Классы в C++

Определение классов

Кроме использования встроенных типов, таких как int, double и т.д., мы можем определять свои собственные типы или **классы**. Класс представляет составной тип, который может использовать другие типы.

Класс, как и структура, задаёт тип данных, но дополнительно определяет его поведение. Переменные этого типа по традиции называются *объектами*.

```
class имя_класса
{
    // компоненты класса
};
```

Для определения класса применяется ключевое слово **class**, после которого идет имя класса:

После названия класса в фигурных скобках располагаются компоненты класса. Причем после закрывающей фигурной скобки идет точка с запятой.

Например, определим простейший класс:

```
class Person { };

int main()
{

}
```

И после определения класса мы можем определять его переменные или константы:

```
class Person
{

};
int main()
{
    Person person;
}
```

Но данный класс мало что делает. Класс может определять переменные и константы для хранения состояния объекта и функции для определения поведения объекта. Поэтому добавим в класс Person некоторое состояние и поведение:

```cpp
#include <iostream>

class Person
{
public:
    std::string name;
    unsigned age;
    void print()
    {
        std::cout << "Name: " << name << "\tAge: " << age << std::endl;
    }
};
int main()
{
    Person person;
    // устанавливаем значения полей класса
    person.name = "Tom";
    person.age = 38;
    // вызываем функцию класса
    person.print();
}
```

Вывод на экран

```
Name: Tom	Age: 38
```

Теперь класс Person имеет две переменных name и age, которые предназначены для хранения имени и возраста человека соответственно. Переменные класса еще называют **полями** класса. Также класс определяет функцию print, которая выводит значения переменных класса 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.

Подобным образом можно получать значения переменных объектов

```cpp
#include <iostream>

class Person
{
public:
    std::string name;
    unsigned age;
    void print()
    {
        std::cout << "Name: " << name << "\tAge: " << age << std::endl;
    }
};
int main()
{
    Person person;
    // устанавливаем значения полей класса
    person.name = "Bob";
    person.age = 42;
    // получаем значения полей
    std::string username = person.name;
    unsigned userage = person.age;
    // выводим полученные данные на консоль
    std::cout << "Name: " << username << "\tAge: " << userage << std::endl;
}
```

```cpp
#include <iostream>

class Person
{
public:
    std::string name{"Undefined"};
    unsigned age{18};
    void print()
    {
        std::cout << "Name: " << name << "\tAge: " << age << std::endl;
    }
};
int main()
{
    Person person;
    person.print(); // Name: Undefined  Age: 18
}
```

Также можно поля класса, как и обычные переменные, инициализировать некоторыми начальными значениями:

Указатели на объекты классов

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 (->):

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

```
#include <iostream>

class Person
{
public:
    std::string name;
    unsigned age;
    void print()
    {
        std::cout << "Name: " << name << "\tAge: " << age << std::endl;
    }
};
int main()
{
    Person person;
    Person *ptr = &person;
    // обращаемся к полям и функции объекта через указатель
    ptr->name = "Tom";
    ptr->age = 22;
    ptr->print();
    // обращаемся к полям объекта
    std::cout << "Name: " << person.name << "\tAge: " << person.age << std::endl;
}
```

Конструкторы и инициализация объектов

Конструкторы представляют специальную функцию, которая имеет то же имя, что и класс, которая не возвращает никакого значения и которая позволяют инициалилизировать объект класса во время его создания и таким образом гарантировать, что поля класса будут иметь определенные значения. При каждом создании нового объекта класса вызывается конструктор класса.

при создании объекта класса Person, который называется person

```
Person person;
```

вызывается **конструктор по умолчанию**.

Если мы не определяем в классе явным образом конструктор, то компилятор автоматически компилирует конструктор по умолчанию.

Подобный конструктор не принимает никаких параметров и по сути ничего не делает.

Теперь определим свой конструктор.

```
#include <iostream>

class Person
{
public:
    std::string name;
    unsigned age;
    void print()
    {
        std::cout << "Name: " << name << "\tAge: " << age << std::endl;
    }
    Person(std::string p_name, unsigned p_age)
    {
        name = p_name;
        age = p_age;
        std::cout << "Person has been created" << std::endl;
    }
};
int main()
{
    Person tom("Tom", 38);  // создаем объект - вызываем конструктор
    tom.print();
}
```

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, а затем выводит сообщение о создании объекта.

Если мы определяем свой конструктор, то компилятор больше не создает конструктор по умолчанию. И при создании объекта нам надо обязательно вызвать определенный нами конструктор.

Вызов конструктора получает значения для параметров и возвращает объект класса:

```
Person tom("Tom", 38);
```

После этого вызова у объекта person для поля name будет определено значение "Tom", а для поля age - значение 38. Вподледствии мы также сможем обращаться к этим полям и переустанавливать их значения.

В качестве альтернативы для создания объекта можно использовать инициализатор в фигурных скобках:

```
Person tom{"Tom", 38};
```

можно присвоить объекту результат вызова конструктора:

```
Person tom = Person("Tom", 38);
```

Вывод на экран

```
Person has been created
Name: Tom       Age: 38
```

Конструкторы облегчают нам создание нескольких объектов, которые должны иметь разные значения:

```cpp
#include <iostream>

class Person
{
public:
    std::string name;
    unsigned age;
    void print()
    {
        std::cout << "Name: " << name << "\tAge: " << age << std::endl;
    }
    Person(std::string p_name, unsigned p_age)
    {
        name = p_name;
        age = p_age;
        std::cout << "Person has been created" << std::endl;
    }
};
int main()
{
    Person tom{"Tom", 38};
    Person bob{"Bob", 42};
    Person sam{"Sam", 25};
    tom.print();
    bob.print();
    sam.print();
}
```

Вывод на экран

```
Person has been created
Person has been created
Person has been created
Name: Tom       Age: 38
Name: Bob       Age: 42
Name: Sam       Age: 25
```

Инициализация констант и списки инициализации

В теле конструктора мы можем передать значения переменным класса. Однако константы требуют особого отношения.

```
1  class Person
2  {
3      const std::string name;
4      unsigned age{};
5  public:
6      void print()
7      {
8          std::cout << "Name: " << name << "\tAge: " << age << std::endl;
9      }
10     Person(std::string p_name, unsigned p_age)
11     {
12         name = p_name;
13         age = p_age;
14     }
15 };
```

Этот класс не будет компилироваться из-за отсутствия инициализации константы name. Хотя ее значение устанавливается в конструкторе, но к моменту, когда инструкции из тела конструктора начнут выполняться, константы уже должны быть инициализированы. И для этого необходимо использовать **списки инициализации**:

```cpp
#include <iostream>

class Person
{
    const std::string name;
    unsigned age{};
public:
    void print()
    {
        std::cout << "Name: " << name << "\tAge: " << age << std::endl;
    }
    Person(std::string p_name, unsigned p_age) : name{p_name}
    {
        age = p_age;
    }
};
int main()
{
    Person tom{"Tom", 38};
    tom.print();    // Name: Tom    Age: 38
}
```

Списки инициализации представляют перечисления инициализаторов для каждой из переменных и констант через двоеточие после списка параметров конструктора:

```cpp
Person(std::string p_name, unsigned p_age) : name{p_name}
```

Здесь выражение **name{p_name}** позволяет инициализировать константу значением параметра p_name. Здесь значение помещается в фигурные скобки, но также можно использовать кругулые:

```cpp
Person(std::string p_name, unsigned p_age) : name(p_name)
```

Списки инициализации пободным образом можно использовать и для присвоения значений переменным:

```
class Person
{
    const std::string name;
    unsigned age;
public:
    void print()
    {
        std::cout << "Name: " << name << "\tAge: " << age << std::endl;
    }
    Person(std::string p_name, unsigned p_age) : name(p_name), age(p_age)
    { }
};
```

При использовании списков инициализации важно учитывать, что передача значений должна идти в том порядке, в котором константы и переменные определены в классе. То есть в данном случае в классе сначала определена константа name, а потом переменная age. Соответственно в таком же порядке идет передача им значений. Поэтому при добавлении дополнительных полей или изменения порядка существующих придется следить, чтобы все инициализировалось в належащем порядке.

Деструктор

Деструктор выполняет освобождение использованных объектом ресурсов и удаление нестатических переменных объекта.

Деструктор автоматически вызывается, когда удаляется объект.

деструктор - это функция, которая называется по имени класса (как и конструктор) и перед которой стоит тильда (~):

```
~имя_класса()
{
    // код деструктора
}
```

Деструктор не имеет возвращаемого значения и не принимает параметров. Каждый класс может иметь только один деструктор.

Обычно деструктор не так часто требуется и в основном используется для освобождения связанных ресурсов. Например, объект класса использует некоторый файл, и в деструкторе можно определить код закрытия файла.

деструктор, который просто уведомляет об уничтожении объекта:

```
~Person()
{
    std::cout << "Person " << name << " deleted" << std::endl;
}
```

```
#include <iostream>

class Person
{
public:
    Person(std::string p_name)
    {
        name = p_name;
        std::cout << "Person " << name << " created" << std::endl;
    }
    ~Person()
    {
        std::cout << "Person " << name << " deleted" << std::endl;
    }
private:
    std::string name;
};

int main()
{
    {
        Person tom{"Tom"};
        Person bob{"Bob"};
    }   // объекты Tom и Bob уничтожаются

    Person sam{"Sam"};
}   // объект Sam уничтожается
```

В функции main создаются три объекта Person. Причем два из них создается во вложенном блоке кода.:

```
int main()
{
    {
        Person tom{"Tom"};
        Person bob{"Bob"};
    }   // объекты Tom и Bob уничтожаются

    Person sam{"Sam"};
}   // объект Sam уничтожается
```

Вывод на экран

```
Person Tom created
Person Bob created
Person Bob deleted
Person Tom deleted
Person Sam created
Person Sam deleted
```

В этом блоке кода задаются границы области видимости, в которой существуют объекты. И когда выполнение блока завершается, то уничтожаются оба объекта, и для каждого вызывается деструктор.

После этого создается третий объект - sam

Поскольку он определен в контексте функции main, то и уничтожается при завершении этой функции.

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

для любого класса, который не определяет собственный деструктор, компилятор сам создает деструктор.

если бы класс Person не имел бы явно определенного деструктора, то для него автоматически создавался бы следующий деструктор:

```
~Person(){}
```

Управление доступом. Инкапсуляция

Класс может определять различное состояние, различные функции. Однако не всегда желательно, чтобы к некоторым компонента класса был прямой доступ извне. Для разграничения доступа к различным компонентам класса применяются **спецификаторы доступа**

Спецификатор **public** делает члены класса - поля и функции открытыми, доступными из любой части программы. Например, возьмем следующий класс Person:

```cpp
#include <iostream>

class Person
{
public:
    std::string name;
    unsigned age;
    void print()
    {
        std::cout << "Name: " << name << "\tAge: " << age << std::endl;
    }
    Person(std::string p_name, unsigned p_age)
    {
        name = p_name;
        age = p_age;
    }
};
int main()
{
    Person tom{"Tom", 38};
    // поля name, age и функция print общедоступные
    tom.name = "Tomas";
    tom.age = 22;
    tom.print();    // Name: Tomas   Age: 22
}
```

в данном случае поля name и age и функция print являются открытыми, общедоступными, и мы можем обращаться к ним во внешнем коде. Однако это имеет некоторые недостатки. Так, мы можем обратиться к полям класса и присвоить им любые значения, даже если они будут не совсем корректными, исходя из логики прогаммы:

```cpp
Person tom("Tom", 22);
tom.name = "";
tom.age = 1001;
```

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

Перепишем класс Person с применением спецификатора **private**:

Все компоненты, которые определяются после спецификатора **private** и идут до спецификатора **public**, являются закрытыми, приватными. Теперь теперь мы не можем обратиться к переменным name и age вне класса Person. Мы можем к ним обращаться только внутри класса Person. А функция print и конструктор по прежнему общедоступные, поэтому мы можем обращаться к ним в любом месте программы.

```cpp
#include <iostream>

class Person
{
private:
    std::string name;
    unsigned age;
public:
    void print()
    {
        std::cout << "Name: " << name << "\tAge: " << age << std::endl;
    }
    Person(std::string p_name, unsigned p_age)
    {
        name = p_name;
        age = p_age;
    }
};
int main()
{
    Person tom{"Tom", 38};
    // функция print общедоступная
    tom.print();    // Name: Tom   Age: 22

    // поля name и age вне класса недоступны
    // tom.name = "";
    // tom.age = 1001;
}
```

Если для каких-то компонентов отсутствует спецификатор доступа, то по умолчанию применяется спецификатор **private**.

```cpp
class Person
{
    std::string name;
    unsigned age;
public:
    void print()
    {
        std::cout << "Name: " << name << "\tAge: " << age << std::endl;
    }
    Person(std::string p_name, unsigned p_age)
    {
        name = p_name;
        age = p_age;
    }
};
```

Стоит отметить, что в данном случае мы все равно можем передать некорректные значения - через конструктор.

можно проверять входные данные и использовать различные стратегии, например, определить только пустой конструктор или передавать объекту данные по умолчанию

Опосредование доступа

```cpp
#include <iostream>

class Person
{
private:
    std::string name;
    unsigned age;
public:
    Person(std::string p_name, unsigned p_age)
    {
        name = p_name;
        if (p_age > 0 && p_age < 110)
            age = p_age;
        else
            age = 18;   // если значение некорректное, устанавливаем значение по умолчанию
    }
    void print()
    {
        std::cout << "Name: " << name << "\tAge: " << age << std::endl;
    }
    void setAge(unsigned p_age)
    {
        if (p_age > 0 && p_age < 110)
            age = p_age;
    }
    std::string getName()
    {
        return name;
    }
    unsigned getAge()
    {
        return age;
    }
};
int main()
{
    Person tom{"Tom", 38};
    // изменяем возраст
    tom.setAge(22);
    tom.setAge(123);
    tom.print();    // Name: Tom   Age: 22

    //отдельно получаем имя
    std::cout << "Person name: " << tom.getName() << std::endl;
}
```

Хотя в предыдущем примере мы избегаем установки некорректных значений, тем не менее иногда может потребоваться доступ к подобным полям. Например, человек стал старше на год - надо изменить возраст. Или мы хотим отдельно получить имя. В этом мы можем определить специальные функции, через которые будем контроллировать доступ к состоянию класса

```cpp
#include <iostream>

class Person
{
private:
    std::string name;
    unsigned age;
public:
    Person(std::string p_name, unsigned p_age)
    {
        name = p_name;
        if (p_age > 0 && p_age < 110)
            age = p_age;
        else
            age = 18;   // если значение некорректное, устанавливаем значение по умолчанию
    }
    void print()
    {
        std::cout << "Name: " << name << "\tAge: " << age << std::endl;
    }
    void setAge(unsigned p_age)
    {
        if (p_age > 0 && p_age < 110)
            age = p_age;
    }
    std::string getName()
    {
        return name;
    }
    unsigned getAge()
    {
        return age;
    }
};
int main()
{
    Person tom{"Tom", 38};
    // изменяем возраст
    tom.setAge(22);
    tom.setAge(123);
    tom.print();    // Name: Tom   Age: 22

    //отдельно получаем имя
    std::cout << "Person name: " << tom.getName() << std::endl;
}
```

состояние класса скрыто извне, к нему можно получить доступ только посредством дополнительно определенных функций, которые представляют интерфейс класса.

Чтобы 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.

Наследование

Наследование (inheritance) представляет один из ключевых аспектов объектно-ориентированного программирования, который позволяет наследовать функциональность одного класса (базового класса) в другом - производном классе (derived class).

Зачем нужно наследование? Рассмотрим небольшую ситуацию, допустим, у нас есть классы, которые представляют человека и сотрудника компании:

```
class Person
{
public:
    void print() const
    {
        std::cout << "Name: " << name << "\tAge: " << age << std::endl;
    }
    std::string name;       //  имя
    unsigned age;           // возраст
};
class Employee
{
public:
    void print() const
    {
        std::cout << "Name: " << name << "\tAge: " << age << std::endl;
    }
    std::string name;       // имя
    unsigned age;           // возраст
    std::string company;    // компания
};
```

В данном случае класс Employee фактически содержит функционал класса Person: свойства name и age и функцию print. В целях демонстрации все переменные здесь определены как публичные. И здесь, с одной стороны, мы сталкиваемся с повторением функционала в двух классах. С другой стороны, мы также сталкиваемся с отношением **is** ("является"). То есть мы можем сказать, что сотрудник компании ЯВЛЯЕТСЯ человеком, так как сотрудник компании имеет в принципе все те же признаки, что и человек (имя, возраст), а также добавляет какие-то свои (компанию). Поэтому в этом случае лучше использовать механизм наследования.

Унаследуем класс Employee от класса Person:

```
class Person
{
public:
    void print() const
    {
        std::cout << "Name: " << name << "\tAge: " << age << std::endl;
    }
    std::string name;       //  имя
    unsigned age;           // возраст
};
class Employee : public Person
{
public:
    std::string company;    // компания
};
```

Для установки отношения наследования после названия класса ставится двоеточие, затем идет спецификатор доступа и название класса, от которого мы хотим унаследовать функциональность. В этом отношении класс Person еще будет называться базовым классом (также называют суперклассом, родительским классом), а Employee - производным классом (также называют подклассом, классом-наследником).

Спецификатор доступа позволяет указать, к каким членам класса производный класс будет иметь доступ. В данном случае используется спецификатор **public**, который позволяет использовать в производном классе все публичные члены базового класса. Если мы не используем модификатор доступа, то класс Employee ничего не будет знать о переменных name и age и функции print.

мы можем убрать из класса Employee те переменные, которые уже определены в классе Person.

```cpp
#include <iostream>

class Person
{
public:
    void print() const
    {
        std::cout << "Name: " << name << "\tAge: " << age << std::endl;
    }
    std::string name;       //  имя
    unsigned age;           // возраст
};
class Employee : public Person
{
public:
    std::string company;    // компания
};

int main()
{
    Person tom;
    tom.name = "Tom";
    tom.age = 23;
    tom.print();    // Name: Tom       Age: 23

    Employee bob;
    bob.name = "Bob";
    bob.age = 31;
    bob.company = "Microsoft";
    bob.print();    // Name: Bob       Age: 31
}
```

Таким образом, через переменную класса Employee мы можем обращаться ко всем открытым членам класса Person.

Конструкторы при наследовании

сделаем все переменные приватными, а для их инициализации добавим конструкторы.

стоит учитывать, что конструкторы при наследовании **не наследуются**. И если базовый класс содержит только конструкторы с параметрами, то производный класс должен вызывать в своем конструкторе один из конструкторов базового класса:

```cpp
#include <iostream>

class Person
{
public:
    Person(std::string name, unsigned age)
    {
        this->name = name;
        this->age = age;
    }
    void print() const
    {
        std::cout << "Name: " << name << "\tAge: " << age << std::endl;
    }
private:
    std::string name;       //  имя
    unsigned age;           // возраст
};
class Employee: public Person
{
public:
    Employee(std::string name, unsigned age, std::string company): Person(name, age)
    {
        this->company = company;
    }
private:
    std::string company;    // компания
};
```

```cpp
int main()
{
    Person person {"Tom", 38};
    person.print();     // Name: Tom       Age: 38

    Employee employee {"Bob", 42, "Microsoft"};
    employee.print();   // Name: Bob       Age: 42
}
```

После списка параметров конструктора производного класса через двоеточие идет вызов конструктора базового класса, в который передаются значения параметров n и a.

Если бы мы не вызвали конструктор базового класса, то это было бы ошибкой.

Вывод на экран

```
Name: Tom       Age: 38
Name: Bob       Age: 42
```

в строке

```
Employee employee {"Bob", 42, "Microsoft"};
```

Вначале будет вызываться конструктор базового класса Person, в который будут передаваться значения "Bob" и 42. И таким образом будут установлены имя и возраст. Затем будет выполняться собственно конструктор Employee, который установит компанию.

Также мы могли бы определить конструктор Employee следующим образом, используя списки инициализации:

```
Employee(std::string name, unsigned age, std::string company): Person(name, age), company(company)
{
}
```

Подключение конструктора базового класса

конструктор Employee отличается от конструктора Person одним параметром - company. Все остальные параметры из Employee передаются в Person. Однако, если бы у нас было бы полное соответствие по параметрам между двумя классами, то мы могли бы и не определять отдельный конструктор для Employee, а подключить конструктор базового класса:

в классе Employee подключаем конструктор базового класса с помощью ключевого слова **using**

Таким образом, класс Employee фактически будет иметь тот же конструктор, что и Person с теми же двумя параметрами.

```cpp
#include <iostream>

class Person
{
public:
    Person(std::string name, unsigned age)
    {
        this->name = name;
        this->age = age;
    }
    void print() const
    {
        std::cout << "Name: " << name << "\tAge: " << age << std::endl;
    }
private:
    std::string name;       //  имя
    unsigned age;           // возраст
};
class Employee: public Person
{
public:
    using Person::Person;   // подключаем конструктор базового класса
};

int main()
{
    Person person {"Tom", 38};
    person.print();     // Name: Tom       Age: 38

    Employee employee {"Bob", 42};
    employee.print();   // Name: Bob       Age: 42
}
```

Наследование деструкторов

Уничтожение объекта производного класса может вовлекать как собственно деструктор производного класса, так и деструктор базового класса. Например, определим в обоих классах деструкторы

```cpp
#include <iostream>

class Person
{
public:
    Person(std::string name, unsigned age)
    {
        this->name = name;
        this->age = age;
        std::cout << "Person created" << std::endl;
    }
    ~Person()
    {
        std::cout << "Person deleted" << std::endl;
    }
    void print() const
    {
        std::cout << "Name: " << name << "\tAge: " << age << std::endl;
    }
private:
    std::string name;
    unsigned age;
};
```

```cpp
class Employee: public Person
{
public:
    Employee(std::string name, unsigned age, std::string company): Person(name, age)
    {
        this->company = company;
        std::cout << "Employee created" << std::endl;
    }
    ~Employee()
    {
        std::cout << "Employee deleted" << std::endl;
    }
private:
    std::string company;
};

int main()
{
    Employee tom{"Tom", 38, "Google"};
    tom.print();
}
```

Вывод на экран

```
Person created
Employee created
Name: Tom       Age: 38
Employee deleted
Person deleted
```

В обоих классах деструктор просто выводит некоторое сообщение. В функции main создается один объект Employee, однако при завершении программы будет вызываться деструктор как из производного, так и из базового класса:

Запрет наследования

Иногда наследование от класса может быть нежелательно. И с помощью спецификатора **final** мы можем запретить наследование:

```
1  class Person final
2  {
3  };
```

После этого мы не сможем унаследовать другие классы от класса Person

ОШИБКА !!!!!

```
1  class Employee : public Person
2  {
3  };
```

Уровень доступа и спецификатор protected

Если переменные или функции в базовом классе являются закрытыми, то есть объявлены со спецификатором private то, производный класс хотя и наследует эти переменные и функции, но не может к ним обращаться.
попробуем определить в производном классе функцию, которая выводит значения приватных переменных базового класса:

```
#include <iostream>

class Person
{
public:
    Person(std::string name, unsigned age)
    {
        this->name = name;
        this->age = age;
    }
    void print() const
    {
        std::cout << "Name: " << name << "\tAge: " << age << std::endl;
    }
private:        // закрытые переменные - доступ из производного класса недоступен
    std::string name;       //  имя
    unsigned age;           // возраст
};

class Employee: public Person
{
public:
    Employee(std::string name, unsigned age, std::string company): Person(name, age)
    {
        this->company = company;
    }
    void printEmployee() const
    {
        std::cout << name << " works in " << company << std::endl;  // ! Ошибка
    }
private:
    std::string company;
};
```

В базовом классе Person определены приватные переменные name и age. В производном классе Employee в функции printEmployee мы пытаемся обратиться к ним, чтобы вывести их значение на консоль. И в данном случае мы столнемся с ошибкой, так как переменные name и age - приватные переменные базового класса Person. И производный класс Employee к ним не имеет доступа.

Однако иногда возникает необходимость в таких переменных и функциях базового класса, которые были бы доступны в производных классах, но не были бы доступны извне. То есть тот же **private**, только с возможностью доступа для производных классов.
И именно для определения уровня доступа подобных членов класса используется спецификатор **protected**.

определим переменную name со спецификатором **protected**:

```
#include <iostream>

class Person
{
public:
    Person(std::string name, unsigned age)
    {
        this->name = name;
        this->age = age;
    }
    void print() const
    {
        std::cout << "Name: " << name << "\tAge: " << age << std::endl;
    }
protected:
    std::string name;   // доступно из производных классов
private:
    unsigned age;
};
class Employee: public Person
{
public:
    Employee(std::string name, unsigned age, std::string company): Person(name, age)
    {
        this->company = company;
    }
    void printEmployee() const
    {
        std::cout << name << " works in " << company << std::endl;
    }
private:
    std::string company;    // компания
};

int main()
{
    Person person {"Tom", 38};
    person.print();     // Name: Tom       Age: 38

    Employee employee {"Bob", 42, "Microsoft"};
    employee.printEmployee();   // Bob works in Microsoft
}
```

мы можем использовать переменную name в производном классе, например, в методе printEmployee, но извне базового и производного классов мы к ней обратиться по-прежнему не можем.

Уровень доступа членов базового класса

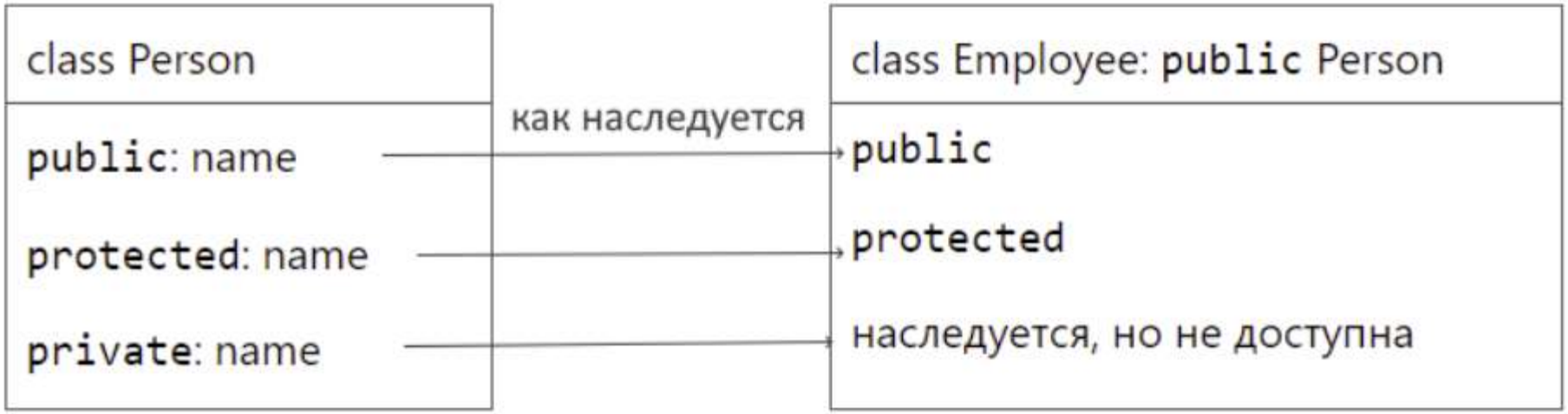
Спецификаторы доступа - **public**, **private**, **protected** играют большую роль в том, к каким именно переменным и функциям базового класса могут обращаться производные классы. Однако на доступ также влияет **спецификатор доступа базового класса**, применяемый при установке наследования:

```
class Employee: public Person
```

мы также можем использовать три варианта: **public**, **protected** или **private**.

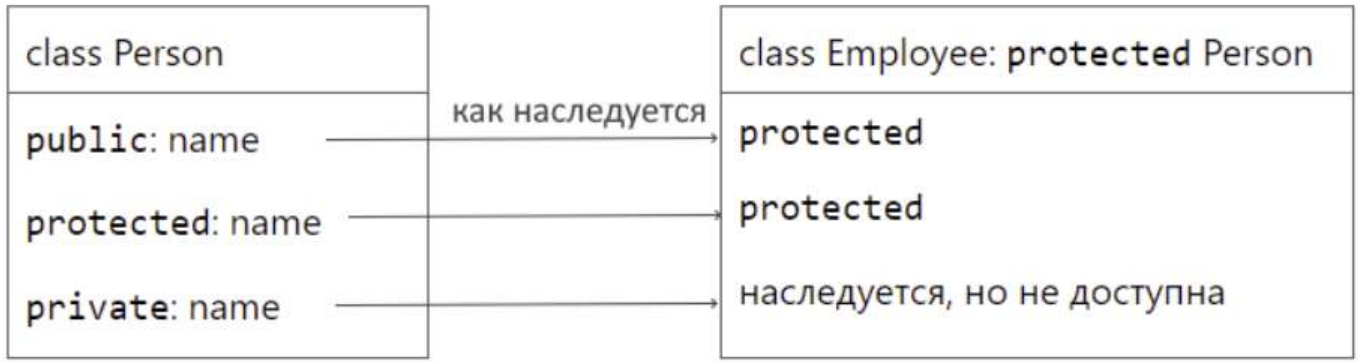
Если спецификатор базового класса явным образом не указан то по умолчанию применяется спецификатор **private**

Таким образом, в базовом классе при определении переменных и функций мы можем использовать три спецификатора для управления доступом: **public**, **protected** или **private**. И те же три спецификатора мы можем использовать при установке наследования от базового класса. Эти спецификаторы накладываются друг на друга и образуют 9 возможных комбинаций.

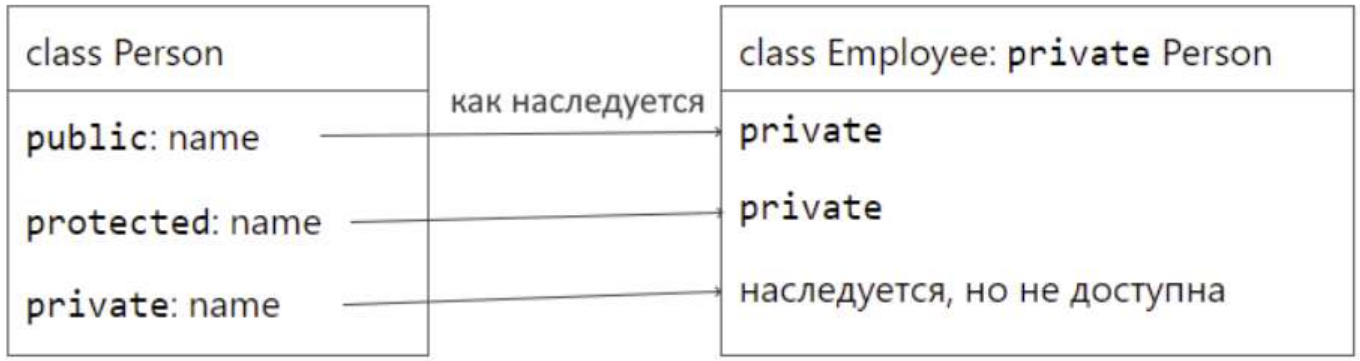


Если спецификатор базового класса - **public**, то уровень доступа унаследованных членов остается неизменным. Таким образом, унаследованные открытые члены являются общедоступными, а унаследованные члены со спецификатором **protected** сохраняют этот спецификатор и в производном классе.

Если члены базового класса определены со спецификатором **private**, то в производном классе они в принципе недоступны независимо от спецификатора доступа к базовому классу.



Если спецификатор базового класса - **protected**, то все унаследованные члены со спецификатором `protected` и `public` в производном классе наследуются как `protected`. Смысл этого состоит в том, что если у производного класса будут свои классы-наследники, то в этих классах-наследниках также можно обращаться к подобным членам базового класса.



Если спецификатор базового класса - **private**, то все унаследованные члены со спецификатором `protected` и `public` в производном классе наследуются как `private`. Они доступны в любой функции производного класса, но вне производного класса (в том числе у его наследников) они не доступны.

Пусть спецификатором базового класса будет **private**

```cpp
#include <iostream>

class Person
{
public:
    Person(std::string name, unsigned age)
    {
        this->name = name;
        this->age = age;
    }
    void print() const
    {
        std::cout << "Name: " << name << "\tAge: " << age << std::endl;
    }
protected:
    std::string name;   // доступно из производных классов
private:
    unsigned age;
};
class Employee: private Person
{
public:
    Employee(std::string name, unsigned age, std::string company): Person(name, age)
    {
        this->company = company;
    }
    void printEmployee() const
    {
        print();    // функция print внутри класса Employee доступна
        std::cout << name << " works in " << company << std::endl;
    }
private:
    std::string company;    // компания
};

int main()
{
    Employee employee {"Bob", 42, "Microsoft"};
    employee.printEmployee();   // Bob works in Microsoft
    // employee.print();       // функция print недоступна
}
```

Поскольку спецификатор базового класса Person - **private**, то класс Employee наследует переменную **name** и функцию **print** как private-члены. К таким переменным и функциям можно обратиться внутри класса Employee. Однако вне класса Employee они будут недоступны

если мы создадим новый класс и унаследуем его от Employee, например, класс Manager:

```cpp
class Manager: public Employee
{
public:
    Manager(std::string name, unsigned age, std::string company): Employee(name, age, company)
    { }
};
```

То приватные переменная name и функция print из Employee в классе Manager будут недоступны.

Установка публичного доступа

если в примере для класса Employee мы все таки хотим вызвать функцию print?

Мы можем восстановить уровень доступа с помощью ключевого слова **using**:

```cpp
#include <iostream>

class Person
{
public:
    Person(std::string name, unsigned age)
    {
        this->name = name;
        this->age = age;
    }
    void print() const
    {
        std::cout << "Name: " << name << "\tAge: " << age << std::endl;
    }
protected:
    std::string name;   // доступно из производных классов
private:
    unsigned age;
};
class Employee: private Person
{
public:
    Employee(std::string name, unsigned age, std::string company): Person(name, age)
    {
        this->company = company;
    }
    using Person::print;
    void printEmployee() const
    {
        std::cout << name << " works in " << company << std::endl;
    }
private:
    std::string company;    // компания
};

int main()
{
    Employee employee {"Bob", 42, "Microsoft"};
    employee.print();  // Name: Bob       Age: 42 - функция доступна
}
```

После этого функция print будет иметь свой первоначальный спецификатор доступа - public и будет доступна вне класса Employee

Подобным образом можно сделать публичной и переменную **name**, несмотря на то, что в базовом классе Person она определена как **protected**: `using Person::name;`

Однако если переменные и функции определены в базовом классе как приватные сделать их публичными нельзя.

Скрытие функционала базового класса

C++ позволяет определять в производном классе переменные и функции с теми же именами, что имеют переменные и функции в базовом классе. В этом случае переменные и функции производного класса будут скрывать одноименные переменные и функции базового класса.

Скрытие функций

Производный класс может определить функцию с тем же именем, что и функция в базовом классе, с тем же или другим списком параметров. Для компилятора такая функция будет существовать независимо от базового класса. И подобное определение функции в производном классе не будет переопределением функции из базового класса.

класс Person, который представляет человека, определяет функцию `print()`, которая выводит значение переменных name и age.

Класс Employee, который представляет сотрудника компании и является производным от класса Person, также определяет функцию `print()`, которая выводит значение переменной company.

```cpp
#include <iostream>

class Person
{
public:
    Person(std::string name, unsigned age) : name(name), age(age)
    {}
    void print() const
    {
        std::cout << "Name: " << name << "\tAge: " << age << std::endl;
    }
private:
    std::string name;
    unsigned age;
};
```

```cpp
class Employee: public Person
{
public:
    Employee(std::string name, unsigned age, std::string company):
        Person(name, age), company(company)
    { }
    void print() const
    {
        std::cout << "Works in " << company << std::endl;
    }
private:
    std::string company;
};

int main()
{
    Employee tom{"Tom", 38, "Google"};
    tom.print(); // Works in Google
}
```

В итоге объект Employee будет использовать реализацию функции print класса Employee, а не класса Person

```
int main()
{
    Employee tom{"Tom", 38, "Google"};
    tom.print(); // Works in Google
}
```

Функция print в Employee скрывает функцию print класса Person. Однако иногда может потребоваться возможность вызвать реализацию функции, которая определена именно в базовом классе. В этом случае можно использовать оператор ::

`базовый_класс::функция`

```
#include <iostream>

class Person
{
public:
    Person(std::string name, unsigned age) : name(name), age(age)
    {}
    void print() const
    {
        std::cout << "Name: " << name << "\tAge: " << age << std::endl;
    }
private:
    std::string name;
    unsigned age;
};
```

```
class Employee: public Person
{
public:
    Employee(std::string name, unsigned age, std::string company):
        Person(name, age), company(company)
    { }
    void print() const
    {
        Person::print();    // вызываем функцию print из базового класса
        std::cout << "Works in " << company << std::endl;
    }
private:
    std::string company;
};

int main()
{
    Employee tom{"Tom", 38, "Google"};
    tom.print();
}
```

```
Name: Tom       Age: 38
Works in Google
```

`Person::print();` представляет обращение к функции print базового класса Person.

Скрытие переменных

Производный класс может иметь переменные с тем же именем, что и базовый класс, Хотя такие ситуации могут привести к путанице, и, возможно, представляют нелучший вариант наименования переменных. Тем не менее мы можем так делать.

```
#include <iostream>

class Integer
{
public:
    Integer(unsigned value): value(value)
    { }
    void printInteger() const
    {
        std::cout << value << std::endl;
    }
protected:
    unsigned value;
};
class Decimal: public Integer
{
public:
    Decimal(unsigned i_value, unsigned d_value): Integer(i_value), value(d_value)
    { }
    void printDecimal() const
    {
        std::cout << Integer::value << "." << value << std::endl;
    }
protected:
    unsigned value;
};

```

```
int main()
{
    Decimal decimal{12345, 3456};
    decimal.printInteger(); // 12345
    decimal.printDecimal(); // 12345.3456
}
```

Здесь класс Integer представляет целое число, значение которого хранится в переменной value. Этот класс наследуется классом Decimal, который представляет дробное число. Целая часть хранится в поле value класса Integer. А для хранения дробной части определена своя переменная **value**. Причем переменная value в Integer имеет спецификатор **protected**, поэтому теоретически мы могли бы обращаться к ней в классе Decimal. Однако поскольку в Decimal определена своя переменная value, то она скрывает переменную value и базового класса.

Чтобы все таки обратиться к переменной value из базового надо использовать оператор **::**

```
Integer::value
```

Geant4

типы переменных

ввод-вывод

библиотека CLHEP

система единиц

Основные типы переменных Geant4

Для достижения переносимости кода в Geant4 переопределены основные типы переменных

G4int, *G4long*, *G4float*, *G4double*,

G4bool, *G4complex*, *G4String*

Ввод и вывод в коде Geant4

Можно использовать обычные printf() и cout

Однако более правильно использовать переопределенные потоки Geant4:

G4cout << *“test”* << *G4endl*;

G4cerr << *“error”* << *G4endl*;

Библиотека CLHEP http://cern.ch/clhep

Class Library for High Energy Physics

используется в коде Geant4

Содержит описание стандартных математических объектов, часто используемых в ФВЭ

- 3-векторы и 4-векторы
- действия с матрицами
- геометрические объекты и преобразования
- генераторы случайных чисел
- система единиц и основные физические константы

G4ThreeVector
- трехкомпонентный (x,y,z) вектор и действия с ним

G4LorenzVector
- четырехкомпонентный (x,y,z,t) вектор

G4RotationMatrix
- матрица 3x3, определяющая вращение 3-вектора

G4LorenzRotation
- матрица 4x4, определяющая вращение 4-вектора

Геометрические объекты и преобразования
- **G4Plane3D, G4Transform3D, G4Normal3D, G4Point3D, G4Vector3D**

Система единиц Geant4

Для некоторых единиц измерения в Geant4 установлено значение «1».

- millimeter (mm) - миллиметр
- nanosecond (ns) - наносекунда
- MeV - Мега электронвольт
- eplus - заряд позитрона
- kelvin - кельвин
- mole - моль
- candela -сила света
- radian - радиан
- steradian - стерадиан

Все остальные единицы задаются через указанные в этом списке величины

единицы измерения описаны в файле G4SystemOfUnits.hh.

Любое число, имеющее размерность, должно быть умножено на соответствующую единицу для перевода во внутреннюю систему единиц Geant4

length = 10.0 * cm;
kinetic_energy = 5.0 * GeV;

```
double length = 5. * cm;
G4cout << length << '\n';
```

Выводимым в консоль результатом будет 50.

Для вывода величины в желаемых единицах следует делить число на единицу измерения

G4cout << eDep / MeV << " [MeV]" << G4endl;

```
double mass = 5. * g;
G4cout << mass / g << '\n';
```

Выводимый результат будет 5

если существует необходимость в добавлении единиц измерения, то можно добавить в файл с описанием геометрии строки

```
#include <G4SystemOfUnits.hh>

static const G4double inch = 2.54*cm;
```

чтобы увидеть весь список доступных единиц измерения, можно воспользоваться командой:

/units/list (например, воспользоваться списком команд в меню визуализатора)

Основные понятия Geant4

Сеанс (Run)

Событие (Event)

Трек (Track)

Шаг (Step)

Срабатывание (Hit)

Сеанс (Run)

- ✓ Период набора статистики, в котором не меняются условия проведения эксперимента (параметры пучка, конфигурация и параметры детектора, материал мишени и т.п.)
- ✓ В Geant4 – самый крупный элемент моделирования, состоящий из последовательности событий
- ✓ Во время сеанса описание геометрии и набор физических процессов остаются неизменными
- ✓ Представлен классом G4Run
- ✓ Управление осуществляется объектом класса G4RunManager

Событие (Event)

- ✓ Единичное независимое измерение физического явления детектором
- ✓ В Geant4 представлено классом G4Event
- ✓ G4Event содержит все входные и выходные характеристики (исходные частицы, срабатывания и т.д.) данного (текущего) события
- ✓ G4Event создается объектом класса G4RunManager и передается объекту класса G4EventManager, который осуществляет управление событием

Структура события

- Первичная вершина и первичная частица
- Траектории
- Коллекция срабатываний
- G4EventManager управляет объектами G4Track, соответствующими данному событию, взаимодействуя с объектами классов G4TrackManager и G4StackManager

Трек (Track) и Шаг (Step)

- Шаг представлен классом G4Step и описывает минимальное продвижение частицы через вещество с учетом различных физических процессов
- Треки представлены классом G4Track и содержат информацию о последнем шаге.
- Объект G4Track, таким образом, описывает полное продвижение частицы в веществе к моменту обращения к данному объекту

Срабатывание (Hit)

- Описывает единичное взаимодействие частицы с веществом в детектирующем объеме
- Содержит информацию о координате и времени взаимодействия, энергии и импульсе частицы в этой точке, энерговыделении, геометрическую информацию (объем, в котором произошло взаимодействие и т.п.)
- Является “истинной” Монте-Карло информацией (Monte-Carlo truth)