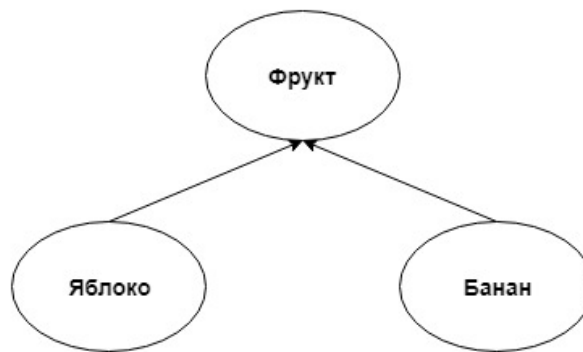


Спадкування

Розглянемо приклад з яблуками і бананами. Хоча яблука і банани - це різні фрукти, але у них обох є одна загальна властивість: вони обидва є *фруктами*. І оскільки яблука і банани - це фрукти, то, за логікою, все, що вірно для фруктів, вірно і для яблук з бананами. Наприклад, всі фрукти мають свою назву, колір і розмір. Яблука і банани також мають свої назви, колір і розмір. Ми можемо сказати, що яблука і банани успадкували (придбали) всі властивості фруктів, тому що вони самі є фруктами. Ми також знаємо, що фрукти піддаються процесу дозрівання, завдяки якому вони стають їстівними. Оскільки яблука і банани є фруктами, то, відповідно, вони також піддаються процесу дозрівання, в результаті чого стають їстівними.

Якщо зобразити відносини між яблуками, бананами і фруктами в діаграмі, то це буде виглядати приблизно так:



2. Базове спадкування

```
1  #include <string>
2
3  class Human
4  {
5  public:
6      std::string m_name;
7      int m_age;
8
9      Human(std::string name = "", int age = 0)
10         : m_name(name), m_age(age)
11     {
12     }
13
14     std::string getName() const { return m_name; }
15     int getAge() const { return m_age; }
16
17 };
```

```

1 class BasketballPlayer
2 {
3 public:
4     double m_gameAverage;
5     int m_points;
6
7     BasketballPlayer(double gameAverage = 0.0, int points = 0)
8         : m_gameAverage(gameAverage), m_points(points)
9     {
10    }
11 };

```

```

1 // BasketballPlayer открыто наследует Human
2 class BasketballPlayer : public Human
3 {
4 public:
5     double m_gameAverage;
6     int m_points;
7
8     BasketballPlayer(double gameAverage = 0.0, int points = 0)
9         : m_gameAverage(gameAverage), m_points(points)
10    {
11    }
12 };

```

Коли **BasketballPlayer** успадковує властивості класу **Human**, то **BasketballPlayer** набуває методи і змінні-члени класу **Human**. Крім того, **BasketballPlayer** має ще два своїх власних члена: **m_gameAverage**, **m_points**. Тут є сенс, тому що ці властивості специфічні тільки для **BasketballPlayer**, а не для кожного **Human**-а. Таким чином, об'єкти **BasketballPlayer** матимуть 4 члени:

- **m_gameAverage** і **m_points** від **BasketballPlayer**;
- **m_name** і **m_age** від **Human**.

```

1  #include <iostream>
2  #include <string>
3
4  class Human
5  {
6  public:
7      std::string m_name;
8      int m_age;
9
10     Human(std::string name = "", int age = 0)
11         : m_name(name), m_age(age)
12     {
13     }
14
15     std::string getName() const { return m_name; }
16     int getAge() const { return m_age; }
17 };
18
19 // BasketballPlayer открыто наследует Human
20 class BasketballPlayer : public Human
21 {
22 public:
23     double m_gameAverage;
24     int m_points;
25
26     BasketballPlayer(double gameAverage = 0.0, int points = 0)
27         : m_gameAverage(gameAverage), m_points(points)
28     {
29     }
30 };
31
32 int main()
33 {
34     // Создаём нового Баскетболиста
35     BasketballPlayer anton;
36     // Присваиваем ему имя (мы можем делать это напрямую, так как m_name является public)
37     anton.m_name = "Anton";
38     // Выводим имя Баскетболиста
39     std::cout << anton.getName() << '\n'; // используем метод getName(), который мы унаследовали от
40
41     return 0;
42 }

```

Результат виконання програми вище:

Anton

Це працює, тому що **anton** є об'єктом класу **BasketballPlayer**, а всі об'єкти класу **BasketballPlayer** мають змінну-член **m_name** і метод **getName ()**, успадковані від класу **Human**.

Тепер напишемо ще один клас, який також буде успадковувати властивості **Human**. Наприклад, клас **Employee** (Працівник). Працівник «є» Людиною, тому використовувати спадкування тут доречно:

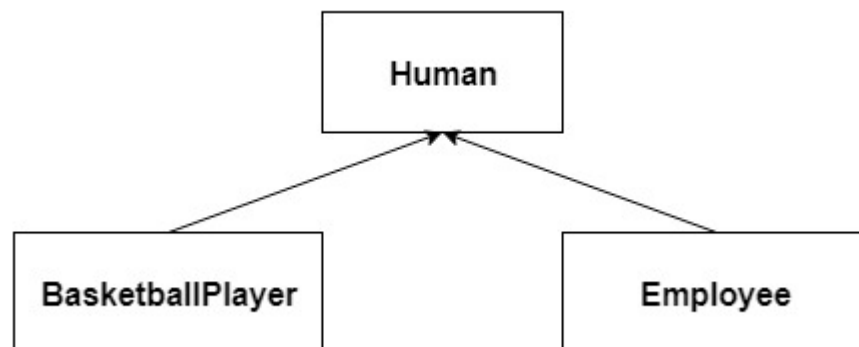
```

1  // Employee открыто наследует Human
2  class Employee: public Human
3  {
4  public:
5      int m_wage;
6      long m_employeeID;
7
8      Employee(int wage = 0, long employeeID = 0)
9          : m_wage(wage), m_employeeID(employeeID)
10     {
11     }
12
13     void printNameAndWage() const
14     {
15         std::cout << m_name << ": " << m_wage << '\n';
16     }
17 };

```

Працівник успадковує **m_name** і **m_age** від **Human**-а (а також два методи) і має ще дві власні змінні-члени і один метод. Зверніть увагу, метод **printNameAndWage ()** використовує змінні як з класу, до якого належить (**Employee::m_wage**), так і з батьківського класу (**Human::m_name**).

Проілюструємо:



Зверніть увагу, **Employee** і **BasketballPlayer** не мають прямих відносин, хоча обидва успадковують властивості класу **Human**.

```

1  #include <iostream>
2  #include <string>
3
4  class Human
5  {
6  public:
7      std::string m_name;
8      int m_age;
9
10     std::string getName() const { return m_name; }
11     int getAge() const { return m_age; }
12
13     Human(std::string name = "", int age = 0)
14         : m_name(name), m_age(age)
15     {
16     }
17 };
18
19 // Employee открыто наследует Human
20 class Employee: public Human
21 {
22 public:
23     int m_wage;
24     long m_employeeID;
25
26     Employee(int wage = 0, long employeeID = 0)
27         : m_wage(wage), m_employeeID(employeeID)
28     {
29     }
30
31     void printNameAndWage() const
32     {
33         std::cout << m_name << ": " << m_wage << '\n';
34     }
35 };
36
37 int main()
38 {
39     Employee ivan(350, 787);
40     ivan.m_name = "Ivan"; // мы можем это сделать, так как m_name является public
41
42     ivan.printNameAndWage();
43
44     return 0;
45 }

```

Результат виконання програми вище:

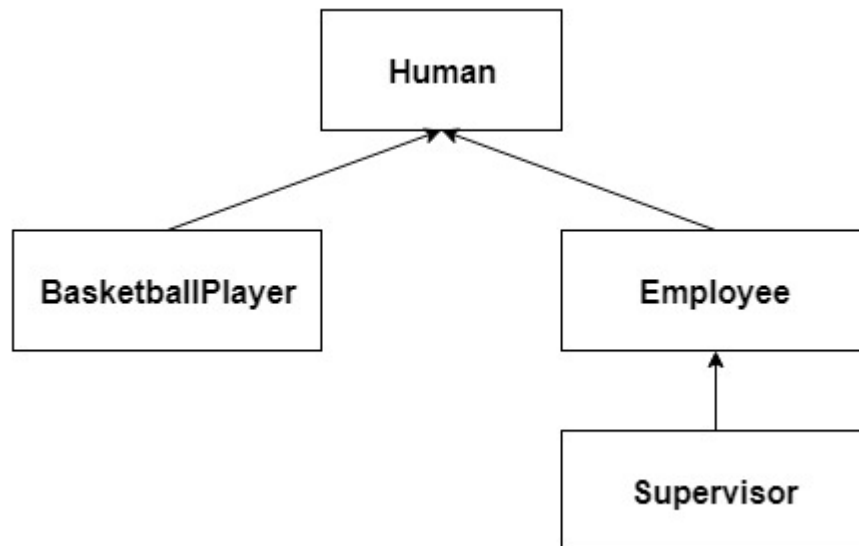
```
Ivan: 350
```

Можна наслідувати від класу, який сам успадковує від іншого класу.

```

1 class Supervisor: public Employee
2 {
3     public:
4         // Этот Супервайзер может наблюдать максимум за 5-тью Работниками
5         long m_overseesIDs[5];
6
7         Supervisor()
8         {
9         }
10
11 };

```



Всі об'єкти **Supervisor** успадковують методи і змінні від **Employee** і **Human**, а також мають свою власну змінну-член **m_nOverseesID**.

Побудувавши такі ланцюжки наслідувань, ми можемо створити набір повторно використовуваних класів, які будуть мати загальні властивості вгорі і стає все більш специфічними на кожному наступному рівні успадкування.

3. Порядок побудови класів в ланцюжку спадкування

Часто буває ситуація, коли одні класи успадковують властивості інших класів, які, в свою чергу, успадковують властивості ще інших класів.

У C ++ завжди йде побудова з «першого» або «топового» класу ієрархії. Потім C ++ переходить до наступного класу в ієрархії і виконує його побудова. Цей процес послідовний.

Проілюструємо порядок побудови класів в ланцюжку спадкування вище:

```

1 class A
2 {
3 public:
4     A()
5     {
6         std::cout << "A\n";
7     }
8 };
9
10 class B: public A
11 {
12 public:
13     B()
14     {
15         std::cout << "B\n";
16     }
17 };
18
19 class C: public B
20 {
21 public:
22     C()
23     {
24         std::cout << "C\n";
25     }
26 };
27
28 class D: public C
29 {
30 public:
31     D()
32     {
33         std::cout << "D\n";
34     }
35 };

```

```

1 int main()
2 {
3     std::cout << "Constructing A: \n";
4     A cA;
5
6     std::cout << "Constructing B: \n";
7     B cB;
8
9     std::cout << "Constructing C: \n";
10    C cC;
11
12    std::cout << "Constructing D: \n";
13    D cD;
14 }

```

результат:

```

Constructing A:
A
Constructing B:
A
B
Constructing C:
A
B
C
Constructing D:
A
B
C
D

```

4. Конструктори і ініціалізація

Розглянемо класи **Parent** і **Child**:

```
1 class Parent
2 {
3 public:
4     int m_id;
5
6     Parent(int id=0)
7         : m_id(id)
8     {
9     }
10
11     int getId() const { return m_id; }
12 };
13
14 class Child: public Parent
15 {
16 public:
17     double m_value;
18
19     Child(double value=0.0)
20         : m_value(value)
21     {
22     }
23
24     double getValue() const { return m_value; }
25 };
```

Об'єкт класу **Parent** створюється наступним чином:

```
1 int main()
2 {
3     Parent parent(7); // вызывается конструктор Parent(int)
4
5     return 0;
6 }
```

Об'єкт класу **child** створюється наступним чином:

```
1 int main()
2 {
3     Child child(1.5); // вызывается конструктор Child(double)
4
5     return 0;
6 }
```

Єдина відмінність між ініціалізацією об'єктів звичайного і дочірнього класу полягає в тому, що при ініціалізації об'єкта дочірнього класу, спочатку виконується конструктор батьківського класу (для ініціалізації частини батьківського класу) і тільки потім вже виконується конструктор дочірнього

класу.

Одним з недоліків дочірнього класу **Child** є те, що ми не можемо ініціювати **m_id** при створенні об'єкта класу **Child**. Що, якщо ми хочемо задати значення як для **m_value**(частини **Child**), так і для **m_id**(частини **Parent**)? Новачки часто намагаються вирішити цю проблему наступним чином:

```
1 class Child: public Parent
2 {
3 public:
4     double m_value;
5
6     Child(double value=0.0, int id=0)
7         // Не працює
8         : m_value(value), m_id(id)
9     {
10    }
11
12     double getValue() const { return m_value; }
13 };
```

```
1 class Child: public Parent
2 {
3 public:
4     double m_value;
5
6     Child(double value=0.0, int id=0)
7         : m_value(value)
8     {
9         m_id = id;
10    }
11
12     double getValue() const { return m_value; }
13 };
```

```
1 class Child: public Parent
2 {
3 public:
4     double m_value;
5
6     Child(double value=0.0, int id=0)
7         : Parent(id), // викликається конструктор Parent(int) зі значенням id!
8         m_value(value)
9     {
10    }
11
12     double getValue() const { return m_value; }
13 };
```

```

1 int main()
2 {
3     Child child(1.5, 7); // вызывается конструктор Child(double, int)
4     std::cout << "ID: " << child.getId() << '\n';
5     std::cout << "Value: " << child.getValue() << '\n';
6
7     return 0;
8 }

```

ID: 7

Value: 1.5

```

1 #include <iostream>
2
3 class Parent
4 {
5 private: // наш m_id теперь закрытый
6     int m_id;
7
8 public:
9     Parent(int id=0)
10        : m_id(id)
11        {
12        }
13
14     int getId() const { return m_id; }
15 };
16 class Child: public Parent
17 {
18 private: // наш m_value теперь закрытый
19     double m_value;
20
21 public:
22     Child(double value=0.0, int id=0)
23        : Parent(id), // вызывается конструктор Parent(int) со значением id!
24        m_value(value)
25        {
26        }
27
28     double getValue() const { return m_value; }
29 };
30
31 int main()
32 {
33     Child child(1.5, 7); // вызывается конструктор Child(double, int)
34     std::cout << "ID: " << child.getId() << '\n';
35     std::cout << "Value: " << child.getValue() << '\n';
36
37     return 0;
38 }

```

ID: 7

Value: 1.5

```

1  #include <string>
2
3  class Human
4  {
5  private:
6      std::string m_name;
7      int m_age;
8
9  public:
10     Human(std::string name = "", int age = 0)
11         : m_name(name), m_age(age )
12     {
13     }
14
15     std::string getName() const { return m_name; }
16     int getAge() const { return m_age; }
17
18 };
19 // BasketballPlayer открыто наследует класс Human
20 class BasketballPlayer: public Human
21 {
22 private:
23     double m_gameAverage;
24     int m_points;
25
26 public:
27     BasketballPlayer(std::string name = "", int age = 0,
28                     double gameAverage = 0.0, int points = 0)
29         : Human(name, age), // вызывается Human(std::string, int) для инициализации
30           m_gameAverage(gameAverage), m_points(points)
31     {
32     }
33
34     double getGameAverage() const { return m_gameAverage; }
35     int getPoints() const { return m_points; }
36 };

```

Створювати об'єкти класу **BasketballPlayer** наступним чином:

```

1  int main()
2  {
3      BasketballPlayer anton("Anton Ivanovuch", 45, 300, 310);
4
5      std::cout << anton.getName() << '\n';
6      std::cout << anton.getAge() << '\n';
7      std::cout << anton.getPoints() << '\n';
8
9      return 0;
10 }

```

Anton Ivanovuch

45

310

Спадкування і специфікатор доступу protected

```
1 class Parent
2 {
3 public:
4     int m_public; // доступ к этому члену открыт для всех объектов
5 private:
6     int m_private; // доступ к этому члену открыт только для других членов класса Parent и для друзей
7 protected:
8     int m_protected; // доступ к этому члену открыт для других членов класса Parent, дружественных классов
9 };
10
11 class Child: public Parent
12 {
13 public:
14     Child()
15     {
16         m_public = 1; // разрешено: доступ к открытым членам родительского класса из дочернего класса
17         m_private = 2; // запрещено: доступ к закрытым членам родительского класса из дочернего класса
18         m_protected = 3; // разрешено: доступ к защищённым членам родительского класса из дочернего класса
19     }
20 };
21
22 int main()
23 {
24     Parent parent;
25     parent.m_public = 1; // разрешено: доступ к открытым членам класса извне
26     parent.m_private = 2; // запрещено: доступ к закрытым членам класса извне
27     parent.m_protected = 3; // запрещено: доступ к защищённым членам класса извне
28 }
```

```
1 // Открытое наследование
2 class Pub: public Parent
3 {
4 };
5
6 // Закрытое наследование
7 class Pri: private Parent
8 {
9 };
10
11 // Защищённое наследование
12 class Pro: protected Parent
13 {
14 };
15
16 class Def: Parent // по умолчанию C++ устанавливает закрытое наследование
17 {
18 };
```

Публічне успадкування - це найпоширеніший тип успадкування. Якщо ви самі не визначили тип спадкування, то в C ++ за замовчуванням буде обраний тип спадкування **private**.

Це дає нам 9 комбінацій: 3 специфікатор доступу (**public**, **private** і **protected**) і 3 типу успадкування (**public**, **private** і **protected**).

Так в чому ж різниця між ними? Якщо коротко, то при спадкуванні специфікатор доступу члена батьківського класу може бути змінений в дочірньому класі (в залежності від типу успадкування). Іншими словами, члени, які були **public** або **protected** в батьківському класі, можуть стати **private** в дочірньому класі.

Спосіб взаємодії специфікаторів доступу, типів успадкування і дочірніх класів може викликати плутанину. Щоб це усунути, з'ясуємо все ще раз:

1. Клас завжди має доступ до своїх власних НЕ успадкованим членам (і друзні йому класи також мають доступ). Специфікатори доступу впливають тільки на те, чи можуть об'єкти поза класом і дочірні класи звертатися до цих членам.
2. Коли дочірні класи успадковують члени батьківських класів, то члени батьківського класу можуть змінювати свої специфікатор доступу в дочірньому класі. Це ніяк не впливає на власні (не успадковане) члени дочірніх класів (які визначені в дочірньому класі і мають свої власні специфікатор доступу). Це впливає тільки на те, чи можуть об'єкти ззовні і класи дочірні нашим дочірнім класами отримати доступ до успадкованих членам батьківського класу.

Загальна таблиця специфікаторів доступу і типів успадкування:

Специфікатор доступу в батьківському класі	Специфікатор доступу при спадкуванні типу public в дочірньому класі	Специфікатор доступу при спадкуванні типу private в дочірньому класі	Специфікатор доступу при спадкуванні типу protected в дочірньому класі
public	public	private	protected
private	недоступний	недоступний	недоступний
protected	protected	private	protected

```
class Mother {
public:
    Mother()
    {
        cout <<"Mother ctor"<<endl;
    }
    ~Mother()
    {
        cout <<"Mother dtor"<<endl;
    }
};
```

```
int main() {
    Mother m;
}
/* Outputs
Mother ctor
Mother dtor
*/
```

```
class Daughter: public Mother {  
public:  
    Daughter()  
    {  
        cout <<"Daughter ctor"<<endl;  
    }  
    ~Daughter()  
    {  
        cout <<"Daughter dtor"<<endl;  
    }  
};
```

```
int main() {  
    Daughter m;  
}  
  
/*Outputs  
Mother ctor  
Daughter ctor  
Daughter dtor  
Mother dtor  
*/
```