**Лекція 26. Спадкування. Типи спадкування та доступ до членів**

**Закрите та відкрите спадкування**

Протокол успадкування в С++ має наступний вигляд:

**class** <Назва похідного класу>: <кваліфікатор> <Назва базового класу>

{

...

};

Поле “кваліфікатор” забезпечує три типи успадкування: **public** (відкрите), **protected** (захищене) і **private**(закрите).

При відкритому успадкуванні властивості і методи базового класу зберігають рівень прав доступу відповідно до кваліфікаторів, з якими вони були оголошені.

При захищеному успадкуванні приватні елементи базового класу залишаються приватними в похідному класі, а відкриті стають захищеними.

При закритому успадкуванні всі елементи базового класу стають приватними в похідному класі.

Для відношення узагальнення (зв’язком “is a”, від анг. is a kind of, наприклад, A rose is a kind of flowers.) рекомендується завжди використовувати відкрите спадкування.

При закритому успадкуванні усі відкриті і захищені поля базового класу стають закритими членами похідного класу. Звернутися до цих полів прямо тепер неможливо — клас як би “запечатується. Всі члени батьківського класу спадкуються як закрити, що означає, що члени private залишаються недоступними, а члени protected и public стають private в похідному класі.

**Специфікатор доступу protected**

В C++ специфікатор доступі **protected** корисний тільки в контексте наслідування. **Специфікатор доступу protected відкриває доступ до членів класу дружнім та похідним класам**. Доступ до члену protected поза тіла класу закритий.

**#include <iostream.h>**

**class Parent**

**{**// доступ до цього члена відкритий для всіх об’єктів

**public: int m\_public;**

/\*доступ до цього члена відкритий тільки інших членів класу Parent та для дружніх класів/функцій (але не для похідних класів)\*/

**private: int m\_private;**

/\* доступ до цього члена відкритий для інших членів класу Parent, дружніх класів/функцій, похідних класів\*/

**protected: int m\_protected;**

**};**

**class Child: public Parent**

**{**

**public:**

**Child()**

**{**

/\* дозволено: доступ до відкритих членів батьківського класуз похідного класу \*/

**m\_public = 1;**

/\* заборонено: доступ до закритих членів батьківського класу з похідного класу \*/

**m\_private = 2;**

/\* дозволено: доступ до захищених членів батьківського класу з похідного класу \*/

**m\_protected = 3;**

**}**

**};**

**int main()**

**{**

**Parent parent;**

**// дозволено: доступ до відкритих членів класу із-зовні**

**parent.m\_public = 1;**

**// заборонено: доступ до закритих членів класу із-зовні**

**parent.m\_private = 2;**

**// заборонено: доступ до захищених членів класу із-зовні**

**parent.m\_protected = 3;**

**system("pause");**

**}**

Член m\_protected класу Parent напряму доступний похідному класу Child, але доступ до нього для членів — закритий.

До членів protected батьківського класу доступ відкритий для членів похідного класу, що означає, що якщо ви пізніше зміните щось в члені protected (тип даних, значення тощо), то вам потрібно буде внести зміни і в батьківський, і у всі похідні класи. Тому використання специфікатору доступу protected найкорисніше, коли ви будете спадкувати тільки свої класи і кількість похідних класів буде невелика. Якщо ви внесете зміни в реалізацію батьківського класу, то вам потрібно оновити всі похідні класи, і це оновлення ви можете зробити самостійно і це не займе багато часу (оскільки похідних класів буде небагато).

Створення членів private надає кращу інкапсуляцію та ізолює батьківські класи від змін, викликаних похідними класами. Ціна цього — створення відкритого або захищеного інтерфейсу (способу взаємодії других об’єктів з класами та їх членами, тобто ***геттери та сеттери***). Це додаткова робота, яку ви можете не виконувати, якщо ви самі працюєте зі своїми класами (чужі класи не звертаються до вашого класу) та кількість похідних класів невелика.

**Типи спадкування та доступ до членів.**

В попередній лекції ми розглядали специфікатори доступу при спадкуванні: **public; private; protected**. Для визначення типу спадкування потрібно просто вказати ключове слово біля класу, що успадковується, наприклад:

class Pri: private Parent {…}; // для закритого спадкування.

Якщо тип наслідування не визначено, то в C++ по замовчуванню буде обрано тип наслідування private (аналогічно членам класу, які по замовчуванню є private, якщо не вказано інше). Це, як бачили 9 комбінаций: 3 специфікатора доступу (public, private и protected) и 3 типи наслідування (public, private и protected).

При спадкуванні специфікатор доступу члена батьківського класу може бути змінений в похідному класі (в залежності від типу наслідування). Тобто, члени, які були public або protected в батьківському класі, можуть стати private в похідному класі.

**Дружні friend функції**

Іноді бажано мати безпосередній доступ ззовні до прихованих полів класу, тобто розширити інтерфейс класу. Для цього служать дружні функції і дружні класи. С++ надає можливість обійти одну із концепцій ООП – концепцію інкапсуляції - за допомогою друзів. С++ дозволяє об’являти два види друзів класу: дружню функцію та дружній клас.

Звичайний доступ до закритих членів класу – використання відкритої функції-члена. Але С++ підтримує інший спосіб одержання доступу до закритих членів класу – за допомогою дружніх функцій. Дружні функції не є членами класу, але тим не менше мають доступ до його закритих членів. Одна така функція може мати доступ до закритих членів декількох класів.

Функція не може стати другом класу “без його згоди”. Для одержання прав друга функція повинна бути описана в тілі класу.

Щоб оголосити функцію дружньою деякому класу, у визначення цього класу включають її прототип, перед яким ставиться ключове слово **friend**.

Дружні функції застосовуються для доступу до прихованих членів класу і є альтернативою методам. Метод, як правило, використовується для реалізації властивостей об'єкту, а у вигляді дружніх функцій оформляються дії, що не представляють властивості класу, але що концептуально входять в його інтерфейс і потребують доступу до його прихованих полів.

***Правила опису і особливості дружніх функцій***.

1. Дружня функція оголошується *усередині класу,* до елементів якого їй потрібен доступ, з ключовим словом **friend**. Як параметр їй повинні передаватися об'єкт або посилання на об'єкт класу, оскільки вказівка **this** їй не передається. Тому, викликаючи дружню функцію, не потрібно вказувати ім’я об’єкта або вказівник на об’єкт і операцію доступу до члена класу (крапку або стрілку).
2. Дружня функція може бути звичайною функцією або методом іншого раніше створеного класу. На неї не розповсюджується дія специфікаторів доступу, місце розміщення її оголошення в класі байдуже.
3. Одна функція може бути дружньою відразу з декількома класами.
4. Обмеження на кількість дружніх функцій відсутні.
5. Дружня функція не успадковується, тобто вона не є такою для похідних класів.
6. Відношення дружності не є транзитивним: Х дружній Y, Y дружній Z, але не обов‘язково Х дружній Z.

***Загальна форма оголошення “дружньої” функції до деякого класу***

Оголошення “дружньої” функції до класу починається з ключового слова friend. Загальна форма оголошення “дружньої” функції до класу має вигляд:

**friend** *type* **FunName**(*parameters*);

де

* *FunName* – ім’я “дружньої” функції;
* *type* – тип, що повертається функцією FunName();
* *parameters* – параметри “дружньої” функції. Щоб отримати об’єкт потрібного класу у функції FunName() доцільно передати у цю функцію посилання (або покажчик) на об’єкт цього класу.

Якщо потрібно оголосити “дружню” функцію у деякому класі, то загальний вигляд такого оголошення наступний:

**class CClass**

**{**

**// ...**

**friend *type* FunName(*parameters*);**

**// ...**

**};**

де

* *FunName* – ім’я “дружньої” функції;
* *type* – тип, що повертається функцією FunName();
* *parameters* – параметри “дружньої” функції.

**Приклад :**

**class Person**

**{**

int secret; // закрита частина класу

friend void Spouse(Person &); // не член класу

public:

Person(){ secret = 10;}

**}**;

Void Spouse(Person &p)

**{**

++p.secret;

cout << p.secret;

**}**

Void main()

**{**

Person ff;

Spouse(ff);

**}**

У даному прикладі функція **Spouse()** є звичайною функцією і не належить жодному класу. Цій функції передається посилання на об'єкт класу **Person**.

**Специфікатор friend:**

* + Оголошує в класі дружню-зовнішню функцію, що має такий же доступ до членів-даних і функцій (приватних та публічних), як власні методи класу
  + Вона не має доступу до this тому має на 1 параметр більше ніж аналогічний метод
  + Вона реалізовується (без вживання friend) і викликається як зовнішня функція

**Приклад:**

class monstr; // *Попереднє оголошення класу*

**class hero**

**{**

**public:**

void kill(monstr &);

…

**}**

**class monstr**

**{**

**int ammo;**

**int health;**

…

**friend int steal\_ammo(monstr &);**

**friend void hero::kill(monstr &);**

// *Клас* hero *має бути визначений раніше*

**};**

…

int steal\_ammo(monstr &M){return --M.ammo;}

void hero::kill(monstr &M){M.health = 0; M.ammo = 0;}

У даному прикладі приведений опис двох функцій, дружніх класу **monstr**. Функція **kill** є методом класу **hero**, а функція **steal\_ammo** не належить жодному класу.

**Приклад** оголошення дружньої функцій для двох класів.

**#include <iostream.h>**

**class myclass1; //неповне оголошення класу**

**class myclass2**

**{**

**int d;**

**public:**

**myclass2 (int d1) { d=d1;}**

**// оголошення дружньої функції**

**friend bool isfactor(myclass1 ob1, myclass2 ob2);**

**};**

**class myclass1**

**{**

**int n;**

**public:**

**myclass1 ( int n1) {n=n1;}**

**// оголошення дружньої фукції**

**friend bool isfactor(myclass1 ob1, myclass2 ob2);**

**};**

**// визначення дружньої фукції**

**bool isfactor (myclass1 ob1, myclass2 ob2)**

**{**

**if (ob1.n % ob2.d ==0) return true;**

**else return false;**

**}**

**void main()**

**{**

**myclass1 ob1(12);**

**myclass2 ob2(3);**

**if (isfactor (ob1, ob2))**

**cout << " 12 ділиться без остачі на 3 \n";**

**else**

**cout << " 12 не ділиться без остачі на 3 \n";**

**}**

  Ця програма демонструє також важливий випадок застосування неповного оголошення класу: без застосування цієї конструкції в даному випадку було б неможливо оголосити дружню функцію для двох класів. Неповне оголошення класу myclass1 дає можливість використовувати його ім’я в оголошенні дружньої функції ще до його визначення.

Функція може бути членом одного класу і дружньою іншому.

Наприклад.

**#include <iostream.h>**

**class myclass1; //неповне оголошення класу**

**class myclass2**

**{**

**int d;**

**public:**

**myclass2 (int d1) { d=d1;}**

**bool isfactor(myclass1 ob1);**

**};**

**class myclass1**

**{**

**int n;**

**public:**

**myclass1 ( int n1) {n=n1;}**

**// оголошення дружньої фукції**

**friend bool myclass2 :: isfactor(myclass1 ob1);**

**};**

**// визначення дружньої фукції**

**bool myclass2::isfactor (myclass1 ob1)**

**{**

**if (ob1.n % d ==0) return true;**

**else return false;**

**}**

**void main()**

**{**

**myclass1 ob1(12);**

**myclass2 ob2(3);**

**//isfactor () викликається як функція-член класу myclass2**

**if (ob2.isfactor (ob1))**

**cout << " 12 ділиться без остачі на 3 \n";**

**else**

**cout << " 12 не ділиться без остачі на 3 \n";**

**}**

  Використання дружніх функцій потрібно по можливості уникати, оскільки вони порушують принцип інкапсуляції і, таким чином, ускладнюють відлагодження та модифікацію програми.

**Friend порушує інкапсуляцію даних !**

Щодо спадкування існують такі **правила**:

*Клас завжди має доступ до своїх (не успадкованих) членів* (та **дружні йому класси** також мають доступ). Специфікатори доступу впливають тільки на те, чи можуть об’єкти поза класом та батьківські класи звертатися до цих членів.

*Доступ до члену класу базується на його специфікаторі доступу.*

*Похідний клас має доступ до успадкованих членів батьківського класу на основі специфікатору доступу цих членів.*

Коли похідні класи успадковують члени батьківських класів, то члени батьківського класу можуть змінювати свої специфікатори доступу в похідному класі. Це не впливає на власні (не успадковані) члени похідних класів (які визначені в похідному класі і мають свої власні специфікатори доступу). Це впливає тільки на те, чи можуть об’єкти із-зовні та класи, похідні до наших похідних класів отримати доступ до успадкованих членів батьківського класу.

**Вирішення конфлікту імен**

При породженні одного [клас](http://moodle.ipo.kpi.ua/moodle/mod/glossary/showentry.php?courseid=508&concept=%D0%9A%D0%BB%D0%B0%D1%81)у з іншого можливі ситуації, коли ім'я елемента класу в похідному класі є таким же, як ім'я елемента в базовому класі. Якщо виник такий конфлікт, C++ завжди використовує елементи похідного класу усередині функцій похідного класу. Розглянемо приклад.

**#include <fstream>**

**#include <string>**

**#include <iostream>**

**using namespace std;**

**class book**

**{ public:**

**book (string a="C++", string b="Brest",**

**int c=100, float d=100.0)**

**{this->title = a; this->author = b;**

**this->pages = c; this->price = d;**

**cout << "constructor class book" << endl;}**

**~book() {cout << "destructor class book" << endl;};**

**protected:**

**string title;**

**string author;**

**int pages;**

**float price; // закупівельна ціна книги**

**};**

**class shop\_book: public book**

**{**

**public:**

**shop\_book(string \*,string \*,int,float);**

**void show\_shop\_book();**

**shop\_book (string a="C++ 2", string b="Brest 2",**

**int c=200, float d=200.0)**

**{this->title = a; this->author = b;**

**this->pages = c; this->price = d;**

**cout << "constructor class shop\_book " << endl;}**

**~ shop\_book () {cout << "destructor shop\_book " << endl;}**

**private:**

**float price; //** продажна ціна книги магазином

**};**

**void shop\_book::show\_shop\_book()**

**{**

**cout<<"title: "<<title<<endl;**

**cout<<"author:"<<author<<endl;**

**cout<<"pages: "<<pages<<endl;**

**cout<<"price: "<<book::price<<endl;**

**cout<<"price for sale: "<<price<<endl;**

**}**

**void main(){**

**shop\_book sb;**

**sb.show\_shop\_book();**

**system("pause");**

**}**

Результат

**constructor class book**

**constructor class shop\_book**

**title: C++ 2**

**author:Brest 2**

**pages: 200**

**price: 100**

**price for sale: 200**

**destructor shop\_book**

**destructor class book**

**Оголошення доступу using**

**class Base**

**{public:**

**void f();**

**void f(int n);**

**};**

**class Derived : private Base**

**{public:**

**Base::f; // Обидві функції Base::f() та Base::f(int) будуть публічними**

**};**

Тут буде попередження компілятора

warning: access declarations are deprecated in favour of using-declarations; suggestion: add the ‘using’ keyword [-Wdeprecated]

**class Base**

**{public:**

**void f();};**

**class Derived : private Base**

**{ public:**

**void f(int n);**

**Base::f; }; // Помилка**

Доступ до члену базового класу не може бути скориговано в похідному класі, якщо в ньому, в той же час, визначається член з тим же ім'ям.

Доступ до члену базового класу в похідних класах можна змінити, згадавши його ім'я в похідному класі (використанням ключового слова using або без нього). Такий запис називається оголошення доступу.

**class Base**

**{public:**

**int n;**

**};**

**class Derived : private Base**

**{ public:**

**Base::n; };**

**// За замовчуванням n був би приватним членом класу Derived**

**Приховування імен**

**class Base {**

**private:**

**int x;**

**public:**

**void mf1();**

**void mf1(int);};**

**class Derived : public Base {**

**public:**

**void mf1();};**

**void main{**

**int x;**

**Derived d;**

**d.mf1(); // викликається Derived::mf1**

**d.mf1(x); // помилка! Derived::mf1 приховує Base::mf1(double)**

**…}**

**Відкриття доступу**

**class Base {**

**private:**

**int x;**

**public:**

**void mf1();**

**void mf1(int);};**

**class Derived : public Base {**

**public:**

**using Base::mf1;**

**void mf1();};**

**int x;**

**d.mf1(); // викликається Derived::mf1**

**d.mf1(x); // викликається Base::mf1(int)**

Використання “using-оголошенню”

Мова C++ надає можливість змінити специфікатор доступу батьківського члена в дочірньому класі. Це робиться за допомогою [**“using-оголошення”**](https://acode.com.ua/urok-57-stejtmenty-using/#toc-0).

У мові C++ неможливо видалити або обмежити функціонал батьківського класу, крім як за допомогою безпосередньої зміни вихідного коду. Однак в дочірньому класі ви можете приховати функціонал, який існує в батьківському класі, так щоб до нього не можна було отримати доступ через об’єкти дочірнього класу. Це робиться шляхом зміни відповідних специфікаторів доступу.

Наприклад, ви можете зробити закритим відкритий член батьківського класу:

**#include <iostream>**

**class Parent**

**{**

**public:**

**int m\_value;**

**};**

**class Child : public Parent**

**{**

**private:**

**using Parent::m\_value;**

**public:**

**Child(int value)**

**// Ми не можемо ініціалізувати m\_value, оскільки це член**

**//класу Parent (Parent повинен ініціалізувати m\_value)**

**{**

**// Але ми можемо присвоїти значення**

**m\_value = value;**

**}**

**};**

**int main()**

**{**

**Child child(9);**

**Parent par;**

**par.m\_value=10;**

**std::cout << par.m\_value;**

**// Наступне не спрацює, оскільки m\_value був**

**// перевизначений як private**

**//std::cout << child.m\_value;**

**return 0;**

**}**

Результат – 10

Якщо розкрити std::cout << child.m\_value;

Компілятор видасть повідомлення

error: ‘int Parent::m\_value’ is inaccessible within this context

Це дозволяє інкапсулювати дані батьківського класу в дочірньому класі. В якості альтернативи можна використати спадкування типу private, що призведе до того, що всі успадковані public- і protected-члени класу Parent стануть private в класі Child.

**Правила відкриття/приховування імен**

Вживання того ж самого імені функції у похідному класі приховує **всі** реалізації цієї функції у базовому.

[**Не приховуйте успадковані імена**](http://cpp.com.ru/meyers_1/ch6.html) **(Правило 33: Майєрс)**

Директива using дозоляє змінити видимість окремих функцій і членів базового класу, однак змінити видимість приватних даних неможливо.

**2 види наслідування С++**

**Наслідування реалізації**

* + всі члени крім стандартних базових типів стають членами похідного типу
  + функціональність базового і похідного типів може відрізнятись при однакових назвах методів

**Наслідування інтерфейсів**

* + базові класи – абстрактні (pure methods)
  + у похідному – реалізація чистих методів
  + фактично можна використати лише функціональність, яка реалізована у похідному типі
  + немає спеціальної конструкції для інтерфейсу

**Наслідування реалізації**

* всі члени (дані і методи) крім стандартних (конструктори, деструктор, присвоєння) базових типів стають членами похідного типу
* специфікатори керують лише доступом до членів
* виклик конструкторів усіх базових типів згідно списку наслідування зліва направо
* виклик деструкторів базових типів у порядку, зворотному до виклику конструкторів
* функціональність базових типів у похідних типах може змінюватися за рахунок перевизначення методів і поліморфізму
* усунення неоднозначності виклику методів:
  + завдання області видимості конкретного класу
  + приведення типу

Розглянемо приклад відкритого успадкування з внутрішнім класом.

#include <stdio.h>

#include <iostream>

#include <Windows.h>

using namespace std;

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()

{system("color F0");

TBase first;

TDerived second;

TDerived2 third;

first.printBase();

second.printDerived();

third.printDerived2();

printf("Sizeof(TInner) = %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;

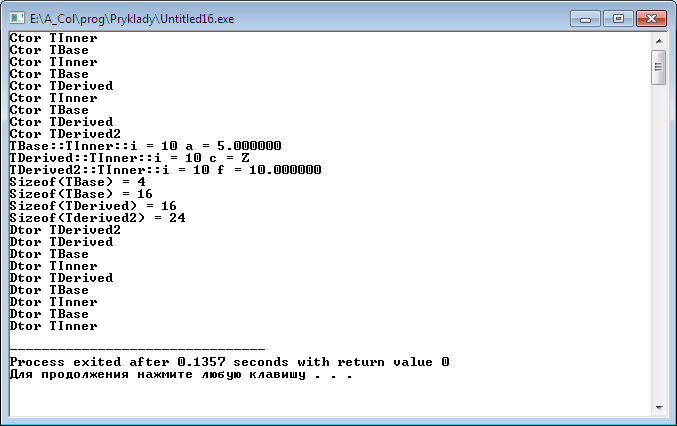
}

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.

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 їх 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 успадкуванні”, 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();

**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 значенні 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 TfDerived стало неправильним. Його функція-член printDerived() 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.

**Захищене успадкування**

Відмінність специфікатор доступу protected від специфікатора private виявляється тільки при успадкуванні класів. у всьому 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;

}

***Контрольні запитання*.**

1. Які типи відношень між класами Ви знаєте?
2. Як пов’язані відношення агрегації і композиції?
3. Наведіть приклад відношення узагальнення.
4. Визначте основні концепції наслідування.
5. В чому полягає принцип підстановки?
6. Які форми наслідування Ви можете визначити?
7. В чому полягає просте спадкування?
8. Як визначається поведінка конструкторів та деструкторів при спадкуванні?
9. Як спадкування подано в C++?
10. Що таке публічне, захищене й приватне спадкування?
11. Що таке клас-нащадок і базовий клас?

***Для самостійного вивчення*:** Поглибити матеріал лекції за наданою літературою. Вивчення лекційного матеріалу та додаткових джерел. Розгляд запитань і виконання завдань для самостійної роботи, запропонованих на лекції.

**Література**

1. Павловская Т.А. С/С++. Программирование на языке высокого уровня. СПб.: Питер, 2003. – 461 с. URL: <http://www.ph4s.ru/bookprogramir_1.html>
2. Страуструп Б. Язык программирования С++. — СПб.; М.: Невский диалект — ЗАО “Изд-во “Бином”, 1999.
3. Джейс Либерти Освой самостоятельно С++ за 21 день: 3-е изд. пер. с англ.: Уч. пос. – М.: Издательский дом „Вильямс”, 2001. – 816 с.: ил..

4. Липпман С. Б., Лажойе Ж. Язык программирования С++: Вводный курс. — М.: ДМК, 2001. URL: <http://www.insycom.ru/html/metodmat/inf/Lipman.pdf>

5. Дейтел Х., Дейтел П. Основы программирования на С++. – М.: Бином, 1999. – 1024 с. URL: <http://ijevanlib.ysu.am/wp-content/uploads/2018/03/deytel.pdf>

6. Буч Г. Объектно-ориентированный анализ и проектирование с примерами приложений на С++ . [2-е изд.] / Буч Г. - СПб.: Невский Диалект, 1998. - 560 с.

7. Бадд Т. Объектно-ориентированное программирование в действии. [2-е изд.] – СПб.: Изд-во "Питер". 1997. URL: <http://khizha.dp.ua/library/Timothy_Budd_-_Introduction_to_OOP_(ru).pdf>

8. Скотт Мейерс*.* Эффективное использование C++. 50 рекомендаций по улучшению ваших программ и проектов. *"ДМК", 2000; 240 с.*