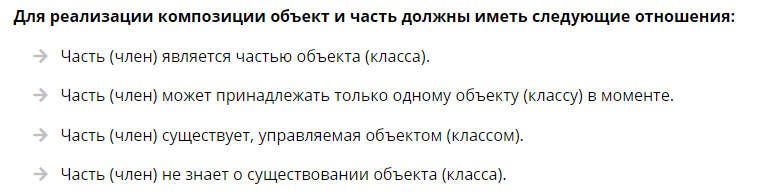
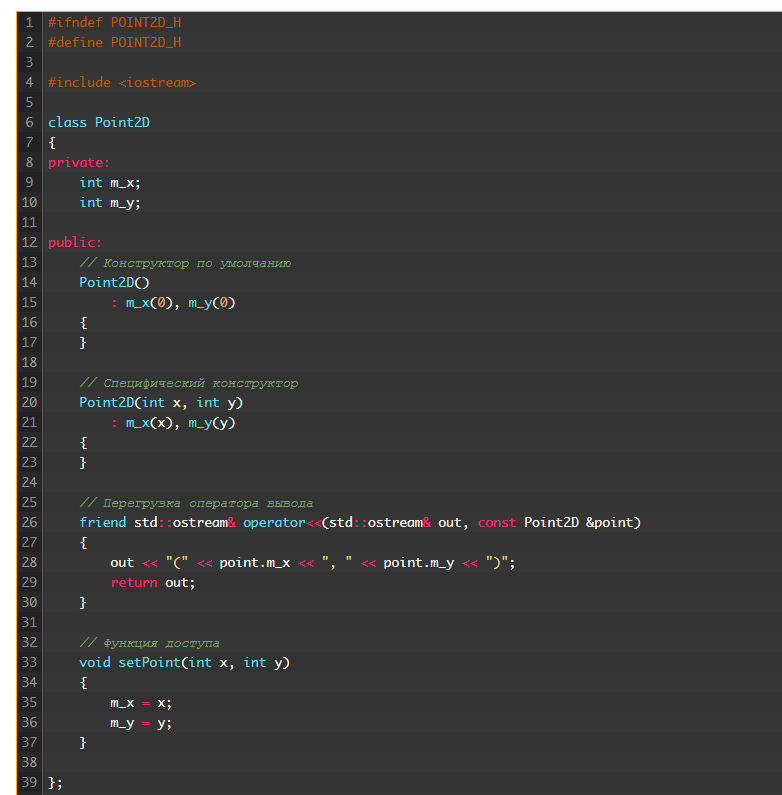
**[Композиция объектов](https://ravesli.com/urok-147-kompozitsiya-obektov/" \t "_blank)** - когда сложные классы состоят из более простых классов и типов данных. Композиция объектов идеально подходит для создания новых объектов, [**типом отношений**](https://ravesli.com/urok-146-otnosheniya-mezhdu-obektami/) которых является «имеет». 





Класс Point2D является целым, которое состоит из частей: x и y, продолжительность жизни которых напрямую зависит от продолжительности жизни объектов класса Point2D. Класс Creature также является целым, которое состоит из частей: m\_name и m\_location, продолжительность жизни которых также зависит от продолжительности жизни объектов класса Creature.

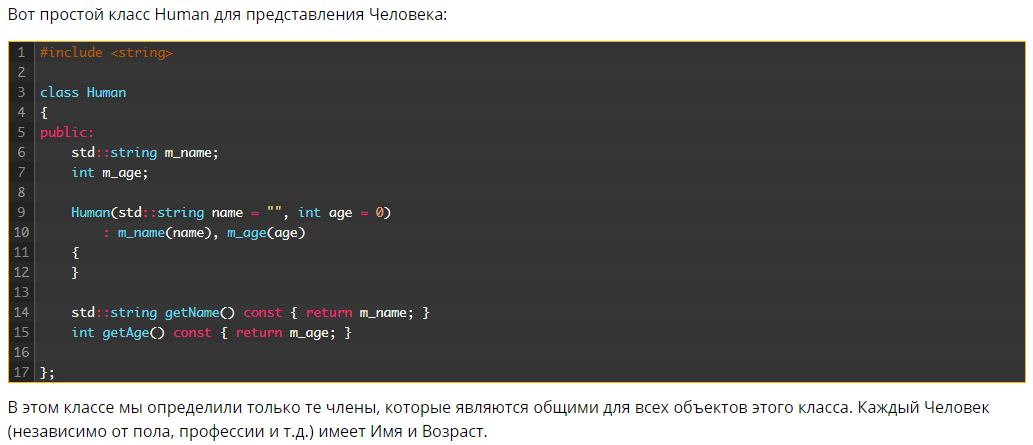
Каждый отдельный класс можно сохранить относительно простым/понятным и сфокусировать на выполнение одной конкретной задачи. Таким образом, писать классы легче и понимать их проще.

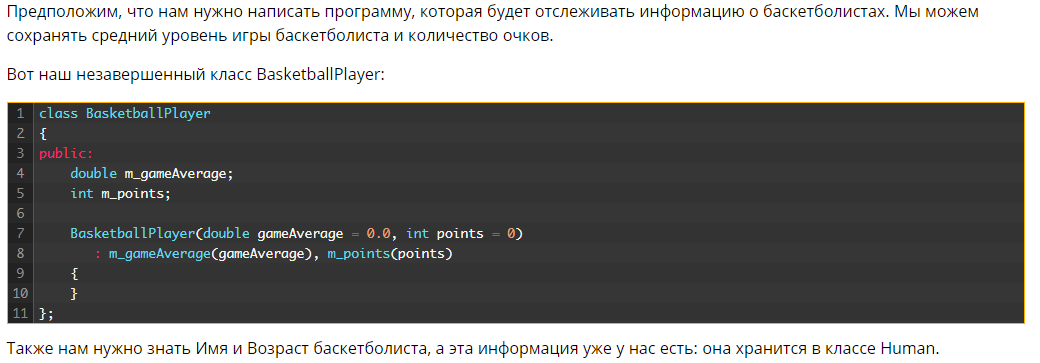
Каждый подкласс может быть автономным, что делает его многоразовым. Например, мы можем повторно использовать наш класс Point2D в совершенно другой программе.

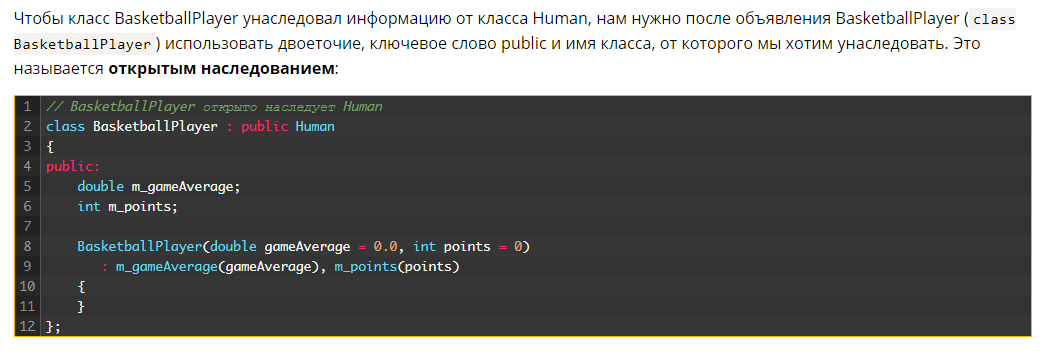
Родительский класс может оставить выполнение большей части сложной работы на подклассы, а сам сосредоточиться на координации потока данных между подклассами.

Однако композиция объектов является лишь одним из двух основных способов, с помощью которых вы можете создавать сложные классы в языке C++. Второй способ — это **наследование**, которое моделирует **тип отношения «является»** между двумя объектами.

В отличие от композиции объектов, которая включает в себя создание новых объектов путем объединения других объектов, наследование включает в себя создание новых объектов путем непосредственного сохранения свойств и поведения других объектов, а затем их расширения или наоборот — конкретизации. Подобно композиции объектов, наследование происходит повсюду в реальной жизни. Например, технологические продукты (компьютеры, смартфоны и т.д.) наследуют функционал от своих предшественников, при этом добавляя что-то свое (новое, уникальное) и сохраняя обратную совместимость.







Когда BasketballPlayer наследует свойства класса Human, то BasketballPlayer приобретает методы и переменные-члены класса Human. Кроме того, BasketballPlayer имеет еще два своих собственных члена: m\_gameAverage и m\_points. Здесь есть смысл, так как эти свойства специфичны только для BasketballPlayer, а не для каждого Human-а.

Таким образом, объекты BasketballPlayer будут иметь 4 члена:

   m\_gameAverage и m\_points от BasketballPlayer; m\_name и m\_age от Human.

Использование наследования означает, что нам не нужно переопределять информацию из родительских классов в дочерних. Мы автоматически получаем методы и переменные-члены суперкласса через наследование, а затем просто добавляем специфичные методы или переменные-члены, которые хотим. Это не только экономит время и усилия, но также является очень эффективным: если мы когда-либо обновим или изменим базовый класс (например, добавим новые функции или исправим ошибку), то все наши производные классы автоматически унаследуют эти изменения!

Например, если мы добавим новый метод в Human, то дочерние классы автоматически получат доступ к нему. Это позволяет создавать новые классы более простым, интуитивно-понятным способом! Наследование позволяет повторно использовать классы путем наследования членов этих классов другими классами.

Наследование класса определяется статически на этапе компиляции, его проще использовать, поскольку оно напрямую поддержано языком программирования. В случае наследования классов упрощается также задача модификации существующей реализации. Если подкласс замещает лишь некоторые операции, то могут оказаться затронутыми и остальные унаследованные операции, поскольку не исключено, что они вызывают замещенные.

Один общий недостаток использования композиции вместо наследования заключается в том, что методы, предоставляемые отдельными компонентами, могут быть реализованы в производном типе, даже если они являются только методами пересылки . Напротив, наследование не требует повторной реализации всех методов базового класса в производном классе. Скорее, производный класс должен только реализовать (переопределить) методы, имеющие другое поведение, чем методы базового класса. Это может потребовать значительно меньших усилий при программировании, если базовый класс содержит много методов, обеспечивающих поведение по умолчанию, и только некоторые из них должны быть переопределены в производном классе.