**Лекція 22. Абстрактні класи. Прийоми програмування**

**Успішне застосування поліморфізму**

Для успішного застосування поліморфізму потрібно розуміти, що базовий і похідний класи утворюють ієрархію, розвиток якої спрямований від більшого до меншого ступеня узагальнення (тобто від базового класу до похідного). У разі коректної реалізації базовий клас забезпечує всі елементи, які похідний клас може використовувати безпосередньо. Він також визначає функції, які похідний клас повинен реалізувати самостійно. Це дає похідному класу гнучкість у визначенні власних методів, але водночас зобов'язує використовувати загальний інтерфейс. Іншими словами, оскільки формат інтерфейсу визначається базовим класом, то будь-який похідний клас повинен розділяти цей загальний інтерфейс. Таким чином, застосування віртуальних функцій дає змогу базовому класу визначати узагальнений інтерфейс, який використовуватиметься всіма похідними класами.

Важливість загального інтерфейсу з множиною реалізацій полягає в тому, що такий інтерфейс дає змогу програмісту справлятися із наростаючою складністю програм. Наприклад, якщо коректно розробити програму, то можна упевнитися в тому, що до всіх об'єктів, виведених з базового класу, можна буде отримати доступ єдиним (загальним для всіх) способом, незважаючи на те, що конкретні дії одного похідного класу можуть відрізнятися від дій іншого. Це означає, що програмісту доведеться пам'ятати тільки один інтерфейс, а не велику їх кількість. Окрім цього, похідний клас має можливість використовувати будь-які або всі функції, надані базовим класом. Іншими словами, розробнику похідного класу не потрібно наново винаходити елементи, які вже є в базовому класі. Понад це, від'єднання інтерфейсу від реалізації дає змогу створювати бібліотеки класів, написанням яких можуть займатися сторонні організації. Коректно реалізовані бібліотеки повинні надавати загальний інтерфейс, який програміст може використовувати для виведення похідних класів відповідно до своїх конкретних потреб.

**Поняття про суто віртуальні функції та абстрактні класи**

Як було зазначено раніше, якщо віртуальна функція, яка не перевизначена у похідному класі, викликається об'єктом цього похідного класу, то використовується та її версія, яку було визначено в базовому класі. Але у багатьох випадках взагалі немає сенсу давати визначення віртуальної функції в базовому класі. У тому, що віртуальна функція не має конкретного визначення у базовому класі, немає нічого незвичайного, це просто заглушка.

Існує два способи оброблення таких ситуацій. Перший полягає у виведенні функцією застережного повідомлення. Можливо, такий підхід і буде корисний у певних ситуаціях, але здебільшого він просто неприйнятний. Наприклад, можна уявити собі віртуальні функції, без визначення яких у існуванні похідного класу взагалі немає ніякого сенсу. У мові програмування C++ для вирішення цього питання і передбачено суто віртуальні функції, які розглядали на минулій лекції.

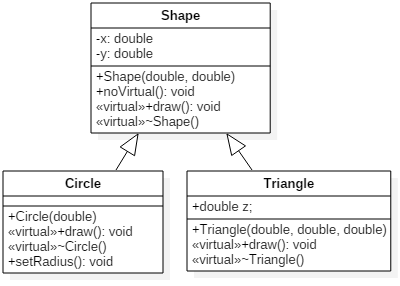
*Суто віртуальна функція –* це віртуальна функція, яку оголошено в базовому класі, але вона не має у ньому ніякого визначення. Тому будь-який похідний тип повинен визначити власну версію цієї функції, адже у нього просто немає ніякої можливості використовувати версію з базового класу (через її відсутність). Щоб оголосити суто віртуальну функцію, використовують такий загальний формат:

**virtual** *тип ім'я\_функції* (*перелік\_параметрів*) = 0;

Таку функцію також називають відкладеним методом, розглядаючи як окремий випадок перевизначення, коли метод базового класу не має реалізації, а будь-яка корисна діяльність задається в методі дочірнього класу

**Абстрактний клас** характеризує одна важлива особливість: у такого класу не може бути об'єктів. Абстрактний клас можна використовувати тільки як базовий, з якого виводитимуться інші похідні класи. Причина того, що абстрактний клас не можна використовувати для побудови об'єктів, полягає, безумовно, у тому, що його одна або декілька функцій не мають визначення. Але навіть якщо базовий клас є абстрактним, то його все одно можна використовувати для оголошення покажчиків і посилань, які необхідні для підтримки динамічного поліморфізму.

Розглянемо більш детально поліморфний базовий клас з віртуальним методом draw() .



Для звичайного об’єкту застосовується статичне зв’язування

**class Shape**

**{ double x, y;**

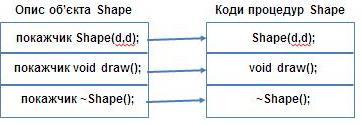
**public:**

**Shape(double\_x = 0, double\_y = 0) :x(x), y(y){};**

**void draw(){puts("Shape");}**

**~Shape(){ printf("~Shape\n"); };**

**};**



Порівняємо з виконанням динамічного зв’язування для віртуальних об’єктів.

**class Shape**

**{ double x, y;**

**public:**

**Shape(double\_x = 0, double\_y = 0) :x(x), y(y){};**

**virtual void draw(){puts("Shape");}**

**virtual ~Shape(){ printf("~Shape\n"); };**

**};**



Зарезервоване cлово virtual вказує компілятору на необхідність побудови таблиці віртуальних правил VTM (virtual method table), що міститиме адреси таких функцій для даного класу. Кожен представник класу з віртуальною функцією містить покажчик VPTR (virtual pointer) на його таблицю віртуальних методів. На етапі компіляції на початок конструктору автоматично вставляється покажчик VPTR на його таблицю віртуальних правил VTM. Аналогічні дії виконуються і для нащадків.

**class Triangle : public Shape**

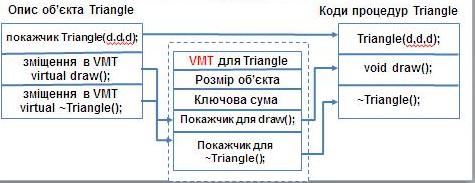
**{double z;**

**public:**

**Triangle(double\_x,double\_y,double\_z):Shape(\_x,\_y),z(\_z){};**

**virtual void draw(){puts("Triangle");}**

**virtual ~Triangle(){ printf("~Triangle\n"); };};**



Адреса кожної віртуальної функції має одне й те ж саме зміщення в таблицях VTM кожного класу конкретної ієрархії.

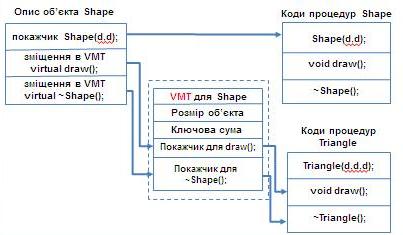
Пошук віртуального метода у VMT.

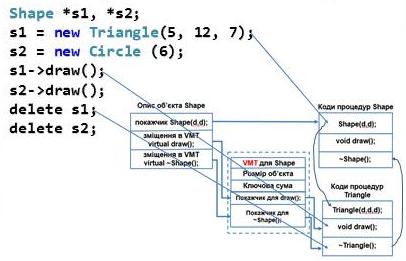
При активації віртуальних методів згенерований код спочатку знаходить покажчик VPTR на таблицю VTM, а потім з таблиці вибирає адресу віртуальної функції і нарешті виконує безпосередній виклик цієї функції. Віртуальний механізм працює лише за допомогою покажчиків (посилань) на об’єкти.

Продовжимо розгляд прикладу.

**Shape \*s;**

**s = new Triangle(5, 12, 7);**





Об’єкт, який віртуальну функцію та визначений через покажчик або посилання, носить назву **поліморфного**. У даному випадку поліморфізм полягає в тому, що за допомогою одного й того ж звертання до методу виконуються різні дії в залежності від типу, на який посилається покажчик в даний момент часу.

**Віртуальний деструктор.**

Якщо у класі є хоча б один віртуальний метод, то деструктор такого класу повинен бути віртуальним. При виклику через покажчик лише віртуальний деструктор забезпечує виклик деструктора базового класу і вірне очищення пам’яті. Розглянемо конкретний приклад.

**#include <iostream>**

**#include <stdio.h>**

**#include <string.h>**

**#include <Windows.h>**

**using namespace std;**

**class TBase**

**{**

**public:**

**char\* a;**

**TBase(const char\* s){a = strdup(s); printf("Ctor TBase\n");}**

**// virtual ~TBase(){delete a; printf("Dtor 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");}**

**// virtual ~TDerived(){delete b; printf("Dtor 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()**

**{ system("color F0");**

**TBase\* pObj = new TDerived2("TBase", "TDerived","TDerived2");**

**delete pObj; // Помилка!**

**system("pause");**

**return 0;**

**}**

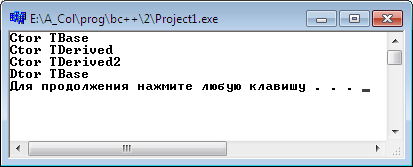
Клас TBase містить один рядок. Його конструктор виділяє для неї стільки пам'яті, скільки займає константний аргумент s, а його деструктор звільняє пам'ять, зайняту рядком a. Клас TDerived успадковує цей рядок із класу TBase і додає свою. Його конструктор спочатку викликає конструктор класу TBase, ініціалізуючи 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.

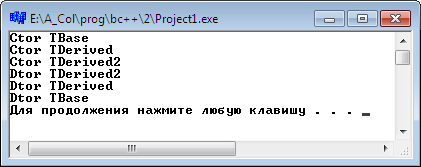
Уся ця конструкція працює цілком надійно, поки програма не спробує знищити вказівник базового класу pObj, що посилається на об'єкт похідного класу; у даному випадку вказівник посилається на об'єкт objDerived2 класу TDerived2.

Програма виводить на екран наступні рядки.



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.

Якщо розкоментувати віртуальні деструктори і закоментувати не віртуальні, то отримаємо результат.

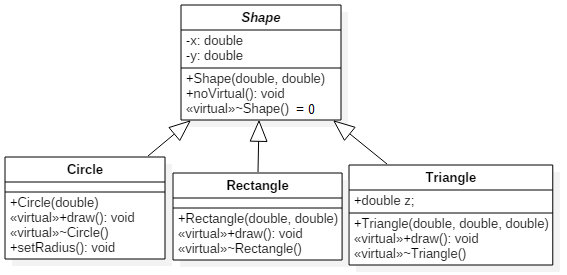


Тепер пам'ять звільнена коректно.

**Поліморфний контейнер та абстрактний клас**

Серед поліморфних змінних можна виділити **поліморфні контейнери.** Поліморфним називається контейнер, при оголошення якого використовується статичний тип, а зберігаються у ньому елементи динамічного типу. Найчастіше поліморфними контейнерами виступають абстрактні базові класи.

Розглянемо абстрактний базовий клас.



**class Shape**

**{ public:**

**virtual ~Shape() = 0;}; // Чистий віртуальний метод**

**Shape \*s[10]; // поліморфний контейнер**

**Приклад поліморфного контейнера**

**Shape \*s[3];**

**s[0] = new Circle(10);**

**s[1] = new Rectangle(1, 8);**

**s[2] = new Triangle(5, 12, 7);**

**for (int i = 0; i < 3; i++)**

**s[i]->draw();**

**for (int i = 0; i < 3; i++)**

**delete s[i];**

**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 <iostream>**

**#include <stdio.h>**

**#include <string.h>**

**#include <Windows.h>**

**using namespace std;**

**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()**

**{ system("color F0");**

**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;**

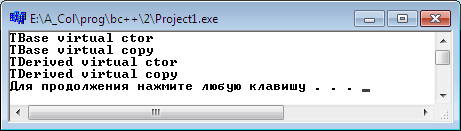
**system("pause");**

**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.



**Перетворення статичного типу до динамічного в мові С ++**

Для приведення статичного типу до динамічного рекомендується використовувати два оператора мови С ++

**dynamic\_cast <type-id>(expression)**

**static\_cast <type-id>(expression)**

Більш безпечно використовувати оператор dynamic\_cast, але він працює тільки з поліморфними типами. Головне достоїнство зазначених операторів полягає в тому, що вони повертають NULL (для покажчиків), якщо приведення неможливо.

**Інформація про тип на етапі виконання.**

Перший підхід заснований на використанні оператора динамічного приведення типу dynamic\_cast. Розглянемо програму.

**#include <iostream>**

**#include <stdio.h>**

**#include <string.h>**

**#include <Windows.h>**

**using namespace std;**

**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()**

**{ system("color F0");**

**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");**

**system("pause");**

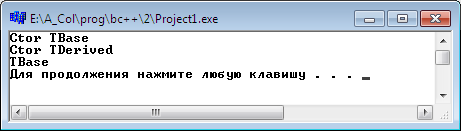
**return 0;**

**}**

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 іншій. До класів, що не містять віртуальних функцій, оператор dynamic\_cast не застосовується.

**Понижуюче, підвищувальне і перехресне приведення**

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

**#include <iostream>**

**#include <stdio.h>**

**#include <string.h>**

**#include <Windows.h>**

**using namespace std;**

**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()**

**{ system("color F0");**

**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");**

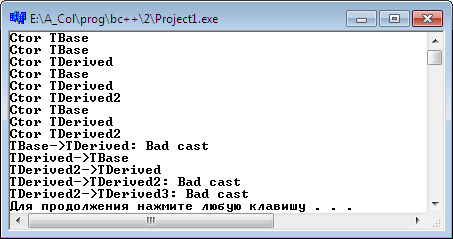
**else printf("TDerived2->TDerived3: Bad cast\n");**

**system("pause");**

**return 0;**

**}**

Програма виводить на екран наступні рядки.



Як бачимо, приведення, що знижують, і перехресні приведення не виконані. 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.

Оператор **typeid**

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 С++ передбачений оператор **typeid**.

Цей оператор повертає посилання на об'єкт класу type\_info, визначеного в заголовку <typeinfo> (потрібно встановити #include <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 практично залишаються незмінними, ми приведемо лише функцію main().

**int main()**

**{**

**TBase objBase("TBase");**

**TBase\* pObj1=&objBase;**

**TDerived objDerived("TBase", "TDerived");**

**pObj1 = &objDerived;**

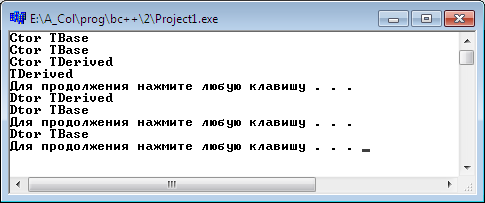
**if(typeid(\*pObj1)==typeid(TDerived)) printf("TDerived\n");**

**else printf("Something else ...");**

**return 0;**

**}**

Програма виводить на екран наступні рядки



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.

**Операції та методи, що підтримує структура type\_info**

|  |  |  |
| --- | --- | --- |
| **operator ==** | **порівнює два типи даних** | **class A a; class B b;**  **if(typeid(a) == typeid(b)) { puts("Type are equivalent"); } else { puts("Type are not equivalent"); }** |
| **operator !=** | **порівнює два типи даних** | **class A a; class B b;**  **if(typeid(a) != typeid(b)) { puts("Type are not equivalent"); } else { puts("Type are equivalent"); }** |
| **name()** | **повертає ім'я класу у вигляді текстового рядка** | **class A a; class B b;**  **puts(typeid(a).name()); //“class A” puts(typeid(b).name()); // “class B”** |

#### static\_cast – унарна операція приведення типів даних в С++

Операція static\_cast може бути використана для перетворення одного типу в інший, але вона не повинна бути використана для виконання неприпустимого перетворення, наприклад, перетворення значення в покажчик або навпаки. Рекомендується користуватися операцією static\_cast, ніж Cі-стилем ((type) val) приведення, тому що static\_cast обмежує неприпустиме приведення типів і, отже – безпечніше.

Операція static\_cast, грубо кажучи, – це шаблон функції, в якій необхідно явно вказати тип даних для перетворення, тобто задати параметр шаблону.

static\_cast<dataType>(value);

Отже, в трикутних дужках вказується тип даних, до якого необхідно перетворити значення value, яке стоїть в круглих дужках. Приклад, який призводить int до типу double для того, щоб при діленні 13-ти на 7 уникнути усічення результату через цілочисельного ділення.

|  |  |
| --- | --- |
|  | double res = static\_cast<double>(13)/7; |

Перетворення, що виконуються за допомогою static\_cast, не є типобезпечними, оскільки перевірка типів відбувається тільки на етапі компіляції; під час виконання можуть приводитися значення несумісних типів.

**Поняття про поліморфізм і пуризм**

*Статичний поліморфізм* (поліморфізм часу компілювання) реалізується у перевизначені функцій і операторів. *Динамічний поліморфізм* (поліморфізм тривалості виконання програми) досягається за рахунок віртуальних функцій. Найзагальніше визначення поліморфізму поміщене у фразі "один інтерфейс, багато методів", і всі згадані вище "знаряддя" поліморфізму відповідають цьому визначенню. Проте під час використання самого терміну *поліморфізм* все ж таки існують деякі розбіжності.

Деякі пуристи (у цьому випадку – борці за чистоту термінології) об'єктно-орієнтованого програмування наполягають на тому, щоб цей термін використовувався тільки для подій, які відбуваються у процесі виконання програм. Вони стверджують, що поліморфізм підтримується тільки віртуальними функціями. Частково ця точка зору ґрунтується на тому факті, що найпершими поліморфічними мовами програмування були інтерпретатори (для них характерним є те, що всі події належать тривалості виконання програми). Поява трансльованих поліморфічних мов програмування розширила концепцію реалізації поліморфізму. Однак все ще не утихають заяви про те, що термін *поліморфізм* повинен застосовуватися виключно до подій періоду виконання програми. Більшість С++-програмістів е погоджуються з цією точкою зору і вважають, що цей термін можна застосовувати до обох видів засобів. Тому не потрібно дивуватися, якщо хтось раптом стане сперечатися на предмет використання цього терміну!

*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. Джейс Либерти Освой самостоятельно С++ за 21 день: 3-е изд. пер. с англ.: Уч. пос. – М.: Издательский дом „Вильямс”, 2001. – 816 с.: ил..
2. Павловская Т.А. С/С++. Программирование на языке высокого уровня. СПб.: Питер, 2003. – 461 с. URL:  <http://www.ph4s.ru/bookprogramir_1.html>
3. Липпман С. Б., Лажойе Ж. Язык программирования С++: Вводный курс. — М.: ДМК, 2001. URL: <http://www.insycom.ru/html/metodmat/inf/Lipman.pdf>
4. 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>
5. 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>.
6. *Герб Саттер,Андрей Александреску* "Стандарты программирования на С++ ", [*Вильямс, 2005*](http://www.williamspublishing.com/)*; 304 с.*
7. *Скотт Мейерс.* Эффективное использование C++. 50 рекомендаций по улучшению ваших программ и проектов. *"ДМК", 2000; 240 с.*
8. *Скотт Мейерс.* Наиболее эффективное использование C++. 35 новых рекомендаций по улучшению ваших программ и проектов.*"ДМК",2000;304 с.*

*Контрольні запитання для самоперевірки*.

1. Який механізм дозволяє створювати ієрархії класів?
2. Які базові класи називаються віртуальними?
3. Що таке поліморфічний кластер?
4. Які класи називаються абстрактними?
5. Що таке таблиця віртуальних функцій?
6. Коли застосовується віртуальний деструктор
7. Опишіть оператор dynamic\_cast.
8. Опишіть три види приведення за допомогою оператора dynamic\_cast в ієрархії класів.
9. Опишіть оператор typeid.

*Контрольні запитання для надання письмових відповідей*.

1. Назвіть види успадкування залежно від кількості базових класів..
2. Опишіть механізм успадкування.
3. Чим динамічний поліморфізм відрізняється від статичного?
4. Що станеться, якщо віртуальна функція не заміщується у якомусь похідному класі?