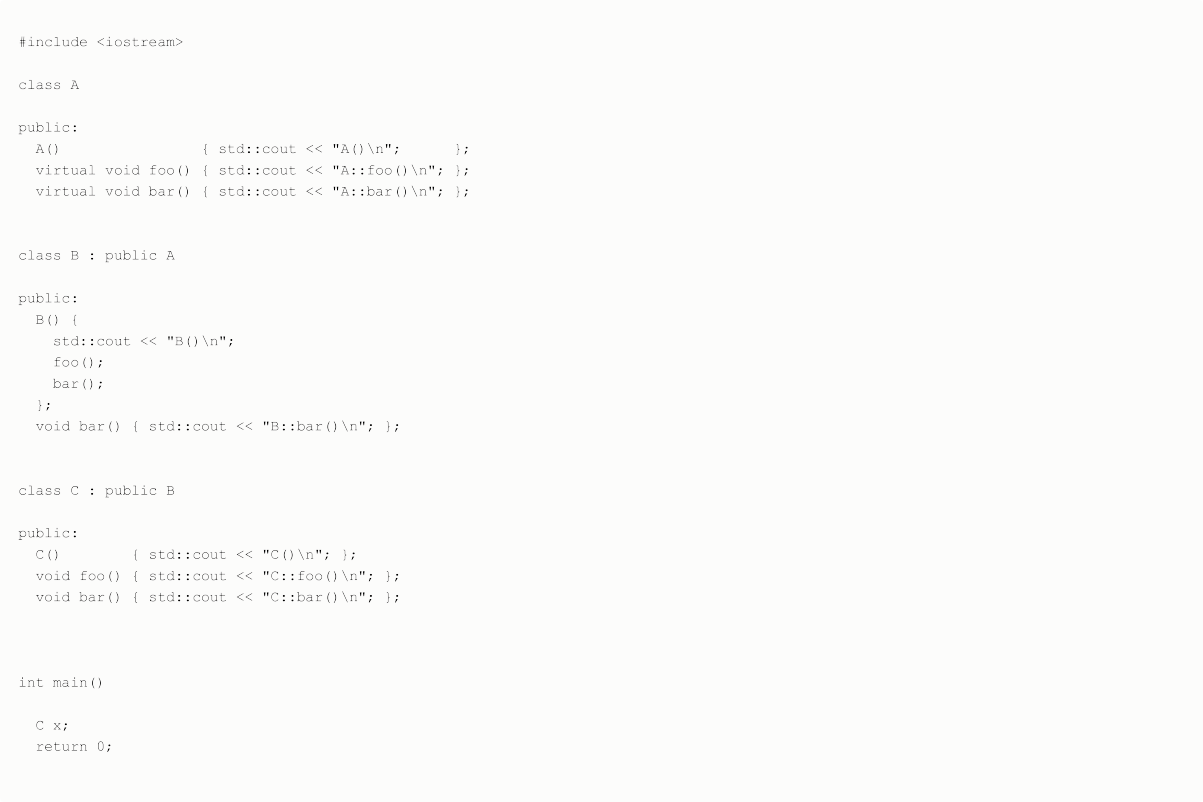
{

}

Проблема с вызовами виртуальных функций в конструкторах. Когда в конструкторе вызывается виртуальная функция, она работает только в пределах базовых или создаваемого в данный момент классов. Конструкторы в классах-наследниках ещё не вызывались, и поэтому реализованные в них виртуальные функции не будут вызваны.



Пояснения: от класса *A* наследуется класс *B*; от класса *B* наследуется класс *C*; функции *foo* и *bar* являются виртуальными; у функции *foo* нет реализации в классе *B*.



{

};

{

};

{

};

{

}

Результат: A() B() A::foo() B::bar() C() . Можно легко забыть и считать, что функции *foo* и *bar* будут вызваны из крайнего наследника, т.е. из класса *C.*

**16.Понятие шаблонов. Виды полиморфизма: статический и динамический.**

Понятие шаблонов.

Существуют шаблоны функций и шаблоны классов.

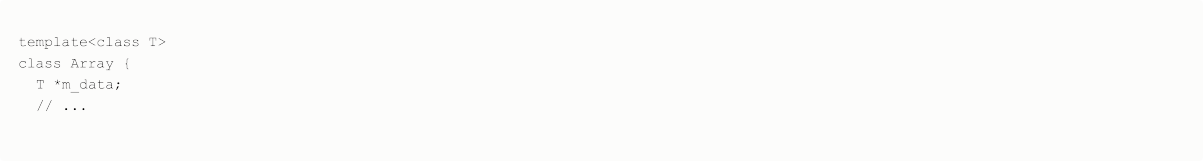
*Шаблоны функций – это обобщенное описание поведения функций, которые могут вызываться для объектов разных типов.*



}



*Шаблоны классов – обобщенное описание пользовательского типа, в котором могут быть параметризованы атрибуты и операции типа.*



}

Виды полиморфизма: статический и динамический.

Статический полиморфизм - Полиморфизм во время компиляции.

Компиляции, компилятор завершает работу во время компиляции. Компилятор может определить, какую функцию вызывать, основываясь на типе аргумента функции (который может быть неявно

преобразован по типу). Функция вызывается, в противном случае возникает ошибка компиляции. Существует два способа реализации статического полиморфизма:

Перегрузка функций: включая перегрузку обычных функций и перегрузку функций-членов Использование шаблонов функций

Динамический полиморфизм - Динамическое связывание

Полиморфизм во время выполнения, во время выполнения программы (период без компиляции), чтобы определить фактический тип ссылочного объекта и вызвать соответствующий метод в

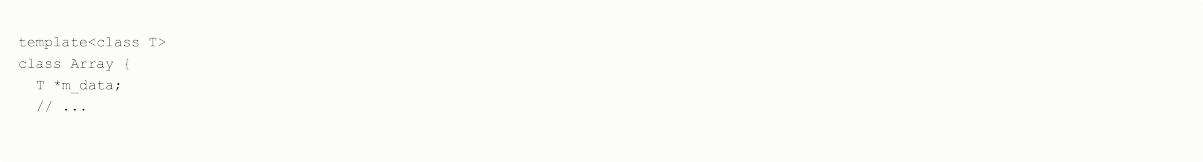
соответствии с его фактическим типом.

**17.Шаблонные классы, синтаксис их определения. Специализация шаблонов, синтаксис определения, пример. Параметры**

**шаблонов, не являющиеся типами, пример.**

Шаблонные классы, синтаксис их определения.

*Шаблоны классов – обобщенное описание пользовательского типа, в котором могут быть параметризованы атрибуты и операции типа.*



}

Специализация шаблонов, синтаксис определения, пример.

Иногда может понадобиться, чтобы реализация шаблона функции для одного типа данных отличалась от реализации шаблона функции для другого типа данных.

Специализация шаблонов именно для этого и предназначена.

{

{

}

{

}

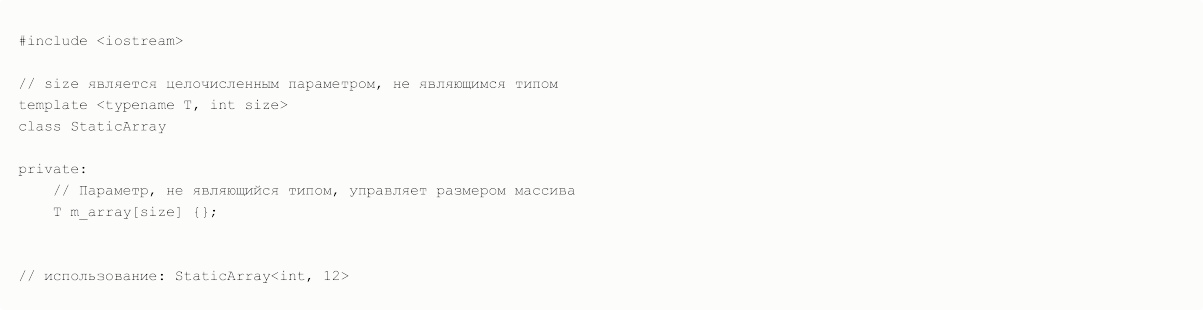
};

{

}

эту функцию, и поэтому он будет использовать именно этот экземпляр, а не копировать общую для всех типов данных версию шаблона функции print().

Параметры шаблонов, не являющиеся типами, пример.

Шаблонный параметр, не являющийся типом, – это параметр шаблона, в котором тип параметра предопределен, и он

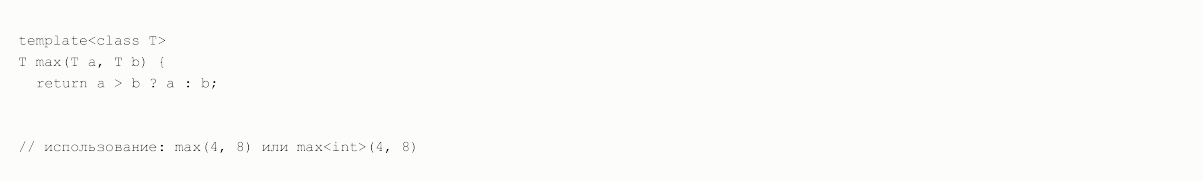
{

}

18.Шаблонные функции, синтаксис объявления и использования.

Шаблонные функции, синтаксис объявления и использования.

*Шаблоны функций – это обобщенное описание поведения функций, которые могут вызываться для объектов разных типов.*

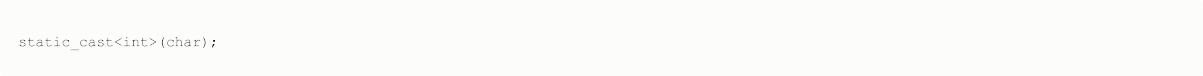


}

**19.Приведение типов: static\_cast, dynamic\_cast, разница между ними, примеры применения.**

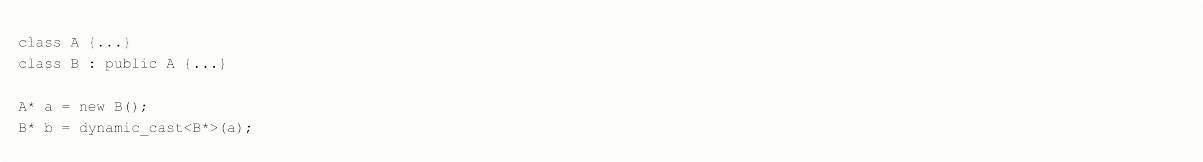
Приведение типов: static\_cast, dynamic\_cast, разница между ними, примеры применения.

static\_cast

Оператор static\_cast используется для явного приведения типов следующим образом:

static\_cast<int>(char);

Основным преимуществом оператора static\_cast является проверка его выполнения компилятором во время компиляции, что усложняет возможность возникновения непреднамеренных проблем.

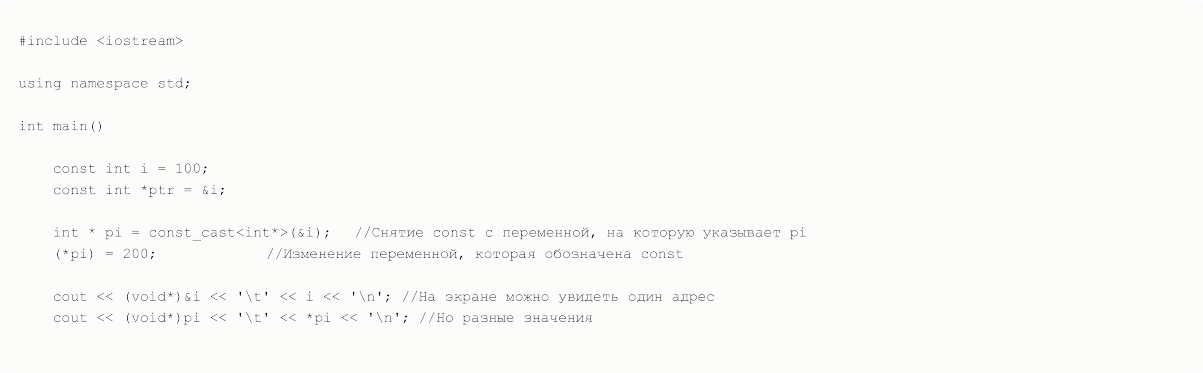
dynamic\_cast dynamic\_cast это ещё один оператор явного преобразования типов. Он используется для преобразования из базового типа к производному. Этот процесс называется приведением к дочернему типу (или *«понижающим приведением типа»*).

Если dynamic\_cast не может выполнить конвертацию, то он возвращает нулевой указатель.

**20.Приведение типов const\_cast, пример применения.**

**Приведение типов reinterpret\_cast, пример применения.**

Приведение типов const\_cast, пример применения.

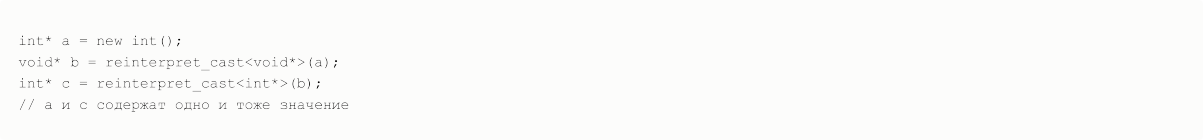
Явное приведение типа const\_cast используют для того, чтобы отбросить квалификатор const у изначально не константных данных или добавить квалификатор const. const\_cast не влияет на оригинальную переменную: не убирает её const и не добавляет ей const

{

}

Приведение типов reinterpret\_cast, пример применения.

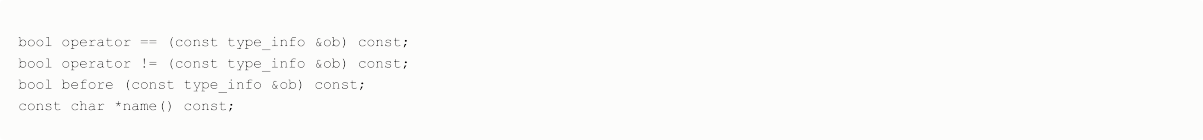
Приведение типов без проверки. reinterpret\_cast — непосредственное указание компилятору. Применяется только в случае полной уверенности программиста в собственных действиях.



**21.Ключевое слово typeid, динамическая проверка типа, вывод имени типа, пример.**

В языке C++ оператор typeid возвращает ссылку на объект type\_info, описывающий тип объекта, к которому принадлежит оператор typeid. общая форма записи оператора typeid





Перегруженные операторы == и != служат для сравнения типов. Функция before() возвращает значение true, если возвращающий объект в порядке сопоставления стоит перед объектом, используемым в качестве параметра. Эта функция предназначена в основном для внутреннего использования. Её

значение возврата не имеет ничего общего с наследованием или иерархией классов. Функция name() возвращает указатель на имя типа.

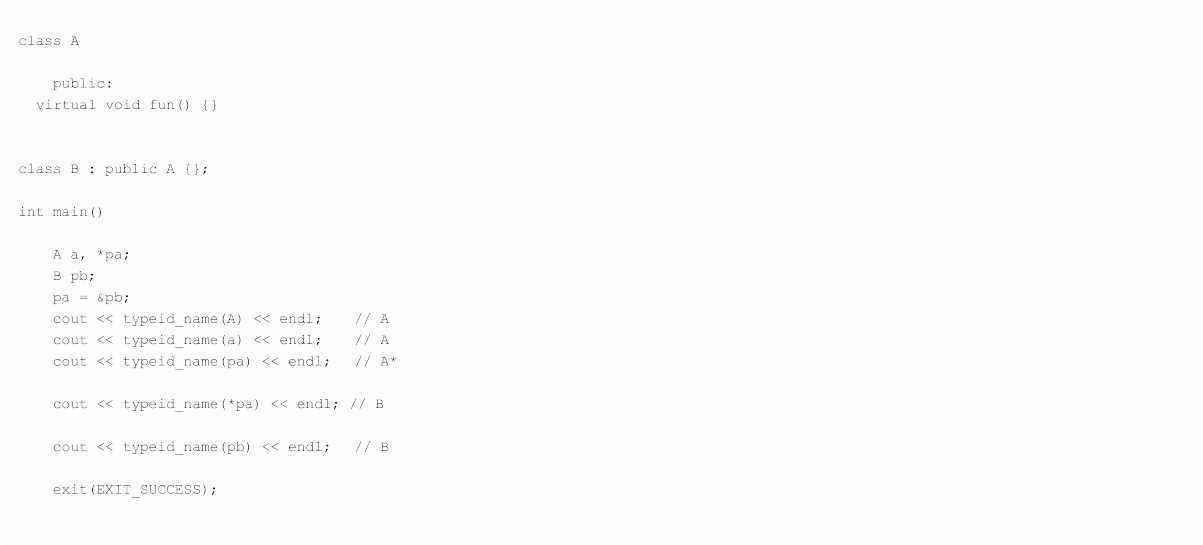
Если оператор typeid применяется к указателю полиморфного класса, он автоматически возвращает тип объекта, на который он указывает. (полиморфный класс — это класс который содержит хотя-бы одну виртуальную функцию.) Следовательно, оператор typeid можно использовать для определения типа

объекта, адресуемого указателем на базовый класс.



{

}



{

};

{

}

Если операнд оператора typeid является типом класса, который содержит хотя бы одну виртуальную функцию, а выражение является ссылкой на базовый класс, оператор typeid указывает тип

производного класса базового объекта.

**22.Понятие итераторов. Классификация итераторов: input,**

**output, однонаправленные, двунаправленные, произвольного доступа. Операции, поддерживаемые итераторами.**

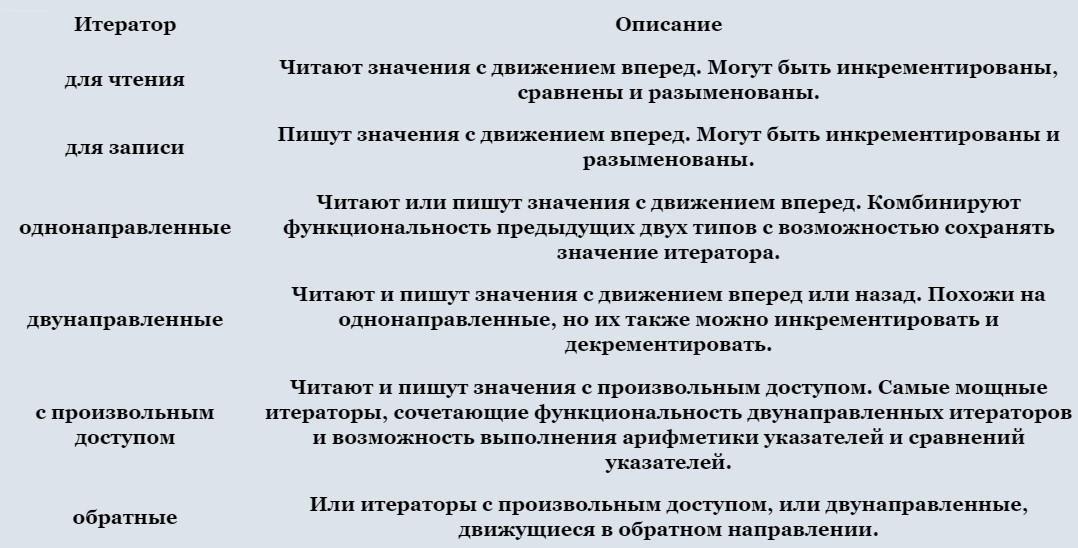
Понятие итераторов.

Итератор — это объект, который способен перебирать элементы контейнерного класса

без необходимости пользователю знать реализацию определенного контейнерного класса. Во многих контейнерах (особенно в списке и в ассоциативных контейнерах) итераторы являются основным

способом доступа к элементам этих контейнеров.

Классификация итераторов: input, output, однонаправленные, двунаправленные, произвольного доступа.



Операции, поддерживаемые итераторами.

Об итераторе можно думать, как об указателе на определенный элемент контейнерного класса с

дополнительным набором перегруженных операторов для выполнения четко определенных функций:

Оператор \* возвращает элемент, на который в данный момент указывает итератор.

Оператор ++ перемещает итератор к следующему элементу контейнера. Большинство итераторов также предоставляют оператор −− для перехода к предыдущему элементу.

Операторы == и != используются для определения того, указывают ли оба итератора на один и тот же элемент или нет. Для сравнения значений, на которые указывают оба итератора, нужно сначала разыменовать эти итераторы, а затем использовать оператор == или оператор != .

Оператор = присваивает итератору новую позицию (обычно начальный или конечный элемент контейнера). Чтобы присвоить значение элемента, на который указывает итератор, другому объекту, нужно сначала разыменовать итератор, а затем использовать оператор = .

Каждый контейнерный класс имеет 4 основных метода для работы с оператором = :

метод begin() возвращает итератор, представляющий начальный элемент контейнера;

метод end() возвращает итератор, представляющий элемент, который находится после последнего элемента в контейнере;

метод cbegin() возвращает константный (только для чтения) итератор, представляющий начальный элемент контейнера;

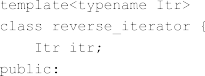
метод cend() возвращает константный (только для чтения) итератор, представляющий элемент, который находится после последнего элемента в контейнере.

**23.Константные и неконстантные итераторы. Класс std::reverse\_iterator, его примерная реализация.**

Константные и неконстантные итераторы.

Константные итераторы, в отличие от неконстантных, не позволяют изменять значения на которые они указывают

Класс std::reverse\_iterator, его примерная реализация.

std::reverse\_iterator это адаптер для итераторов, который меняет направление данного итератора

}

}

}

|  |  |
| --- | --- |
| 24.Классы итераторов для вставок (insert\_iterator), их схема  реализации и использования. Функции std::inserter, std::front\_inserter и std::back\_inserter, | |
| схема их реализации и пример использования. |  |

Чтобы было возможно иметь дело с вставкой таким же образом, как с записью в массив, в библиотеке обеспечивается специальный вид адаптеров итераторов, называемых *итераторами вставки* (*insert iterators*). (могли вставлять данные, а не перезаписывать их)

Итератор вставки создаётся из контейнера и, возможно, одного из его итераторов, указывающих, где вставка происходит, если это ни в начале, ни в конце контейнера. Итераторы вставки удовлетворяют требованиям итераторов вывода. operator\* возвращает непосредственно сам итератор вставки.

Присваивание *operator=(const T& х)* определено для итераторов вставки, чтобы разрешить запись в них, оно вставляет х прямо перед позицией, куда итератор вставки указывает. Другими словами, итератор вставки подобен курсору, указывающему в контейнер, где происходит вставка. back\_insert\_iterator вставляет элементы в конце контейнера, front\_insert\_iterator вставляет элементы в начале контейнера, а insert\_iterator вставляет элементы, куда итератор указывает в контейнере. back\_inserter, front\_inserter и inserter - три функции, создающие итераторы вставки из контейнера.



**25.Понятие исключений, общая идея, аргументы за и против применения исключений. Оператор throw и конструкция try...catch, схема ее работы.**

**Правила генерации и перехвата исключений, производные исключения.**

Понятие исключений, общая идея, аргументы за и против применения исключений. Исключения - ошибки, которые могут вызывать как вручную, так и непосредственно системой. Они требуют от вызывающего кода обработки или завершают выполнение программы.

В C++ в качестве сигнала об исключении может использоваться любой тип данных. Минусы использования исключений:

Очистка памят

Одной из самых больших проблем используя исключения, является проблема очистки выделенных ресурсов после генерации исключения.

Исключения и деструкторы

В отличие от конструкторов, где генерация исключений может быть полезным способом указать, что создать объект не удалось, исключения *никогда* не должны генерироваться в деструкторах.

Проблема возникает, когда исключение генерируется в деструкторе во время раскручивания стека.

Если это происходит, то компилятор оказывается в ситуации, когда он не знает, продолжать ли процесс раскручивания стека или обработать новое исключение. Конечным результатом будет немедленное прекращение выполнения вашей программы.

Проблемы с производительностью

Исключения имеют свою небольшую цену производительности. Они увеличивают размер вашего исполняемого файла и также могут заставить его выполняться медленнее из-за дополнительной проверки, которая должна быть выполнена.

Оператор throw и конструкция try...catch, схема ее работы.

В языке C++ оператор throw используется для сигнализирования о возникновении исключения или ошибки. Сигнализирование о том, что произошло исключение, называется генерацией исключения (или *«выбрасыванием исключения»*).

Для использования оператора throw применяется ключевое слово throw, а за ним указывается значение любого типа данных, которое вы хотите задействовать, чтобы сигнализировать об ошибке. Как правило, этим значением является код ошибки, описание проблемы или настраиваемый класс-исключение.

В языке C++ мы используем ключевое слово try для определения блока стейтментов (так называемого

*«блока try»*). Блок try действует как наблюдатель в поисках исключений, которые были выброшены каким-либо из операторов в этом же блоке try.

Ключевое слово catch используется для определения блока кода (так называемого *«блока catch»*), который обрабатывает исключения определенного типа данных.

При выбрасывании исключения (оператор throw), точка выполнения программы немедленно переходит к ближайшему блоку try. Если какой-либо из обработчиков catch, прикрепленных к блоку try,

обрабатывает этот тип исключения, то точка выполнения переходит в этот обработчик и, после выполнения кода блока catch, исключение считается обработанным.

Если подходящих обработчиков catch не существует, то выполнение программы переходит к

следующему блоку try. Если до конца программы не найдены соответствующие обработчики catch, то программа завершает свое выполнение с ошибкой исключения.

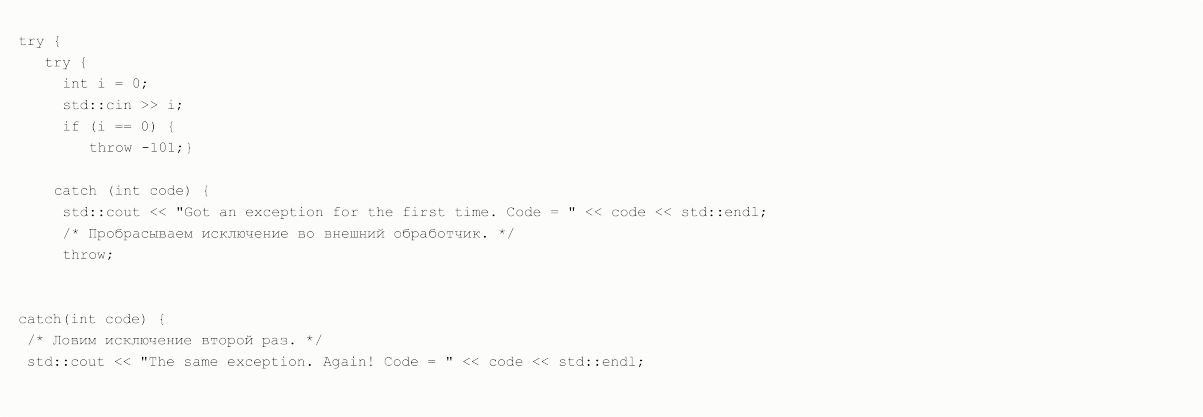
Правила генерации и перехвата исключений, производные исключения.

При наличии нескольких блоков catch вызывается только один, тот, который подходит по типу исключения и находится выше остальных по коду.

**26.Проброс и повторная генерация исключений, перехват любых исключений. Особенности передачи исключений по указателю, по ссылке и по значению.**

Проброс генерации исключений. В некоторых случаях возникает необходимость поймать исключение в обработчике, а затем передать это исключение дальше в следующий обработчик (если он есть). Речь

идет о ситуации, когда в некотором внешнем блоке try написана группа внутренних блоков try/catch. Для проброса исключения из внутреннего обработчика во внешний нужно написать конструкцию throw; (без аргумента) в соответствующем внутреннем обработчике.



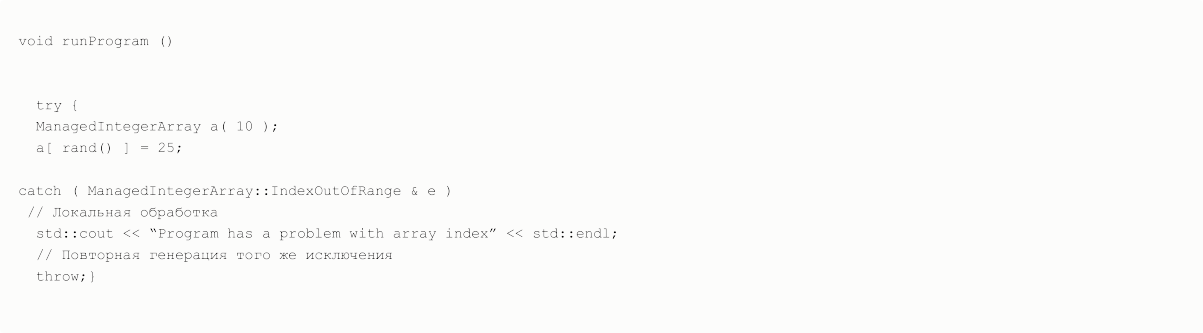
}

}

}

}

Повторная генерация исключений. При повторной генерации исключения используйте ключевое слово throw без указания какого-либо идентификатора.



{

}

{

}

Перехват Исключений. Блок catch, следующий за блоком try, ловит любое исключение. Вы можете указать, какой тип исключения вы хотите перехватить, и это определяется объявлением исключения, которое появляется в круглых скобках после ключевого слова Если нужно, чтобы блок catch

обрабатывал любой тип исключения, вызванного в блоке try, нужно поставить многоточие,…, между скобками, заключающими объявление исключения следующим образом:

try {

// protected code

} catch(…) {

// code to handle any exception

}

Особенности передачи исключений по указателю, по ссылке и по значению. v

Если не удалять объекты-исключения, перехватываемые по указателю, то это может привести к утечкам памяти, когда на стороне выброса использовалось динамическое выделение памяти. Если удалять все объекты-исключения, можно получить фатальный сбой при попытке удаления объекта, не хранящегося в динамической памяти, такого как globalException.

Перехватывать объекты-исключения можно по значению либо по ссылке. Если принимать исключение по значению, будет выполняться копирование его содержимого. Если же принимать исключение по

ссылке, копирования происходить не будет, обработчик будет работать с содержимым объекта- исключения в специальной служебной памяти. Обычно используют константные ссылки, чтобы избежать случайной модификации объекта-исключения.

**27.Неожиданные исключения, функция unexpected. Функции terminate и set\_terminate. Особенности исключений в**

**конструкторах и деструкторах, функция uncaught\_exception.**

Неожиданные исключения, функция unexpected.

Когда не один из блоков catch не готов обработать возникшее исключение среда выполнения C++ делает следующее:

Вызывается функция unexpected

Функция unexpected вызывает функцию, указанную в unexpected\_handler

Чтобы зарегистрировать unexpected\_handler используется функция set\_unexpected(handler)

Функции terminate и set\_terminate.

Функция terminate вызывается в следующих случаях:

Исключение выбрасывается и не перехватывается (определяется реализацией, выполняется ли в этом случае какая-либо раскрутка стека)

Функция, непосредственно вызываемая механизмом обработки исключений при обработке исключения, которое еще не было перехвачено, завершается через исключение (например,

деструктор некоторого локального объекта или конструктор копирования, создающий параметр catch-clause)

Конструктор или деструктор статического или локального (начиная с C++11) объекта выдает исключение

Функция, зарегистрированная с помощью std::atexit или std::at\_quick\_exit (начиная с C++11), выдает исключение

Функция set\_terminate устанавливает новый terminate\_handler и возвращает предыдущий terminate\_handler

Особенности исключений в конструкторах и деструкторах, функция uncaught\_exception.

Если конструктор класса завершает работу исключением, значит он не завершает свою работу — следовательно объект не будет создан. Из-за этого могут возникать утечки памяти, т.к. для не

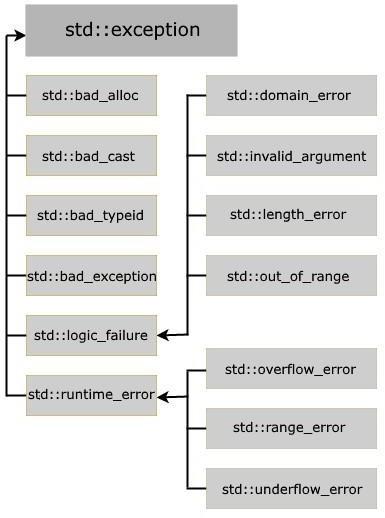
полностью сконструированных объектов не будет вызван деструктор.

Если деструктор выбросит исключение, так же произойдёт утечка памяти, в памяти останутся части текущего и базового классов.

Стандартная библиотека предоставляет функцию std::uncaught\_exception, которая в деструкторе

позволяет узнать, почему уничтожается объект, из-за выброшенного исключения, или же по какой-либо другой причине.

**28.Примеры стандартных исключений и операторов или методов STL, которые их генерируют. Отличия между исключениями и ошибками времени выполнения.**

Примеры стандартных исключений и операторов или методов STL, которые их генерируют.

Основные из них: runtime\_error: общий тип исключений, которые возникают во время выполнения



range\_error: исключение, которое возникает, когда полученный результат превосходит допустимый диапазон

overflow\_error: исключение, которое возникает, если полученный результат превышает допустимый диапазон

underflow\_error: исключение, которое возникает, если полученный в вычислениях результат

имеет недопустимые отрицательное значение (выход за нижнюю допустимую границу значений) logic\_error: исключение, которое возникает при наличии логических ошбок к коде программы domain\_error: исключение, которое возникает, если для некоторого значения, передаваемого в функцию, не определено результата invalid\_argument: исключение, которое возникает при

передаче в функцию некорректного аргумента length\_error: исключение, которое возникает при попытке создать объект большего размера, чем допустим для данного типа out\_of\_range:

исключение, которое возникает при попытке доступа к элементам вне допустимого диапазона

Отличия между исключениями и ошибками времени выполнения.

**29.Понятие контейнера. Последовательные контейнеры.**

**Контейнер std::vector, схема его реализации. Схема работы и асимптотика методов push\_back, pop\_back, insert, erase.**

Контейнер - это класс STL (стандартная библиотека шаблонов), реализующий функциональность некоторой структуры данных, то есть хранилища нескольких элементов. Примеры разных контейнеров: vector, stack, queue, deque, string, set, map и т.д.

Различные контейнеры имеют различные способы доступа к элементом. Для обращения к элементам контейнеров существует понятие итератора. Итератор является обобщением идеи доступа к элементу по индексу и обобщением указателей языка C. Можно рассматривать итераторы, как "умные" указатели.

Последовательные контейнеры обеспечивают хранение конечного количества однотипных объектов в виде непрерывной последовательности. К базовым после- довательным контейнерам относятся векторы (vector), списки (list) и двусторонние очереди (deque).

Контейнер вектор является аналогом обычного массива, за исключением того, что он автоматически выделяет и освобождает память по мере

необходимости. Контейнер эффективно обрабатывает произвольную выборку элементов с помощью операции индексации [] или метода at. Однако вставка элемента в любую позицию, кроме конца вектора, неэффективна. Для этого

потребуется сдвинуть все последующие элементы путем копирования их значений. По этой же причине неэффективным является удаление любого элемента, кроме последнего.

Асимптотика методов push\_back, pop\_back, insert, erase: (Временная Сложность) push\_back() - для добавления элементов в вектор, в который передается добавляемый элемент -O(1) pop\_back() - удаляет последний элемент вектора -O(1) insert(pos, value) - вставляет элемент value на позицию, на которую указывает итератор pos, аналогично функции emplace insert(pos, n, value) - вставляет n

элементов value начиная с позиции, на которую указывает итератор pos insert(pos, begin, end) -

вставляет начиная с позиции, на которую указывает итератор pos, элементы из другого контейнера

из диапазона между итераторами begin и end insert(pos, values) - вставляет список значений начиная с позиции, на которую указывает итератор pos erase(p) - удаляет элемент, на который указывает

итератор p. Возвращает итератор на элемент, следующий после удаленного, или на конец контейнера, если удален последний элемент -O(n) erase(begin, end) - удаляет элементы из

диапазона, на начало и конец которого указывают итераторы begin и end. Возвращает итератор на элемент, следующий после последнего удаленного, или на конец контейнера, если удален

последний элемент

**30.Понятие контейнера. Последовательные контейнеры.**

**Контейнер std::vector, схема его реализации. Методы size, capacity, resize, reserve, shrink\_to\_fit.**

Понятие контейнера.

Контейнерный класс (или *«класс-контейнер»*) в языке C++ — это класс, предназначенный для хранения и организации нескольких объектов определенного типа данных (пользовательских или фундаментальных).

Последовательные контейнеры.

Последовательные контейнеры (или *«контейнеры последовательности»*) — это контейнерные классы, элементы которых находятся в последовательности. Их определяющей характеристикой является то, что вы можете добавить свой элемент в любое место контейнера. Наиболее

распространенным примером последовательного контейнера является массив

Контейнер std::vector, схема его реализации.

std::vector (или просто *«вектор»*) — это тот же динамический массив, но который может сам управлять выделенной себе памятью. Это означает, что вы можете создавать массивы, длина которых задается во время выполнения, без использования операторов new и delete

Методы size, capacity, resize, reserve, shrink\_to\_fit. size - Возвращает количество элементов в векторе.

capacity - Возвращает количество элементов, которое может содержать вектор до того, как ему

потребуется выделить больше места. resize - Изменяет размер вектора на заданную величину. reserve - Устанавливает минимально возможное количество элементов в векторе. shrink\_to\_fit - Уменьшает количество используемой памяти за счёт освобождения неиспользованной

**31.Понятие контейнера. Последовательные контейнеры.**

**Контейнер std::vector, схема его реализации. Методы для обращения по индексу, для обращения к первому и**

**последнему элементам. Метод swap, его схема реализации.**

Понятие контейнера.

Контейнерный класс (или *«класс-контейнер»*) в языке C++ — это класс, предназначенный для хранения и организации нескольких объектов определенного типа данных (пользовательских или фундаментальных).

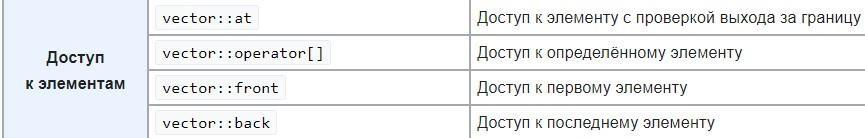
Последовательные контейнеры.

Последовательные контейнеры (или *«контейнеры последовательности»*) — это контейнерные классы, элементы которых находятся в последовательности. Их определяющей характеристикой является то, что вы можете добавить свой элемент в любое место контейнера. Наиболее

распространенным примером последовательного контейнера является массив

Контейнер std::vector, схема его реализации.

std::vector (или просто *«вектор»*) — это тот же динамический массив, но который может сам управлять выделенной себе памятью. Это означает, что вы можете создавать массивы, длина которых задается во время выполнения, без использования операторов new и delete

Методы для обращения по индексу, для обращения к первому и последнему элементам.

Метод swap, его схема реализации.

Метод swap - Обменять содержимое двух векторов

32.Контейнер vector<bool> и его отличия от обычного vector.

std::vector<bool> — это более компактная специализация std::vector для типа bool.

Способ, которым std::vector<bool> экономит пространство (а также то, оптимизируется ли он вообще), определяется реализацией. Одна потенциальная оптимизация включает в себя объединение векторных элементов таким образом, чтобы каждый элемент занимал один бит вместо sizeof(bool) байтов.

std::vector<bool> ведет себя аналогично std::vector, но для экономии места он:

Не обязательно сохраняет свои элементы как непрерывный массив.

Предоставляет класс std::vector<bool>::reference как метод доступа к отдельным битам. В частности, объекты этого класса возвращаются оператором [] по значению.

Не использует std::allocator\_traits::construct для построения битовых значений.

33.Контейнер std::deque, схема его реализации, асимптотика

работы методов. Отличия deque от vector с точки зрения

интерфейса. Контейнеры std::list и std::forward\_list, схема их реализации, асимптотика работы методов. Отличия list от

deque и vector с точки зрения интерфейса.

Контейнер двусторонняя очередь во многом аналогичен вектору, элементы хранятся

в непрерывной области памяти, элементы можно добавлять и удалять как в начало, так и в конец, то есть дисциплинами обслуживания являются одновременно FIFO и LIFO. Но в отличие от вектора

двусторонняя очередь эффективно поддерживает вставку и удаление первого элемента (так же, как и последнего).

push\_back(T&key) - добавление в конец (O(1)) pop\_back() - удаление(O(1)) push\_front(T&key) - добавление в начало (O(1)) pop\_front() - удаление из начала (O(1)) insert - вставка в произвольное место

(insert(iter, val) - O(1) insert(iter, ne, val) - O

insert(iter, inItA, inItB) - O(n))

erase - удаление из произвольного места [], at(O(1)) - доступ к произвольному элементу front(): возвращает первый элемент - O(1) back(): возвращает последний элемент - O(1) assign(n, value) - заменяет содержимое контейнера n элементами, которые имеют

значение value emplace\_back(val)/emplace\_front(val) - добавляет значение val в конец очереди и в начало очереди -O(1)

Контейнер список организует хранение объектов в виде двусвязного списка. Каждый элемент списка содержит три поля: значение элемента, указатель на предшествующий и указатель на последующий элементы списка. Вставка и удаление работают эффективно для любой позиции

элемента в списке. Однако список не поддерживает произвольного доступа к своим элементам: например, для выборки n-го элемента нужно

последовательно выбрать предыдущие п-1

элементов. методы такие же как и в двусторонней очереди (нету [] и at) +

Контейнер forward\_list реализует односвязный список, элементы которого хранят указатель только на следующий элемент. Вставка и удаление элементов происходит быстро, достаточно изменить ссылку, но получить доступ к элементу списка по индексу нельзя, т. к. элементы могут быть расположены в

разных местах памяти. Перебирать элементы можно только в прямом направлении. resize(n) - оставляет в списке n первых элементов assign() ; push\_front (); emplace(); insert(); clear(); swap()

**34.Методы, специфичные для list: sort, merge, splice, remove, unique, reverse, схема их реализации. Отличия list от forward\_list**

sort() - сортирует список merge (list &ob) - объединяет упорядоченный список, содержащийся в объекте ob, с данным упорядоченным списком.

splice(iterator i, list &ob) - вставляет содержимое списка ob в данный список в позиции, указанной итератором i. После выполнения этой операции список ob остается пустым splice(iterator i, list &ob,

iterator el) - удаляет из списка ob элемент, адресуемый итератором el, и сохраняет его в данном списке в позиции, адресуемой итератором i remove(val) - удаляет из списка элементы со значением val unique() - удаляет из списка элементы-дубликаты reverse() - обратный список

List основан на двусвязном списке, а forward\_list основан на односвязном списке (нет возможности получить доступ к отдельному методу нельзя повторить в обратном порядке).

**35.Понятие адаптеров над контейнерами. Схема реализации**

**std::stack, std::queue, std::priority\_queue, асимптотика работы методов, требования к используемому контейнеру.**

Понятие адаптеров над контейнерами.

Адаптеры — это специальные предопределенные контейнерные классы, которые адаптированы для выполнения конкретных заданий. Самое интересное заключается в том, что вы сами можете выбрать, какой последовательный контейнер должен использовать адаптер.

Схема реализации std::stack, std::queue, std::priority\_queue, асимптотика работы методов, требования к используемому контейнеру.

stack

Методы:

empty() – Returns whether the stack is empty – Time Complexity : O(1)

size() – Returns the size of the stack – Time Complexity : O(1)

top() – Returns a reference to the top most element of the stack – Time Complexity : O(1)

push(g) – Adds the element ‘g’ at the top of the stack – Time Complexity : O(1)

pop() – Deletes the top most element of the stack – Time Complexity : O(1)

queue

Методы:

empty() - Returns whether the queue is empty. It return true if the queue is empty otherwise returns false.

- Time Complexity : O(1)

size() - Returns the size of the queue. - Time Complexity : O(1)

emplace() - Exchange the contents of two queues but the queues must be of the same data type, although sizes may differ. - Time Complexity : O(1)

front() - Insert a new element into the queue container, the new element is added to the end of the queue. - Time Complexity : O(1)

back() - Returns a reference to the first element of the queue. - Time Complexity : O(1)

push(g) - Adds the element ‘g’ at the end of the queue. - Time Complexity : O(1)

pop() - Deletes the first element of the queue. - Time Complexity : O(1)

priority\_queue

Методы:

empty() - Returns true if the priority queue is empty and false if the priority queue has at least one element. Its time complexity is O(1).

pop() - Removes the largest element from the priority queue. Its time complexity is O(logN) where N is the size of the priority queue.

push() - Inserts a new element in the priority queue. Its time complexity is O(logN) where N is the size of the priority queue.

size() - Returns the number of element in the priority queue. Its time complexity is O(1).

top() - Returns a reference to the largest element in the priority queue. Its time complexity is O(1).

**36.Ассоциативные контейнеры. Класс std::map, его идея, его назначение и схема реализации. Асимптотика и принцип**

**работы методов [], find, count, insert, erase. Класс std::pair и его связь с map. Функция make\_pair.**

Ассоциативные контейнеры обеспечивают быстрый доступ к данным по ключу. Эти контейнеры построены на основе сбалансированных деревьев. Существует пять типов ассоциативных контейнеров: словари (mар), словари с дубликатами

(multimap), множества (set), множества с дубликатами (multiset) и битовые множества (bitset).

map (или *«ассоциативный массив»*) — это set, в котором каждый элемент является парой «ключ-

значение». «Ключ» используется для сортировки и индексации данных и должен быть уникальным, а

«значение» — это фактические данные. Позволяет записать значение по некоторому ключу,

посмотреть значение по ключу, а также удалить пару с заданным ключом. Ключи в этом контейнере являются уникальными, их нельзя модифицировать

Благодаря упорядоченной структуре контейнера map операции поиска или внесения нужного элемента могут быть выполнены за логарифмическое время O(log(n)).

count( ): выполняет поиск элементов, сопоставленных по заданному ключу, и возвращает количество совпадений.Поскольку map хранит каждый элемент с уникальным ключом, то вернет 1, если

совпадение найдено, в противном случае вернет 0. O(logN).

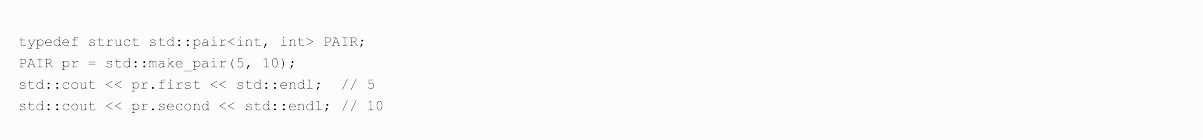
find(k) - возвращает итератор, указывающий на значение, соответствующее значению ключа k . Если такого значения в контейнере нет, то возвращается итератор end insert(pair(k,v)) - вставляет в контейнер пару (k,v) , возвращая адрес его позиции erase(p) - удаляет из контейнера элемент, на который указывает итератор p

*Класс pair* (пара) стандартной библиотеки С++ позволяет нам определить одним объектом пару

значений, если между ними есть какая-либо семантическая связь. Эти значения могут быть одинакового или разного типа. Первое значение доступно через свойство first

, а второе значение — через свойство second *pair* - одна пара ключ -> значений, *map* - коллекция пар ключ -> значений





**37.Ассоциативные контейнеры. Класс std::set, его отличия от**

**std::map. Классы std::multimap и std::multiset, их отличия от std:map и std::set.**

Ассоциативные контейнеры обеспечивают быстрый доступ к данным по ключу. Эти контейнеры построены на основе сбалансированных деревьев.

set — это контейнер, в котором хранятся только уникальные элементы, и повторения запрещены. Элементы сортируются в соответствии с их значениями.

multiset — это set, но в котором допускаются повторяющиеся элементы.

map (или *«ассоциативный массив»*) — это set, в котором каждый элемент является парой «ключ-

значение». «Ключ» используется для сортировки и индексации данных и должен быть уникальным, а

«значение» — это фактические данные.

multimap (или *«словарь»*) — это map, который допускает дублирование ключей. Все ключи отсортированы в порядке возрастания, и вы можете посмотреть значение по ключу.

**38.Класс std::unordered\_map, его идея, схема реализации, асимптотика работы методов, отличия от std::map.**

unordered\_map — это связанный контейнер, в котором хранятся элементы, образованные комбинацией ключ-значение и отображаемое значение. Значение ключа используется для уникальной

идентификации элемента, а сопоставленное значение — это содержимое, связанное с ключом. И ключ, и значение могут быть любого типа, предопределенными или определенными пользователем.

map представляет собой упорядоченную последовательность уникальных ключей, тогда как в unordered\_map ключ может храниться в любом порядке.

**39.Стандартные строки. Шаблонный класс basic\_string, его методы, отличия от vector. Класс string. Класс char\_traits, его специализации, его назначение.**

Класс basic\_string по сути является контейнером. Это значит, что итераторы и алгоритмы STL могут обеспечить работу со строками. Однако строки обладают дополнительными возможностями.

Тип basic\_string использует класс char\_traits, который определяет ряд атрибутов символов, составляющих строку.

Отличия от vector:

basic\_string не вызывает конструкторы и деструкторы своих элементов. vector вызывает.

Замена basic\_string делает недействительными итераторы (включая оптимизацию небольших строк), замена векторов - нет.

Символоподобные объекты в объекте basic\_string должны храниться непрерывно.

basic\_string имеет интерфейс для строковых операций. vector нет.

basic\_string может использовать стратегию копирования при записи. vector не может.

Класс string был введен как альтернативный вариант для работы со строками типа char\*. Строки, которые завершаются символом ‘\0’ еще называются C-строками. Поскольку, string есть классом, то можно объявлять объекты этого класса.

Основным недостатком типа string в сравнении с типом char\*, есть замедленная скорость обработки данных.

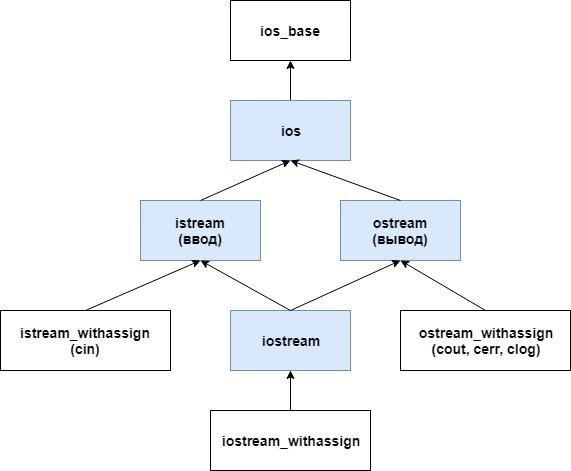
Класс char\_traits - это шаблон класса traits, который абстрагирует основные символьные и строковые операции для данного типа символов. Определенный набор операций таков, что универсальные алгоритмы почти всегда могут быть реализованы с его точки зрения.

**40.Концепция потоков ввода-вывода. Иерархия классов**

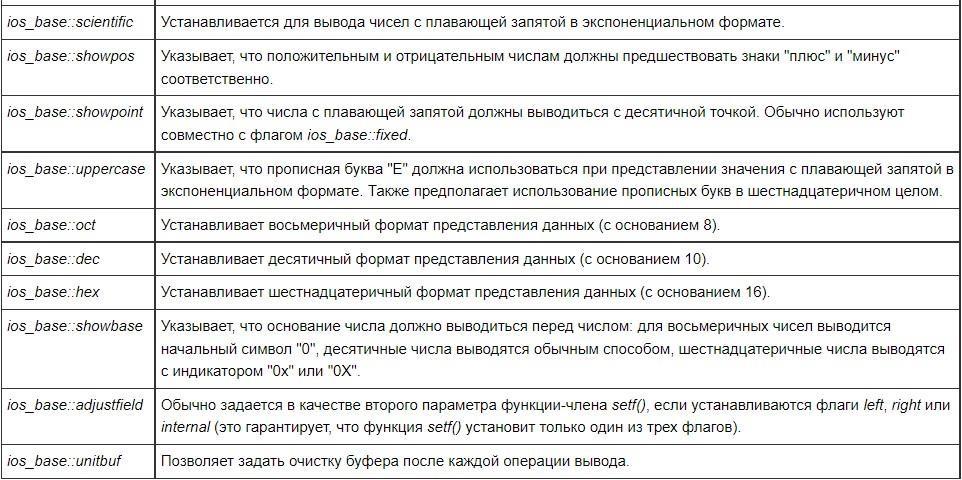
**стандартных потоков. Класс ios\_base. Флаги потоков. Методы precision и width. Синхронизация с stdio.**

Поток — это последовательность символов, к которым можно получить доступ.

Поток ввода (или *«входной поток»*) используется для хранения данных, полученных от источника данных: клавиатуры, файла, сети и т.д.

Поток вывода (или *«выходной поток»*) используется для хранения данных, предоставляемых конкретному потребителю данных: монитору, файлу, принтеру и т.д. При записи данных на устройство вывода, это устройство может быть не готовым принять данные немедленно.

ios\_base. Класс описывает хранилище и функции-члены, общие как для входных, так и для выходных потоков, которые не зависят от параметров шаблона.



Допускается объединение разных флагов в одно значение (типа *long*) путем использования операции побитовое ИЛИ

(|); можно также использовать операцию "запятая precision - Позволяет управлять точностью печатаемых чисел с плавающей запятой *width - Установка ширины поля*

Синхронизация с stdin можно включить или отключить искользуя функцию sync\_with\_stdio(bool). Устанавливает, синхронизируются ли стандартные потоки C++ со стандартными потоками C после каждой операции ввода/вывода.

**41.Классы basic\_ios и ios. Состояние потоков, проверка и**

**изменение состояния. Классы basic\_istream, basic\_iostream, basic\_ostream.**

Шаблон класса basic\_ios<>, производный от ios\_base<>, определяет общие свойства всех потоковых классов,зависящие от типа и трактовок символов (определяют признак конца файла и способы

копирования/ перемещения ряда символов). В число этих свойств входит определение буфера, используемого потоком данных.

Стандартная библиотека потокового ввода-вывода для языка C++ ориентирована, прежде всего, на

следующую схему работы с потоком ввода: «читать данные из потока, пока очередная операция чтения не закончится неудачей». Поэтому никакая ошибка, связанная с потоками, не приводит к аварийному

завершению программы. Вместо этого изменяется состояние соответствующего потока, после чего все операции, связанные с этим потоком, блокируются.

Для обнаружения ошибки программа должна явным образом проверять состояние потока, а для продолжения работы с потоком (если это возможно) — сбрасывать состояние ошибки.

Средства, предусмотренные для проверки и изменения состояния потока; все эти средства реализованы в классе ios

(точнее, в классе-шаблоне basic\_ios) и поэтому

доступны для всех рассматриваемых далее потоков:

good: в данный момент ошибок, связанных с потоком, не обнаружено;

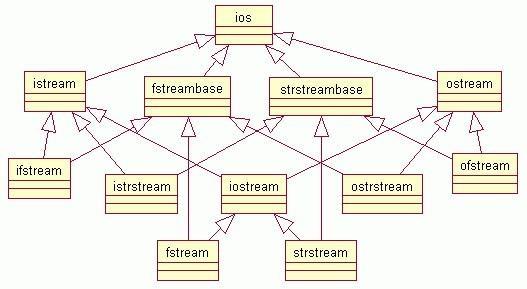
bad: при работе с потоком произошла фатальная ошибка, которую, скорее всего, не удастся исправить;

eof: прочесть очередной элемент данных не удалось, так как обнаружен конец потока;

fail: произошла какая-то ошибка; возможно, это фатальная ошибка (тогда значение true вернет и функция bad), возможно, это ошибка, связанная с концом потока (тогда значение true вернет и функция eof), возможно, это другая ошибка ввода-вывода или преобразования данных (в этой

ситуации можно попытаться продолжить работу с потоком, сбросив состояние ошибки).

Класс ios содержит средства для форматированного ввода-вывода и проверки ошибок.



Стандартные потоки (istream, ostream, iostream) служат для работы с терминалом. Строковые потоки (istrstream, ostrstream, strstream) служат для ввода-вывода из строковых буферов, размещенных в

памяти. Файловые потоки (ifstream, ofstream, fstream ) служат для работы с файлами. ios (базовый

потоковый класс), streambuf (буферизация потоков), istream (потоки ввода), ostream (потоки вывода),

iostream (двунаправленные потоки), istrstream (строковые потоки ввода), ostrstream (строковые потоки вывода), strstream (двунаправленные строковые потоки), ifstream (файловые потоки ввода), ofstream (файловые потоки вывода), fstream (двунаправленные файловые потоки).

Потоковые классы делятся на три группы (шаблонов классов):

basic\_istream, basic\_ostream - общие потоковые классы, которые могут быть связаны с любым буферным объектом; basic\_iostream - потоковые классы для считывания и записи файлов; basic\_istringstream, basic\_ostringstream - потоковые классы для объектов-строк.

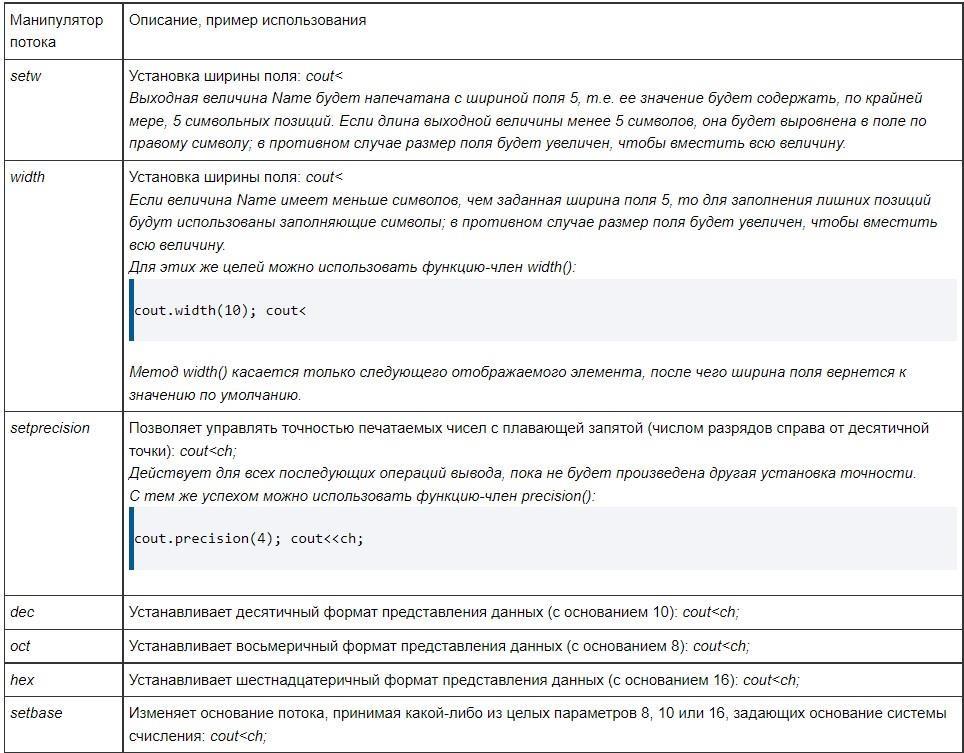
**42.Классы ostream, istream, iostream, их определения.**

**Манипуляторы потоков, схема их работы. Примеры**

**манипуляторов. Манипуляторы с параметрами, заголовочный файл <iomanip>. Файловые потоки и строковые потоки.**

ifstream, производный от istream, связывает ввод программы с файлом; ofstream, производный от ostream, связывает вывод программы с файлом;

fstream, производный от iostream, связывает как ввод, так и вывод программы с файлом.



Есть три основных класса файлового ввода/вывода в языке C++: ifstream (является [дочерним](https://ravesli.com/urok-154-bazovoe-nasledovanie-v-c/) классу [istream](https://ravesli.com/urok-208-funktsional-klassa-istream/));

ofstream (является дочерним классу [ostream](https://ravesli.com/urok-209-funktsional-klassov-ostream-i-ios-formatirovanie-vuvoda/));

fstream (является дочерним классу iostream ).

Строковые потоки — *это универсальные средства перевода внутренних двоичных значений в строки (выходной строковый поток) и обратно (входной строковый поток)*.

можно в качестве объектов объявлять строковые потоки трех видов: входной istringstream

выходной ostringstream двунаправленный stringstream

**43.Буфера потоков, классы basic\_streambuf и streambuf. Итераторы потоков: istream\_iterator и ostream\_iterator, их схема**

**реализации.**

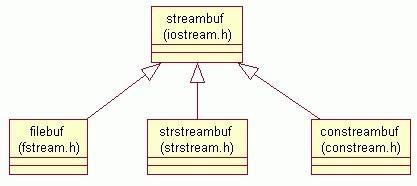
Буфера потоков. Потоки выполняют обмен данными между прикладной программой и файлом через буферы, которые являются объектами классов filebuf и stringbuf соответственно. Эти классы являются производными от абстрактного класса streambuf. Каждый буфер содержит в зависимости от режима доступа один или два символьных массива: один для ввода, а второй для вывода данных. Каждый

поток поддерживает внутренний указатель на буфер. Ввод-вывод данных из буфера в файл или в строку называется *синхронизацией буфера с внешним устройством ввода-вывода.*

Классы basic\_streambuf и streambuf. Вся система ввода-вывода C++ построена на двух связанных, но различных иерархиях шаблонных классов. Первая выведена из класса ввода-вывода нижнего уровня basic\_streambuf. Этот класс поддерживает базовые операции ввода и вывода нижнего уровня и

обеспечивает основополагающую поддержку для всей системы ввода-вывода C++.

Класс streambuf обеспечивает организацию и взаимосвязь буферов ввода-вывода, размещаемых в

памяти, с физическими устройствами ввода-вывода. Методы и данные класса streambuf программист явно обычно не использует. Этот класс нужен другим классам библиотеки ввода-вывода. Он доступен и программисту для создания новых классов на основе уже существующих.

Итераторы потоков: istream\_iterator и ostream\_iterator, их схема реализации.

Стандартные потоковые итераторы istream\_iterator<T> и ostream\_iterator<T> (шаблонные классы) определены в заголовочном файле <iterator>.

Имеются два варианта конструктора для итератора потокового чтения istream\_iterator: вариант с

параметром-потоком stream создает итератор для чтения из данного потока, вариант без параметров создает итератор, обозначающий конец потока.

Ниже перечислены свойства потоковых итераторов чтения:

тип T определяет тип элементов данных, которые считываются из потока;

чтение элемента из потока выполняется в начальный момент работы с итератором, а затем при каждой операции инкремента ++;

имеются два варианта операции ++: префиксный инкремент (++p) и постфиксный инкремент (p++);

при достижении конца потока итератор становится равным итератору конца потока;

Для итератора потоковой записи ostream\_iterator<T> также определены два конструктора: первый конструктор содержит единственный параметр stream, задающий поток вывода, а второй конструктор дополнительно к параметру stream содержит второй параметр delim, задающий разделитель, который добавляется в поток вывода после каждого выведенного элемента (если параметр delim не указан, то между выводимыми элементами никакой разделитель не добавляется).

Ниже перечислены свойства потоковых итераторов записи:

специальный конструктор для создания итератора конца потока вывода не предусмотрен;

операции \* и ++ не выполняют никаких действий и просто возвращают сам итератор;

операция присваивания p = выражение (где p – имя итератора записи) записывает значение выражения в поток вывода.