

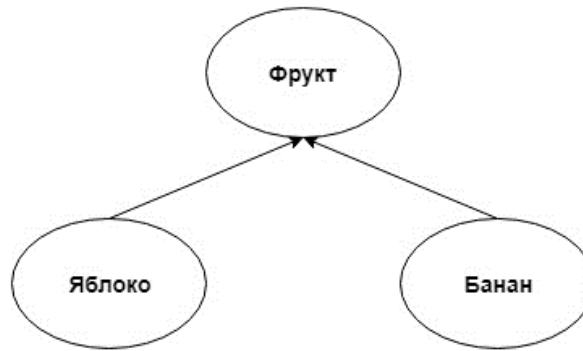
Спадкування

1. Загальні поняття
2. Базове спадкування
3. Порядок побудови класів в ланцюжку спадкування
4. Конструктори і ініціалізація
5. Спадкування і специфікатор доступу `protected`

На відміну від композиції об'єктів, яка включає створення нових об'єктів шляхом об'єднання і з'єднання інших об'єктів, спадкування включає в себе створення нових об'єктів шляхом безпосереднього збереження властивостей і поведінки інших об'єктів, а потім їх розширення або навпаки - конкретизації. Подібно композиції об'єктів, спадкування відбувається всюди в реальному житті. Наприклад, при народженні ви успадкували гени від ваших батьків і придбали певні фізичні властивості в обох з них (схильність до хвороб, видам діяльності і т.д.), але потім ви додали свою особистість до всього придбаному. Технологічні продукти (комп'ютери, смартфони і т.д.) успадковують функціонал від своїх попередників, при цьому додаючи щось своє (нове, унікальне), зберігаючи сумісність, наприклад, процесор Intel Pentium успадкував багато функціональні властивості від процесора Intel 486, який, в свою чергу, успадкував свій функціонал від більш ранніх процесорів. C++ багато успадкував від мови C, на якому він базується, а C успадкував багато властивостей від інших мов програмування, які були до нього.

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

Спадкування в C++ відбувається між класами і має тип відносин «є». Клас, від якого успадковують, називається **батьківським** (або ще «**базовим**» або «**суперкласом**»), а клас, який успадковує, називається **дочірнім** (або ще «**похідним**» або «**підкласом**»). У діаграмі (рис. 1) фрукт є батьківським класом, а Яблоко та Банан- дочірніми. дочірній клас успадковує як поведінку (методи), так і властивості (змінні-члени) від батька (з урахуванням деяких обмежень доступу). Ці методи і змінні стають членами дочірнього класу.

Оскільки дочірні класи є повноцінними класами, то вони можуть (звичайно) мати і свої власні члени.

Ось простий клас Human для подання Людини:

```
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 };
```

У цьому класі ми визначили тільки ті члени, які є загальними для всіх об'єктів цього класу. Кожен Людина (незалежно від статі, професії і т.д.) має Ім'я та Вік. У прикладі вище все змінні-члени і методи класу зроблені відкритими. Це зроблено заради простоти прикладу. Зазвичай змінні-члени потрібно робити private.

Припустимо, що нам потрібно написати програму, яка буде відслідковувати інформацію про баскетболістів. Ми можемо зберігати середній рівень гри баскетболіста і кількість очок.

Ось наш незавершений клас **BasketballPlayer**:

```
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 };
```

Також нам потрібно знати **Ім'я** та **Вік** баскетболіста, а ця інформація вже у нас є: вона зберігається в класі **Human**.

У нас є три варіанти додавання **Імені** і **Віку** в **BasketballPlayer**:

- Додати **Ім'я** та **Вік** в клас **BasketballPlayer** безпосередньо в якості членів. Це найгірший варіант, тому що відбудеться дублювання коду, який вже існує в класі **Human**. Будь-які оновлення в **Human** також повинні бути продубльовані і в **BasketballPlayer**.
- Додати клас **Human** як члена в клас **BasketballPlayer**, використовуючи композицію. Але слід питання: «Чи може **BasketballPlayer** мати **Human**?». Ні, це некоректно.
- Зробити так, щоб **BasketballPlayer** успадкував необхідні атрибути від **Human**. Пам'ятайте, що тип відносин в спадкуванні - «є». Чи є **BasketballPlayer Human** (тобто Людиною)? Звичайно! Тому вибір - успадкування.

Щоб клас **BasketballPlayer** успадкував інформацію від класу **Human**, потрібно після оголошення `class BasketballPlayer` використовувати двокрапку, ключове слово **public** і **ім'я** класу, від якого ми хочемо успадкувати. Це називається **відкритим успадкуванням**:

```
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**.

Повний код програми:

```
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
20 // BasketballPlayer открыто наследует Human
21 class BasketballPlayer : public Human
22 {
23 public:
24     double m_gameAverage;
25     int m_points;
26
27     BasketballPlayer(double gameAverage = 0.0, int points = 0)
28         : m_gameAverage(gameAverage), m_points(points)
29     {
30     }
31 };
32
33 int main()
34 {
35     // Создаём нового Баскетболиста
36     BasketballPlayer anton;
37     // Присваиваем ему имя (мы можем делать это напрямую, так как m_name является public)
38     anton.m_name = "Anton";
39     // Выводим имя Баскетболиста
40     std::cout << anton.getName() << '\n'; // используем метод getName(), который мы унаследовали от
41
42     return 0;
43 }
```

Результат виконання програми вище:

Anton

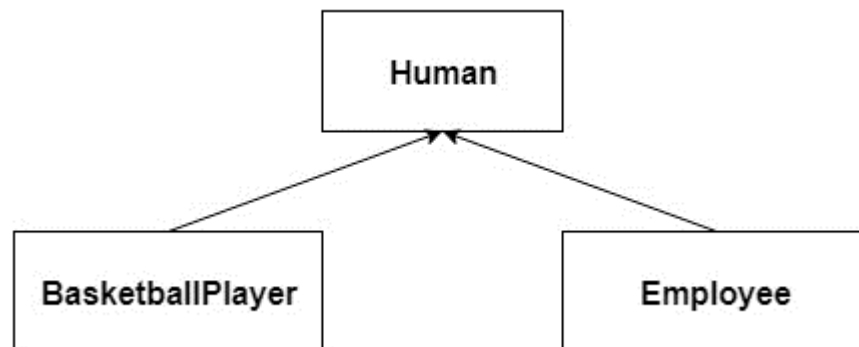
Це працює, тому що **anton** є об'єктом класу **BasketballPlayer**, а всі об'єкти класу **BasketballPlayer** мають змінну-член **m_name** і метод **getName ()**, успадковані від класу **Human**.

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 }

```

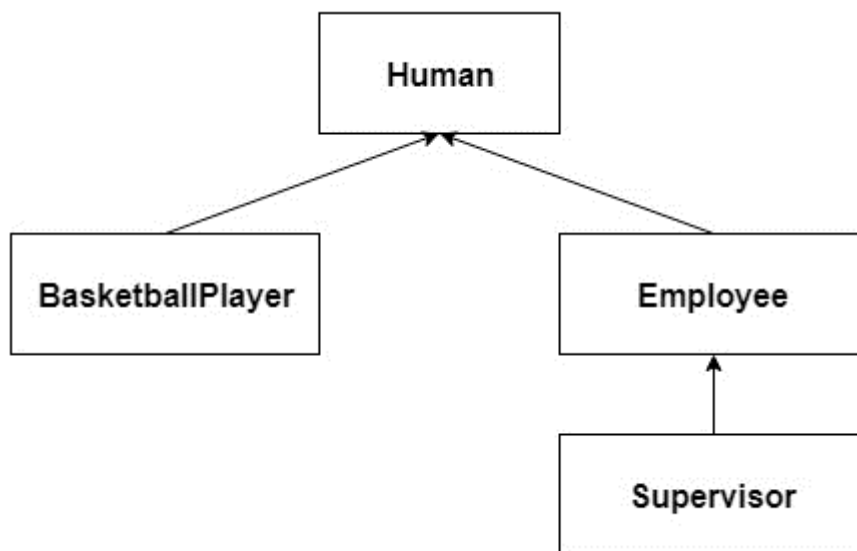
Результат виконання програми вище:

```
Ivan: 350
```

Можна наслідувати від класу, який сам успадковує від іншого класу. При цьому нічого примітного або чого-небудь особливого не відбувається - все

аналогічно тому, що ми розглянули вище. Наприклад, напишемо клас **Supervisor**. Супервайзер - це Працівник, який «є» Людиною. Ми вже написали клас **Employee**, тому будемо його використовувати в якості батьківського 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

```


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 };
```

C++ забороняє дочірнім класами формувати успадковані змінні-члени батьківського класу в списку ініціалізації свого конструктора. Іншими словами, значення змінної може бути поставлено лише в списку ініціалізації конструктора, що належить до того ж класу, що і змінна-член.

Обмежуючи ініціалізацію змінних конструктором класу, до якого належать ці змінні, C++ гарантує, що всі змінні будуть ініційовані тільки один раз.

Кінцевим результатом виконання коду вище є помилка, так як **m_id** успадкований від класу **Parent**, а тільки неуспадковані змінні-члени можуть бути змінені в списку ініціалізації конструктора класу **Child**.

Однак успадковані змінні можуть як і раніше змінювати свої значення в тілі конструктора через операцію присвоювання. Отже, новачки часто намагаються Зробити наступне:

```
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 };
```

Хоча це дійсно спрацює в цьому випадку, але це не спрацює, якщо **m_id** буде константою. Це також неефективно, тому що для **m_id** привласнюють значення двічі: перший раз в списку ініціалізації конструктора класу **Parent**, а потім - в тілі конструктора класу **Child**. І, нарешті, що, якщо класу **Parent** необхідний доступ до цього значення під час ініціалізації?

Отже, як правильно формувати **m_id** при створенні об'єкта класу **Child**?

У всіх наших прикладах, при створенні об'єкта класу **Child**, викликався конструктор за замовчуванням класу **Parent**. Чому так? Тому що ми не вказували інакше!

C++ надає нам можливість явно вибрати конструктор класу **Parent** для виконання ініціалізації частини **Parent**! Для цього необхідно просто додати виклик потрібного конструктора в список ініціалізації конструктора дочірнього класу:

```
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), // вызывается конструктор Parent(int) со значением id!
8           m_value(value)
9     {
10    }
11
12     double getValue() const { return m_value; }
13 };
```

Тепер при виконанні наступного коду:

```
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 }
```

Конструктор **Parent(int)** буде використовуватися для ініціалізації **m_id** значенням 7, а конструктор дочірнього класу буде використовуватися для ініціалізації **m_value** значенням 1.5! Разом, результат виконання програми:

```
ID: 7
Value: 1.5
```

Тепер, коли ми знаємо, що про ініціалізації членів батьківського класу, немає ніякої необхідності зберігати наші змінні-члени відкритими. Ми зробимо їх **private**, як і повинно бути.

Нагадування. Доступ до членів **public** відкритий для всіх. Доступ до членів **private** відкритий тільки для інших членів цього ж класу. Зверніть увагу, це означає, що дочірні класи не можуть безпосередньо звертатися до закритих членам батьківського класу! Дочірнім класами потрібно використовувати **геттери** і **сеттери** для доступу до цих членам.

```
1  #include <iostream>
2
3  class Parent
4  {
5  private: // наш m_id теперь закритий
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 закритий
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 }
```

У коді робимо **m_id** і **m_value** закритими. Для їх ініціалізації використовуються відповідні конструктори, а для доступу - відкриті функції доступу (геттери). Результат виконання програми вище:

ID: 7

Value: 1.5

Розглянемо ще пару класів, з якими ми працювали раніше:

```
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 членов name и age
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 }
```

Результат виконання програми вище:

```
Anton Ivanovuch
45
310
```

Варто відзначити, що конструктори дочірнього класу можуть викликати конструктори тільки того батьківського класу, від якого вони безпосередньо успадковують. При знищенні дочірнього класу, кожен **деструктор** викликається в зворотному порядку побудови класів.

При ініціалізації об'єктів дочірнього класу, конструктор дочірнього класу відповідає за те, який конструктор батьківського класу викликати. Якщо цей конструктор явно не вказано, то викликається конструктор за замовчуванням батьківського класу. Якщо ж компілятор не може знайти конструктор за замовчуванням батьківського класу (або цей конструктор не може бути створений автоматично), то компілятор видасть помилку.

Спадкування і специфікатор доступу **protected**

У C ++ є третій специфікатор доступу, який корисний тільки в контексті успадкування. Специфікатор доступу **protected** відкриває доступ до членів класу дружнім і дочірнім класами. Доступ до члена **protected** поза тілом класу закритий.

```
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 дру
7 protected:
8     int m_protected; // доступ к этому члену открыт для других членов класса Parent, дружественных
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 клас
17         m_private = 2; // запрещено: доступ к закрытым членам родительского класса из дочернего клас
18         m_protected = 3; // разрешено: доступ к защищённым членам родительского класса из дочернего
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 }
```

Можемо бачити, що член **m_protected** класу **Parent** безпосередньо доступний дочірньому класу **Child**, але доступ до нього для членів ззовні - закритий.

Використання специфікатору доступу **protected** найбільш корисно, коли ви будете наслідувати тільки свої ж класи і кількість дочірніх класів буде невелике.

Таким чином, якщо ви внесете зміни в реалізацію батьківського класу, і вам знадобиться оновити всі дочірні класи, то ви зможете зробити ці оновлення самі і це не займе багато часу (так як дочірніх класів буде небагато).

Типи наслідувань. Доступ до членів

Існує три типи наслідувань класів:

public;
private;
protected .

Для визначення типу успадкування потрібно просто вказати потрібне ключове слово біля успадкованого класу:

```
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 };
```

Якщо ви самі не визначили тип спадкування, то в C ++ за замовчуванням буде обраний тип спадкування **private** (аналогічно членам класу, які за умовчанням є **private**, якщо не вказано інакше).

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 друзі

- 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 <code>public</code> в дочірньому класі	Специфікатор доступу при спадкуванні типу <code>private</code> в дочірньому класі	Специфікатор доступу при спадкуванні типу <code>protected</code> в дочірньому класі
<code>public</code>	<code>public</code>	<code>private</code>	<code>protected</code>
<code>private</code>	недоступний	недоступний	недоступний
<code>protected</code>	<code>protected</code>	<code>private</code>	<code>protected</code>