9. ОБЪЕКТЫ И КЛАССЫ

Оглавление

Классы и объекты являются основными концепциями объектно-ориентированного программирования — ООП.

Объектно-ориентированное программирование (ООП) — расширение структурного программирования, в котором основными концепциями являются понятия классов и объектов. Основное отличие языка программирования C++ от C

состоит в том, что в языке программирования С нет классов, а, следовательно, язык С не поддерживает ООП, в отличие от С++.

§9.1 Определение объекта

Объект – это достаточно общее и широкое понятие, которое может иметь разные трактовки. В широком смысле слова **объект – это любая сущность, имеющая некоторый набор свойств (параметров, характеристик) и обладающая некоторым поведением (функциональностью).**

Отсюда следует, что объектом может быть практически все, что угодно – **материальные** предметы (компьютер, автомобиль, человек), **логические** понятия (файл, документ, таблица, список, управляющая кнопка, окно в многооконной графической системе, форма, приложение), **математические** понятия (например, геометрические объекты типа отрезок или окружность).

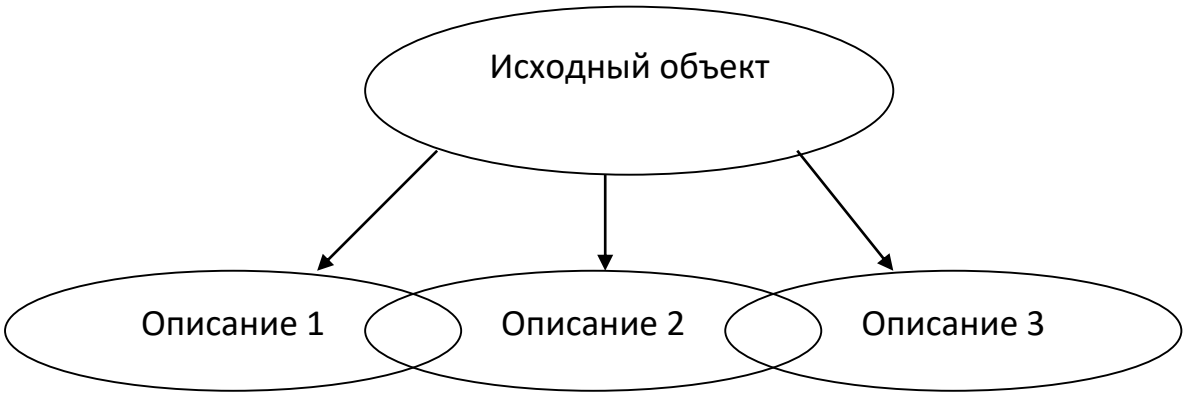
Например, когда вы смотрите на человека, вы видите его как объект. При этом объект определяется двумя компонентами: свойствами и поведением. У человека имеются такие свойства как цвет глаз, возраст, вес и т.д. Человек также обладает поведением, то есть он ходит, говорит, дышит и т.д.

Свойства и поведение объекта содержатся в объекте **одновременно**. Слово «одновременно» в данном случае определяет ключевую разницу между ООП и другими методологиями программирования. При ООП атрибуты и поведения размещаются в рамках одного объекта, в то время как при процедурном или структурном программировании атрибуты и поведение обычно разделяются.

Под объектом в узком смысле можно понимать некоторое **формализованное описание** рассматриваемой сущности, т.е. **модель** исходного объекта**.** Для такого описания нужен некоторый язык, например – язык программирования. В этом случае объект становится элементом языка, и именно в таком контексте он и будет рассматриваться далее.

Моделирование реальных объектов – это важнейший этап разработки объектных программ, во многом определяющий успех всего проекта. Необходимо понимать, что любая модель – это только **часть** исходного моделируемого объекта, отражающая те свойства и атрибуты объекта, которые **наиболее важны** с точки зрения решаемой задачи. Изменение постановки задачи может привести к существенному изменению

модели. Особенно это характерно для сложных объектов. Вряд ли имеет смысл строить всеобъемлющую модель «на все случаи жизни», поскольку такая модель будет очень громоздкой и практически непригодной для использования. Поэтому один из важнейших принципов построения моделей – это принцип **абстрагирования**, т.е. выделение наиболее существенных для решаемой задачи черт исходного объекта и отбрасывание второстепенных деталей. Отсюда следует, что для одного и того же исходного объекта можно построить **разные** модели, каждая из которых будет отражать только наиболее важные в данный момент черты объекта.



Например, для исходного объекта «человек» можно построить следующие информационные модели:

- человек как студент учебного заведения
- человек как сотрудник организации
- человек как пациент учреждений здравоохранения
- человек как клиент государственных органов управления и т.д.

Из этого примера видно, что весьма часто различные модели одной и той же исходной сущности «пересекаются», имеют общие свойства (это отражено и на приведенной выше схеме): все студенты, сотрудники, клиенты имеют фамилию, дату рождения, место проживания, пол и т.д.

Как показала практика последних 10-15 лет, использование объектов при разработке **сложных** информационных систем оказалось очень удачным решением. Заказчики информационных систем, как правило, не являются специалистами в области программного обеспечения и поэтому не воспринимают такие программистские термины как переменные, подпрограммы, массивы, файлы и т.д. Гораздо легче воспринимаются **объекты** как элементы предметной области.

Разработчики на основе общения с заказчиком и на основе анализа предметной области должны построить **адекватные описания основных сущностей** решаемой задачи, перевести эти описания в необходимый **формальный** вид и создать объектную программу как **набор взаимодействующих объектов**.

Достоинством объектных программ является их четкая структуризация – каждый объект выполняет свою строго определенную задачу, а их совместная работа позволяет достичь поставленных целей.

Программный объект – это условное понятие, с которым связывается набор некоторых данных и программный код обработки этих данных

объект : данные + программный код

Основное преимущество ООП заключается в том, что и данные, и операции (код), используемые для манипулирования ими, инкапсулируются в одном объекте. Например, при перемещении объекта по сети он передается целиком, включая данные и поведение. Объединение в рамках объекта некоторых данных и соответствующего программного кода рассматривается как проявление одного из базовых принципов объектного подхода – принципа **инкапсуляции** (**encapsulation**).

Инкапсуляция позволяет рассматривать объект в виде некоторой достаточно самостоятельной **программной единицы**, полностью отвечающей за хранение и обработку своих данных и предоставляющей посторонним пользователям четко определенный набор услуг. Однако объект — это не только поставщик услуг, но чаще всего еще и потребитель услуг других объектов.

Хорошим примером этой концепции является объект, загружаемый браузером. Часто бывает так, что браузер заранее не знает, какие действия будет выполнять определенный объект, поскольку он еще «не видел» кода. Когда объект загрузится, браузер выполнит код, содержащийся в этом объекте, а также использует заключенные в нем данные.

Относительно связываемых с объектом **данных** надо отметить следующее:

1. каждый элемент данных часто называют **свойством** объекта (хотя есть и различия в трактовке этого термина)
2. объект может содержать **любое разумное** число данных-свойств

3. набор данных-свойств определяется при описании объекта и при выполнении программы изменяться не может, могут изменяться лишь **значения** свойств
4. текущие **значения** свойств определяют текущее **состояние** объекта
5. для хранения значений свойств необходима **память** (как для обычных переменных), и поэтому объекты программы **потребляют** оперативную память
6. свойства могут иметь разные типы: **простейшие** (целочисленные, символьные, логические), **структурные** (строки, массивы, списки) и даже **объектные**
7. в соответствии с принципом **инкапсуляции** элементы данных **рекомендуется** делать **недоступными** для прямого использования за пределами объекта (**закрытость** данных)

В качестве **примеров** свойств можно привести:

- объект «Студент»: фамилия, имя, дата рождения, место проживания, группа, специальность, массив оценок;
- объект «Файл»: имя, размер, дата создания, тип;
- объект «Окно»: положение на экране, размеры, тип, фон заполнения, тип рамки, оформление заголовка;
- объект «Автомобиль»: стоимость, тип, марка, мощность двигателя, цвет и т.д.;
- объект «Окружность»: координаты центра, радиус, цвет.

Относительно связываемого с объектом **программного кода** необходимо отметить следующие моменты:

- программный код разбивается на отдельные **подпрограммы**, которые принято называть **методами**
- набор методов определяет выполняемые объектом функции и тем самым реализует поведение объекта
- набор методов определяется при описании объекта и при выполнении программы уже не изменяется
- методы могут иметь **ограничения по доступности**: некоторые методы можно сделать недоступными (**закрытыми**) за пределами объекта, но всегда должен быть определен набор **открытых** методов, образующих внешний интерфейс объекта

- вызов одного из открытых методов соответствует запросу услуги, реализуемой данным методом
- среди методов выделяют один или несколько специальных **методов-конструкторов** и иногда – один **метод-деструктор**, назначение которых рассматривается чуть дальше
- в соответствии с принципом **инкапсуляции** для доступа извне к закрытым данным могут вводиться специальные **методы доступа**

Еще одна группа очень часто используемых методов – это **методы доступа** к закрытым свойствам объекта. Введение таких методов позволяет организовать **контролируемый** доступ к внутренним данным объекта. В общем случае для закрытого свойства можно ввести **два** метода доступа:

- метод для **чтения** хранящегося в свойстве значения (часто такие методы называют **get-методами**)
- метод для **изменения** значения свойства (**set-метод**)

Набор используемых с каждым свойством методов доступа определяется при разработке объекта и позволяет для каждого свойства организовать **необходимый уровень доступа**. Если объявлены оба метода, то пользователи имеют **полный** доступ к соответствующему свойству. Если объявлен **только get-метод**, то пользователь может лишь **просматривать** значение свойства, но не изменять его. Наконец, если для свойства нет ни одного метода, то такое свойство **полностью закрыто** для пользователя.

Иногда set-методы кроме изменения значений свойств выполняют некоторую **дополнительную** работу, например – проверяют новые значения свойств.

Кроме конструкторов и методов доступа, объекты практически всегда имеют еще и некоторый набор **специфических** методов, определяющих **функциональность** объекта. Например:

- объект «Окружность»: отображение на экране, перемещение, растяжение/сжатие, вычисление длины окружности.
- объект «Список»: добавление элемента, удаление элемента, поиск элемента, сортировка элементов;

- объект «Окно»: отображение, перемещение, изменение атрибутов, изменение размеров;
- объект «Студент»: посещение занятий, выполнение заданий, сдача контрольных точек, оплата обучения;

В итоге объектная программа представляет собой **набор взаимодействующих объектов,** которые обращаются друг к другу за выполнением необходимых действий.

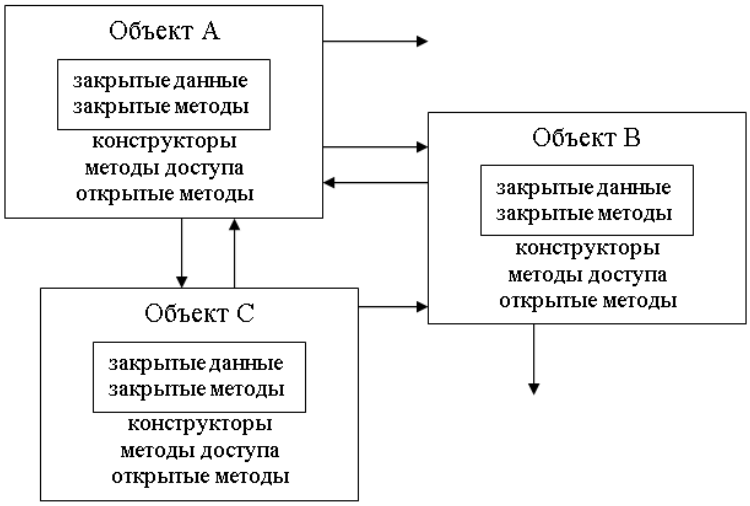


Рисунок 9.1

При этом каждый объект проходит определенный **жизненный цикл:**

1. объект **создается** методом-конструктором
2. объект **используется** другими объектами, предоставляя им свои открытые методы и неявно – закрытые данные
3. объект **уничтожается** (явно деструктором или неявно механизмом сборки мусора)

Очевидно, что при работе объектной программы одновременно может существовать **множество** однотипных программных объектов (множество файлов, множество окон, множество студентов). Поэтому необходим инструмент **формального описания** таких **однотипных объектов** и в качестве такого инструмента выступает следующее важнейшее понятие объектного подхода - **класс.**

§9.2 Определение классов

Класс представляет собой **формализованный способ описания однотипных объектов**, т.е. объектов с **одинаковым** набором свойств и методов**.** Именно при описании класса перечисляются свойства и реализуются методы соответствующих

объектов. Разработка объектной программы начинается с описания необходимых классов.

На основе одного класса можно создать **любое разумное** (все ресурсы вычислительной системы конечны!) число объектов, называемых **экземплярами** этого класса. Все используемые в программе объекты должны быть экземплярами некоторых классов, стандартных или собственных. Соответствие между понятиями «объект» и «класс» аналогично соответствию между понятиями «переменная» и «тип данных».

Правила описания классов различны для разных языков, тем не менее можно отметить ряд **общих** моментов.

Описание класса включает в себя:

- **заголовок** класса, включающий специальную директиву **class** и имя класса; практически всегда в заголовке класса задается дополнительная информация о классе;
- **тело** класса, содержащее перечень **свойств** (полей данных), **заголовки** методов и их **программную реализацию** (не всегда).

В классах допускается объявлять так называемые **абстрактные** методы, у которых есть **только заголовок** (имя и параметры), но **нет программной реализации**. Если класс содержит **хотя бы один** абстрактный метод, он сам считается **абстрактным**. Важная особенность абстрактных классов состоит в том, что **объекты-экземпляры на их основе создавать нельзя**! Абстрактные классы используются для описания абстрактных понятий.

Общие правила описания свойств:

- каждое свойство объявляется как обычная переменная, т.е. для нее обязательно задается **имя** и **тип** (простейший, структурный или объектный)
- имена всех свойств в классе должны быть различными
- свойства рекомендуется объявлять **закрытыми** с помощью специальной директивы **private** (такие свойства доступны для прямого использования только внутри методов данного класса); открытые свойства (если необходимо нарушить принцип инкапсуляции) объявляются с помощью директивы **public**.

Общие правила описания методов:

- метод оформляется как обычная подпрограмма, с указанием **имени**, формальных **параметров** (если необходимо) и типа возвращаемого значения

- допускается объявлять **несколько** методов с **одним и тем же** именем, но **разными** наборами формальных параметров (так называемая **перегрузка** методов, **overloading**); наборы параметров должны отличаться либо их числом, либо типами параметров
- каждый метод должен быть объявлен либо как **открытый** (директива **public**), либо как **закрытый (private**); закрытые методы могут использоваться только другими методами данного класса
- метод может быть объявлен **статическим**; такой метод можно вызывать **БЕЗ** создания объектов!
- **абстрактные** методы объявляются **с помощью** директивы **abstract**
- методы доступа для **чтения** значений закрытых свойств (Get-методы) **рекомендуется** именовать с префиксом **Get** (например, GetColor, GetSize) и оформлять как **функцию** без параметров, возвращающую тип соответствующего свойства
- методы доступа для **изменения** значений закрытых свойств (Set-методы) **рекомендуется** именовать с префиксом **Set** (например, SetColor, SetSize) и оформлять как процедуру (void-функцию) с одним **входным** параметром

В ООП существует три основных принципа построения классов:

1. **Инкапсуляция** — это свойство, позволяющее объединить в классе и данные, и методы, работающие с ними и скрыть детали реализации от пользователя.
2. **Наследование** — это свойство, позволяющее создать новый класс-потомок на основе уже существующего, при этом все характеристики класса родителя присваиваются классу-потомку.
3. **Полиморфизм** — свойство классов, позволяющее использовать объекты классов с одинаковым интерфейсом без информации о типе и внутренней структуре объекта.

§9.3 Классы (C++)

§9.3.1 Объявление классов

При объявлении класса используется ключевое слово `class`, сопровождаемое именем класса и блоком операторов `{...}`, который заключает в фигурные скобки набор свойств и методов, а завершается точкой с запятой.

Объявление класса сродни объявлению функции. Оно оповещает компилятор о классе и его свойствах. Только объявление класса никак не влияет на выполнение программы, поскольку класс следует использовать, как и функцию следует вызвать.

Например, определим простейший класс:

```
class Person
{
        //модификатор_доступа:
        //набор свойств
        //набор методов
                                };
```

В данном случае класс называется `Person` и представляет человека. После названия класса идет блок кода, который определяет тело класса.

После определения класса мы можем создавать его переменные:

```
1  class Person
2  {
3  };
4  int main()
5  {
6      Person person;
7      return 0;
8  }
```

Переменная класса называется **экземпляром** (или «объектом») **класса**. Точно так же, как определение переменной фундаментального типа данных (например, `int x`) приводит к выделению памяти для этой переменной, так же и создание объекта класса (например, `Person person`) приводит к выделению памяти для этого объекта. Класс можно представлять себе, как нечто вроде типа данных более высокого уровня.

Но данный класс мало что делает. Теперь изменим его:

```
 1  #include <iostream>
 2  #include <string>
 3  using namespace std;
 4  class Person
 5  {
 6  public:
 7  string name;
 8  int age;
```

```
 9  void move() {
10  cout << name << " is moving"<< endl;
11  }
12  };
13  int main()
14  {
15  Person person;
16  person.name = "Tom";
17  person.age = 22;
18  cout << "Name: " << person.name << "\tAge: " << person.age << endl;
19  person.move();
20  return 0;
21  }
```

Теперь класс `Person` имеет две переменных-свойства `name` и `age`, которые предназначены для хранения имени и возраста человека соответственно. Также класс определяет функцию-метод `move`, которая выводит строку на консоль. Также стоит обратить внимание на модификатор доступа `public:`, который указывает, что идущие после него переменные и функции будут доступны извне, из внешнего кода.

Затем в функции `main` создается один объект класса `Person`. К членам (переменным и функциям) класса **доступ осуществляется через оператор выбора членов (`.`)**. Например, через выражение

```
person.name = "Tom";
```

Можно передать значение переменной `name`. А с помощью выражения

```
string personName = person.name;
```

Получить это значение в какую-нибудь переменную. Ну и также мы можем вызывать функции у объекта:

```
person.move();
```

На объекты классов, как и на объекты других типов, можно определять указатели. Затем через указатель можно обращаться к членам класса - переменным и методам. Однако если при обращении через обычную переменную используется символ точка (`.`), то для обращения к членам класса через указатель применяется стрелка (`->`):

```
1  #include <iostream>
2  #include <string>
3  using namespace std;
4  class Person
```

```
5   {
6   public:
7   string name;
8   int age;
9   void move() {
10  cout << name << " is moving" << endl;
11  }
12  };
13  int main()
14  {
15  Person person;
16  Person *ptr = &person;
17  ptr->name = "Tom";
18  ptr->age = 22;
19  ptr->move();
20  cout << "Name: " << ptr->name << "\tAge: " << ptr->age << endl;
21  cout << "Name: " << person.name << "\tAge: " << person.age << endl;
22  return 0;
23  }
```

Изменения по указателю `ptr` в данном случае приведут к изменениям объекта `person`.

В языке Си структуры могут только хранить данные и не могут иметь связанных методов. В языке C++, после проектирования классов (используя ключевое слово `class`), Бьёрн Страуструп размышлял о том, нужно ли, чтобы структуры (которые были унаследованы из языка Си) имели связанные методы. После некоторых размышлений он решил, что нужно. Поэтому в программах, приведенных выше, мы также можем использовать ключевое слово `struct`, вместо `class`, и всё будет работать.

§9.3.2 Get/Set-методы

Концепция геттеров и сеттеров поддерживает концепцию сокрытия данных. В зависимости от класса, может быть уместным (в контексте того, что делает класс) иметь возможность получать или устанавливать значения закрытым переменным-членам класса.

Функция доступа — это короткая открытая функция, задачей которой является получение или изменение значения закрытой переменной-члена класса. Например:

```
1   class MyString
2   {
```

```
3  private:
4      char *m_string; // динамически выделяем строку
5      int m_length; // используем переменную для отслеживания длины строки
6
7  public:
8      int getLength() { return m_length; } // функция доступа для получения значения m_length
9  };
```

Здесь getLength() является функцией доступа, которая просто возвращает значение m_length.

Функции доступа обычно бывают двух типов:

- Get-геттеры (геттеры) — это функции, которые возвращают значения закрытых переменных-членов класса;
- Set-методы (сеттеры) — это функции, которые позволяют присваивать значения закрытым переменным-членам класса.

Вот пример класса, который использует геттеры и сеттеры для всех своих закрытых переменных-членов:

```
1  class Date
2  {
3  private:
4      int m_day;
5      int m_month;
6      int m_year;
7  public:
8      int getDay() { return m_day; } // геттер для day
9      void setDay(int day) { m_day = day; } // сеттер для day
10     int getMonth() { return m_month; } // геттер для month
11     void setMonth(int month) { m_month = month; } // сеттер для month
12     int getYear() { return m_year; } // геттер для year
13     void setYear(int year) { m_year = year; } // сеттер для year
14 };
```

В этом классе нет никаких проблем с тем, чтобы пользователь мог напрямую получать или присваивать значения закрытым переменным-членам этого класса, так как есть полный набор геттеров и сеттеров. В примере с классом `MyString` для переменной

`m_length` не было предоставлено сеттера, так как не было необходимости в том, чтобы пользователь мог напрямую устанавливать длину.

Предоставляйте функции доступа только в том случае, когда нужно, чтобы пользователь имел возможность получать или присваивать значения членам класса.

Хотя иногда вы можете увидеть, что геттер возвращает неконстантную ссылку на переменную-член — этого следует избегать, так как в таком случае нарушается инкапсуляция, позволяя caller-у изменять внутреннее состояние класса вне этого же класса. Лучше, чтобы ваши геттеры использовали тип возврата по значению или по константной ссылке.

§9.3.3 Модификаторы доступа

У каждого из нас есть много информации, часть которой доступна для окружающих людей, например, наши имена. Эта информация может быть названа открытой. Но есть информация, которую вы не захотите предоставлять всему миру или даже соседу, например, ваш доход. Эта информация является закрытой и часто хранится в тайне.

Языки программирования позволяет организовать контролируемый доступ к данным с помощью модификаторов доступа.

В C++ и C# члены класса классифицируются в соответствии с правами доступа на следующие три категории: **открытые** (public), **закрытые** (private) и **защищенные** (protected).

Закрытые члены классы, объявленные с помощью модификатора **`private`**, могут быть прочитаны и изменены только классом, к которому принадлежат. Уровень доступа может быть изменен при помощи соответствующих ключевых слов.

Модификатор **`public`** является спецификатором доступа (access specifier), то есть определяет параметры доступа к членам класса - переменным и функциям. В частности, он делает их доступными из любой части программы. По сути спецификатор `public` определяет общедоступный интерфейс класса.

Модификатор **`protected`** открывает доступ классам, производным от данного. То есть, производные классы получают свободный доступ к таким свойствам или метода. Все другие классы такого доступа не имеют.

То есть на примерах, рассмотренных в предыдущих параграфах, поля name и age являются открытыми, и мы можем присвоить им во внешнем коде любые значения.

В том числе можно присвоить какие-то недопустимые значения, например, отрицательное значение для возраста пользователя. Естественно это не очень хорошая ситуация.

Однако с помощью другого модификатора private мы можем скрыть реализацию членов класса, то есть сделать их закрытыми, инкапсулировать внутри класса.

Перепишем класс Person с исключением модификатора private:

```
1  #include <iostream>
2  #include <string>
3  using namespace std;
4  class Person
5  {
6  public:
7  Person(string n, int a)
8  {
9  name = n; age = a;
10 }
11 void move()
12 {
13 cout << name << " is moving" << endl;
14 }
15 void setAge(int a)
16 {
17 if (a > 0 && a < 100) age = a;
18 }
19 string getName()
20 {
21 return name;
22 }
23 int getAge()
24 {
25 return age;
26 }
27 private:
28 string name;
29 int age;
30 };
31 int main()
```

```
32  {
33  Person tom("Tom", 22);
34  // string personName = tom.name;
35  cout << "Name: " << tom.getName() << "\tAge: " << tom.getAge() << endl;
36  tom.setAge(31);
37  cout << "Name: " << tom.getName() << "\tAge: " << tom.getAge() << endl;
38  tom.setAge(291);
39  cout << "Name: " << tom.getName() << "\tAge: " << tom.getAge() << endl;
40  return 0;
41  }
```

Теперь переменные `name` и `age` в классе `Person` являются закрытыми, поэтому мы не можем обратиться к ним напрямую. Мы можем к ним обращаться только внутри класса.

Чтобы все-таки можно было получить извне значения переменных `name` и `age`, определены дополнительные функции `getAge` и `getName`. Установить значение переменной `name` напрямую можно только через конструктор, а значение переменной `age` - через конструктор или через функцию `setAge`. При этом функция `setAge` устанавливает значение для переменной `age`, если оно соответствует определенным условиям.

Таким образом, состояние класса скрыто извне, к нему можно получить доступ только посредством дополнительно определенных функций (set-методов, get-методов и других), который представляют интерфейс класса.

Открытые члены классов составляют **интерфейс класса**. Поскольку доступ к открытым членам класса может осуществляться извне класса, то открытый интерфейс и определяет, как программы, использующие класс, будут взаимодействовать с этим же классом.

Если модификатор доступа явно необъявлен, то класс по умолчанию устанавливает всем своим членам спецификатор доступа `private`. Структура же по умолчанию устанавливает всем своим членам спецификатор доступа `public`.

§9.4 Классы (C#)

§9.4.1 Объявление классов

C# является полноценным объектно-ориентированным языком. Это значит, что программу на C# можно представить в виде взаимосвязанных взаимодействующих между собой объектов.

По умолчанию проект консольного приложения уже содержит один класс `Program`, с которого и начинается выполнение программы.

По сути класс представляет новый тип, который определяется пользователем. Класс определяется с помощью ключевого слова `class`.

Давайте напишем общую структуру класса, которая содержит только 2 переменные класса и 2 метода (чтобы было проще понимать, методы — это те же самые функции, но реализованные внутри класса):

```
1  class имя_класса
2  {
3  уровень_доступа тип переменная1;
4  уровень_доступа тип переменная2;
5  уровень_доступа тип_возвращаемого_значени метод1(параметры) {
6  //тело метода
7  }
8  уровень_доступа тип_возвращаемого_значени метод2(параметры) {
9  //тело метода
10 }
11 }
```

Класс можно определять внутри пространства имен, вне пространства имен, внутри другого класса. Как правило, классы помещаются в отдельные файлы. Но в данном случае поместим новый класс в файле, где располагается класс `Program`. То есть файл Program.cs будет выглядеть следующим образом:

```
1  using System;
2  namespace HelloApp
3  {
4      class Person
5      {
6
7      }
8      class Program
9      {
10         static void Main(string[] args)
```

```
11          {
12          }
13      }
14  }
```

Вся функциональность класса представлена его членами:

1. полями (полями называются переменные класса);
2. свойствами (поля с методами доступа);
3. методами;
4. событиями.

Поле – это переменная, объявленная внутри класса. Как правило, поля объявляются с модификаторами доступа `private` либо `protected`, чтобы запретить прямой доступ к ним. Для получения доступа к полям следует использовать свойства или методы.

Пример объявления полей в классе:

```
class Student
{
   private string firstName;
   private string lastName;
   private int age;
   public string group; // не рекомендуется использовать public для поля
}
```

Например, определим в классе `Person` поля и метод:

```
 1  using System;
 2  namespace HelloApp
 3  {
 4      class Person
 5      {
 6          public string name; // имя
 7          public int age = 18;     // возраст
 8          public void GetInfo()
 9          {
10              Console.WriteLine($"Имя: {name}  Возраст: {age}");
11          }
12      }
```

```
13      class Program
14      {
15          static void Main(string[] args)
16          {
17              Person tom;
18          }
19      }
20  }
```

В данном случае класс `Person` представляет человека. Поле `name` хранит имя, а поле `age` - возраст человека. А метод `GetInfo` выводит все данные на консоль. Чтобы все данные были доступны вне класса `Person` переменные и метод определены с модификатором `public`. Поскольку поля фактически те же переменные, им можно присвоить начальные значения, как в случае выше, поле `age` инициализировано значением 18.

Так как класс представляет собой новый тип, то в программе мы можем определять переменные, которые представляют данный тип. Так, здесь в методе `Main` определена переменная `tom`, которая представляет класс `Person`. Но пока эта переменная не указывает ни на какой объект и по умолчанию она имеет значение `null`. Поэтому вначале необходимо создать объект класса `Person`.

§9.4.2 Свойства

Кроме обычных методов в языке C# предусмотрены специальные методы доступа, которые называют **свойства**. Они обеспечивают контролируемый доступ к полям классов и структур, узнать их значение или выполнить их установку.

Стандартное описание свойства имеет следующий синтаксис:

```
[модификатор_доступа] возвращаемый_тип произвольное_название
                    {
                // код свойства
                    }
```

Например:

```
1   class Person
2   {
3       private string name;
4       public string Name
5       {
```

```
6           get
7           {
8               return name;
9           }
10          set
11          {
12              name = value;
13          }
14      }
15  }
```

Здесь у нас есть закрытое поле `name` и есть общедоступное свойство `Name`. Хотя они имеют практически одинаковое название за исключением регистра, но это не более чем стиль, названия у них могут быть произвольные и не обязательно должны совпадать.

Через это свойство мы можем управлять доступом к переменной `name`. Стандартное определение свойства содержит блоки `get` и `set`. В блоке `get` мы возвращаем значение поля, а в блоке `set` устанавливаем. Параметр `value` представляет передаваемое значение.

Мы можем использовать данное свойство следующим образом:

```
Person p = new Person();
// Устанавливаем свойство - срабатывает блок Set
// значение "Tom" и есть передаваемое в свойство value
p.Name = "Tom";
// Получаем значение свойства и присваиваем его переменной - срабатывает блок Get
string personName = p.Name;
```

Свойства также позволяют вложить дополнительную логику, которая может быть необходима, например, при присвоении переменной класса какого-либо значения. Например, нам надо установить проверку по возрасту:

```
1   class Person
2   {
3       private int age;
4       public int Age
5       {
6           set
```

```
7           {
8               if (value < 18)
9               {
10                  Console.WriteLine("Возраст должен быть больше 17");
11              }
12              else
13              {
14                  age = value;
15              }
16          }
17          get { return age; }
18      }
19  }
```

Если бы переменная age была бы публичной, то мы могли бы передать ей извне любое значение, в том числе отрицательное. Свойство же позволяет скрыть данные объекты и опосредовать к ним доступ.

Блоки set и get не обязательно одновременно должны присутствовать в свойстве. Если свойство определяют только блок get, то такое свойство доступно только для чтения - мы можем получить его значение, но не установить. И, наоборот, если свойство имеет только блок set, тогда это свойство доступно только для записи - можно только установить значение, но нельзя получить.

Можно применять модификаторы доступа не только ко всему свойству, но и к отдельным блокам - либо get, либо set.

```
1   class Person
2   {
3       private string name;
4       public string Name
5       {
6           get
7           {
8               return name;
9           }
10          private set
11          {
12              name = value;
13          }
14      }
15  }
```

Теперь закрытый блок `set` мы сможем использовать только в данном классе - в его методах, свойствах, конструкторе, но никак не в другом классе:

```
Person p = new Person("Tom");
// Ошибка - set объявлен с модификатором private
//p.Name = "John";
Console.WriteLine(p.Name);
```

При использовании модификаторов в свойствах следует учитывать ряд ограничений:

- Модификатор для блока `set` или `get` можно установить, если свойство имеет оба блока (и `set`, и `get`);
- Только один блок `set` или `get` может иметь модификатор доступа, но не оба сразу;
- Модификатор доступа блока `set` или `get` должен быть более ограничивающим, чем модификатор доступа свойства. Например, если свойство имеет модификатор public, то блок set/get может иметь только модификаторы `internal`, `protected`, `private`.

Свойства управляют доступом к полям класса. Однако, что, если у нас с десяток и более полей, то определять каждое поле и писать для него однотипное свойство было бы утомительно. Поэтому в фреймворк .NET были добавлены **автоматические свойства**. Они имеют сокращенное объявление:

```
1  class Person
2  {
3      public string Name { get; set; }
4      public int Age { get; set; }
5      public Person(string name, int age)
6      {
7          Name = name;
8          Age = age;
9      }
10 }
```

На самом деле тут также создаются поля для свойств, только их создает не программист в коде, а компилятор автоматически генерирует при компиляции.

Преимущества автоматического свойства в том, что в любой момент времени при необходимости мы можем развернуть автосвойство в обычное свойство, добавить в него какую-то определенную логику.

Стоит учитывать, что нельзя создать автоматическое свойство только для записи, как в случае со стандартными свойствами.

Автосвойствам можно присвоить значения по умолчанию (инициализация автосвойств):

```
1  class Person
2  {
3      public string Name { get; set; } = "Tom";
4      public int Age { get; set; } = 23;
5  }
```

И если мы не укажем для объекта `Person` значения свойств `Name` и `Age`, то будут действовать значения по умолчанию.

Автосвойства также могут иметь модификаторы доступа.

```
1  class Person
2  {
3      public string Name { private set; get;}
4      public Person(string n)
5      {
6          Name = n;
7      }
8  }
```

§9.5 Конструкторы и инициализация объектов

Конструктор отвечает за **динамическое** создание нового объекта при **выполнении** программы, которое включает в себя:

- **выделение памяти**, необходимой для хранения значений свойств создаваемого объекта (совместно с операционной системой и средой поддержки выполнения программ)
- **занесение** в выделенную область **начальных значений** свойств создаваемого объекта

Важно понимать, что пока объект не создан, он не может предоставлять никакие услуги и поэтому конструктор должен вызываться **раньше** всех остальных методов. Важность конструктора определяется тем, что он **гарантированно** создает объект с какими-то **начальными** значениями свойств. Это позволяет устранить целый класс

неприятных логических ошибок, таких как использование неинициализированных данных.

Для одного и того же объекта можно предусмотреть **несколько** различных конструкторов, которые **по-разному** инициализируют свойства создаваемого объекта. Начальные значения свойств часто передаются конструктору как входные параметры.

§9.5.1 Конструкторы и инициализация объектов (C++)

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, или для выполнения любых шагов настройки, необходимых для используемого класса (например, открыть определенный файл или базу данных).

В отличие от обычных методов, конструкторы имеют определенные правила по поводу их имен:

- конструкторы всегда должны иметь то же имя, что и класс (учитываются верхний и нижний регистры);
- конструкторы не имеют типа возврата (даже `void`).

Так, изменим код программы, разработанный в прошлом параграфе, следующим образом:

```
1  #include <iostream>
2  #include <string>
3  using namespace std;
4  class Person
5  {
6  public:
7  string name;
8  int age;
9  Person(string n, int a)
10 {
11 name = n; age = a;
12 }
13 void move()
14 {
15 cout << name << " is moving" << endl;
16 }
```

```
};
int main()
{
Person person = Person("Tom", 22);
cout << "Name: " << person.name << "\tAge: " << person.age << endl;
person.name = "Bob";
person.move();
return 0;
}
```

Теперь в классе Person определен конструктор:

```
Person(string n, int a)
{
name = n; age = a;
}
```

По сути конструктор представляет функцию, которая может принимать параметры, и которая должна называться по имени класса. В данном случае конструктор принимает два параметра и передает их значения полям `name` и `age`.

Если в классе определены конструкторы, то при создании объекта этого класса необходимо вызвать один из его конструкторов.

Вызов конструктора получает значения для параметров и возвращает объект класса:

```
Person person = Person("Tom", 22);
```

После этого вызова у объекта `person` для поля `name` будет определено значение "Tom", а для поля `age` - значение 22. Впоследствии мы также сможем обращаться к этим полям и переустанавливать их значения.

Конструктор, который не имеет параметров (или имеет параметры, все из которых имеют значения по умолчанию), называется **конструктором по умолчанию**. Он вызывается, если пользователем не указаны значения для инициализации.

Подобным образом мы можем определить несколько конструкторов и затем их использовать:

```
#include <iostream>
#include <string>
using namespace std;
class Person
{
```

```
6   public:
7   string name;
8   int age;
9   Person(string n, int a)
10  {
11  name = n; age = a;
12  cout << "First constructor" << endl;
13  }
14  Person(string n): Person(n, 18) // вызов первого конструктора
15  {
16  cout << "Second constructor" << endl;
17  }
18  Person() : Person("Bob")    // вызов второго конструктора
19  {
20  cout << "Third constructor" << endl;
21  }
22  void move()
23  {
24  cout << name << " is moving" << endl;
25  }
26  };
27  int main()
28  {
29  Person tom("Tom", 22);
30  cout << "Name: " << tom.name << "\tAge: " << tom.age << endl;
31  Person sam("Sam");
32  cout << "Name: " << sam.name << "\tAge: " << sam.age << endl;
33  Person bob = Person();
34  cout << "Name: " << bob.name << "\tAge: " << bob.age << endl;
35  return 0;
36  }
```

Запись `Person(string n): Person(n, 18)` представляет вызов конструктора, которому передается значение параметра `n` и число 18. То есть второй конструктор делегирует действия по инициализации переменных первому конструктору. При этом второй конструктор может дополнительно определять какие-то свои действия.

Таким образом, следующее создание объекта

```
Person bob = Person();
```

будет использовать третий конструктор, который в свою очередь вызывает второй конструктор, а тот обращается к первому конструктору.

Для инициализации членов класса также можно использовать **списки инициализации**. Список инициализации характеризуется двоеточием (:) с последующим объявлением параметров, содержащихся в круглых скобках (…), индивидуальными переменными-членами и значениями для инициализации. Например, объявления вышеописанного конструктора через списки инициализации будут выглядеть следующим образом:

```
Person() : name(n), age(a)
{}
```

§9.5.2 Конструкторы и инициализация объектов (C#)

Если в классе не определено ни одного конструктора, то для этого класса автоматически создается конструктор по умолчанию. Такой конструктор не имеет параметров и не имеет тела.

Выше класс `Person` не имеет никаких конструкторов. Поэтому для него автоматически создается **конструктор по умолчанию**. И мы можем использовать этот конструктор. В частности, создадим один объект класса `Person`:

```
1  class Person
2  {
3  public string name; // имя
4  public int age;     // возраст
5  public void GetInfo()
6  {
7  Console.WriteLine($"Имя: {name}  Возраст: {age}");
8  }
9  }
10 class Program
11 {
12 static void Main(string[] args)
13 {
14 Person tom = new Person();
15 tom.GetInfo();      // Имя: Возраст: 0
16 tom.name = "Tom";
17 tom.age = 34;
18 tom.GetInfo();  // Имя: Tom Возраст: 34
19 Console.ReadKey();
20 }
21 }
```

Для создания объекта `Person` используется выражение `new Person()`. Оператор new выделяет память для объекта `Person`. И затем вызывается конструктор по умолчанию, который не принимает никаких параметров. В итоге после выполнения данного выражения в памяти будет выделен участок, где будут храниться все данные объекта `Person`. А переменная `tom` получит ссылку на созданный объект.

Если конструктор не инициализирует значения переменных объекта, то они получают значения по умолчанию. Для переменных числовых типов это число 0, а для типа `string` и классов - это значение `null` (то есть фактически отсутствие значения).

После создания объекта мы можем обратиться к переменным объекта `Person` через переменную `tom` и установить или получить их значения, например, tom.name = "Tom";.

Выше для инициализации объекта использовался конструктор по умолчанию. Однако мы сами можем определить **собственные конструкторы**:

```
1  class Person
2  {
3      public string name;
4      public int age;
5      public Person() { name = "Неизвестно"; age = 18; }      // 1 конструктор
6      public Person(string n) { name = n; age = 18; }         // 2 конструктор
7      public Person(string n, int a) { name = n; age = a; }   // 3 конструктор
8      public void GetInfo()
9      {
10         Console.WriteLine($"Имя: {name}  Возраст: {age}");
11     }
12 }
```

Теперь в классе определено три конструктора, каждый из которых принимает различное количество параметров и устанавливает значения полей класса. Используем эти конструкторы:

```
1  static void Main(string[] args)
2  {
3      Person tom = new Person();
       // вызов 1-ого конструктора без параметров
4      Person bob = new Person("Bob");
       //вызов 2-ого конструктора с одним параметром
```

```
5       Person sam = new Person("Sam", 25);
        // вызов 3-его конструктора с двумя параметрами
6       bob.GetInfo();
        // Имя: Bob  Возраст: 18
7       tom.GetInfo();
        // Имя: Неизвестно  Возраст: 18
8       sam.GetInfo();
        // Имя: Sam  Возраст: 25
9   }
```

При этом если в классе определены конструкторы, то при создании объекта необходимо использовать один из этих конструкторов.

§9.6 Деструктор

Деструктор отвечает за уничтожение объекта, т.е. освобождение памяти, выделенной объекту. В отличие от конструкторов, механизм деструкторов реализован не во всех языках: деструкторы есть в языке C++, в языке Delphi Pascal, предусмотрены (но практически не используются) в языке C# и отсутствуют в языке Java. В последних двух языках вместо деструкторов реализован специальный механизм «**сборки мусора**» (garbage collect), который способен при необходимости выполнять автоматическое освобождение памяти.

Чаще всего деструктор используют тогда, когда в конструкторе, при создании объекта класса, динамически был выделен участок памяти и необходимо эту память очистить, если эти значения уже не нужны для дальнейшей работы программы.

Важно запомнить следующее:

1. конструктор и деструктор, мы всегда объявляем в разделе public;
2. при объявлении конструктора, тип данных возвращаемого значения не указывается;
3. у деструктора также нет типа данных для возвращаемого значения, к тому же деструктору нельзя передавать никаких параметров;
4. имя класса и конструктора должно быть идентично;
5. имя деструктора идентично имени конструктора, но с приставкой ~ ;
6. в классе допустимо создавать несколько конструкторов, если это необходимо. Имена, согласно пункту 2 нашего списка, будут одинаковыми. Компилятор будет их различать по передаваемым параметрам (как при перегрузке

функций). Если мы не передаем в конструктор параметры, он считается конструктором по умолчанию;

7. обратите внимание на то, что в классе может быть объявлен только один деструктор.

Деструктор выполняет освобождение использованных объектом ресурсов и удаление нестатических переменных объекта. По сути деструктор - это функция, которая называется по имени класса (как и конструктор) и перед которой стоит тильда (~). Деструктор не имеет возвращаемого значения и не принимает параметров. Каждый класс может иметь только один деструктор.

Деструктор автоматически вызывается, когда удаляется объект. Удаление объекта происходит в следующих случаях:

1. когда завершается выполнение области видимости, внутри которой определены объекты
2. когда удаляется контейнер (например, массив), который содержит объекты
3. когда удаляется объект, в котором определены переменные, представляющие другие объекты
4. динамически созданные объекты удаляются при применении к указателю на объект оператора delete

Рассмотрим следующую ситуацию:

```
1  #include <iostream>
2  #include <string>
3  using namespace std;
4  class Person
5  {
6  public:
7  Person(std::string n)
8  {
9  name = n;
10 }
11 ~Person()
12 {
13 cout << "Destructor called for Person " << name << endl;
14 }
15 private:
16 string name;
17 };
```

```
18  int main()
19  {
20  Person tom("Tom");
21  Person *sam = new Person("Sam");
22  delete sam;     // вызывается деструктор для объекта sam
23  cout << "End of Main" << endl;
24  return 0;
25  } // вызывается деструктор для объекта tom
```

В классе Person определен деструктор:

```
1  ~Person()
2  {
3  cout << "Destructor called for Person " << name << endl;
4  }
```

Все, что делает данный деструктор, это выводит на консоль соответствующее сообщение. Как правило, деструкторы определяют код для освобождения ресурсов, но в данном случае нам не надо освобождать никаких ресурсов, и мы могли бы определить даже пустой деструктор.

В функции `main` создается переменная `tom`, которая хранит объект `Person`. Это обычная переменная, не указатель. И для нее деструктор будет вызываться, когда завершит выполнение та область видимости, где эта переменная определена, то есть функция `main`.

Также здесь определен указатель `bob`, который указывает на объект `Person`. Это динамический объект, который определяется с помощью ключевого слова `new`. Когда такие объекты выходят из области видимости, то для них автоматически не выполняется деструктор. Поэтому для вызова деструктора и удаления таких объектов применяется оператор delete:

```
Person *sam = new Person("Sam");
delete sam; // вызывается деструктор для объекта sam
```

При этом выполнение самого деструктора еще не удаляет сам объект. Непосредственно удаление объекта производится в ходе явной фазы удаления, которая следует после выполнения деструктора.

Стоит также отметить, что для любого класса, который не определяет собственный деструктор, компилятор по умолчанию сам создает деструктор.

§9.7 Типы значений и ссылочные типы в C#

Ранее мы рассматривали следующие элементарные типы данных: int, byte, double, string, object и др. Также есть сложные типы: структуры, перечисления, классы. Все эти типы данных можно разделить на типы значений, еще называемые значимыми типами, (value types) и ссылочные типы (reference types). Важно понимать между ними различия.

Типы значений:

1. Целочисленные типы (byte, sbyte, short, ushort, int, uint, long, ulong)
2. Типы с плавающей запятой (float, double)
3. Тип decimal
4. Тип bool
5. Тип char
6. Структуры (struct)

Ссылочные типы:

1. Тип object
2. Тип string
3. Классы (class)
4. Интерфейсы (interface)
5. Делегаты (delegate)

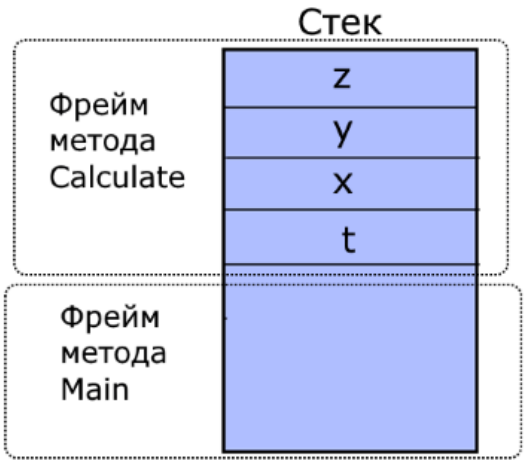
Для понимания различия между ними надо понять организацию памяти в .NET. Здесь память делится на два типа: стек и куча (heap). Параметры и переменные метода, которые представляют типы значений, размещают свое значение в стеке. Стек представляет собой структуру данных, которая растет снизу-вверх: каждый новый добавляемый элемент помещается поверх предыдущего. Время жизни переменных таких типов ограничено их контекстом. Физически стек - это некоторая область памяти в адресном пространстве.

Когда программа только запускается на выполнение, в конце блока памяти, зарезервированного для стека, устанавливается указатель стека. При помещении данных в стек указатель переустанавливается таким образом, что снова указывает на новое свободное место. При вызове каждого отдельного метода в стеке будет выделяться область памяти или фрейм стека, где будут храниться значения его параметров и переменных.

Например:

```
1  class Program
2  {
3      static void Main(string[] args)
4      {
5          Calculate(5);
6          Console.ReadKey();
7      }
8      static void Calculate(int t)
9      {
10         int x = 6;
11         int y = 7;
12         int z = y + t;
13     }
14 }
```

При запуске такой программы в стеке будут определяться два фрейма - для метода `Main` (так как он вызывается при запуске программы) и для метода `Calculate`:



При вызове этого метода `Calculate` в его фрейм в стеке будут помещаться значения t, x, y и z. Они определяются в контексте данного метода. Когда метод отработает, область памяти, которая выделялась под стек, впоследствии может быть использована другими методами.

Причем если параметр или переменная метода представляет тип значений, то в стеке будет храниться непосредсвенное значение этого параметра или переменной. Например, в данном случае переменные и параметр метода `Calculate` представляют значимый тип - тип int, поэтому в стеке будут храниться их числовые значения.

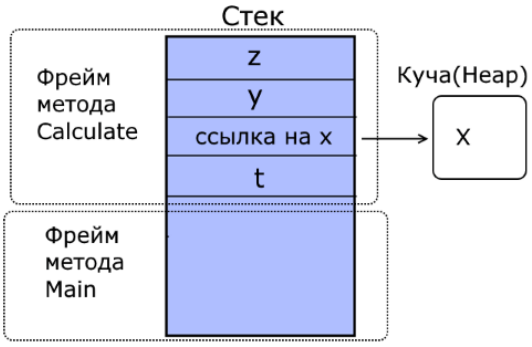
Ссылочные типы хранятся в куче или хипе, которую можно представить как неупорядоченный набор разнородных объектов. Физически это остальная часть памяти, которая доступна процессу.

При создании объекта ссылочного типа в стеке помещается ссылка на адрес в куче (хипе). Когда объект ссылочного типа перестает использоваться, в дело вступает автоматический сборщик мусора: он видит, что на объект в хипе нету больше ссылок, условно удаляет этот объект и очищает память - фактически помечает, что данный сегмент памяти может быть использован для хранения других данных.

Так, в частности, если мы изменим метод `Calculate` следующим образом:

```
1  static void Calculate(int t)
2  {
3      object x = 6;
4      int y = 7;
5      int z = y + t;
6  }
```

То теперь значение переменной x будет храниться в куче, так как она представляет ссылочный тип object, а в стеке будет храниться ссылка на объект в куче.

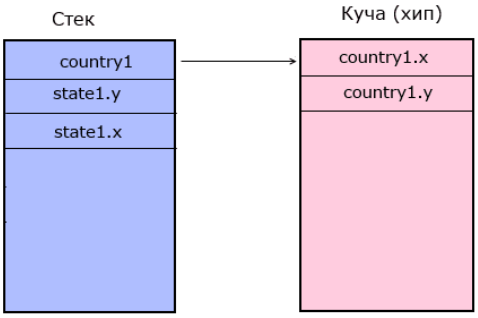


Теперь рассмотим ситуацию, когда тип значений и ссылочный тип представляют составные типы - структуру и класс:

```
1  class Program
2  {
3      private static void Main(string[] args)
4      {
5          State state1 = new State();
   // State - структура, ее данные размещены в стеке
6          Country country1 = new Country();
   // Country - класс, в стек помещается ссылка на адрес в хипе
7  // а в хипе располагаются все данные объекта country1
8      }
9  }
```

```
10  struct State
11  {
12      public int x;
13      public int y;
14      public Country country;
15  }
16  class Country
17  {
18      public int x;
19      public int y;
20  }
```

Здесь в методе `Main` в стеке выделяется память для объекта `state1`. Далее в стеке создается ссылка для объекта `country1` (`Country country1`), а с помощью вызова конструктора с ключевым словом new выделяется место в хипе (`new Country()`). Ссылка в стеке для объекта `country1` будет представлять адрес на место в хипе, по которому размещен данный объект.

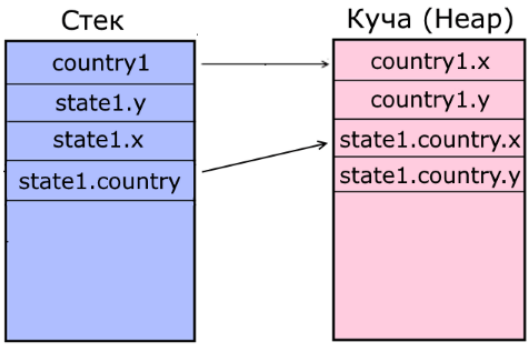


Таким образом, в стеке окажутся все поля структуры state1 и ссылка на объект `country1` в хипе.

Однако в структуре `State` также определена переменная ссылочного типа `Country`. Где она будет хранить свое значение, если она определена в типе значений?

```
1  private static void Main(string[] args)
2  {
3      State state1 = new State();
4      state1.country = new Country();
5      Country country1 = new Country();
6  }
```

Значение переменной `state1.country` также будет храниться в куче, так как эта переменная представляет ссылочный тип:



Копирование значений

Тип данных надо учитывать при копировании значений. При присвоении данных объекту значимого типа он получает копию данных. При присвоении данных объекту ссылочного типа он получает не копию объекта, а ссылку на этот объект в хипе. Например:

```
1  private static void Main(string[] args)
2  {
3      State state1 = new State(); // Структура State
4      State state2 = new State();
5      state2.x = 1;
6      state2.y = 2;
7      state1 = state2;
8      state2.x = 5; // state1.x=1 по-прежнему
9      Console.WriteLine(state1.x); // 1
10     Console.WriteLine(state2.x); // 5
11     Country country1 = new Country(); // Класс Country
12     Country country2 = new Country();
13     country2.x = 1;
14     country2.y = 4;
15     country1 = country2;
16     country2.x = 7; // теперь и country1.x = 7, так как обе
   ссылки и country1 и country2
17     // указывают на один объект в хипе
18     Console.WriteLine(country1.x); // 7
19     Console.WriteLine(country2.x); // 7
20     Console.Read();
21 }
```

Так как `state1` - структура, то при присвоении `state1 = state2` она получает копию структуры `state2`. А объект класса `country1` при присвоении `country1 = country2` получает ссылку на тот же объект, на который указывает `country2`. Поэтому с изменением `country2`, так же будет меняться и `country1`.

Объекты классов как параметры методов

Организацию объектов в памяти следует учитывать при передаче параметров по значению и по ссылке. Если параметры методов представляют объекты классов, то использование параметров имеет некоторые особенности. Например, создадим метод, который в качестве параметра принимает объект Person:

```
1   class Program
2   {
3       static void Main(string[] args)
4       {
5           Person p = new Person { name = "Tom", age=23 };
6           ChangePerson(p);
7           Console.WriteLine(p.name); // Alice
8           Console.WriteLine(p.age); // 23
9           Console.Read();
10      }
11      static void ChangePerson(Person person)
12      {
13          // сработает
14          person.name = "Alice";
15          // сработает только в рамках данного метода
16          person = new Person {name="Bill", age=45 };
17          Console.WriteLine(person.name); // Bill
18      }
19  }
20  class Person
21  {
22      public string name;
23      public int age;
24  }
```

При передаче объекта класса по значению в метод передается копия ссылки на объект. Эта копия указывает на тот же объект, что и исходная ссылка, потому мы можем изменить отдельные поля и свойства объекта, но не можем изменить сам объект. Поэтому в примере выше сработает только строка `person.name = "Alice"`.

А другая строка

```
person = new Person { name = "Bill", age = 45 }
```

создаст новый объект в памяти, и person теперь будет указывать на новый объект в памяти. Даже если после этого мы его изменим, то это никак не повлияет на ссылку `p` в методе `Main`, поскольку ссылка `p` все еще указывает на старый объект в памяти.

Но при передаче параметра по ссылке (с помощью ключевого слова **ref**) в метод в качестве аргумента передается сама ссылка на объект в памяти. Поэтому можно изменить как поля и свойства объекта, так и сам объект:

```
1  class Program
2  {
3      static void Main(string[] args)
4      {
5          Person p = new Person { name = "Tom", age=23 };
6          ChangePerson(ref p);
7          Console.WriteLine(p.name); // Bill
8          Console.WriteLine(p.age); // 45
9          Console.Read();
10     }
11     static void ChangePerson(ref Person person)
12     {
13         person.name = "Alice";
14         person = new Person { name = "Bill", age = 45 };
15     }
16 }
17 class Person
18 {
19     public string name;
20     public int age;
21 }
```

Операция new создаст новый объект в памяти, и теперь ссылка `person` (она же ссылка `p` из метода `Main`) будет указывать уже на новый объект в памяти.

§9.8 Ключевое слово this

§9.8.1 Ключевое слово this в C++

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

В пределах метода класса, когда вызывается другой метод, компилятор неявно передает ему в вызове указатель this как невидимый параметр:

```
1  #include <iostream>
2  using namespace std;
3  class Another
4  {
5  private:
```

```
6     int m_number;
7  public:
8     Another(int number)
9     {
10        setNumber(number);
11    }
12    void setNumber(int number) { m_number = number; }
13    int getNumber() { return m_number; }
14 };
15 int main()
16 {
17    Another another(3);
18    another.setNumber(4);
19    cout << another.getNumber() << '\n';
20    return 0;
21 }
```

При вызове `another.setNumber(4);` C++ понимает, что функция `setNumber()` работает с объектом `another`, а `m_number` — это фактически `another.m_number`. Рассмотрим детально, как это всё работает.

Возьмем, к примеру, следующую строку:

```
another.setNumber(4);
```

Хотя на первый взгляд кажется, что у нас здесь только один аргумент, но на самом деле у нас их два! Во время компиляции строка `another.setNumber(4);` конвертируется компилятором в следующее:

```
setNumber(&another, 4);
```

где объект `another` конвертировался из объекта, который находился перед точкой, в аргумент функции!

Поскольку в вызове функции теперь есть два аргумента, то и метод нужно изменить соответствующим образом (чтобы он принимал два аргумента). Следовательно, следующий метод:

```
void setNumber(int number) { m_number = number; }
```

Конвертируется компилятором в:

```
void setNumber(Another* const this, int number)
        { this->m_number = number; }
```

При компиляции обычного метода, компилятор неявно добавляет к нему параметр *this. Указатель *this — это скрытый константный указатель, содержащий адрес объекта, который вызывает метод класса.

Внутри метода также необходимо обновить все члены класса (функции и переменные), чтобы они ссылались на объект, который вызывает этот метод. Это легко сделать, добавив префикс this-> к каждому из них. Рассмотрим пример инициализации полей класса через константный указатель this:

```
1  #include <iostream>
2  using namespace std;
3  class Point
4  {
5  public:
6  Point(int x, int y)
7  {
8  this->x = x;
9  this->y = y;
10 }
11 void showCoords()
12 {
13 cout << "Coords x: " << this->x << "\t y: " << y << endl;
14 }
15 private:
16 int x;
17 int y;
18 };
19 int main()
20 {
21 Point p1(20, 50);
22 p1.showCoords();
23 return 0;
24 }
```

В данном случае определен класс Point, который представляет точку на плоскости. И для хранения координат точки в классе определены переменные x и y.

Для обращения к переменным используется ключевое слово this, который возвращает на текущий объект данного класса. Причем после this ставится не точка, а стрелка ->.

В большинстве случаев для обращения к членам класса вряд ли понадобится ключевое слово this. Но оно может быть необходимо, если параметры функции или

переменные, которые определяются внутри функции, называются также, как и переменные класса. К примеру, чтобы в конструкторе разграничить параметры и переменные класса как раз и используется указатель `this`.

Иногда бывает полезно, чтобы метод класса возвращал объект, с которым работает, в виде возвращаемого значения. Основной смысл здесь — это позволить нескольким методам объединиться в «цепочку», работая при этом с одним объектом.

Если функция будет возвращать ссылку на текущий объект класса, то можно связать вызовы методов в цепочку. Например,

```
1  #include <iostream>
2  using namespace std;
3  class Point
4  {
5  public:
6  Point(int x, int y)
7  {
8  this->x = x;
9  this->y = y;
10 }
11 void showCoords()
12 {
13 cout << "Coords x: " << x << "\t y: " << y << endl;
14 }
15 Point &move(int x, int y)
16 {
17 this->x += x;
18 this->y += y;
19 return *this;
20 }
21 private:
22 int x;
23 int y;
24 };
25 int main()
26 {Point p1(20,50);
27 p1.move(10,5).move(10,10);
28 p1.showCoords(); // x: 40  y: 65
29 return 0;
30 }
```

Здесь метод `move` с помощью указателя `this` возвращает ссылку на объект текущего класса, осуществляя условное перемещение точки. Таким образом, мы можем по цепочке для одного и того же объекта вызывать метод `move`:

```
p1.move(10,5).move(10,10);
```

Здесь также важно отметить возвращение не просто объекта `Point`, а ссылки на этот объект. Так, в данном случае выше определенная строка фактически будет аналогично следующему коду:

```
p1.move(10, 5);
p1.move(10, 10);
```

Но если бы метод move возвращал бы не ссылку, а просто объект:

```
1  Point move(int x, int y)
2  {
3  this->x += x;
4  this->y += y;
5  return *this;
6  }
```

То вызов `p1.move(10, 5).move(10, 10)` был бы фактически эквивалентен следующему коду:

```
Point temp = p1.move(10,5)
temp.move(10,10)
```

Где второй вызов метода `move` вызывался бы для временной копии и никак бы не затрагивал переменную `p1`.

§9.8.2 Ключевое слово this в C#

Стоит заметить, что ключевое слово `this` в C# представляет схожую функциональность - оно представляет ссылку на текущий экземпляр класса. Использование ключевого слова `this` отличается одной особенностью – для обращения к членам класса после ключевого слова будет использоваться оператор выбора членов (.).

```
1  class Person
2  {
3      public string name;
4      public int age;
5  public Person(string name, int age)
6      {
```

```
7           this.name = name;
8           this.age = age;
9       }
10  }
```

Для разграничения параметров и полей класса, к полям класса обращение идет через ключевое слово `this`. Так, в выражении `this.name = name;` первая часть `this.name` означает, что `name` - это поле текущего класса, а не название параметра `name`. Если бы у нас параметры и поля назывались по-разному, то использовать слово `this` было бы необязательно. Также через ключевое слово `this` можно обращаться к любому полю или методу.

§9.9 Статические члены класса

§9.9.1 Статические члены класса (C++)

Кроме переменных и методов, которые относятся непосредственно к объекту, C++ позволяет определять переменные и методы, которые относятся непосредственно к классу или иначе говоря **статические члены класса**. Статические переменные и методы относят в целом ко всем объектам классу. Для их определения используется ключевое слово **`static`**.

Например, в банке может быть множество различных вкладов, однако все вклады будут иметь какие-то общие процентные ставки. Так, для описания банковского счета определим и используем следующий класс:

```
1   #include <iostream>
2   using namespace std;
3   class Account
4   {
5   public:
6   Account(double sum)
7   {
8   this->sum = sum;
9   }
10  static int getRate()
11  {
12  return rate;
13  }
14  static void setRate(int r)
15  {
16  rate = r;
17  }
```

```
18  double getIncome()
19  {
20  return sum + sum * rate / 100;
21  }
22  private:
23  double sum;
24  static int rate;
25  };
26  int Account::rate = 8;
27  int main()
28  {
29  Account account1(20000);
30  Account account2(50000);
31  Account::setRate(5); // переустанавливаем значение rate
32  cout << "Rate: " << Account::getRate() << endl;
33  cout << "Rate: " << account1.getRate() << " Income: " << account1.getIncome() << endl;
34  cout << "Rate: " << account2.getRate() << " Income: " << account2.getIncome() << endl;
35  return 0;
36  }
```

В классе `Account` определена одна статическая переменная `rate` и две статических функции для управления этой переменной. При определении статических функций стоит учитывать, что внутри них мы можем использовать только статические переменные класса, как например, переменную `rate`. Нестатические переменные использовать в статических функциях нельзя.

Кроме того, в статических функциях нельзя использовать указатель `this`, что в принципе и так очевидно, так как `this` указывает на текущий объект, а статические функции относятся в целом ко всему классу.

Поскольку статические переменные-члены не являются частью отдельных объектов класса (они обрабатываются аналогично глобальным переменным и инициализируются при запуске программы), то вы должны явно определить статический член вне тела класса — в глобальной области видимости:

```
int Account::rate = 8;
```

Присваивать начальное значение переменной необязательно.

Есть исключение при определении статических членов внутри тела класса. Если статический член является константным интегральным типом (к которому относятся и char, и bool), то статический член может быть инициализирован внутри тела класса.

Также стоит отметить, что так как статические члены относятся в целом ко всему классу, то для обращения к статическим членам используется имя класса, после которого идет **оператор разрешения области видимости (`::`)**. Либо мы можем также обращаться к публичным членам класса через переменные данного класса.

§9.8.2 Статические члены класса (C#)

Кроме обычных полей, методов, свойств класс может иметь статические поля, методы, свойства. Статические поля, методы, свойства относятся ко всему классу и для обращения к подобным членам класса необязательно создавать экземпляр класса.

Например:

```
1  class Account
2  {
3      public static double bonus = 100;
4      public double totalSum;
5      public Account(double sum)
6      {
7          totalSum = sum + bonus;
8      }
9  }
10 class Program
11 {
12     static void Main(string[] args)
13     {
14         Console.WriteLine(Account.bonus); // 100
15         Account.bonus += 200;
16         Account account1 = new Account(150);
17         Console.WriteLine(account1.totalSum);   // 450
18         Account account2 = new Account(1000);
19         Console.WriteLine(account2.totalSum);  // 1300
20         Console.ReadKey();
21     }
22 }
```

В данном случае класс `Account` имеет два поля: `bonus` и `totalSum`. Поле `bonus` является статическим, поэтому оно хранит состояние класса в целом, а не отдельного объекта. И поэтому мы можем обращаться к этому полю по имени класса:

```
Console.WriteLine(Account.bonus);
Account.bonus += 200;
```

На уровне памяти для статических полей будет создаваться участок в памяти, который будет общим для всех объектов класса. При этом память для статических переменных выделяется даже в том случае, если не создано ни одного объекта этого класса.

Подобным образом мы можем создавать и использовать статические методы и свойства:

```
1  class Account
2  {
3      public Account(double sum, double rate)
4      {
5          if (sum < MinSum)
   throw
   new Exception("Недопустимая сумма!");
6          Sum = sum; Rate = rate;
7      }
8      private static double minSum = 100; // минимальная
   допустимая сумма для всех счетов
9      public static double MinSum
10     {
11         get { return minSum; }
12         set { if(value>0) minSum = value; }
13     }
14     public double Sum { get; private set; }    // сумма на счете
15     public double Rate { get; private set; }    // процентная
   ставка
16     // подсчет суммы на счете через определенный период по
   определенной ставке
17     public static double GetSum(double sum, double rate, int
   period)
18     {
19         double result = sum;
20         for (int i = 1; i <= period; i++)
21             result = result + result * rate / 100;
22         return result;
23     }
24 }
```

Переменная `minSum`, свойство `MinSum`, а также метод `GetSum` здесь определены с ключевым словом `static`, то есть они являются статическими.

Переменная `minSum` и свойство `MinSum` представляют минимальную сумму, которая допустима для создания счета. Этот показатель не относится к какому-то конкретному счету, а относится ко всем счетам в целом. Если мы изменим этот показатель для одного счета, то он также должен измениться и для другого счета. То есть в отличии от свойств `Sum` и `Rate`, которые хранят состояние объекта, переменная `minSum` хранит состояние для всех объектов данного класса.

То же самое с методом `GetSum` - он вычисляет сумму на счете через определенный период по определенной процентной ставке для определенной начальной суммы. Вызов и результат этого метода не зависит от конкретного объекта или его состояния.

Таким образом, переменные и свойства, которые хранят состояние, общее для всех объектов класса, следует определять как статические. И также методы, которые определяют общее для всех объектов поведение, также следует объявлять как статические.

Статические члены класса являются общими для всех объектов этого класса, поэтому к ним надо обращаться по имени класса:

```
Account.MinSum = 560;
double result = Account.GetSum(1000, 10, 5);
```

Следует учитывать, что статические методы могут обращаться только к статическим членам класса. **Обращаться к нестатическим методам, полям, свойствам внутри статического метода мы не можем.**

Нередко статические поля применяются для хранения счетчиков. Например, пусть у нас есть класс `User`, и мы хотим иметь счетчик, который позволял бы узнать, сколько объектов `User` создано:

```
1  class User
2  {
3      private static int counter = 0;
4      public User()
5      {
6          counter++;
7      }
8      public static void DisplayCounter()
9      {
10         Console.WriteLine($"Создано {counter} объектов User");
11     }
```

```
}
class Program
{
    static void Main(string[] args)
    {
        User user1 = new User();
        User user2 = new User();
        User user3 = new User();
        User user4 = new User();
        User user5 = new User();
        User.DisplayCounter(); // 5
        Console.ReadKey();
    }
}
```

Кроме обычных конструкторов у класса также могут быть статические конструкторы. Статические конструкторы имеют следующие отличительные черты:

- Статические конструкторы не должны иметь модификатор доступа и не принимают параметров;
- Как и в статических методах, в статических конструкторах нельзя использовать ключевое слово `this` для ссылки на текущий объект класса и можно обращаться только к статическим членам класса;
- Статические конструкторы нельзя вызвать в программе вручную. Они выполняются автоматически при самом первом создании объекта данного класса или при первом обращении к его статическим членам (если таковые имеются).

Статические конструкторы обычно используются для инициализации статических данных, либо же выполняют действия, которые требуется выполнить только один раз.

Объявление статического конструктора выглядит следующим образом:

```
static User()
{
Console.WriteLine("Создан первый пользователь");
}
```

В C# можно объявлять статические классы. Статические классы объявляются с модификатором `static` и могут содержать только статические поля, свойства и методы. Например, если бы класс `Account` имел бы только статические переменные, свойства и методы, то его можно было бы объявить как статический. В C#

показательным примером статического класса является класс Math, который применяется для различных математических операций.

§9.10 Перегрузка операторов

В языках программирования имеется готовый набор лексем, используемых для выполнения базовых операций над встроенными типами. Например, известно, что операция + может применяться к двум целым, чтобы дать их сумму:

```
1 // Операция + с целыми.
2 int a = 100;
3 int b = 240;
4 int c = a + b; //c теперь равно 340
```

Здесь нет ничего нового, но задумывались ли вы когда-нибудь о том, что одна и та же операция + может применяться к большинству встроенных типов данных C#? Например, рассмотрим такой код:

```
1 // Операция + со строками.
2 string si = "Hello";
3 string s2 = " world!";
4 string s3 = si + s2;   // s3 теперь содержит "Hello world!"
```

По сути, функциональность операции + уникальным образом базируются на представленных типах данных (строках или целых в данном случае). Когда операция + применяется к числовым типам, мы получаем арифметическую сумму операндов. Однако, когда та же операция применяется к строковым типам, получается конкатенация строк.

По сути, функциональность операции + уникальным образом базируются на представленных типах данных (строках или целых в данном случае). Когда операция + применяется к числовым типам, мы получаем арифметическую сумму операндов. Однако, когда та же операция применяется к строковым типам, получается конкатенация строк.

В C++ и C# разрешено перегружать встроенные операции – это одно из проявлений полиморфизма. Наличие перегрузки операций обеспечивает дополнительный уровень удобства при использовании новых типов данных.

§9.10.1 Перегрузка операторов в C++

При перегрузке операции нужно учитывать следующие ограничения:

1. Запрещена перегрузка следующих операций:

- sizeof() – определение размера переменной;
- . (точка) – селектор компонента объекта;
- ?: - условная операция;
- :: - указание области видимости;
- .* - выбор компоненты класса через указатель;
- # - операции препроцессора.

2. Операции можно перегружать только для нового типа данных – нельзя перегружать операцию для встроенного типа.

3. Нельзя изменить приоритет перегружаемой операции и количество операндов. Унарная операция обязана иметь один операнд, бинарная – два операнда; не разрешается использовать параметры по умолчанию. Единственная операция, которая не имеет фиксированного количества операндов – это операция вызова функции (). Операции «+», «-», «*», «&» допускается перегружать и как унарные, и как бинарные.

4. Операции можно перегружать либо как независимые внешние функции, либо как методы класса. Четыре операции:

- присваивания =;
- вызова функции ();
- индексирования [];
- доступа по указателю ->;

допускается перегружать только как методы класса.

5. Если операция перегружается как метод класса, то левым (или единственным) аргументом обязательно будет объект класса, для которого перегружается операция.

Прототип функции-операции выглядит следующим образом:

```
тип operator @ (список_параметров) {}
```

где **@** - символ операции. Слово **operator** является зарезервированным словом и может использоваться только в определении или в функциональной форме вызова операции.

Перегрузка операций внешними функциями

Прототип бинарной операции, перегружаемой как внешняя независимая функция, выглядит так:

```
тип operator @ (параметр1, параметр2) {}
```

Для обеспечения коммутативности бинарной операции с параметрами разного типа, как правило, требуется реализовать две функции-операции:

```
тип operator @ (параметр1, параметр2) {}
тип operator @ (параметр2, параметр1) {}
```

Прототип унарной операции, перегружаемой независимой внешней функцией, отличается только количеством параметров.

```
тип operator @ (параметр) {}
```

Хотя бы один из параметров должен быть нового типа. Параметры можно передавать любым допустимым способом. Обращение к операции выполняется двумя способами:

1) инфиксная форма **параметр1 @ параметр2 / @параметр**;

2) функциональная форма **operator @ (параметр1, параметр2) / operator @ (параметр)**.

Операции инкремента ++ и декремента -- имеют две формы: префиксную и постфиксную. В определении постфиксной формы операции должен быть объявлен дополнительный параметр типа int.

```
тип operator @ (параметр, int) {}
```

Этот параметр не должен использоваться в теле функции. Инфиксная форма обращения к такой операции - **параметр@**; при функциональной форме обращения необходимо задавать торой фиктивный аргумент, например

```
operator @ (параметр, 0)
```

Рассмотрим пример с классом Counter, который представляет секундомер и хранит количество секунд:

```
1  #include <iostream>
2  using namespace std;
3  class Counter
4  {
5  public:
6      Counter(int sec)
7      {
8          seconds = sec;
9      }
10     void display()
11     {
```

```
12          cout << seconds << " seconds" << endl;
13      }
14  int seconds;
15  };
16  Counter operator + (Counter c1, Counter c2)
17  {
18      return Counter(c1.seconds + c2.seconds);
19  }
20  int main()
21  {
22      Counter c1(20);
23      Counter c2(10);
24      Counter c3 = c1 + c2;
25      c3.display();   // 30 seconds
26      return 0;
27  }
```

Здесь функция оператора не является частью класса Counter и определена вне его. Данная функция перегружает оператор сложения для типа Counter. Она является бинарной, поэтому принимает два параметра. В данном случае мы складываем два объекта Counter. Возвращает функция также объект Counter, который хранит общее количество секунд. То есть по сути здесь операция сложения сводится к сложению секунд обоих объектов.

При этом необязательно возвращать объект класса. Это может быть и объект встроенного примитивного типа. И также мы можем определять дополнительные перегруженные функции операторов:

```
1  int operator + (Counter c1, int s)
2  {
3      return c1.seconds + s;
4  }
```

Данная версия складывает объект Counter с числом и возвращает также число. Поэтому левый операнд операции должен представлять тип Counter, а правый операнд - тип int. И, к примеру, мы можем применить данную версию оператора следующим образом:

```
1  Counter c1(20);
2  int seconds = c1 + 25;  // 45
3  std::cout << seconds << std::endl;
4  Counter c1(20);
```

У методов один параметр – ссылка на текущий экземпляр класса – определен по умолчанию. Унарные операции выполняются для текущего объекта, который и является единственным аргументом. Поэтому унарные операции, перегружаемые как методы класса, не имеют параметров. Прототип унарной операции, реализованной как метод класса, выглядит следующим образом:

```
тип operator @ () {}
```

Возвращаемое значение может быть любого типа, в том числе и определяемого класса.

Прототип бинарной операции имеет один аргумент и выглядит следующим образом:

```
тип operator @ (параметр) {}
```

Параметры разрешается передавать любым удобным способом: по значению, по ссылке или по указателю.

При реализации метода-операции вне класса требуется в заголовке указывать префикс-имя класса:

```
тип имя_класса::operator@ () {} //унарная операция
тип имя_класса::operator@ (параметр) {} //бинарная операция
```

Вызов унарной операции для объекта имеет следующую форму:

```
@объект
объект@
```

в зависимости от того, какой вид операции используется – префиксный или постфиксный. Функциональный вызов для постфиксной операции имеет форму:

```
объект.operator@ ()
```

Для постфиксной операции в функциональной форме вызова нужно задавать фиктивный аргумент:

```
объект.operator@ (0)
```

Функциональная форма вызова для бинарной операции, определенной как метод класса, выглядит так:

```
объект.operator@ (аргумент)
```

Если функция оператора определена как член класса, то левый операнд доступен через указатель this и представляет текущий объект, а правый операнд передается в подобную функцию в качестве единственного параметра:

```
#include <iostream>
using namespace std;
class Counter
{
public:
    Counter(int sec)
    {
        seconds = sec;
    }
    void display()
    {
        cout << seconds << " seconds" << endl;
    }
    Counter operator + (Counter c2)
    {
        return Counter(this->seconds + c2.seconds);
    }
    int operator + (int s)
    {
        return this->seconds + s;
    }
int seconds;
};
int main()
{
    Counter c1(20);
    Counter c2(10);
    Counter c3 = c1 + c2;
    c3.display();           // 30 seconds
    int seconds = c1 + 25;  // 45
    return 0;
}
```

В данном случае к левому операнду в функциях операторов мы обращаемся через указатель this.

Какие операторы где переопределять? Операторы присвоения, индексирования ([]), вызова (()), доступа к члену класса по указателю (->) следует определять в виде функций-членов класса. Операторы, которые изменяют состояние объекта или

непосредственно связаны с объектом (инкремент, декремент), обычно также определяются в виде функций-членов класса. Все остальные операторы чаще определяются как отдельные функции, а не члены класса.

§9.10.2 Перегрузка операторов в C#

Язык C# предоставляет возможность строить специальные классы и структуры, которые также уникально реагируют на один и тот же набор базовых лексем (вроде операции +). Имейте в виду, что абсолютно каждую встроенную операцию C# перегружать нельзя. В следующей таблице описаны возможности перегрузки основных операций:

Операция C#	**Возможность перегрузки**
+, -, !, ++, --, true, false	Этот набор унарных операций может быть перегружен
+, -, *, /, %, &, |, ^, <<, >>	Эти бинарные операции могут быть перегружены
==, !=, <, >, <=, >=	Эти операции сравнения могут быть перегружены. C# требует совместной перегрузки "подобных" операций (т.е. < и >, <= и >=, == и !=)
[],()	Эти операции не могут быть перегружены
+=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=	Сокращенные операции присваивания не могут перегружаться; однако вы получаете их автоматически, перегружая соответствующую бинарную операцию

Перегрузка операторов тесно связана с перегрузкой методов. Для перегрузки оператора служит ключевое слово `operator`, определяющее операторный метод, который, в свою очередь, определяет действие оператора относительно своего класса. Существуют две формы операторных методов (`operator`): одна - для унарных операторов, другая - для бинарных. Ниже приведена общая форма для каждой разновидности этих методов:

```
1  // Общая форма перегрузки унарного оператора.
```

```
2   public static возвращаемый_тип operator @(тип_параметра операнд)
3   {
4   // операции
5   }
6   // Общая форма перегрузки бинарного оператора.
7   public static возвращаемый_тип operator @(тип_параметра1 операнд1, тип_параметра2 операнд2)
8   {
9   // операции
10  }
```

Здесь вместо **@** подставляется перегружаемый оператор, например, + или /, а возвращаемый_тип обозначает конкретный тип значения, возвращаемого указанной операцией. Это значение может быть любого типа, но зачастую оно указывается такого же типа, как и у класса, для которого перегружается оператор. Такая корреляция упрощает применение перегружаемых операторов в выражениях. Для унарных операторов операнд обозначает передаваемый операнд, а для бинарных операторов то же самое обозначают операнд1 и операнд2.

Операторный метод должен иметь модификаторы public static, так как перегружаемый оператор будет использоваться для всех объектов данного класса.

Например, перегрузим ряд операторов для класса Counter:

```
1   class Counter
2   {
3       public int Value { get; set; }
4       public static Counter operator +(Counter c1, Counter c2)
5       {
6           return new Counter { Value = c1.Value + c2.Value };
7       }
8       public static bool operator >(Counter c1, Counter c2)
9       {
10          return c1.Value > c2.Value;
11      }
12      public static bool operator <(Counter c1, Counter c2)
13      {
14          return c1.Value < c2.Value;
15      }
16  }
```

Так как в случае с операцией сложения мы хотим сложить два объекта класса Counter, то оператор принимает два объекта этого класса. И так как мы хотим в

результате сложения получить новый объект Counter, то данный класс также используется в качестве возвращаемого типа. Все действия этого оператора сводятся к созданию, нового объекта, свойство Value которого объединяет значения свойства Value обоих параметров.

Также переопределены две операции сравнения. Если мы переопределяем одну из этих операций сравнения, то мы также должны переопределить вторую из этих операций. Сами операторы сравнения сравнивают значения свойств Value и в зависимости от результата сравнения возвращают либо true, либо false.

Теперь используем перегруженные операторы в программе:

```
1  static void Main(string[] args)
2  {
3      Counter c1 = new Counter { Value = 23 };
4      Counter c2 = new Counter { Value = 45 };
5      bool result = c1 > c2;
6      Console.WriteLine(result); // false
7      Counter c3 = c1 + c2;
8      Console.WriteLine(c3.Value);  // 23 + 45 = 68
9      Console.ReadKey();
10 }
```

Стоит отметить, что так как по сути определение оператора представляет собой метод, то этот метод мы также можем перегрузить, то есть создать для него еще одну версию. Например, добавим в класс Counter еще один оператор:

```
1  public static int operator +(Counter c1, int val)
2  {
3      return c1.Value + val;
4  }
```

Данный метод складывает значение свойства Value и некоторое число, возвращая их сумму. И также мы можем применить этот оператор:

```
1  Counter c1 = new Counter { Value = 23 }
2  int d = c1 + 27 // 50
3  Console.WriteLine(d)
```

Следует учитывать, что при перегрузке не должны изменяться те объекты, которые передаются в оператор через параметры. Например, мы можем определить для класса Counter оператор инкремента:

```
1  public static Counter operator ++(Counter c1)
2  {
```

```
3       c1.Value += 10;
4       return c1;
5   }
```

Поскольку оператор унарный, он принимает только один параметр - объект того класса, в котором данный оператор определен. Но это неправильное определение инкремента, так как оператор не должен менять значения своих параметров.

И более корректная перегрузка оператора инкремента будет выглядеть так:

```
1   public static Counter operator ++(Counter c1)
2   {
3       return new Counter { Value = c1.Value + 10 };
4   }
```

При этом нам не надо определять отдельно операторы для префиксного и для постфиксного инкремента (а также декремента), так как одна реализация будет работать в обоих случаях.

Например, используем операцию префиксного инкремента:

```
1   Counter counter = new Counter() { Value = 10 };
2   Console.WriteLine($"{counter.Value}");      // 10
3   Console.WriteLine($"{(++counter).Value}");  // 20
4   Console.WriteLine($"{counter.Value}");      // 20
```

Теперь используем постфиксный инкремент:

```
1   Counter counter = new Counter() { Value = 10 };
2   Console.WriteLine($"{counter.Value}");      // 10
3   Console.WriteLine($"{(counter++).Value}");  // 10
4   Console.WriteLine($"{counter.Value}");      // 20
```

При перегрузке операторов также следует помнить, что мы не можем изменить приоритет оператора или его ассоциативность, мы не можем создать новый оператор или изменить логику операторов в типах, который есть по умолчанию в .NET.

§9.11 Наследование

Одной из наиболее сильных сторон объектно-ориентированного программирования, пожалуй, является повторное использование кода. При структурном проектировании повторное использование кода допускается в известной мере: вы можете написать функцию, а затем применять ее столько раз, сколько пожелаете. Однако объектно-ориентированное проектирования делает важный шаг вперед, позволяя вам определять отношения между классами, которые не только облегчают повторное использование кода, но и способствуют созданию лучшей общей

конструкции путем упорядочения и учета общности разнообразных классов. Основное средство обеспечения такой функциональности – *наследование*.

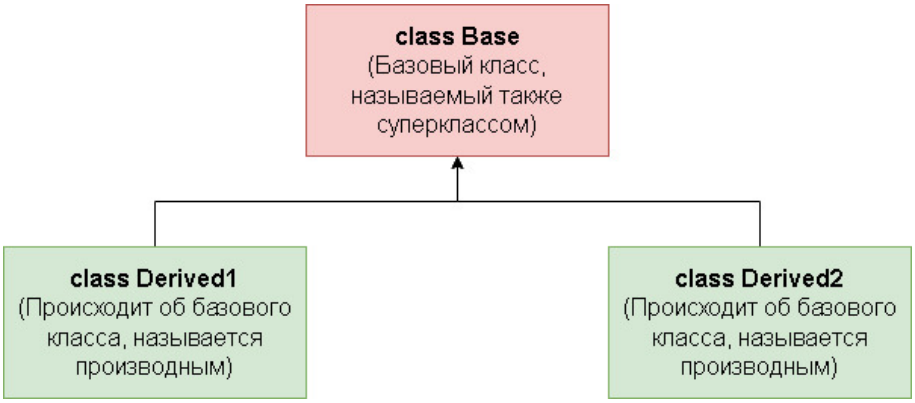


Рисунок 9.2 – Схема наследования классов

На рисунке 9.2 приведена схема наследования между базовым классом (base class) и происходящими от него производными классами (derived class).

Наследование позволяет производному классу наследовать свойства и методы другого класса. Это дает возможность создавать абсолютно новые классы путем абстрагирования общих атрибутов и поведений.

Одна из основных задач проектирования при объектно-ориентированном программировании заключается в выделении общности разнообразных классов. Допустим, у вас есть класс Dog и класс Cat, каждый из которых будет содержать атрибут eyeColor. При процедурной модели код как для Dog, так и для Cat включал бы этот атрибут. При объектно-ориентированном проектировании атрибут, связанный с цветом, можно перенести в класс с именем Mammal наряду со всеми прочими общими атрибутами и методами. В данном случае оба класса – Dog и Cat – будут наследовать от класса Mammal, как показано на рисунке 9.3.

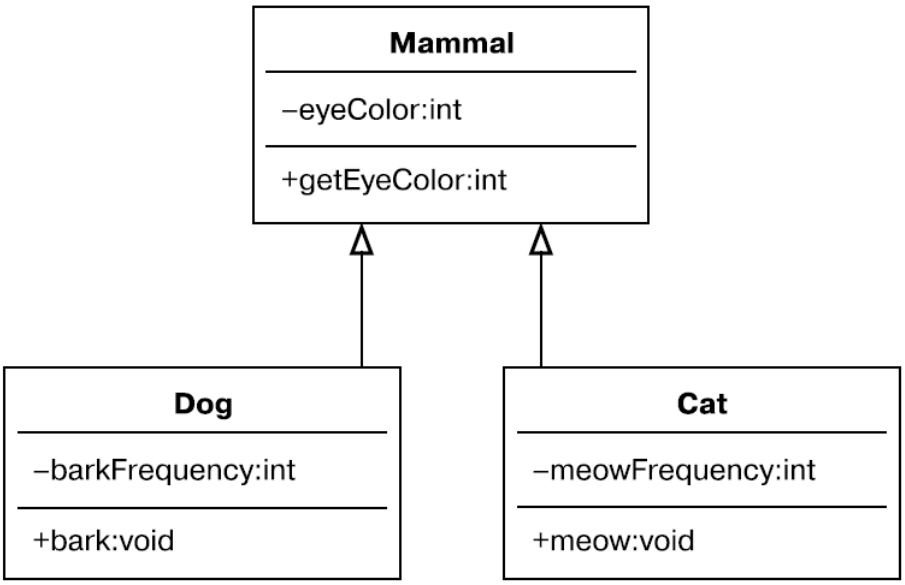


Рисунок 9.3 – Иерархия классов млекопитающих

Итак, оба класса наследуют от Mammal. Это означает, что в итоге класс Dog будет содержать следующие атрибуты:

```
eyeColor //унаследован от Mammal
barkFrequency //определен только для Dog
```

В том же духе объект Dog будет содержать следующие методы:

```
getEyeColor //унаследован от Mammal
bark //определен только для Dog
```

Создаваемые экземпляр объекта Dog или Cat будет содержать все, что есть в его собственном классе, а также все имеющиеся в родительском классе. Таким образом, Dog будет включать все свойства своего определения класса, а также свойства, унаследованные от класса Mammal.

Суперкласс, или базовый класс (иногда называемый родительским), содержит все атрибуты и поведения, общие для классов, которые наследуют от него. Например, в случае с классом Mammal все классы млекопитающих содержат аналогичные атрибут, такие как eyeColor и hairColor, а также поведения вроде generateInternalHeat и growHair. Все классы млекопитающих включают эти атрибуты и поведения, поэтому нет необходимости их дублировать, спускаясь по дереву наследования, для каждого типа млекопитающих. Дублирование потребует много дополнительной работы и может привести к ошибкам и несоответствиям.

Подкласс, или производный класс (иногда называемый дочерним), представляет собой расширение суперкласса. Таким образом, классы Dog и Cat наследуют все общие атрибуты и поведения от класса Mammal. Класс Mammal считается суперклассом подклассов, или дочерних классов, Dog и Cat.

Дерево наследования может разрастись довольно сильно. Когда классы Mammal и Cat будут готовы, добавить другие классы млекопитающих не составит особого труда. Класс Cat может также выступать в роли базового класса. Например, может потребоваться дополнительно абстрагировать Cat, чтобы обеспечить классы для различных пород кошек. Точно так же, как и Cat, класс Dog может выступать в роли базового класса для других классов, например, GermanShepherd и Poodle (Рисунок 9.4). Мощность наследования заключается в его методиках абстрагирования и организации.

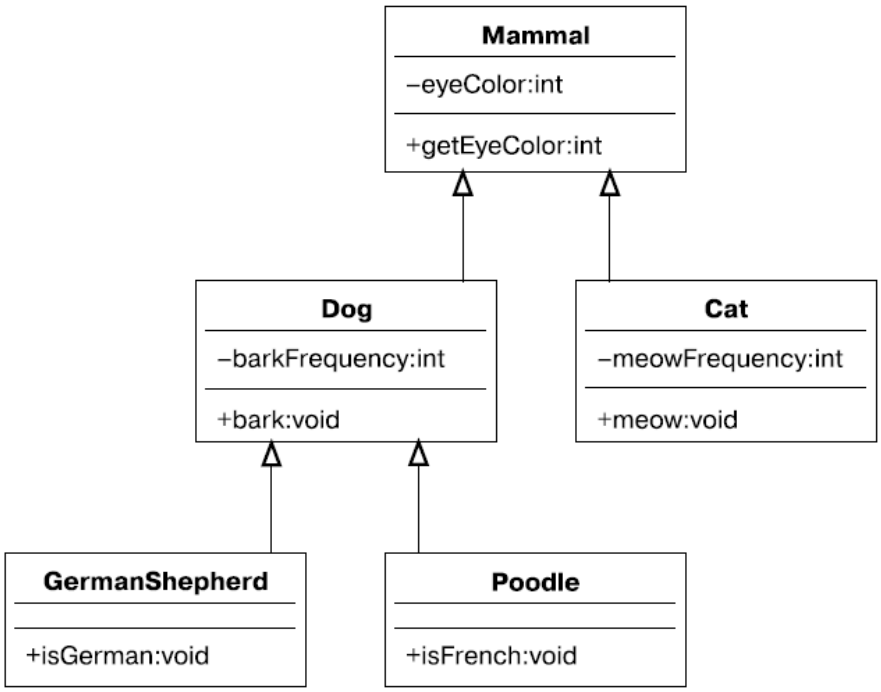


Рисунок 9.4 – Диаграмма классов млекопитающих

Обратите внимание, что оба класса - GermanShepherd и Poodle – наследуют от Dog – каждый содержит только один метод. Однако, поскольку они наследуют от Dog, они также наследуют от Mammal. Таким образом, классы GermanShepherd и Poodle включают в себя все свойства и методы, содержащиеся в Dog и Mammal, а также свои собственные.

§9.11.1 Наследование в C++

Рассмотрение принципа наследования начнем с открытого наследованиям, как его наиболее распространенной формы.

Простым (или одиночным) наследованием называется наследование, при котором производный класс имеет только одного родителя.

Формально наследование одного класса от другого задается следующей конструкцией:

```
1  class имя_базового_класса
2  {
3  //тело базового класса
4  };
5  class имя_производного класса : модификатор_доступа имя_базового_класса
6  {
7  //тело производного класса
8  };
```

Модификатор доступа определяет доступность элемента базового класса в классе-наследнике.

При разработке классов используется два модификатора доступа к элементам класса: public (открытый), private (закрытый). При наследовании применяется еще один: protected (защищенный). Защищенные элементы класса доступны только прямым наследникам и никому другому.

Доступность элементов базового класса из классов-наследников изменяется в зависимости от модификаторов доступа в базовом классе и модификатора наследования. Каким бы ни был модификатор наследования, приватные элементы базового класса недоступны в его потомках. В таблице 9.1 приведены все варианты доступности элементов базового класса в производном классе при любых вариантах модификатора наследования.

Если модификатор наследования – public, наследование называется открытым. Модификатор protected определяет защищенное наследование, а слово private – закрытое наследование.

Таблица 9.1. Доступ к элементам базового класса в классе-наследнике

Модификатор доступа в базовом классе	**Модификатор наследования**	**Доступ в производном классе**	
		struct	**class**
public	отсутствует	public	private

protected	отсутствует	public	private
private	отсутствует	недоступны	недоступны
public	public	public	public
protected	public	protected	protected
private	public	недоступны	недоступны
public	protected	protected	protected
protected	protected	protected	protected
private	protected	недоступны	недоступны
public	private	private	private
protected	private	private	private
private	private	недоступны	недоступны

Конструкторы не наследуются – они создаются в производном классе (если не определены программистом явно). Система поступает конструкторами следующим образом:

- Если в базовом классе нет конструкторов или есть конструктор без аргументов (или аргументы присваиваются по умолчанию), то в производном классе конструктор можно не писать – будут созданы конструктор копирования и конструктор без аргументов;

- Если в базовом классе все конструкторы с аргументами, производный класс обязан иметь конструктор, в котором явно должен быть вызван конструктор базового класса;

- При создании объекта производного класса сначала вызывается конструктор базового класса, затем – производного.

Рассмотрим пример с конструкторами в наследнике.

```
1  class Money
2  {
3      public:
4          Money(const double &d = 0.0)
5          {    Summa = d;    }
6      private:
7          double Summa;
8  };
9  class Backs : public Money{ };
```

```
10  class Basis
11  {
12      int a, b;
13      public:
14          Basis(int x, int y)
15          {
16              a = x;
17              b = y;
18          }
19  }
20  class Inherit : public Basis
21  {
22      int sum;
23      public:
24          Inherit(int x, int y, int s) : Basis (x, y)
25          {    sum = s; }
26  }
```

В классе Money определен конструктор инициализации, который играет роль и конструктора без аргументов, так как аргумент задан по умолчанию. В классе-наследнике Backs конструкторы можно не писать. Во втором случае в классе Basis определен только конструктор инициализации. В классе `Inherit` конструктор базового класса явно вызывается в списке инициализации.

Деструктор класса, как и конструкторы, не наследуется, а создается. С деструкторами система поступает следующим образом:

- при отсутствии деструктора в производном классе система создает деструктор по умолчанию;

- деструктор базового класса вызывается в деструкторе производного класса автоматически независимо от того, определен он явно или создан системой;

- деструкторы вызываются (для уничтожения объектов) в порядке, обратном вызову конструкторов).

Создание и уничтожения объектов выполняется по принципу LIFO: последним создан – первым уничтожен.

Поля и методы при наследовании

Класс-потомок наследует структуру (все элементы данных) и поведение (все методы) базового класса. Класс-наследник получает в наследство все поля базового класса (хотя, если они приватные, доступа к ним не имеет). Если новые поля не

добавляются, размер класса-наследника совпадает с размером базового класса. Порожденный класс может добавить собственные поля и методы:

```
1  class A
2  {
3      protected:
4      int x; int y;
5      public: //...
6  };
7  class B: public A
8  {
9      int z;
10     public: //...
11 }
```

Добавленные поля должен инициализировать конструктор наследника.

Дополнительные поля производного класса могут совпадать и по имени, и по типу с полями базового класса – в этом случае новое поле скрывает поле базового класса, поэтому для доступа к последнему полю в классе-наследнике необходимо использовать префикс-квалификатор базового класса.

Если в классе-наследнике имя методы и его прототип совпадают с именем метода базового класса, то говорят, что метод производного класса скрывает метод базового класса. Чтобы вызвать метод родительского класса, нужно указывать его с квалификатором класса.

Рассмотрим пример:

```
1  #include <iostream>
2  using namespace std;
3  class A
4  {
5      protected:
6      int a;
7      public:
8      A(int a){this->a=a;}
9      void getA(){cout << this->a;}
10 };
11 class B : public A
12 {
13     public:
14     B(int a) : A(a){}
15     void getA()
16     {cout << this->a + 10;}
```

```
17  };
18  int main()
19  {
20      B b(5);
21      b.getA();
22      b.A::getA();
23  }
```

В строке 21 вызывается метод `getA()`, определенный в классе B,так как из-за совпадающий прототип перекрывает метод родительского класса. В строке 22 мы указываем квалификатор родительского класса `A::` для вызова метода `getA()`, определенного в родительском классе.

Принцип подстановки

Открытое наследование устанавливает между классами отношение «является»: класс-наследник является разновидностью класса-родителя. Это означает, что везде, где может быть использован объект базового класса (при присваивании, при передаче параметров и возврате результата), вместо него разрешается подстановка объектов производного класса. Данное положение называется принципов подстановки и поддерживается компилятором. Этот принцип работает и для ссылок, и для указателей: вместо ссылки (указателя) на базовый класс может быть подставлена ссылка (указатель) на класс-наследник. Обратное – неверно! Например, спортсмен является человеком, но не всякий человек – спортсмен. Здесь человек – базовый класс, а спортсмен – производный.

Помимо конструкторов, не наследуются два вида функций: операция присваивания и дружественные функции. Операция присваивания, как и конструкторов с деструктором, для любого класса создается автоматически и имеет прототип:

```
класс& класс::operator = (const класс &t);
```

Операция бинарная, левым аргументом является текущий объект. Эта стандартная операция обеспечивает копирование значений полей объекта t в поля текущего объекта. Для базового класса `Base` и его наследника `Derive` соответствующие операции присваивания имеют вид:

```
Base &Base::operator(const Base &t);
Derive &Derive::operator(const Derive &t);
```

Наличие этих функций позволяет нам выполнить следующие присваивания:

```
Base b1, b2; //переменные базового класса
Derive d1, d2; //переменные производного класса
b1 = b2; //операция базового класса
d1 = d2; //операция производного класса
b1 = d2; //базовый = производный: операция базового класса
```

В последнем случае работает принцип подстановки: справа от знака присваивания задан объект производного класса, который подставляется на место аргумента базового класса в операции присваивания базового класса. Преобразование типа при этом указывать не нужно – это происходит автоматически.

Однако нельзя выполнить присваивание

```
d1 = b1; // производный = базовый
```

Чтобы оно работало, требуется в производном классе определить операцию присваивания со следующим определением

```
Derive &Derive::operator(const Base &t)
{this->Base::operator=(t);
Return *this;}
```

В теле дочерней операции выполняется вызов родительской операции в функциональной форме. Эта операция похожа на операцию преобразования объектов базового класса в производный.

Дружественные функции

Дружественная функция — это функция, которая имеет доступ к закрытым членам класса, как если бы она сама была членом этого класса. 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. Неважно, объявляете ли вы её в `public` или в `private` зоне класса.

Например:

```
1  class Anything
2  {
3  private:
4      int m_value;
5  public:
```

```
6     Anything() { m_value = 0; }
7     void add(int value) { m_value += value; }
8     friend void reset(Anything &anything);
9  };
10 void reset(Anything &anything)
11 {
12    anything.m_value = 0;
13 }
14 int main()
15 {
16    Anything one;
17    one.add(4);
18    reset(one);
19    return 0;
20 }
```

Функция `reset()` принимает объект класса `Anything` и устанавливает для свойства `m_value` значение 0. Поскольку `reset()` не является членом класса `Anything`, то, в обычной ситуации, `reset()` не имел бы доступа к закрытым членам `Anything`. Однако, поскольку эта функция является дружественной классу `Anything`, она имеет доступ к закрытым членам `Anything`.

В связи с тем, что `reset()` не является методом класса, необходимо передавать объект `Anything` в `reset()` в качестве параметра. Он не имеет указателя `*this` и кроме как передачи объекта, он не сможет взаимодействовать с классом.

Закрытое наследование

Закрытое наследование – это наследование реализации, в котором класс реализован посредством класса. Оно принципиально отличается от открытого: принцип подстановки не соблюдается. Это означает, что нельзя присвоить (во всяком случае, без явного преобразования типа) объект производного класса базовому.

При закрытом наследовании все элементы класса-наследника становятся приватными и недоступными программе-клиенту.

```
1  class Base
2  { public:
3       void method1();
4       void method2();
5  };
6  class Derive: private Base // наследуемые методы недоступны клиенту
```

```
7  { };
```

Программа, использующая класс `Derive`, не может вызвать ни `method1()`, ни `method2()`. В наследнике нужно заново реализовать нужные методы, вызвав в методах наследника методы родителя.

```
1  class Derive: private Base // наследуемые методы недоступны клиенту
2  {  public:
3      void method1() { Base::method1(); }
4      void method2() { Base::method2(); };
5  };
```

Префикс при вызове задавать обязательно, иначе возникает рекурсивное обращение к методу-наследнику.

Можно открыть методы базового класса с помощью using-объявления, которое имеет следующий синтаксис:

```
using <имя базового класса>::<имя в базовом классе>;
```

В наследуемом классе это делается так:

```
1  class Derive: private Base // наследуемые методы недоступны клиенту
2  {  public:
3      using Base::method1();
4  };
```

Таким образом, закрытое наследование позволяет ограничить предоставляемую производным классам функциональность.

Множественное наследование

Язык C++ предоставляет возможность выполнения множественного наследования. Множественное наследование позволяет одному дочернему классу иметь несколько родителей. Предположим, что мы хотим написать программу для отслеживания работы учителей. Учитель — это Human. Тем не менее, он также является Сотрудником (Employee).

Множественное наследование может быть использовано для создания класса Teacher, который будет наследовать свойства как Human, так и Employee. Для использования множественного наследования нужно просто указать через запятую тип наследования и второй родительский класс:

```
1  #include <string>
2  using namespace std;
```

```
3  class Human
4  {
5  private:
6  string m_name;
7  int m_age;
8  public:
9  Human(string name, int age): m_name(name), m_age(age)
10 {
11 }
12 string getName() { return m_name; }
13 int getAge() { return m_age; }
14 };
15 class Employee
16 {
17 private:
18 string m_employer;
19 double m_wage;
20 public:
21 Employee(string employer, double wage) : m_employer(employer), m_wage(wage)
22 {
23 }
24 string getEmployer() { return m_employer; }
25 double getWage() { return m_wage; }
26 };
27 // Класс Teacher открыто наследует свойства классов Human и Employee
28 class Teacher: public Human, public Employee
29 {
30 private:
31 int m_teachesGrade;
32  public:
33 Teacher(string name, int age, string employer, double wage, int teachesGrade) : Human(name, age), Employee(employer, wage), m_teachesGrade(teachesGrade)
34 {
35 }
36 };
```

Большинство задач, решаемых с помощью множественного наследования, могут быть решены и с использованием одиночного наследования. Многие объектно-ориентированные языки программирования (например, PHP) даже не поддерживают множественное наследование. Многие, относительно современные языки, такие как

Java и C#, ограничивают классы одиночным наследованием обычных классов, но допускают множественное наследование интерфейсных классов. Суть идеи, запрещающей множественное наследование в этих языках, заключается в том, что это лишняя сложность, которая порождает больше проблем, чем удобств.

Многие опытные программисты считают, что множественное наследование в C++ следует избегать любой ценой из-за потенциальных проблем, которые могут возникнуть. Однако всё же остаётся вероятность, когда множественное наследование будет лучшим решением, нежели придумывание двухуровневых костылей.

§9.11.2 Наследование в C#

В C# наследование является одиночным, то есть нельзя наследоваться от двух и более классов. Наследование определяется через ":".

Пусть у нас есть следующий класс `Person`, который описывает отдельного человека:

```
1  class Person
2  {
3      private string _name;
4      public string Name
5      {
6          get { return _name; }
7          set { _name = value; }
8      }
9      public void Display()
10     {
11         Console.WriteLine(Name);
12     }
13 }
```

Но вдруг нам потребовался класс, описывающий сотрудника предприятия - класс `Employee`. Поскольку этот класс будет реализовывать тот же функционал, что и класс `Person`, так как сотрудник - это также и человек, то было бы рационально сделать класс `Employee` производным (или наследником, или подклассом) от класса `Person`, который, в свою очередь, называется базовым классом или родителем (или суперклассом):

```
1  class Employee : Person
2  {}
```

После двоеточия мы указываем базовый класс для данного класса. Для класса `Employee` базовым является `Person`, и поэтому класс `Employee` наследует все те же свойства, методы, поля, которые есть в классе `Person`. Единственное, что не передается при наследовании, это конструкторы базового класса.

Таким образом, наследование реализует отношение is-a (является), объект класса `Employee` также является объектом класса `Person`:

```
1  static void Main(string[] args)
2  {
3      Person p = new Person { Name = "Tom"};
4      p.Display();
5      p = new Employee { Name = "Sam" };
6      p.Display();
7      Console.Read();
8  }
```

И поскольку объект `Employee` является также и объектом `Person`, то мы можем так определить переменную: `Person p = new Employee()`.

Все классы по умолчанию могут наследоваться. Однако здесь есть ряд ограничений:

- Не поддерживается множественное наследование, класс может наследоваться только от одного класса.
- При создании производного класса надо учитывать тип доступа к базовому классу - тип доступа к производному классу должен быть таким же, как и у базового класса, или более строгим.
- Если класс объявлен с модификатором sealed, то от этого класса нельзя наследовать и создавать производные классы. Например, следующий класс не допускает создание наследников:

```
sealed class Admin
{}
```

- Нельзя унаследовать класс от статического класса.

Класс `Object` и его методы

По умолчанию все классы наследуются от базового класса `Object`, даже если мы явным образом не устанавливаем наследование. Даже если мы не указываем класс `Object` в качестве базового, по умолчанию неявно класс `Object` все равно стоит на

вершине иерархии наследования. Поэтому все типы и классы могут реализовать те методы, которые определены в классе `System.Object`.

В классе `Object` реализованы следующие методы: `ToString()`, `Equals()`, `GetHashCode()` и `GetType()`.

Метод `ToString()` служит для получения строкового представления данного объекта. Для базовых типов просто будет выводиться их строковое значение:

```
1  int i = 5
2  Console.WriteLine(i.ToString()) // выведет число 5
3  double d = 3.5
4  Console.WriteLine(d.ToString()) // выведет число 3,5
```

Для классов же этот метод выводит полное название класса с указанием пространства имен, в котором определен этот класс. И мы можем переопределить данный метод. Посмотрим на примере:

```
1  using System;
2  namespace FirstApp
3  {
4      class Program
5      {
6          private static void Main(string[] args)
7          {
8              Person person = new Person { Name = "Tom" };
9              Console.WriteLine(person.ToString());
   // выведет название класса Person
10             Clock clock = new Clock { Hours = 15, Minutes = 34,
   Seconds = 53 };
11             Console.WriteLine(clock.ToString());
   // выведет 15:34:53
12             Console.Read();
13         }
14     }
15     class Clock
16     {
17         public int Hours { get; set; }
18         public int Minutes { get; set; }
19         public int Seconds { get; set; }
20         public override string ToString()
21         {
22             return $"{Hours}:{Minutes}:{Seconds}";
23         }
24     }
```

```
    class Person
    {
        public string Name { get; set; }
    }
}
```

Для переопределения метода `ToString` в классе `Clock`, который представляет часы, используется ключевое слово `override`. В данном случае метод `ToString()` возвращает значение свойств `Hours`, `Minutes`, `Seconds`, объединенные в одну строку.

Класс `Person` не переопределяет метод `ToString`, поэтому для этого класса срабатывает стандартная реализация этого метода, которая выводит просто название класса.

Стоит отметить, что различные технологии на платформе .NET активно используют метод `ToString()` для разных целей. В частности, тот же метод `Console.WriteLine()` по умолчанию выводит именно строковое представление объекта. Поэтому, если нам надо вывести строковое представление объекта на консоль, то при передаче объекта в метод `Console.WriteLine()` необязательно использовать метод `ToString()` - он вызывается неявно:

```
private static void Main(string[] args)
{
    Person person = new Person { Name = "Tom" };
    Console.WriteLine(person);
    Clock clock = new Clock { Hours = 15, Minutes = 34, Seconds = 53 };
    Console.WriteLine(clock); // выведет 15:34:53
    Console.Read();
}
```

Метод `GetHashCode()` позволяет возвратить некоторое числовое значение, которое будет соответствовать данному объекту или его хэш-код. По данному числу, например, можно сравнивать объекты. Можно определять самые разные алгоритмы генерации подобного числа или взять реализации базового типа.

Метод `GetType()` позволяет получить тип данного объекта:

```
Person person = new Person { Name = "Tom" }
Console.WriteLine(person.GetType()) // Person
```

Этот метод возвращает объект `Type`, то есть тип объекта.

С помощью ключевого слова typeof мы получаем тип класса и сравниваем его с типом объекта. И если этот объект представляет тип Client, то выполняем определенные действия.

```
1  object person = new Person { Name = "Tom" };
2  if (person.GetType() == typeof(Person))
3      Console.WriteLine("Это реально класс Person");
```

Причем поскольку класс Object является базовым типом для всех классов, то мы можем переменной типа object присвоить объект любого типа. Однако для этой переменной метод GetType все равно вернет тот тип, на объект которого ссылается переменная. То есть в данном случае объект типа Person.

В отличие от методов ToString, Equals, GetHashCode метод GetType не переопределяется.

Метод Equals позволяет сравнить два объекта на равенство:

```
1  class Person
2  {
3      public string Name { get; set; }
4      public override bool Equals(object obj)
5      {
6          if (obj.GetType() != this.GetType()) return false;
7          Person person = (Person)obj;
8          return (this.Name == person.Name);
9      }
10 }
```

Метод Equals принимает в качестве параметра объект любого типа, который мы затем приводим к текущему, если они являются объектами одного класса. Затем сравниваем по именам. Если имена равны, возвращаем true, что будет говорить, что объекты равны. Однако при необходимости реализацию метода можно сделать более сложной, например, сравнивать по нескольким свойствам при их наличии.

Применение метода:

```
1  Person person1 = new Person { Name = "Tom" }
2  Person person2 = new Person { Name = "Bob" }
3  Person person3 = new Person { Name = "Tom" }
4  bool p1Ep2 = person1.Equals(person2) // false
5  bool p1Ep3 = person1.Equals(person3) // true
```

И если следует сравнивать два сложных объекта, как в данном случае, то лучше использовать метод Equals, а не стандартную операцию ==.

Доступ к членам базового класса из класса-наследника

Вернемся к нашим классам `Person` и `Employee`. Хотя `Employee` наследует весь функционал от класса `Person`, посмотрим, что будет в следующем случае:

```
1  class Employee : Person
2  {
3      public void Display()
4      {Console.WriteLine(_name);}
5  }
```

Этот код не сработает и выдаст ошибку, так как переменная `_name` объявлена с модификатором `private` и поэтому к ней доступ имеет только класс `Person`. Но зато в классе `Person` определено общедоступное свойство `Name`, которое мы можем использовать, поэтому следующий код у нас будет работать нормально:

```
1  class Employee : Person
2  {
3      public void Display()
4      {Console.WriteLine(Name);}
5  }
```

Таким образом, производный класс может иметь доступ только к тем членам базового класса, которые определены с модификаторами `public` и `protected`.

Ключевое слово `base`

Ключевое слово `base` используется для доступа к членам базового из производного класса в следующих случаях:

- Вызов метода базового класса, который был переопределен другим методом.
- Определение конструктора базового класса, который должен вызываться при создании экземпляров производного класса.

Доступ к базовому классу разрешен только в конструкторе, методе экземпляра или методе доступа к свойству экземпляра.

Использование ключевого слова `base` в статическом методе является недопустимым.

Доступ осуществляется к базовому классу, заданному в объявлении класса. Например, если указать `class ClassB : ClassA`, члены `ClassA` будут доступны из `ClassB` независимо от базового класса `ClassA`.

Теперь добавим в наши классы конструкторы:

```
1  class Person
2  {
3      public string Name { get;  set; }
4      public Person(string name)
5      {
6          Name = name;
7      }
8      public void Display()
9      {
10         Console.WriteLine(Name);
11     }
12 }
13 class Employee : Person
14 {
15     public string Company { get; set; }
16     public Employee(string name, string company) : base(name)
17     {
18         Company = company;
19     }
20 }
```

Класс `Person` имеет конструктор, который устанавливает свойство `Name`. Поскольку класс `Employee` наследует и устанавливает то же свойство `Name`, то логично было бы не писать по сто раз код установки, а как-то вызвать соответствующий код класса `Person`. К тому же свойств, которые надо установить в конструкторе базового класса, и параметров может быть гораздо больше.

С помощью ключевого слова `base` мы можем обратиться к базовому классу. В нашем случае в конструкторе класса `Employee` нам надо установить имя и компанию. Но имя мы передаем на установку в конструктор базового класса, то есть в конструктор класса `Person`, с помощью выражения `base(name)`.

```
1  static void Main(string[] args)
2  {
3      Person p = new Person("Bill");
4      p.Display();
5      Employee emp = new Employee ("Tom", "Microsoft");
6      emp.Display();
7      Console.Read();
8  }
```

В приведенном ниже примере метод `Getinfo` присутствует как в базовом классе `Person`, так и в производном классе `Employee`. С помощью ключевого слова `base` можно вызвать метод `Getinfo` базового класса из производного класса.

```
1  public class Person
2  {
3  protected string ssn = "444-55-6666";
4  protected string name = "John L. Malgraine";
5  public virtual void GetInfo()
6  {
7  Console.WriteLine("Name: {0}", name);
8  Console.WriteLine("SSN: {0}", ssn);
9  }
10 }
11 class Employee : Person
12 {
13 public string id = "ABC567EFG";
14 public override void GetInfo()
15 {
16 // Calling the base class GetInfo method:
17 base.GetInfo();
18 Console.WriteLine("Employee ID: {0}", id);
19 }
29 }
21 class TestClass
22 {
23 static void Main()
24 {
25 Employee E = new Employee();
26 E.GetInfo();
27 }
28 }
```

Конструкторы в производных классах

Конструкторы не передаются производному классу при наследовании. И если в базовом классе не определен конструктор по умолчанию без параметров, а только конструкторы с параметрами (как в случае с базовым классом `Person`), то в производном классе мы обязательно должны вызвать один из этих конструкторов через ключевое слово `base`. Например, из класса `Employee` уберем определение конструктора:

```
1  class Employee : Person
```

```
2  {
3      public string Company { get; set; }
4  }
```

В данном случае мы получим ошибку, так как класс `Employee` не соответствует классу `Person`, а именно не вызывает конструктор базового класса. Даже если бы мы добавили какой-нибудь конструктор, который бы устанавливал все те же свойства, то мы все равно бы получили ошибку:

```
1  public Employee(string name, string company)
2  {
3      Name = name;
4      Company = company;
5  }
```

То есть в классе `Employee` через ключевое слово `base` надо явным образом вызвать конструктор класса `Person`:

```
1  public Employee(string name, string company) : base(name)
2  {
3      Company = company;
4  }
```

При вызове конструктора класса сначала отрабатывают конструкторы базовых классов и только затем конструкторы производных.

Либо в качестве альтернативы мы могли бы определить в базовом классе конструктор без параметров:

```
1  class Person
2  {
3      // остальной код класса
4      // конструктор по умолчанию
5      public Person()
6      {
7          FirstName = "Tom";
8          Console.WriteLine("Вызов конструктора без
   параметров");
9      }
10 }
```

Тогда в любом конструкторе производного класса, где нет обращения конструктору базового класса, все равно неявно вызывался бы этот конструктор по умолчанию. Например, следующий конструктор

```
1  public Employee(string company)
2  {
```

```
3       Company = company;
4   }
```

Фактически был бы эквивалентен следующему конструктору:

```
1   public Employee(string company):base()
2   {
3       Company = company;
4   }
```