Лаб. 5. Множинне успадкування. Проблема ромба у C++. Віртуальне успадкування.

Мета: Проробити на практиці можливі варіанти множинного успадкування та методи уникнення проблем із ним.

Завдання: До створеної ієрархії класів у лаб. 4. добавити класи таким чином, щоб отримати множинне успадкування. Переробити класи так, щоб отримати ромбовидне успадкування. Класи мають мати окрім конструкторів і звичайних методів ще й деструктори. Навчитись керувати порядком виклику конструкторів та деструкторів класів.

1. Приклад діаграми класів

Найпростіший набір класів, який реалізує множинне успадкування може виглядати наступним чином:

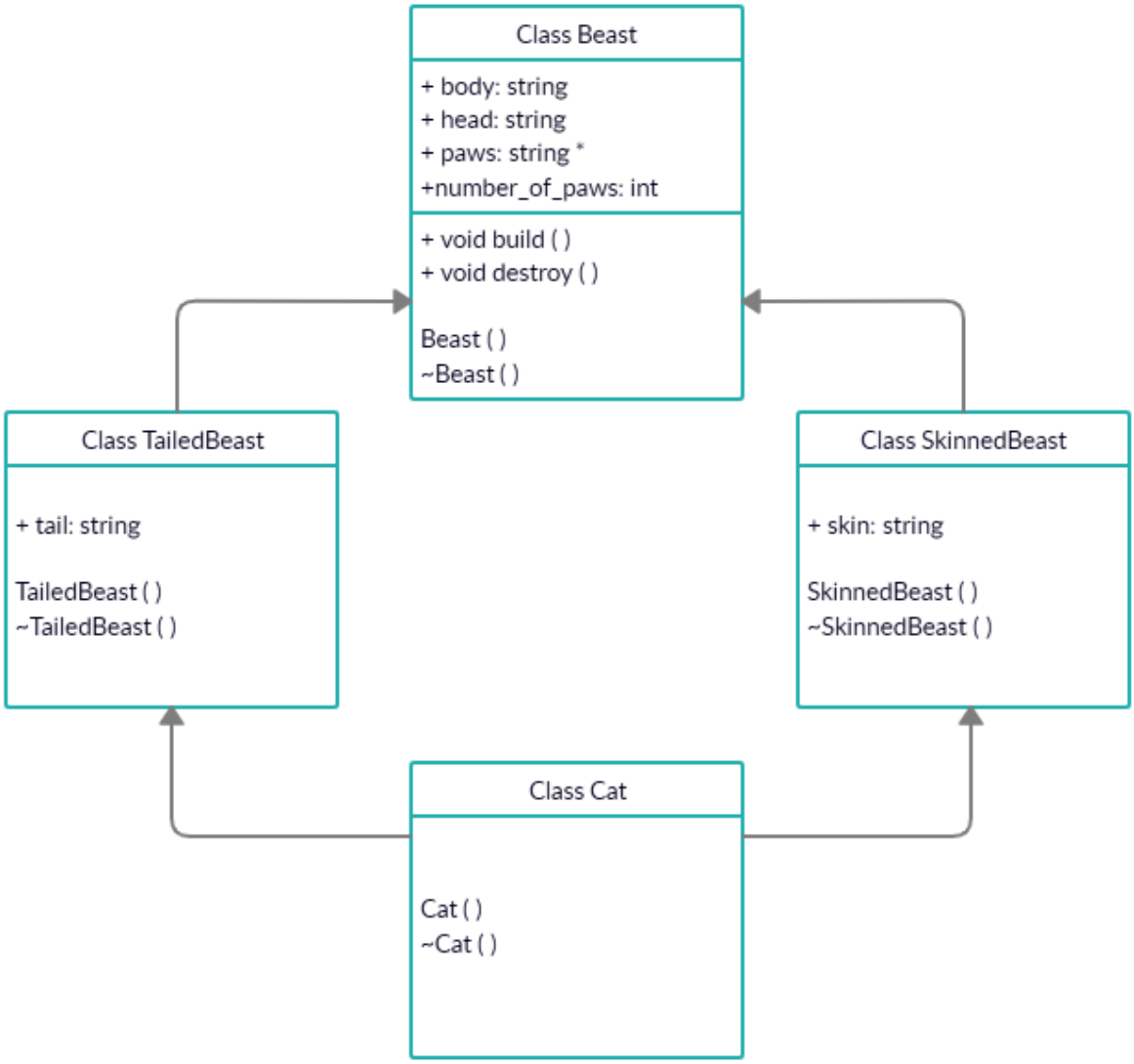

Рис. 1. Діаграма класів.

2. Основні класи

Для початку, у нас є клас `Beast` у якому створюється (та знищується) наш звір.

Цей клас має приватні поля, одне з яких має тип вказівника:

```
std::string body;
std::string head;
std::string *paws;
int number_of_paws;
```

Та два приватні методи

```
void build ()
{
  std::cout << "Створено:\n" << body << "\n" << head << "\nлапи \n" ;
  for (int i = 0; i < number_of_paws; ++i)  std::cout << paws[i] << "\n";
};
void destroy ()
{
  std::cout << "Знищено:\n" << body << "\n" << head << "\nлапи \n" ;
  for (int i = 0; i < number_of_paws; ++i)  std::cout << paws[i] << "\n";
};
```

Конструктор та деструктор мають модифікатор public:

```
public:
 Beast()
  {
    body = "тіло";
    head = "голова";
    number_of_paws = 4;
    paws = new std::string [number_of_paws];
    paws[0] = "передня права";
    paws[1] = "передня ліва";
    paws[2] = "задня права";
    paws[3] = "задня ліва";
    //std::cout << "Beast";
    build();
  };
  ~Beast()
  {
    //std::cout << "beast;
    destroy();
    delete[] paws;
  };
};
```

Треба зауважити, що масив стрічок `paws` був створений оператором `new` у конструкторі, його треба знищити у деструкторі. Тут ми використаємо оператор `delete[]`, бо звільняємо пам'ять від масиву об’єктів.

Якщо викликати цей клас

```
int main() {
  Beast beast;
}
```

Отримаємо наступний вивід на екран

```
Створено:
тіло
голова
лапи
передня права
передня ліва
задня права
задня ліва
Знищено:
тіло
голова
лапи
передня права
передня ліва
задня права
задня ліва
```

Тобто при створенні змінної типу Beast був викликаний конструктор, коли ж закінчилось виконання коду області видимості цієї змінної, був викликаний деструктор. Перевіримо це, добавимо стрічку після створення екземпляру нашого класу:

```
int main() {
  Beast beast;
  std::cout << "Кіт живе: Мяв!)\n";
}
```

Вивід зміниться на наступний:

```
Створено:
тіло
голова
лапи
передня права
передня ліва
задня права
задня ліва
Кіт живе: Мяв!)
Знищено:
тіло
голова
лапи
передня права
передня ліва
задня права
задня ліва
```

Бачимо, що виведена стрічка, як і передбачалося, розмістилася після конструктора і перед деструктором.

Створено ще два класи-нащадки від `Beast`:

```
class TailedBeast : public Beast
```

та

```
class SkinnedBeast : public Beast
```

.

Кожен з цих класів має одне поле, конструктор та деструктор.

Приклад виводу на екран при створенні екземпляру першого класу буде наступним:

```
Створено:
тіло
голова
лапи
передня права
передня ліва
задня права
задня ліва
добавлено хвіст
Кіт живе: Мяв!)
видалено хвіст
Знищено:
тіло
голова
лапи
передня права
передня ліва
задня права
задня ліва
```

3. Проблема ромба

Для наглядності спростимо поки вивід – залишимо вивід лише у конструкторах (з великої літери) та деструкторах (з малої літери): закоментовані стрічки у коді. Тоді після такого ж виклику, як і у попередньому випадку

```
int main() {
  TailedBeast beast;
  std::cout << "Кіт живе: Мяв!) ";
}
```

отримаємо:

```
Beast Tail Кіт живе: Мяв!) tail beast
```

Тобто спрацював конструктор батьківського класу, потім конструктор викликаного класу, спрацював код після створення об’єкту. Коли починає знищуватись екземпляр класу, викликається спочатку деструктор дочірнього класу, потім батьківського.

Така поведінка характерна для усіх об’єктно-орієнтованих мов програмування і не залежить від довжини ланцюга успадкування. Що ж буде, якщо наш клас буде успадковуватись від двох (чи більше) класів, у яких один батьківський клас (така структура, як на схемі Рис. 1)? За якою”доріжкою”

будуть викликатись конструктори та деструктори? Ці питання і є основою так званої проблеми ромба, або Diamond problem.

Насправді, у різних мовах програмування ця проблема вирішується по різному, наприклад, у мовах високого рівня, як Python часто будуються графи успадкування класів (щось схоже на Рис. 1) та робить повний обхід графу з викликом батьківського класу лише один раз (останній у гілці). Подивимось, що буде у C++: створимо екземпляр класу `Cat`.

```
int main() {
  Cat beast;
  std::cout << "Кіт живе: Мяв!) ";
}
```

Отримаємо таке:

```
Beast Tail Beast Skin Cat Кіт живе: Мяв!) cat skin beast tail beast
```

Бачимо, що конструктор та деструктор батьківського класу викликався двічі. Це характерно для мови C++, компілятор ігнорує проблему ромбу і покладає рішення, як вирішувати задачу, коли треба мати один екземпляр батьківського класу, на програміста. Нащо це потрібно? Якщо ми матимемо хоча б один метод батьківського класу з модифікатором public, який буде успадковувались двома шляхами, виникне неоднозначність, який екземпляр методу суперкласу (від якого усі успадковуються) використовувати і компілятор покаже помилку.

Для керування успадкуванням використовують так зване віртуальне успадкування через ключове слово `virtual`. Віртуальним можна визначити метод (тоді його треба обов'язково перевантажити у класі нащадку, тобто це заготовка, яка нічого не робить, але повинна бути описана далі, фактично є лише прототипом функції). Зв'язування таких функцій проходить на етапі виконання коду, а не компіляції. Це породжує механізми керування кодом, які вивчатимо пізніше, коли говоритимемо про абстрактні класи.

Ромбовидне ж успадкування у C++ виникає тоді, коли мінімум два класи нащадки віртуально успадкують вихідний клас – суперклас.

Подивимось, як це працює:

Добавимо ключове слово `virtual` при успадкуванні обидвома класами

```
class TailedBeast : virtual public Beast
```

та

```
class SkinnedBeast : virtual public Beast
```

Тоді отримаєм при запуску програми наступне

```
Beast Tail Skin Cat Кіт живе: Мяв!) cat skin tail beast
```

Тобто конструктор і деструктор батьківського класу викликаний один раз, а це означає, що існує один єдиний його екземпляр, неоднозначності не буде.

Зробимо віртуальним тільки перше успадкування, отримаємо наступне:

```
Beast Tail Beast Skin Cat Кіт живе: Мяв!) cat skin beast tail beast
```

У цьому випадку створено два екземпляри суперкласу: один звичайний та один віртуальний. Аналогічний результат буде і у випадку віртуального успадкування другим класом, лише зміниться порядок створення віртуального екземпляра:

```
Beast Beast Tail Skin Cat Кіт живе: Мяв!) cat skin tail beast beast
```

Така поведінка пояснюється пізнім з'єднанням (на етапі виконання) віртуальних функцій класів.

Повернемо обидвом шляхам успадкування слово `virtual` та наш початковий вивід у консоль. Отримаємо:

```
Створено:
тіло
голова
лапи
передня права
передня ліва
задня права
задня ліва
добавлено хвіст
добавлено шкіра
добавлено няшність
У нас є КІТ!

Кіт живе: Мяв!)

Зденекотення:
видалено шкіра
видалено хвіст
Знищено:
тіло
голова
лапи
передня права
передня ліва
задня права
задня ліва
```

Додаткове завдання:

Реалізувати клас, який успадковується від трьох батьківських класів. Дослідити, цьому випадку порядок викликів конструкторів та деструкторів, як на це впливає віртуальне успадкування?

Додаток – лістінг коду

```
[1] #include <iostream>
[2]
[3] class Beast
```

```
 [4] {
 [5] std::string body;
 [6] std::string head;
 [7] std::string *paws;
 [8] int number_of_paws;
 [9] void build ()
[10] {
[11] std::cout << "Створено:\n" << body << "\n" << head << "\nлапи \n" ;
[12] for (int i = 0; i < number_of_paws; ++i)  std::cout << paws[i] <<
     "\n";
[13] };
[14] void destroy ()
[15] {
[16] std::cout << "Знищено:\n" << body << "\n" << head << "\nлапи \n" ;
[17] for (int i = 0; i < number_of_paws; ++i)  std::cout << paws[i] <<
     "\n";
[18] };
[19] public:
[20] Beast()
[21] {
[22] body = "тіло";
[23] head = "голова";
[24] number_of_paws = 4;
[25] paws = new std::string [number_of_paws];
[26] paws[0] = "передня права";
[27] paws[1] = "передня ліва";
[28] paws[2] = "задня права";
[29] paws[3] = "задня ліва";
[30] //std::cout << "Beast ";
[31] build();
[32] };
[33] ~Beast()
[34] {
[35] //std::cout << "beast ";
[36] destroy();
[37] delete[] paws;
[38] };
[39] };
[40]
[41] class TailedBeast : virtual  public Beast
[42] {
[43] std::string tail;
[44] public:
[45] TailedBeast()
[46] {
[47] tail = "хвіст";
[48] std::cout << "добавлено " << tail << "\n";
[49] //std::cout << "Tail ";
[50] };
[51] ~TailedBeast()
[52] {
[53] std::cout << "видалено " << tail << "\n";
[54] //std::cout << "tail ";
```

```
[55] };
[56] };
[57]
[58] class SkinnedBeast : virtual public Beast
[59] {
[60] std::string skin;
[61] public:
[62] SkinnedBeast()
[63] {
[64] skin = "шкіра";
[65] std::cout << "добавлено " << skin << "\n";
[66] //std::cout << "Skin ";
[67] };
[68] ~SkinnedBeast()
[69] {
[70] std::cout << "видалено " << skin << "\n";
[71] //std::cout << "skin ";
[72] };
[73] };
[74]
[75] class Cat : TailedBeast, SkinnedBeast
[76] {
[77] public:
[78] Cat()
[79] {
[80] //std::cout << "Cat ";
[81] std::cout << "добавлено няшність\nУ нас є КІТ!\n\n";
[82] }
[83] ~Cat()
[84] {
[85] //std::cout << "cat ";
[86] std::cout << "Зденекотення:\n";
[87] }
[88] };
[89]
[90] int main() {
[91] Cat beast;
[92] std::cout << "Кіт живе: Мяв!)\n\n";
[93] }
```