Лекція 9

Успадкування

У цій лекції...

Принципи інкапсуляції, приховання інформації й абстракції даних забезпечують захист об'єктів від несанкціонованого втручання. Однак це лише одна зі сторін об'єктно-орієнтованого програмування. Вона дозволяє перейти від індивідуального будівництва програм до великоблочного, але в той же час сковує можливості програміста. Дійсно, одержавши готовий модуль із дружелюбним і зрозумілим інтерфейсом, ви можете лише пристосувати його до своїх потреб. Якщо ж вам знадобиться розширити його функціональні можливості, прийдеться знову звернутися до розроблювача з проханням модифікувати чи переписати модуль самому. Ані те, ані інше не занадто вигідно. Набагато ефективніше надати програмісту можливість використовувати готовий модуль як базу для власних розробок, не переробляючи його код. Цей підхід називається *повторним використанням коду*.

Крім того, досить часто в багатьох додатках інформація уже організована у виді визначеної структури, чи ієрархії. Систему, що має бути змоделюваною, іноді можна подати у виді об'єктів, зв'язаних особливими відношеннями, які можна було б назвати *подібністю*. Таку подібність можна знайти навіть у математичних задачах, де коло об'єктів сильно обмежене. Наприклад, вектор можна вважати матрицею, що складається з одного стовпця і декількох рядків, а число — вектором, що складається з одного рядка й одного стовпця. При розв'язанні математичних задач це не має ніякого значення, однак у процесі програмування цю обставину можна використовувати для підвищення ефективності програми.

В основі механізму, що дозволяє створювати ієрархії класів, лежить принцип *успадкування*. Клас, що лежить в основі ієрархії, називається *базовим*. Класи, що успадковують властивості базового класу, називаються *похідними*. Похідні класи, у свою чергу, можуть бути базовими стосовно своїх спадкоємців, що в результаті приводить до ланцюжка успадкування. Процес утворення похідного класу на основі базового називається *виводом* класу. З одного базового класу можна вивести декілька похідних. Крім того, похідний клас може бути спадкоємцем декількох базових класів.

Успадкування буває *одиночним* і *множинним*. При одиночному успадкуванні в кожного похідного класу є лише *один* базовий клас, а при множинному — *декілька*.

Розглянемо механізм успадкування ближче.

9.1. ПОХІДНІ КЛАСИ

Усі члени базового класу автоматично стають членами похідного. Керуючись оголошенням похідного класу, компілятор ніби збирає його з різних частин — спочатку він бере усі властивості базового класу, а потім додає до них нові функціональні можливості похідного.

Для цього використовується наступна синтаксична конструкція.

```
class ім'я_похідного_класу:специфікатор_доступу ім'я_базового_класу
{
  // тіло класу
};
```

Хоча всі члени базового класу автоматично стають членами похідного класу, однак доступ до цих членів визначається видом успадкування. У залежності від специфікатора доступу, зазначеного при оголошенні похідного класу, розрізняють *відкрите*, *закрите* і *захищене* успадкування. За замовчуванням використовується закрите успадкування (специфікатор `private`).

9.1.1. Відкрите успадкування

Розглянемо приклад відкритого успадкування.

Використання відкритого успадкування

```
#include <stdio.h>

class TInner
{
public:
   int i;
   TInner(int n):i(n){ printf("Ctor TInner\n");}
   TInner(TInner& x){ *this = x; printf("Copy ctor TInner\n");}
   ~TInner(){printf("Dtor TInner\n");}
};
```

```
class TBase
{
public:
   double a;
   TInner b;
   TBase():b(10),a(5.0){printf("Ctor TBase\n");}
   ~TBase(){printf("Dtor TBase\n");}
    TBase(TBase& x):b(10){ *this = x; printf("Copy ctor TBase\n");}
   void printBase() {printf("TBase::TInner::i = %d a = %lf\n",b.i, a);}
};

class TDerived: public TBase
{
public:
   char c;
   TDerived():c('Z'){printf("Ctor TDerived\n");}
   TDerived(TDerived& x){ *this = x; printf("Copy ctor TDerived\n");}
   ~TDerived(){printf("Dtor TDerived\n");}
   void printDerived() {printf("TDerived::TInner::i = %d c = %c\n",b.i,c);}
};

class TDerived2: public TDerived
{
public:
   float f;
   TDerived2():f(10.0){printf("Ctor TDerived2\n");}
   ~TDerived2(){printf("Dtor TDerived2\n");}
   void printDerived2() {printf("TDerived2::TInner::i = %d f = %lf\n",b.i,f);}
};

int main()
{
   TBase first;
   TDerived second;
   TDerived2 third;
   first.printBase();
   second.printDerived();
   third.printDerived2();
   printf("Sizeof(TBase) = %d\n", sizeof(TInner));
   printf("Sizeof(TBase) = %d\n", sizeof(TBase));
   printf("Sizeof(TDerived) = %d\n", sizeof(TDerived));
   printf("Sizeof(Tderived2) = %d\n", sizeof(TDerived2));

   return 0;
}
```

У цій програмі описано чотири класи: `TInner`, `TBase`, `TDerived`, `TDerived2`. Клас `TBase` є *контейнерним*; він містить об'єкт класу `TInner`. Крім того, клас `TBase` є базовим стосовно класу `TDerived`. У свою чергу, клас `TDerived` є базовим стосовно класу `TDerived2`.

У функції `main()` створюється три об'єкти — `first`, `second` і `third` — класів `TBase`, `TDerived` і `TDerived2` відповідно. Потім викликаються функції, що виводять на екран вміст цих об'єктів.

Проаналізуємо результати роботи цієї програми (номери не є частиною виводу — вони додані для зручності). Програма виконувалася під керуванням 32-розрядної операційної системи.

```
1.  Ctor Inner
2.  Ctor TBase
3.  Ctor Inner
4.  Ctor TBase
5.  Ctor TDerived
6.  Ctor Inner
7.  Ctor TBase
9.  Ctor TDerived
9.  Ctor TDerived2
10. TBase::TInner::i = 10 a = 5.000000
11. TDerived::TInner::i = 10 c = Z
12. TDerived2::TInner::i = 10 f = 10.000000
13. Sizeof(TBase) = 4
14. Sizeof(TBase) = 16
```

```
15. Sizeof(TDerived) = 24
16. Sizeof(TDerived2) = 32
17. Dtor TDerived2
19. Dtor TDerived
19. Dtor TBase
20. Dtor TInner
21. Dtor TDerived
22. Dtor TBase
23. Dtor TInner
24. Dtor TBase
25. Dtor TInner
```

У рядку 1 ми бачимо повідомлення конструктора класу `TInner`, що генерується при створенні об'єкта `first`. Повідомлення про створення об'єкта `first` міститься в другому рядку.

Рядки 3–5 містять повідомлення від конструкторів, що створюють об'єкт `second` — спочатку найбільше глибоко вкладений об'єкт класу `TInner`, потім об'єкт контейнерного класу `TBase` і, нарешті, об'єкт похідного класу `TDerived`.

Аналогічно в рядках 6–9 наведені повідомлення від конструкторів `TInner`, `TBase`, `TDerived` і `TDerived2`, що послідовно викликаються при створенні об'єкта `third`.

Рядки 10–12 являють собою результат роботи функцій-членів `printBase()`, `printDerived()` і `PrintDerived2()`.

Рядки 13–16 демонструють розміри класів. Як бачимо, хоча члени класу не копіюються (інакше ми одержали би повідомлення від конструктора копіювання), розміри класу враховують розміри успадкованих полів. Крім того, необхідно помітити, що ці розміри обчислюються з урахуванням вирівнювання, тому, наприклад, розмір класу `TBase` дорівнює 16 байт, хоча розмір поля типу `double` дорівнює 9 байт, а розмір класу `TInner` — 4 байт.

У рядках 17–25 містяться повідомлення від деструкторів. Вони наочно показують, що деструктори об'єктів викликаються в зворотному порядку.

І головне: класи `TDerived` і `TDerived2` створені за допомогою відкритого копіювання. Це означає, що всі поля их базових класів *зберігають* свій рівень доступу.

Відзначимо, що в нашій програмі всі члени класів поміщені у відкритих розділах. Як правило, усе обстоїть навпаки — дані *необхідно* ховати. Якби ми підкорилися цій вимозі і помістили змінні `a` і `b` у закритий розділ класу `TBase`, вони були б сховані від класу `TDerived`. Інакше кажучи, хоча закриті члени класу `TBase` успадковуються класом `TDerived`, функції-члени похідного класу не мають прямого доступу до закритих полів — інкапсуляція продовжує діяти! Об'єкти класу `TDerived` одержують поля класу `TBase` "в упакуванні", і звертатися до них можуть лише через функції відкритого інтерфейсу. Таким чином, закриті члени базового класу не рівноцінні закритим членам похідного класу. Вони доступні лише членам базового класу. Що ж у такому випадку означає вираження *члени базового класу автоматично стають членами похідного класу*? Повною мірою це відноситься лише до відкритих і захищених об'єктів, інакше успадкування вступило б у протиріччя з принципом приховання інформації.

До речі, відкрите успадкування дозволяє обійтися без створення об'єктів класів `TBase` і `TDerived`. Звернутися до функцій-членів базових класів можна безпосередньо через об'єкт похідного класу `TDerived2`.

```
   TDerived2 obj;
  obj.printBase();
  obj.printDerived();
  obj.printDerived2();
```

9.1.2. Закрите успадкування

Припустимо, що ми застосували закрите успадкування. У цьому випадку усі відкриті і захищені поля базового класу стали б закритими членами похідного класу. Звернутися до цих полів прямо тепер неможливо — клас як би "запечатується".

Применим закрите успадкування при висновку класу `TDerived`.

```
class TDerived: private TBase     // Закрите успадкування
{
public:
   char c;
   TDerived():c('Z'){printf("Ctor TDerived\n");}
   TDerived(TDerived& x):b(10){ *this = x; printf("Copy ctor TDerived\n");}
   ~TDerived(){printf("Dtor TDerived\n");}
   void printDerived() {printf("TDerived::TInner::i = %d c = %c\n",b.i,c);}
};
```

На його відношення до полів класу `TBase` це ніяк не впливає. Однак тепер визначення класу `TfDerived2` стало неправильним. Його функція-член `printDerived2()` не має доступу до поля `b`, що стало закритим.

Існує два способи вирішення цієї проблеми. Перший — переписати класи `TDerived` і `TDerived2`, відкривши шлях до поля `b`. Другий — відкрити поле `b` насильно, використовувавши *уточнення*.

```
  class TDerived: private TBase
{
public:
   char c;
   TBase::b;  // Уточнення
   TDerived():c('Z'){printf("Ctor TDerived\n");}
   TDerived(TDerived& x){*this = x; printf("Copy ctor TDerived");}
   ~TDerived(){printf("Dtor TDerived\n");}
   void printDerived() {printf("TDerived::TInner::i = %d c = %c\n",b.i,c);}
};
```

Виділений рядок називається уточненням поля `b`. Незважаючи на те що відповідно до правил закритого успадкування цей член вважається закритим, він повторно з'являється у відкритому розділі класу `TDerived` і стає відкритим.

9.1.3. Захищене успадкування

Ми уже не раз згадували специфікатор доступу `protected`, але лише тепер прийшла пора розкрити його зміст. Це пояснюється тим, що його відмінність від специфікатора `private` виявляється тільки при успадкуванні класів. В всем іншому специфікатори `protected` і `private` еквівалентні. При захищеному успадкуванні відкриті члени базового класу стають захищеними членами похідного класу. Вони видні лише функціям базового і похідного класів і є невидимими з інших точок програми. Таким чином, програма як і раніше має доступ до відкритих членів базового класу через його об'єкти, але втрачає доступ до цих полів, якщо доступ здійснюється через об'єкти похідного класу.

Уявимо собі, що клас `TDerived` оголошений захищеним.

```
  class TDerived: protected TBase
{
 public:
   char c;
   TDerived():c('Z'){printf("Ctor TDerived\n");}
   TDerived(TDerived& x){*this = x; printf("Copy ctor TDerived");}
   ~TDerived(){printf("Dtor TDerived\n");}
   void printDerived() {printf("TDerived::TInner::i = %d c = %c\n",b.i,c);}
};
```

Функція `printDerived()` як і раніше має доступ до поля `b.i`, однак прямого доступу до цього *відкритого* поля класу `TBase` через об'єкти класів `TDerived` і `TDerived2` тепер немає.

```
  int main()
{
   TBase first;
   TDerived second;
   TDerived2 third;
   printf("TBase::TInner::b = %d \n",first.b.i);
   printf("TDerived::TInner::b = %d \n",second.b.i); // Помилка! Нет доступу!
   printf("TDerived2::TInner::b = %d \n",third.b.i); // Помилка! Нет доступу!
   return 0;
}
```

9.1.4. Множинне успадкування

У похідного класу може бути декілька базових. У цьому випадку члени базових класів стають членами похідного.

```
  class ім'я_похідного_класу:
      специфікатор_доступу ім'я_базового_класу1,
      ...,
      специфікатор_доступу ім'я_базового_класса
{
  // тіло класу
};
```

Розглянемо наступну програму.

Використання множинного успадкування

```
  #include <stdio.h>

class TInner
{
public:
   int i;
```

```
    TInner(int n):i(n){printf("Ctor Inner\n");}
    TInner(TInner& x){ *this = x; printf("Copy ctor TInner\n");}
    ~TInner(){printf("Dtor TInner\n");}
 };

 class TBase1
 {
 public:
    double a;
    TInner b;
    TBase1():b(10),a(5.0){printf("Ctor TBase1\n");}
    TBase1(TBase1& x):b(10){ *this = x; printf("Copy ctor TBase1");}
    ~TBase1(){printf("Dtor TBase1\n");}
    void printBase() {printf("TBase1::TInner::i = %d a = %lf\n",b.i, a);}
 };

 class TBase2
{
 public:
    char c;
    TBase2():c('Z'){printf("Ctor TBase2\n");}
    TBase2(TBase2& x){*this = x; printf("Copy ctor TBase2");}
    ~TBase2(){printf("Dtor TBase2\n");}
    void printBase2() {printf("TBase2::c = c = %c\n",c);}
 };

 class TDerived: public TBase1, public TBase2
 {
 public:
    float f;
    TDerived():f(10.0){printf("Ctor TDerived\n");}
    ~TDerived(){printf("Dtor TDerived\n");}
    void printDerived() {printf("TDerived::%lf\n",f);}
 };

 int main()
 {
    TDerived obj;
    printf("TBase1::a = %lf \n",obj.a);
    printf("TBase1::b = %d \n",obj.b.i);
    printf("TBase2::c = %c \n",obj.c);
    printf("TDerived::f = %lf \n",obj.f);
    return 0;
 }
```

У класу `TDerived2` є два базових класи, кожний з який є відкритим. Отже, об'єкт класу `TDerived2` містить усі поля, що належать класам `TBase1` і `TBase2`, причому усі вони відкриті.

Програма виводить на екран наступні повідомлення.

```
1.   Ctor Inner
2.   Ctor TBase1
3.   Ctor TBase2
4.   Ctor TDerived
5.   TBase1::a = 5.000000
6.   TBase1::b = 10
7.   TBase2::c = Z
8.   TDerived2::f = 10.000000
9.   Dtor TDerived
10.  Dtor TBase2
11.  Dtor TBase1
12.  Dtor TInner
```

Перші чотири рядки являють собою повідомлення від конструкторів у порядку створення об'єктів — внутрішній, два базових і похідний. Потім функція `printf()` виводить на екран поля об'єкта `obj`, що він успадкував від класів `TBase1` і `TBase2`, а також його власне поле. Наприкінці програми викликаються деструктори, що знищують об'єкти в зворотному порядку.

9.1.5. Конструктори і деструктори

Як ми бачили, при створенні об'єктів похідного класу необхідно спочатку створити проміжний об'єкт базового класу. Це зрозуміло — об'єкт похідного класу являє собою модифікований об'єкт базового класу. Він не може виникнути з повітря. Отже, спочатку викликається конструктор базового класу, а потім — конструктор похідного класу. Якщо базових класів декілька, вони викликаються в порядку их перерахування в списку наслідуваних класів.

Деструктори викликаються в зворотному порядку.

Зверніть увагу на оголошення конструктора похідного класу.

```
ім'я_похідного_класу:
   ім'я_базового класу1(параметри), ..., ім'я_базового класса(параметри)
{
   тіло конструктора
}
```

Наприклад, у приведеній вище програмі конструктор похідного класу визначається в такий спосіб.

```
TDerived():TBase1(),TBase2(){f=15.0; printf("Ctor TDerived\n");}
```

Утім, конструктори базових класів можуть викликатися неявно.

```
TDerived():f(15.0) { printf("Ctor TDerived\n");}
```

Варто мати на увазі, що конструктори базових класів у списку не можна перемежовувати списками ініціалізації. Наприклад, що випливає конструкція непрацездатна.

```
TDerived():TBase1(),TBase2():f(15.0) {printf("Ctor TDerived\n");}
```

Дотепер ми вивчали класи, конструктори яких не мали параметрів. Розглянемо тепер конструктори з параметрами.

Застосування конструкторів з параметрами

```
#include <stdio.h>

class TInner
{
public:
   int i;
   TInner(int n):i(n){printf("Ctor Inner\n");}
   TInner(TInner& x){ *this = x; printf("Copy ctor TInner\n");}
   ~TInner(){printf("Dtor TInner\n");}
};

class TBase1
{
public:
   double a;
   TInner b;
   TBase1(int x, double y):b(x),a(y){printf("Ctor TBase1\n");}
   TBase1(TBase1& x):b(10){ *this = x; printf("Copy ctor TBase1");}
   ~TBase1(){printf("Dtor TBase1\n");}
   void printBase() {printf("TBase1::TInner::i = %d a = %lf\n",b.i, a);}
};

class TBase2
{
public:
   char c;
   TBase2(char x):c(x){printf("Ctor TBase2\n");}
   TBase2(TBase2& x){*this = x; printf("Copy ctor TBase2");}
   ~TBase2(){printf("Dtor TBase2\n");}
   void printBase2() {printf("TBase2::c = c = %c\n",c);}
};

class TDerived: public TBase1, public TBase2

{
public:
   float f;
   TDerived(float x):TBase1(20.0, 30),TBase2('X')
    {  f = x;printf("Ctor TDerived\n"); }
   ~TDerived(){printf("Dtor TDerived\n");}
   void printDerived2() {printf("TDerived2::%lf\n",f);}
};
```

```
int main()
{
   TDerived obj(15.0);
   printf("TBase1::a = %lf \n",obj.a);
   printf("TBase1::b = %d \n",obj.b.i);
   printf("TBase2::c = %c \n",obj.c);
   printf("TDerived2::f = %lf \n",obj.f);
   return 0;
}
```

Результат приведений нижче.

```
Ctor Inner
Ctor TBase1
Ctor TBase2
Ctor TDerived
TBase1::a = 30.000000
TBase1::b = 20
TBase2::c = X
TDerived2::f = 15.000000
Dtor TDerived
Dtor TBase2
Dtor TBase1
Dtor TInner
```

Параметри конструктора базового класу можна передавати конструкторам базових класів.

Передача параметрів конструкторам базових класів

```
#include <stdio.h>

class TInner
{
public:
    int i;
    TInner(int n):i(n){printf("Ctor Inner\n");}
    TInner(TInner& x){ *this = x; printf("Copy ctor TInner\n");}
    ~TInner(){printf("Dtor TInner\n");}
};

class TBase1
{
public:
    double a;
    TInner b;
    TBase1(int x, double y):b(x),a(y){printf("Ctor TBase1\n");}
    TBase1(TBase1& x):b(10){ *this = x; printf("Copy ctor TBase1");}
    ~TBase1(){printf("Dtor TBase1\n");}
    void printBase() {printf("TBase1::TInner::i = %d a = %lf\n",b.i, a);}
};

class TBase2
{
public:
    char c;
    TBase2(char x):c(x){printf("Ctor TBase2\n");}
    TBase2(TBase2& x){*this = x; printf("Copy ctor TBase2");}
    ~TBase2(){printf("Dtor TBase2\n");}
    void printBase2() {printf("TBase2::c = c = %c\n",c);}
};

class TDerived: public TBase1, public TBase2
{
public:
    float f;
    TDerived(int x, double y, char z, float w):TBase1(x, y),TBase2('z')
    { f = w;printf("Ctor TDerived\n"); }
    ~TDerived(){printf("Dtor TDerived\n");}
    void printDerived2() {printf("TDerived2::%lf\n",f);}
};
```

```
int main()
{
    TDerived obj(1.0, 2, 'Y', 3.0);
    printf("TBase1::a = %lf \n",obj.a);
    printf("TBase1::b = %d \n",obj.b.i);
    printf("TBase2::c = %c \n",obj.c);
    printf("TDerived2::f = %lf \n",obj.f);
    return 0;
}
```

9.1.6. Віртуальні базові класи

Множинне успадкування часте приводить до неоднозначностей. Як відомо, множинне успадкування і вказівники — це дві властивості мови C++, що зазнавали запеклої критики. У результаті цих дискусій була розроблена мова програмування, яка не підтримує ані вказівники, ані множинне успадкування, — мова Java.

Однак для вирішення неоднозначностей можна обійтися менш радикальними мірами. Розглянемо класичний варіант, що одержав назву брильянтового успадкування, — клас `TBase` є базовим для класів `TDerived1` і `TDerived2`, а ті, у свою чергу, — базовими для класу `TNext`. Відповідно до механізму успадкування всі члени класу `TBase` стають членами класів `TDerived1` і `TDerived2`, потім вони успадковуються класом `TNext`. Оскільки при першому успадкуванні (одиночному) члени класу `TBase` "роздвоюються", при множинному успадкуванні в класі `TNext` виникає неоднозначність — як бути з членами класу `TBase`?

Розглянемо приклад.

Діамантове успадкування

```
  #include <stdio.h>

 class TInner
 {
 public:
    int i;
    TInner(int n):i(n){printf("Ctor Inner\n");}
    TInner(TInner& x){ *this = x; printf("Copy ctor TInner\n");}
    ~TInner(){printf("Dtor TInner\n");}
 };

 class TBase
 {
 public:
    double a;
    TInner b;
    TBase(int x, double y):b(x),a(y){printf("Ctor TBase\n");}
    TBase(TBase& x):b(10){ *this = x; printf("Copy ctor TBase");}
    ~TBase(){printf("Dtor TBase\n");}
    void printBase() {printf("TBase::TInner::i = %d a = %lf\n",b.i, a);}
 };

 class TDerived1: public TBase
{
 public:
    char c;
    TDerived1(int x, int y, char z):TBase(x,y){c = z; printf("Ctor
TDerived1\n");}
    ~TDerived1(){printf("Dtor TDerived1\n");}
    void printDerived1() {printf("TDerived1::c = %c\n",c);}
 };

 class TDerived2: public TBase
{
 public:
    double d;
    TDerived2(int x, int y, double z):TBase(x,y){d = x; printf("Ctor
TDerived2\n");}
    ~TDerived2(){printf("Dtor TDerived2\n");}
    void printDerived2() {printf("TDerived2::c = c = %lf\n",d);}
 };

 class TNext:public TDerived1, public TDerived2
```

```
 {
 public:
    float f;
    TNext(int x, int y, char z, double w):TDerived1(x,y,z),TDerived2(x,y,w)
    {f = w;printf("Ctor TDerived\n");}
    ~TNext(){printf("Dtor TNext\n");}
    void printNext() {printf("Tnext::%lf\n",f);}
 };


 int main()
 {
    TNext obj(1.0, 2, 'Y', 3.0);
 //   printf("TBase::a = %lf \n",obj.a);        // Неоднозначність!
 //   printf("TDerived1::b = %d \n",obj.b.i);   // Неоднозначність!
    printf("TDerived2::c = %c \n",obj.c);
    printf("TNext::f = %lf \n",obj.f);

    return 0;
 }
```

Очевидно, що поля `a` і `b` попадають із класу `TBase` у клас `TNext` двома шляхами — через клас `TDerived1` і `TDerived2`. Отже, звертання `obj.a` і `obj.b.i` є неоднозначними.

Вирішити проблему можна двома шляхами. Наприклад, можна уточнити ім'я поля за допомогою оператора дозволу бачимості.

```
   printf("TBase::a = %lf \n",obj.TDerived1::a);
printf("TDerived1::b = %d \n",obj.TDerived1::b.i);
```

Однак залишається проблема, зв'язана з розміром класу. Хоча неоднозначність усунута, клас `TNext` як і раніше містить подвоєний набір змінних, успадкованих від класу `TBase`. Існує більш елегантний спосіб рішення цієї проблеми — *віртуальні базові класи*.

Для того щоб запобігти дублювання, у списку успадкування перед ім'ям базового класу варто вказати ключовому слову `virtual`. Правда, тепер необхідно самому передбачити конструктор за замовчуванням.

Віртуальний базовий клас

```
  #include <stdio.h>

 class TInner
 {
 public:
    int i;
    TInner(int n):i(n){printf("Ctor Inner\n");}
    TInner(TInner& x){ *this = x; printf("Copy ctor TInner\n");}
    ~TInner(){printf("Dtor TInner\n");}
 };

 class TBase
 {
 public:
    double a;
    TInner b;
    TBase():a(0),b(0){}
    TBase(int x, double y):b(x),a(y){printf("Ctor TBase\n");}
    TBase(TBase& x):b(10){ *this = x; printf("Copy ctor TBase");}
    ~TBase(){printf("Dtor TBase\n");}
    void printBase() {printf("TBase::TInner::i = %d a = %lf\n",b.i, a);}
 };

 class TDerived1: virtual public TBase
{
 public:
    char c;
    TDerived1(int x, int y, char z):TBase(x,y){c = z; printf("Ctor
TDerived1\n");}
    ~TDerived1(){printf("Dtor TDerived1\n");}
    void printDerived1() {printf("TDerived1::c = %c\n",c);}
 };
```

```
 class TDerived2: virtual public TBase
{
 public:
    double d;
    TDerived2(int x, int y, double z):TBase(x,y)
    { d = x; printf("Ctor TDerived2\n"); }
    ~TDerived2(){printf("Dtor TDerived2\n");}
    void printDerived2() {printf("TDerived2::c = c = %lf\n",d);}
 };

 class TNext:public TDerived1, public TDerived2
 {
 public:
    float f;
    TNext(int x, int y, char z, double w):TDerived1(x,y,z),TDerived2(x,y,w)
    {f = w;printf("Ctor TNext\n");}
    //TDerived(float x):f(x) { printf("Ctor TDerived\n");}
    ~TNext(){printf("Dtor TNext\n");}
    void printNext() {printf("TNext::%lf\n",f);}
 };

 int main()
 {
    TNext obj(1.0, 2, 'Y', 3.0);
    printf("TBase::a = %lf \n",obj.a);
    printf("TDerived1::b = %d \n",obj.b.i);
    printf("TDerived2::c = %c \n",obj.c);
    printf("TNext::f = %lf \n",obj.f);
    printf("Sizeof(TNext) = %d\n",sizeof(TNext));
    return 0;
 }
```

Результати роботи цієї програми приведені нижче.

```
  Ctor Inner
 Ctor TDerived1
 Ctor TDerived2
 Ctor TNext
 TBase::a = 0.000000
 TDerived1::b = 0
 TDerived2::c = Y
 TNext::f = 3.000000
 Sizeof(TNext) = 49
 Dtor TNext
 Dtor TDerived2
 Dtor TDerived1
 Dtor TBase
 Dtor TInner
```

Як бачимо, механізм віртуальних базових класів дозволяє не тільки усунути неоднозначність, але і взагалі запобігти дублювання наслідуваних змінних — розмір класу `TNext` тепер дорівнює не 56 байт, а лише 49.

9.2. ДИНАМІЧНИЙ ПОЛІМОРФІЗМ

Перш ніж приступити до обговорення такої важливої теми, як динамічний поліморфізм, повернемося до програми, розглянутої вище при аналізі відкритого одиночного успадкування.

Відкрите одиночне успадкування

```
  #include <stdio.h>

 class TInner
 {
 public:
    int i;
    TInner(int n):i(n){}
    ~TInner(){printf("Dtor TInner\n");}
 };

 class TBase
 {
 public:
    double a;
```

```
    TInner b;
    TBase():b(10),a(5.0){printf("Ctor TBase\n");}
    ~TBase(){printf("Dtor TBase\n");}
    void printBase() {printf("TBase::TInner::i = %d a = %lf\n",b.i, a);}
};

class TDerived: public TBase
{
public:
    char c;
    TDerived():c('Z'){printf("Ctor TDerived\n");}
    ~TDerived(){printf("Dtor TDerived\n");}
    void printDerived() {printf("TDerived::TInner::i = %d c = %c\n",b.i,c);}
};

class TDerived2: public TDerived
{
public:
    float f;
    TDerived2():f(10.0){printf("Ctor TDerived2\n");}
    ~TDerived2(){printf("Dtor TDerived2\n");}
    void printDerived2() {printf("TDerived2::TInner::i = %d f = %lf\n",b.i,f);}
};

int main()
{
    TDerived2 obj;
    obj.printBase();
    obj.printDerived();
    obj.printDerived2();
    return 0;
}
```

У кожнім із класів `TBase`, `TDerived` і `TDerived2` передбачена своя функція, що виводить на екран уміст полів класу — `printBase()`, `printDerived()` і `printDerived2()`. Об'єкт `obj` належить класу `TDerived2`, тому він успадковує функції `printBase()` і `printDerived()`. Це дозволяє забезпечити однаковий виклик цих функцій, інакше довелося б створювати окремі об'єкти класів `TBase` і `TDerived`, як це зроблено в першому варіанті програми.

Спробуємо уніфікувати інтерфейс, використовуючи однакові імена функцій висновку. Наприклад, назвемо їх ім'ям `print()`. Зовсім очевидно, що тепер виклик `obj.print()` відноситься лише до функції `print()` із класу `TDerived2`, а функції `print()` із класів `TBase` і `TDerived` "маскуються". Ця ситуація дуже нагадує маскування глобальних змінних локальними.

Конечно, можна скористатися уже відомим нам прийомом і застосувати оператор дозволу області бачимоості, вказавши ім'я класу.

```
  obj.TBase::print();
 obj.TDerived::print();
 obj.print();
```

Однак такий спосіб позбавляє програму гнучкості. В ідеалі, хотілося б, щоб об'єкт класу, що входить в ієрархію, сам визначав, з якого класу викликати функцію, аналізуючи контекст, як це роблять перевантажені функції й оператори. Таке поводження сутностей програми називається *поліморфізмом*.

Оскільки перевантаження функцій і операторів визначається в ході компіляції, такий вид поліморфізму називається *статичним*. (Усі процеси, виконувані компілятором, називаються *статичними*.) Процедура зв'язування типів з різними версіями функції на етапі компіляції називається *раннім зв'язуванням*. Однак ми хотіли б, щоб об'єкт реагував на контекст програми *в ході її виконання*. Це явище називається *динамічним поліморфізмом*, чи *пізнім зв'язуванням*.

В основі механізму динамічного поліморфізму лежить механізм *успадкування* і *віртуальних функцій*.

9.2.1. Віртуальні функції-члени

Віртуальна функція з'являється в базовому класі за допомогою ключового слова `virtual`. У похідних класах слово `virtual` можна не вживати. Програміст може перевизначити цю функцію в кожнім з похідних класів, настроївши її на рішення нової задачі. Якщо функція викликається з об'єкта відповідного класу, що входить в ієрархію, її поводження нічим не відрізняється від звичайної функції-члена. Віртуальність виявляється лише при виклику через вказівник. Помітимо, що перевизначення функцій можна здійснювати і без допомоги ключового слова `virtual` (наприклад, варіюючи тип і кількість параметрів). Однак у цьому випадку буде задіяний механізм раннього зв'язування, тобто зіставлення параметрів і функції-члена буде виконано на етапі

компіляції, а не на етапі виконання програми. На противагу перевантаженню, при перевизначенні в похідних класах прототипи віртуальних функцій повинні точно збігатися з прототипом віртуальної функції в базовому класі. Такий процес називається *заміщенням*.

Крім того, на віртуальні функції накладаються наступні обмеження.

1. Вони не можуть бути статичними.
2. Вони не можуть бути дружніми.
3. Конструктори не можуть бути віртуальними.

Оскільки об'єкти похідного класу, по суті, являють собою модифіковані об'єкти базового класу, на них у мові C++ можна *посилатися за допомогою вказівника на об'єкти базового класу*. Якщо вони містять віртуальні функції, их вибір буде виконаний на етапі виконання програми. Ієрархія класів, зв'язаних визначеною віртуальною функцією, називається *поліморфичним кластером*.

Продемонструємо механізм заміщення, переписавши нашу програму.

Використання віртуальних функцій

```
  #include <stdio.h>

class TInner
{
public:
    int i;
    TInner(int n):i(n){}
    ~TInner(){printf("Dtor TInner\n");}
};

class TBase
{
public:
    double a;
    TInner b;
    TBase():b(10),a(5.0){printf("Ctor TBase\n");}
    ~TBase(){printf("Dtor TBase\n");}
    virtual void print() {printf("TBase::TInner::i = %d a = %lf\n",b.i, a);}
};

class TDerived: public TBase
{
public:
    char c;
    TDerived():c('Z'){printf("Ctor TDerived\n");}
    ~TDerived(){printf("Dtor TDerived\n");}
    void print() {printf("TDerived::TInner::i = %d c = %c\n",b.i,c);}
};

class TDerived2: public TDerived
{
public:
    float f;
    TDerived2():f(10.0){printf("Ctor TDerived2\n");}
    ~TDerived2(){printf("Dtor TDerived2\n");}
    void print() {printf("TDerived2::TInner::i = %d f = %lf\n",b.i,f);}
};

int main()
{
     TBase objBase;
     TDerived objDerived;
     TDerived2 objDerived2;

     TBase* pObj = &objBase;
     pObj->print();

     pObj=&objDerived;
     pObj->print();

     pObj=&objDerived2;
     pObj->print();
```

```
    return 0;
}
```

Програма працює правильно, як і колись. Переміщаючи вказівник базового класу `pObj` по ієрархії, ми одержуємо доступ до усіх функцій `print()`, що заміщають віртуальну функцію в похідних класах.

Ця програма ілюструє один важливий аспект віртуальних функцій. Зверніть увагу на те, що в класі `TDerived` перед оголошенням функції `print()` ключове слово `virtual` *не зазначене*. Як ми уже відзначали, у похідних класах це робити не обязательно. Однак клас `TDerived` є базовим стосовно класу `TDerived2`, отже, ключове слово `virtual` повинне було б забезпечувати заміщення функції `print()` у класі `TDerived2`. Тим часом програма як і раніше успішно посилається на заміщену функцію `print()` в об'єкті `objDerived2`. Це відбувається тому, що *віртуальність успадковується*.

Цікаво, а що відбудеться, якщо на якомусь рівні ієрархії ми не будемо заміщати віртуальну функцію? Чи відбудеться розрив ланцюжка заміщень? Припустимо, що в попередній програмі ми не заміщали функцію `print()`. (Для більшої наочності ми не видалили, а просто закоментували відповідне визначення функції `print`.)

```
  class TDerived2: public TDerived
 {
 public:
    float f;
    TDerived2():f(10.0){printf("Ctor TDerived2\n");}
    ~TDerived2(){printf("Dtor TDerived2\n");}
 //    void print() {printf("TDerived2::TInner::i = %d f = %lf\n",b.i,f);}
 };
```

Результати роботи програми тепер виглядають так.

```
1.   Ctor TBase
2.   Ctor TBase
3.   Ctor TDerived
4.   Ctor TBase
5.   Ctor TDerived
6.   Ctor TDerived2
7.   TBase::TInner::i = 10 a = 5.000000
8.   TDerived::TInner::i = 10 c = Z
9.   TDerived::TInner::i = 10 c = Z
10.  Dtor TDerived2
11.  Dtor TDerived
12.  Dtor TBase
13.  Dtor TInner
14.  Dtor TDerived
15.  Dtor TBase
16.  Dtor TInner
17.  Dtor TBase
18.  Dtor TInner
```

Рядок 9 свідчить про те, що при виклику функції `print()` через вказівник на об'єкт `objDerived2` виробляється виклик попередньої заміщеної версії — з об'єкта `objDerived`.

Отже, *віртуальні функції утворять ієрархію*.

9.2.2. Суто віртуальні функції й абстрактні класи

Уявимо собі, що базовий клас настільки абстрактний, що визначити заздалегідь, як повинна виглядати його віртуальна функція-член, неможливо. У таких ситуаціях її конкретне втілення конкретизується лише в похідних класах на основі додаткової інформації. Крім того, іноді необхідно забезпечити заміщення віртуальних функцій в усіх без винятку похідних класах. Як ми бачили вище, механізму звичайних віртуальних функцій для цього недостатньо — компілятор ніяк не реагує на відсутність заміщеної версії функції `print()` у класі `TDerived2`. Для вирішення цієї проблеми в мові C++ реалізований механізм *суто віртуальних функцій*.

Суто віртуальною функцією називається віртуальна функція-член базового класу, що не має визначення. Оголошення суто віртуальної функції виглядає в такий спосіб.

`virtual` *тип ім'я_функції*`(`*параметри*`) = 0;`

Допустимо, нам невідома конкретна реалізація класу `TInner`. Закроем очі на те, що зараз він містить поле типу `int`. Можливо, замість цілого числа там буде число типу `double` чи об'єкт іншого класу. У такому випадку клас `TBase` не повинний залежати від конкретної реалізації класу `TInner`. Тоді незрозуміло, як виводити на екран поля класу `TBase`. У цьому випадку необхідно оголосити функцію `print()` суто віртуальною, поклавши відповідальність за вивід на екран полів базового класу на функції, що заміщають

віртуальну функцію `print()` у базовому класі. Зрозуміло, при розробці похідних класів протоколи класів `TInner` і `TBase` вважаються відомими.

Клас, що містить хоча б одну суто віртуальну функцію, називається *абстрактним*. Оскільки він визначений не цілком, його об'єкти створити неможливо, хоча можна повідомляти вказівники і посилання абстрактного класу. Наприклад, функцію `main()` можна визначити в такий спосіб.

Використання абстрактних класів: перший варіант

```
 int main()
{
   TBase* pObj;
   TDerived objDerived;
   TDerived2 objDerived2;

   pObj = &objDerived;
   pObj->print();
   pObj = &objDerived2;
   pObj->print();

   return 0;
}
```

Розглянемо програму, у якій клас `TBase` є абстрактним.

Використання абстрактних класів: другий варіант

```
 #include <stdio.h>

class TInner
{
public:
   int i;
   TInner(int n):i(n){}
   ~TInner(){printf("Dtor TInner\n");}
};

class TBase
{
public:
   double a;
   TInner b;
   TBase():b(10),a(5.0){printf("Ctor TBase\n");}
   ~TBase(){printf("Dtor TBase\n");}
   virtual void print() = 0;
};

class TDerived: public TBase
{
public:
   char c;
   TDerived():c('Z'){printf("Ctor TDerived\n");}
   ~TDerived(){printf("Dtor TDerived\n");}
   void print() {printf("TDerived::TInner::i = %d c = %c\n",b.i,c);}
};

class TDerived2: public TDerived
{
public:
   float f;
   TDerived2():f(10.0){printf("Ctor TDerived2\n");}
   ~TDerived2(){printf("Dtor TDerived2\n");}
};

int main()
{
    //TBase objBase;            Помилка! Клас TBase — абстрактний!
    TDerived objDerived;
    TDerived2 objDerived2;

    objDerived.print();
```

```
    objDerived2.print();

    return 0;
}
```

Варто мати на увазі, що похідні класи повинні заміщати суто віртуальну функцію. Це анітрошки не суперечить наведеній вище програмі, оскільки в сформульованому вище правилі маються на увазі *безпосередні спадкоємці* базового класу. Оскільки клас `TDerived2` є похідним від класу `TDerived`, на нього це правило не поширюється. Однак якби клас `TDerived2` був прямим спадкоємцем класу `TBase`, виникла б помилка компіляції.

```
class TDerived2: public TBase
{
public:
   float f;
   TDerived2():f(10.0){printf("Ctor TDerived2\n");}
   ~TDerived2(){printf("Dtor TDerived2\n");}
  // Відсутнє перевизначення суто віртуальних функцій.
};
```

9.2.3. Механізм віртуальних функцій

Об'єкт класу розміщається в суцільній області пам'яті, адреса якої зберігається в неявному вказівнику `this`. При виклику звичайної функції-члена цей вказівник передається їй як додатковий аргумент. Створюючи об'єкт похідного класу, компілятор поєднує його поля і поля базового класу в одне ціле. Якщо базовий клас містить віртуальні функції, їх адреси заносяться в *таблицю віртуальних функцій*. Усі класи, що утворюють поліморфічний кластер, містять вказівник на цю таблицю, у якій зберігаються вказівники на усі віртуальні функції-члени класів.

Розглянемо наступну програму.

Вказівник на таблицю віртуальних функцій

```
#include <stdio.h>

class TBase
{
public:
   char a;
   TBase():a('X'){printf("Ctor TBase\n");}
   ~TBase(){printf("Dtor TBase\n");}
   //virtual void print() {printf("TBase::a = %c\n",a);}
};

class TDerived: public TBase
{
public:
   char b;
   TDerived():b('Y'){printf("Ctor TDerived\n");}
   ~TDerived(){printf("Dtor TDerived\n");}
   void print() {printf("TDerived::b = %c\n",b);}
};

class TDerived2: public TDerived
{
public:
   char c;
   TDerived2():c('Z'){printf("Ctor TDerived2\n");}
   ~TDerived2(){printf("Dtor TDerived2\n");}
   void print() {printf("TDerived2::c %c \n",c);}
};

int main()
{
    printf("Sizeof(char)      = %d\n",sizeof(char));
    printf("Sizeof(TBase)     = %d\n",sizeof(TBase));
    printf("Sizeof(TDerived)  = %d\n",sizeof(TDerived));
    printf("Sizeof(TDerived2) = %d\n",sizeof(TDerived2));

    return 0;
}
```

Виконання цієї програми супроводжується наступними повідомленнями.

```
  Sizeof(char)      = 1
Sizeof(TBase)     = 1
Sizeof(TDerived)  = 2
Sizeof(TDerived2) = 3
```

Отже, розмір типу `char` дорівнює одному байту. Клас `TBase` містить одне поле типу `char`, отже, його розмір також дорівнює одному байту. Клас `TDerived` успадковує поле типу `char`, додаючи його до власного члена `b`. Естественно, розмір збільшується вдвічі. Аналогічна ситуація виникає в класі `TDerived2`, що містить три поля типу `char` — одне своє і два успадкованих.

Тепер знімемо коментар з наступної рядка.

```
// virtual void print() {printf("TBase::a = %c\n",a);}
```

Результати роботи програми стануть зовсім іншими.

```
  Sizeof(char)      = 1
Sizeof(TBase)     = 9
Sizeof(TDerived)  = 12
Sizeof(TDerived2) = 16
```

Розмір вказівника на таблицю віртуальних функцій дорівнює 4 байт, отже, справжній розмір класу `TBase` повинний бути дорівнює 5 байт. Однак за рахунок вирівнювання машинного слова він збільшується до 9 байт. Отже, розмір класу `TDerived` повинний бути рівним 9+4=12 байт. Як бачимо, це відповідає дійсності. Аналогічно, `sizeof(TDerived2)= sizeof(TDerived)+sizeof(вказівник)=` 12+6=16 байт. Таким чином, ця програма підтверджує загальновідомий факт, що кожен об'єкт поліморфічного кластера містить вказівник на таблицю віртуальних функцій. Виключення перевизначених версій з похідних класів нічого не змінює — віртуальні функції утворять ієрархію й успадковуються всеми похідними класами.

9.2.4. Віртуальний деструктор

Якщо вказівник базового класу посилається на об'єкт похідного класу, необхідно застосовувати віртуальний деструктор. Розглянемо конкретний приклад.

Віртуальний деструктор

```
  #include <stdio.h>
#include <string.h>

class TBase
{
public:
   char* a;
   TBase(const char* s){a = strdup(s); printf("Ctor TBase\n");}
   ~TBase(){delete a; printf("Dtor TBase\n");}
   virtual void print() {printf("TBase::a      = %s\n",a);}
};

class TDerived: public TBase
{
public:
   char* b;
   TDerived(const char* s1, const char* s2):TBase(s2)
   { b = strdup(s2); printf("Ctor TDerived\n");}
   ~TDerived(){delete b; printf("Dtor TDerived\n");}
   void print() {printf("TDerived::b   = %s\n",b);}
};

class TDerived2: public TDerived
{
public:
   char* c;
   TDerived2(const char* s1,const char* s2, const char* s3):TDerived(s2,s3)
   {c = strdup(s3); printf("Ctor TDerived2\n");}
   ~TDerived2(){delete c; printf("Dtor TDerived2\n");}
   void print() {printf("TDerived2::c  = %s \n",c);}
};

int main()
{
    TBase* pObj = new TDerived2("TBase", "TDerived","TDerived2");
     delete pObj; // Помилка!
```

```
      return 0;
 }
```

Клас `TBase` містить один рядок. Його конструктор виділяє для неї стільки пам'яті, скільки займає константний аргумент `s`, а його деструктор звільняє пам'ять, зайняту рядком `a`.

Клас `TDerived` успадковує цей рядок із класу `TBase` і додає свою. Його конструктор спочатку викликає конструктор класу `TBase`, ініціалізуючи успадковане поле `a`, а потім виділяє пам'ять для власного рядка `b`, ініціалізуючи її константним аргументом.

Клас `TDerived2` успадковує цей рядок із класу `TDerived` і додає свою. Його конструктор спочатку викликає конструктор класу `TDerived`, ініціалізуючи успадковані поля `a` і `b`, а потім виділяє пам'ять для власного рядка `c`, ініціалізуючи її константним аргументом.

Помітимо, що всі деструктори видаляють лише поля, виділені для власних членів класу, думаючи, що за успадковані поля відповідальність несе деструктор базового класу.

Уся ця конструкція працює цілком надійно, поки програма не спробує знищити вказівник базового класу `pObj`, що посилається на об'єкт похідного класу; у даному випадку вказівник посилається на об'єкт `objDerived2` класи `TDerived2`.

Програма виводить на екран наступні рядки.

```
  Ctor TBase
 Ctor TDerived
 Ctor TDerived2
 Dtor TBase
```

Як бачимо, пам'ять звільнена не цілком. Що відбулося? Деструктор звільнить пам'ять, виділену конструктором базового класу, але для звільнення пам'яті, зайнятої додатковими полями, необхідно викликати деструктори похідних класів! У звичайного деструктора немає такої можливості. Для цього його оголошують *віртуальним*.

Не будемо приводити практично незмінний текст програми — просто поставимо перед деструкторами класів `TBase` і `TDerived` (вони обоє є базовими для класів `TDerived` і `TDerived2` відповідно) ключове слово `virtual`.

```
  class TBase
{
public:
     ...


    virtual  ~TBase(){delete a; printf("Dtor TBase\n");}
    ...
};

class TDerived: public TBase
{
public:
    ...
    virtual ~TDerived(){delete b; printf("Dtor TDerived\n");}
    ...
 };
```

Це приведе до наступного результатам.

```
  Ctor TBase
  Ctor TDerived
  Ctor TDerived2
  Dtor TDerived2
  Dtor TDerived
  Dtor TBase
```

Тепер пам'ять звільнена корректно.

9.2.5. Прийоми програмування

Дотепер ми вивчали, як слід і як не слід родити, працюючи з ієрархією класів. Зокрема, не можна створювати віртуальні конструктори. По визначенню віртуальний конструктор — це оксиморон, інакше кажучи, внутрішньо суперечлива сутність. Віртуальність функції виявляється, коли, знаходячись у межах ієрархії класів, необхідно послатися на існуючий об'єкт, тип якого заздалегідь невідомий. У той же час конструктор викликається, коли об'єкта ще не існує (власне, саме для його створення він і викликається!). Це явне протиріччя.

Однак це протиріччя можна зняти, використовувавши *узагальнене рішення*. Ми будемо так називати синтаксичну конструкцію, що імітує поводження забороненої сутності, будучи зовсім коректною із

синтаксичної точки зору. Інакше кажучи, узагальнене рішення — це спосіб обдурити занадто строгий компілятор.

Отже, чого ми хочемо від віртуального конструктора? Щоб він створював об'єкти різного типу — у залежності від одержуваних аргументів. Спробуємо його реалізувати.

Узагальнене рішення

```
  #include <stdio.h>

 class TBase {
 public:
    char *a;
    virtual ~TBase(){}
    virtual TBase* virtual_copy()  const
    {
      printf("TBase virtual copy\n");
      return new TBase(*this);
    }
    virtual TBase* virtual_ctor()  const
    {
      printf("TBase virtual ctor \n");
      return new TBase();
    }
 };

class TDerived : public TBase {
 public:
    char* b;
    TDerived* virtual_copy()  const
    {
      printf("TDerived virtual copy\n");
      return new TDerived(*this);
   };

   TDerived* virtual_ctor()  const
   {
     printf("TDerived virtual ctor\n");
     return new TDerived();
   }
 };

int main()
 {
    TBase objBase;
    TDerived objDerived;

    TBase* pBase = &objBase;

    TBase* s2 = pBase->virtual_ctor();
    TBase* s1 = pBase->virtual_copy();

    pBase = &objDerived;

    TDerived* s3 = (TDerived*)pBase->virtual_ctor();
    TDerived* s4 = (TDerived*)pBase->virtual_copy();

    delete s1;
    delete s2;
    delete s3;
    delete s4;

    return 0;
 }
```

Зрозуміло, конструктор і конструктор копіювання віртуальними не стали. Просто віртуальні функції, що заміщаються в похідному класі, звертаються до відповідного конструктора, а звертатися до цих функцій можна за допомогою вказівника на базовий клас.

У результаті виконання програми на екрані з'являться наступні рядки.

```
TBase virtual ctor
```

```
TBase virtual copy
TDerived virtual ctor
TDerived virtual copy
```

В ідіомі віртуального конструктора використовуються ослаблені обмеження на типи значень, що повертаються віртуальними функціями. Інакше кажучи, тепер прототипи віртуальних функцій у базовому і похідному класах не зобов'язані збігатися. На жаль, компілятор Microsoft C++ 6.0 не підтримує цю можливість. Для того щоб реалізувати ідіому віртуального конструктора, слід скористатися більш сучасними компіляторами.

9.3. ІНФОРМАЦІЯ ПРО ТИП НА ЕТАПІ ВИКОНАННЯ

Як визначити тип об'єкта під час виконання програми? Для рішення цієї задачі можна скористатися двома методами.

9.3.1. Оператор dynamic_cast

Перший підхід заснований на використанні оператора динамічного приведення типу `dynamic_cast`. Якщо операція типу повертає правильний вказівник, виходить, об'єкт має відповідний тип, якщо нульовий, — немає. Отже, можна написати таку програму.

Розпізнавання типу об'єкта на етапі виконання програми

```
   #include <stdio.h>
#include <string.h>

class TBase
{
public:
   char* a;
   TBase(const char* s){a = strdup(s); printf("Ctor TBase\n");}
   virtual ~TBase(){delete a; printf("Dtor TBase\n");}
   virtual void print() {printf("TBase::a      = %s\n",a);}
};

class TDerived: public TBase
{
public:
   char* b;
   TDerived(const char* s1, const char* s2):TBase(s2)
   { b = strdup(s2); printf("Ctor TDerived\n");}
   virtual ~TDerived(){delete b; printf("Dtor TDerived\n");}
   void print() {printf("TDerived::b   = %s\n",b);}
};

class TDerived2: public TDerived
{
public:
   char* c;
   TDerived2(const char* s1,const char* s2, const char* s3):TDerived(s2,s3)
   {c = strdup(s3); printf("Ctor TDerived2\n");}
   ~TDerived2(){delete c; printf("Dtor TDerived2\n");}
   void print() {printf("TDerived2::c  = %s \n",c);}
};

int main()
{
    TDerived* pObj = new TDerived("TBase", "TDerived");
//   TBase* q = dynamic_cast<TBase *>(pObj);
    if (TBase* q = dynamic_cast<TBase *>(pObj))
        printf("TBase\n"); else printf("Not TBase");
    return 0;
}
```

Оператор `dynamic_cast` має два операнди: у кутових дужках указується тип, а в круглих дужках — вказівник. Наприклад, в операторі

```
dynamic_cast<TBase*>(pObj)
```

`TBase*` — тип, а `pObj` — вказівник.

Якщо вказівник `pObj` посилається на об'єкт базового класу `TBase` чи похідного, результатом застосування оператора `dynamic_cast` буде вказівник на базовий клас `TBase`.

За допомогою оператора `dynamic_cast` віртуальні базові класи можна перетворювати в похідні (*понижуюче приведення*), похідні — у базовий (*підвищувальне приведення*), а також один похідний клас — в інший. До класів, що не містять віртуальних функцій, оператор `dynamic_cast` не застосовується.

Розглянемо наступну ієрархію класів. Базовий клас `TBase` має похідний клас `TDerived`. Клас `TDerived` є базовим стосовно двох довільних класів: `TDerived2` і `TDerived3`.

Понижуюче, підвищувальне і перехресне приведення

```
   #include <stdio.h>
#include <string.h>
#include <iostream.h>

class TBase
{
public:
   char* a;
   TBase(const char* s){a = strdup(s); printf("Ctor TBase\n");}
   ~TBase(){delete a; printf("Dtor TBase\n");}
   virtual void print() {printf("TBase::a      = %s\n",a);}
};

class TDerived: public TBase
{
public:
   char* b;
   TDerived(const char* s1, const char* s2):TBase(s2)
   { b = strdup(s2); printf("Ctor TDerived\n");}
   ~TDerived(){delete b; printf("Dtor TDerived\n");}
   void print() {printf("TDerived::b   = %s\n",b);}
};

class TDerived2: public TDerived
{
public:
   char* c;
   TDerived2(const char* s1,const char* s2, const char* s3):TDerived(s2,s3)
   {c = strdup(s3); printf("Ctor TDerived2\n");}
   ~TDerived2(){delete c; printf("Dtor TDerived2\n");}
   void print() {printf("TDerived2::c  = %s \n",c);}
};

class TDerived3: public TDerived
{
public:
   char* c;
   TDerived3(const char* s1,const char* s2, const char* s3):TDerived(s2,s3)
   {c = strdup(s3); printf("Ctor TDerived2\n");}
   ~TDerived3(){delete c; printf("Dtor TDerived2\n");}
   void print() {printf("TDerived3::c  = %s \n",c);}
};

int main()
{
   TBase* pObj1 = new TBase("TBase");
   TDerived* pObj2 = new TDerived("TBase", "TDerived");
   TDerived2* pObj3 = new TDerived2("TBase", "TDerived","TDerived2");
   TDerived3* pObj4 = new TDerived3("TBase", "TDerived","TDerived2");

  // Понижуюче приведення
  if(TDerived* q = dynamic_cast<TDerived*>(pObj1))
  printf("TBase->TDerived\n");
     else printf("TBase->TDerived: Bad cast\n");

  // Підвищувальне приведення
  if(TBase* pObj5 = dynamic_cast<TBase *> (pObj2))
  printf("TDerived->TBase\n");
     else printf("TDerived->TBase: Bad cast\n");

  // Підвищувальне приведення
  if(TDerived* pObj6 = dynamic_cast<TDerived*>(pObj3))
  printf("TDerived2->TDerived\n");
```

```
      else printf("TDerived2->TDerived: Bad cast\n");

    // Понижуюче приведення
    if(TDerived2* pObj7 = dynamic_cast<TDerived2*>(pObj2))
    printf("TDerived->TDerived2\n");
      else printf("TDerived->TDerived2: Bad cast\n");

    // Перехресне приведення
   if(TDerived3* pObj9 = dynamic_cast<TDerived3*>(pObj3))
   printf("TDerived2->TDerived3\n");
       else printf("TDerived2->TDerived3: Bad cast\n");
   return 0;
 }
```

Програма виводить на екран наступні рядки.

```
Ctor TBase
Ctor TBase
Ctor TDerived
Ctor TBase
Ctor TDerived
Ctor TDerived2
Ctor TBase
Ctor TDerived
Ctor TDerived2
TBase->TDerived: Bad cast
TDerived->TBase
TDerived2->TDerived
TDerived->TDerived2: Bad cast
TDerived2->TDerived3: Bad cast
```

Як бачимо, приведення, що знижують, і перехресні приведення не виконані. Зауважимо, що, виконуючи понижуюче приведення, необхідно установити опцію компілятора `/Gr`, інакше виникає непередбачена виняткова ситуація.

Це пояснюється тим, що понижуюче і перехресне приведення можливе лише тоді, коли вказівник на базовий клас, що підлягає приведенню, дійсно посилається на об'єкт похідного класу.

9.3.2. Оператор typeid

Отже, оператор `dynamic_cast` додає програмам, що працюють з ієрархіями віртуальних класів, додаткову гнучкість. Однак про точний тип об'єкта з його допомогою довідатися складно. Оператор `dynamic_cast` у кращому випадку дозволяє розпізнати базовий клас. Для цього в мові C++ передбачений оператор `typeid`.

Цей оператор повертає посилання на об'єкт класу `type_info`, визначеного в заголовку `<typeinfo>`. Клас `typeinfo`, поряд з ім'ям типу, містить визначення операцій порівняння, тому його можна використовувати в логічних виразах.

Розглянемо приклад, у якому досліджується описана вище ієрархія класів. Оскільки визначення класів залишаються незмінними, ми приведемо лише функцію `main()`.

```
  int main()
{
     TBase objBase("TBase");
     TBase* pObj1=&objBase;
     TDerived objDerived("TBase", "TDerived");
     pObj1 = &objDerived;
     if(typeid(*pObj1)==typeid(TDerived)) printf("TDerived\n");
        else printf("Something else ...");

     return 0;
 }
```

Програма виводить на екран наступні рядки.

```
Ctor TBase
Ctor TBase
Ctor TDerived
TBase->TDerived
Dtor TDerived
Dtor TBase
Dtor TBase
```

Четвертий рядок свідчить про те, що вказівник `pObj1` дійсно посилається на об'єкт класу `TDerived`.

Знаючи пристрій класу `type_info`, можна спробувати вивести на екран назву елементарного типу чи класу.

Застосування оператора typeid

```
  #include <stdio.h>
#include <typeinfo>

int main()
{
   double X;
   printf("%s",typeid(X).name());

   return 0;
}
```

Ця програма виводить на екран рядок `double`.

Застосовний тепер оператор `typeid` не до елементарного типу, а до класу.

```
int main()
{
   TBase objBase("TBase");
   printf("%s",typeid(objBase).name());
   return 0;
}
```

На екран виводяться такі рядки.

```
Ctor TBase
class TBase
Dtor TBase
```

Перше повідомлення виводиться конструктором класу `TBase`, другий рядок — результат застосування оператора `typeid` до об'єкта `objBase`, а останній рядок являє собою повідомлення деструктора класу `TBase`.

9.4. РЕЗЮМЕ

- Можливість використовувати готовий модуль як базу для власних розробок, не переробляючи його код називається *повторним використанням коду*.
- В основі механізму, що дозволяє створювати ієрархії класів, лежить принцип *успадкування*.
- Клас, що лежить в основі ієрархії, називається *базовим*.
- Класи, що успадковують властивості базового класу, називаються *похідними*.
- Похідні класи, у свою чергу, можуть бути базовими стосовно своїх спадкоємців, що в результаті приводить до ланцюжка успадкування. Процес утворення похідного класу на основі базового називається *виводом* класу. З одного базового класу можна вивести декілька похідних. Крім того, похідний клас може бути спадкоємцем декількох базових класів.
- Успадкування буває *одиночним* і *множинним*. При одиночному успадкуванні в кожного похідного класу є лише *один* базовий клас, а при множинному — *декілька*.
- Усі члени базового класу автоматично стають членами похідного. Керуючись оголошенням похідного класу, компілятор спочатку він бере усі властивості базового класу, а потім додає до них нові функціональні можливості похідного.

  Для цього використовується наступна синтаксична конструкція.

```
class ім'я_похідного_класу:специфікатор_доступу ім'я_базового_класу
{
  // тіло класу
};
```

- Хоча всі члени базового класу автоматично стають членами похідного класу, однак доступ до цих членів визначається видом успадкування. У залежності від специфікатора доступу, зазначеного при оголошенні похідного класу, розрізняють *відкрите*, *закрите* і *захищене* успадкування. За замовчуванням використовується закрите успадкування (специфікатор `private`).
- При відкритому успадкуванні всі відкриті і захищені члени базового класу стають відкритими і захищеними членами похідного класу відповідно.
- При захищеному успадкуванні всі відкриті і захищені члени базового класу стають захищеними членами похідного класу.
- При закритому успадкуванні всі відкриті і захищені члени базового класу стають закритими членами похідного класу відповідно.
- При успадкуванні рівень доступу дол членів похідного класу може лише зменшитись. “Все можна ще більше закрити, але не відкрити”.
- Закртиі члени базового клсу є доступними лише членам базового класу. Успадкування не може суперечити принципу приховання даних.

- При створенні об'єктів похідного класу необхідно спочатку створити проміжний об'єкт базового класу. Отже, спочатку викликається конструктор базового класу, а потім — конструктор похідного класу. Якщо базових класів декілька, вони викликаються в порядку их перерахування в списку наслідуваних класів.
- Деструктори викливаєьбся у порядку, зворотньому відносно до конструкторів.
- Множинне успадкування часте приводить до неоднозначностей. Однак для вирішення неоднозначностей можна використати *віртуальні базові класи*. Для того щоб запобігти дублювання, у списку успадкування перед ім'ям базового класу варто вказати ключовому слову `virtual`, передбачивши конструктор за замовчуванням.
- Оскільки перевантаження функцій і операторів визначається в ході компіляції, такий вид поліморфізму називається *статичним*. (Усі процеси, виконувані компілятором, називаються *статичними*.) Процедура зв'язування типів з різними версіями функції на етапі компіляції називається *раннім зв'язуванням*. Однак ми хотіли б, щоб об'єкт реагував на контекст програми *в ході її виконання*. Це явище називається *динамічним поліморфізмом*, чи *пізнім зв'язуванням*.
- В основі механізму динамічного поліморфізму лежить механізм *успадкування* і *віртуальних функцій*. *Віртуальна функція* з'являється в базовому класі за допомогою ключового слова `virtual`. У похідних класах слово `virtual` можна не вживати. Програміст може перевизначити цю функцію в кожнім з похідних класів, настроївши її на рішення нової задачі. Якщо функція викликається з об'єкта відповідного класу, що входить в ієрархію, її поводження нічим не відрізняється від звичайної функції-члена. Віртуальність виявляється лише при виклику через вказівник.
- На противагу перевантаженню, при перевизначенні в похідних класах прототипи віртуальних функцій повинні точно збігатися з прототипом віртуальної функції в базовому класі. Такий процес називається *заміщенням*.
- На віртуальні функції накладаються наступні обмеження.

  Вони не можуть бути статичними.

  Вони не можуть бути дружніми.

  Конструктори не можуть бути віртуальними.
- Оскільки об'єкти похідного класу, по суті, являють собою модифіковані об'єкти базового класу, на них у мові C++ можна *посилатися за допомогою вказівника на об'єкти базового класу*. Якщо вони містять віртуальні функції, их вибір буде виконаний на етапі виконання програми. Ієрархія класів, зв'язаних визначеною віртуальною функцією, називається *поліморфичним кластером*.
- Якщо віртуальна функція не заміщується у якомусь похідному класі, викливається її попередня заміщена версія. Отже, *віртуальні функції утворять ієрархію, тобто успадковуються.*
- Якщо віртуальна функція зовсім не заміщується у похідних классах — це помилка.
- Якщо базовий клас настільки абстрактний, що визначити заздалегідь, як повинна виглядати його віртуальна функція-член, неможливо, його конкретне втілення конкретизується лише в похідних класах на основі додаткової інформації. Такій класи називаються *абстрактними*. Для вирішення цієї проблеми в мові C++ реалізований механізм *суто віртуальних функцій*.
- *Суто віртуальною функцією* називається віртуальна функція-член базового класу, що не має визначення. Оголошення суто віртуальної функції виглядає в такий спосіб.

  `virtual` *`тип ім'я_функції`*`(`*`параметри`*`) = 0;`
- Об'єкт класурозміщається в суцільній області пам'яті, адреса якої зберігається в неявному вказівнику `this`. При виклику звичайної функції-члена цей вказівник передається їй як додатковий аргумент. Створюючи об'єкт похідного класу, компілятор поєднує його поля і поля базового класу в одне ціле. Якщо базовий клас містить віртуальні функції, їх адреси заносяться в *таблицю віртуальних функцій*. Усі класи, що утворюють поліморфічний кластер, містять вказівник на цю таблицю, у якій зберігаються вказівники на усі віртуальні функції-члени класів.
- Якщо вказівник базового класу посилається на об'єкт похідного класу, необхідно застосовувати віртуальний деструктор. Розглянемо конкретний приклад.
- Оператор `dynamic_cast` має два операнди: у кутових дужках указується тип, а в круглих дужках — вказівник. Наприклад, в операторі

  `dynamic_cast<TBase*>(pObj)`

  `TBase*` — тип, а `pObj` — вказівник.
- Якщо вказівник `pObj` посилається на об'єкт базового класу `TBase` чи похідного, результатом застосування оператора `dynamic_cast` буде вказівник на базовий клас `TBase`.
- За допомогою оператора `dynamic_cast` віртуальні базові класи можна перетворювати в похідні (*понижуюче приведення*), похідні — у базовий (*підвищувальне приведення*), а також один похідний клас — в іншій. До класів, що не містять віртуальних функцій, оператор `dynamic_cast` не застосовується.

- Оператор `typeid` повертає посилання на об'єкт класу `type_info`, визначеного в заголовку `<typeinfo>`. Клас `typeinfo`, поряд з ім'ям типу, містить визначення операцій порівняння, тому його можна використовувати в логічних вираженнях.

9.4. КОНТРОЛЬНІ ПИТАННЯ

1. Що називається повторним використанням коду?
2. Який механізм дозволяє створювати ієрархії класів?
3. Який клас називається базовим?
4. Який клас називається похідним?
5. Що називається виводом класу?
6. Назвіть види успадкування залежно від кількості базових класів.
7. Опишіть механізм успадкування.
8. Назвіть види успадкування залежно від рівня доступу.
9. Опишіть відкрите успадкування.
10. Опишіть захищене успадкування.
11. Опишіть закрите успадкування.
12. При успадкуванні рівень доступу дол членів похідного класу може лише зменшитись. “Все можна ще більше закрити, але не відкрити”.
13. Як викликаються конструктори і деструктори в ієрахії класів.
14. Які базові класи називаються віртуальними?
15. Чим динамічний поліморфізм відрізняється від статичного?
16. Яка функція називається віртуальною?
17. Що таке заміщення?
18. Які обмеження накладаються на віртуальні функції?
19. Щотаке поліморфічний кластер?
20. Що станеться, якщо віртуальна функція не заміщується у якомусь похідному класі?
21. Які класи називаються абстрактними?
22. Яка функція називається суто віртуальною
23. Що таке таблиця віртуальних функцій?
24. Коли застосовується віртуальний деструктор.
25. Опишіть оператор `dynamic_cast`.
26. Опишіть три види приведення за допомогою оператора `dynamic_cast` в ієрархії класів.
27. Опишіть оператор `typeid`.