Лекція 10: Перетворення типів

Неявне перетворення

Неявні перетворення здійснюються автоматично, коли значення копіюється на сумісний тип. Наприклад:
short a=2000;
int b;
b=a;

Тут значення a підвищується від короткого до int без необхідності явного оператора. Це відомо як стандартне перетворення. Стандартні перетворення впливають на основні типи даних і дозволяють перетворювати між числовими типами (короткий до int, int для плавання, подвійний до int ...), до або з bool та деякими перетвореннями вказівників.

Перетворення в int з якогось меншого цілого типу, або в подвійне з float, відоме як просування, і гарантується, що воно буде точно таким же значенням у типі призначення. Інші перетворення між арифметичними типами не завжди можуть точно представляти одне значення:
Якщо негативне ціле значення перетворюється в беззнаковий тип, то результуюче значення відповідає побітовий зміні його перешого біту до 1 (тобто -1 стає найбільшим значенням, що представляється типом, -2 другим за величиною, ...).

• Перетворення з / до bool вважають false еквівалент нуля (для числових типів) та нульовий вказівник (для типу вказівника); true еквівалентно всім іншим значенням і перетворюється в еквівалент 1.

• Якщо перетворення відбувається від типу з плаваючою точкою до цілочисельного типу, значення обрізається (десяткова частина видаляється). Якщо результат лежить за межами діапазону репрезентативних значень за типом, перетворення викликає невизначену поведінку.

• В іншому випадку, якщо перетворення відбувається між числовими типами одного і того ж виду (цілочисельне або ціле число або плаваюче-плаваюче), перетворення є дійсним, але значення є специфічним для реалізації (і може бути не портативним).

Деякі з цих перетворень можуть означати втрату точності, про що компілятор може сигналізувати з попередженням. Цього попередження можна уникнути за допомогою явного перетворення.

Для не фундаментальних типів масиви та функції неявно перетворюються на

вказівники, а вказівники взагалі дозволяють проводити такі перетворення:
• Нульові вказівники можуть бути перетворені в вказівники будь-якого типу
• Вказівники на будь-який тип можуть бути перетворені в вказівники на void
• Перетворення вказівників: вказівники на похідний клас можуть бути перетворені на вказівник доступного та однозначного базового класу, не змінюючи його const або volatile специфікації.

Неявні перетворення з класами
У світі класів неявні перетворення можна керувати за допомогою трьох функцій-членів:
• Конструктори з одним аргументом: дозволяють неявне перетворення з певного типу для ініціалізації об'єкта.
• Оператор присвоєння: дозволяє непряме перетворення з певного типу на завдання.
• Оператор-перетворення типів: дозволяє непряме перетворення до певного типу.

```
// implicit conversion of classes:
#include <iostream>
using namespace std;

class A {};

class B {
public:
  // conversion from A (constructor):
  B (const A& x) {}
  // conversion from A (assignment):
  B& operator= (const A& x) {return *this;}
  // conversion to A (type-cast operator)
  operator A() {return A();}
};

int main ()
{
  A foo;
  B bar = foo;    // calls constructor
  bar = foo;      // calls assignment
  foo = bar;      // calls type-cast operator
  return 0;
}
```

Оператор type-cast operator використовує певний синтаксис: він використовує ключове слово оператора, за яким слід тип призначення та порожній набір дужок. Зауважте, що тип повернення - тип призначення, і тому він не вказується перед ключовим словом оператора.

ключове слово explicit

Під час виклику функції C ++ дозволяє здійснювати одне неявне перетворення для кожного аргументу. Це може бути дещо проблематичним для занять, адже це не завжди те, що призначено. Наприклад, якщо до останнього прикладу додамо таку функцію:
void fn (B arg) {}

Ця функція бере аргумент типу B, але її також можна назвати об'єктом типу A як аргумент:
fn (foo);
Це може бути або не бути тим, що було призначено. Але, у будь-якому випадку, це можна запобігти, позначивши порушений конструктор ключовим словом explicit:

```
// explicit:
#include <iostream>
using namespace std;

class A {};

class B {
public:
  explicit B (const A& x) {}
  B& operator= (const A& x) {return *this;}
  operator A() {return A();}
};

void fn (B x) {}

int main ()
{
  A foo;
  B bar (foo);
  bar = foo;
  foo = bar;

//  fn (foo);  // not allowed for explicit ctor.
  fn (bar);

  return 0;
}
```

Крім того, конструктори, позначені *explicit*, не можуть бути викликані синтаксисом, подібним до призначення; У наведеному вище прикладі bar не міг бути побудований із:
B bar = foo;

Функції членів типу (ті, що описані в попередньому розділі) також можна вказати як явні. Це запобігає неявним перетворенням так само, як це роблять чітко визначені конструктори для типу призначення.

Перетворення типів

C ++ - мова сильного типу. Багато конверсій, особливо ті, що передбачають різну інтерпретацію значення, вимагають явного перетворення, відомого в C ++ як типовий кастинг. Існує два основні синтаксиси для загального кастингу типу: функціональний та c-подібний:

```
double x = 10.3;
int y;
y = int (x);    // functional notation
y = (int) x;    // c-like cast notation
```

Функціональність цих загальних форм кастингу типів достатня для більшості потреб із основними типами даних. Однак ці оператори можуть бути застосовані без розбору на класи та вказівники на класи, що може призвести до коду, який, в той час як синтаксично правильний, - може спричинити помилки під час виконання. Наприклад, такий код компілюється без помилок:

```
// class type-casting
#include <iostream>
using namespace std;

class Dummy {
   double i,j;
};

class Addition {
   int x,y;
 public:
   Addition (int a, int b) { x=a; y=b; }
   int result() { return x+y;}
};

int main () {
  Dummy d;
```

```
  Addition * padd;
  padd = (Addition*) &d;
  cout << padd->result();
  return 0;
}
```

Програма оголошує вказівник на Addition, але потім присвоює йому посилання на об'єкт іншого неспорідненого типу за допомогою явного кастингу типу:
padd = (Addition*) &d;

Неограниченное явне введення типів дозволяє перетворити будь-який вказівник в будь-який інший тип вказівника, незалежно від типів, на які вони вказують. Подальший виклик результату учасника призведе до помилки під час виконання або інших несподіваних результатів.

Для того, щоб контролювати такі типи перетворень між класами, у нас є чотири конкретні оператори кастингу: динамический_cast, реінтерпрет_каст, статичний_каст і const_cast. Їх формат повинен відповідати новому типу, укладеному між кутовими дужками (<>) та одразу після цього, виразом для перетворення між дужками.

- dynamic_cast <new_type> (expression)
- reinterpret_cast <new_type> (expression)
- static_cast <new_type> (expression)
- const_cast <new_type> (expression)

Традиційними перетвореннями типів (type-casting) будуть:

- (new_type) expression
- new_type (expression)

static_cast

static_cast може здійснювати перетворення між вказівниками на пов’язані класи, не тільки перетворення вверх (від вказівника до похідного класу до вказівника на базовий тип), але і вниз (від вказівника до базового класу до вказівника на похідний клас). Жодні перевірки не виконуються під час виконання, щоб гарантувати, що об’єкт, який перетворюється, насправді є повноцінним об'єктом типу призначення. Тому програміст повинен забезпечити безпеку перетворення. З іншого боку, він не несе накладних витрат на перевірку безпеки типу динамічних передач.

```
class Base {};
class Derived: public Base {};
Base * a = new Base;
Derived * b = static_cast<Derived*>(a);
```

Це був би коректний код, хоча b вказував би на незавершений об'єкт класу та може призвести до помилок виконання під час дереференції.

Отже, static_cast може виконувати за допомогою вказівників класів не тільки конверсії, дозволені неявно, але й їх протилежні перетворення.

Також static_cast може виконувати всі перетворення, дозволені неявно (не тільки ті, що мають вказівники на класи), а також здатні виконувати протилежні від них. Зокрема:

• Перетворити з void * на будь-який тип вказівника. У цьому випадку гарантується, що якщо значення void * було отримано шляхом перетворення з того самого типу вказівника, результуюче значення вказівника буде однаковим.

• Перетворювати цілі, значення з плаваючою комою та перерахування на типи перерахування.

Крім того, static_cast також може виконувати наступне:

• Явно викликати конструктор з одним аргументом або оператор перетворення.

• Перетворити на посилання rvalue.

• Перетворення значень класу перерахування в цілі або значення з плаваючою комою.

• Перетворити будь-який тип на void, оцінюючи та відкидаючи значення.

Таким чином, static_cast здатний виконувати з вказівниками на класи не тільки перетворення, що допускаються неявно, але і їх протилежні перетворення.

reinterpret_cast

reinterpret_cast перетворює будь-який тип вказівника в будь-який інший тип вказівника, навіть неспоріднених класів. Результат операції - це проста двійкова копія значення з одного вказівника на інший. Дозволені всі перетворення вказівників: не вказується ані зміст, ані сам тип вказівника.

Він також може наводити вказівники на цілі типи або з них. Формат, у якому це ціле значення представляє вказівник, залежить від платформи. Єдиною гарантією є те, що вказівник, переданий на цілий тип, достатньо великий, щоб повністю містити його (наприклад, intptr_t), гарантовано зможе повернутися до дійсного вказівника.

Конверсії, які можна виконати reinterpret_cast, але не static_cast, - це операції низького рівня, засновані на реінтерпретації бінарних представлень типів, що

в більшості випадків призводить до коду, який є системним і, таким чином, не портативним. Наприклад:

```
class A { /* ... */ };
class B { /* ... */ };
A * a = new A;
B * b = reinterpret_cast<B*>(a);
```

Цей код компілюється, хоча це не має особливого сенсу, оскільки тепер b вказує на об'єкт абсолютно неспорідненого і, ймовірно, несумісного класу. Таке перетворення b небезпечно і компівлятор в цьому випадку не несе ніякої гарантії що перетворення відбулося коректно насправді.

const_cast

Цей тип кастингу маніпулює константністю об'єкта, вказуваної вказівником, тобото може встановити його, або навпаки видалити. Наприклад, для передачі вказівника const на функцію, яка очікує аргумент non-const:

```
// const_cast
#include <iostream>
using namespace std;

void print (char * str)
{
  cout << str << '\n';
}

int main () {
  const char * c = "sample text";
  print ( const_cast<char *> (c) );
  return 0;
}
```

Результат:

```
sample text
```

Наведений вище приклад гарантовано спрацьовує, оскільки функція друку не пише на вказаний об'єкт..

typeid

typeid дозволяє перевіряти тип виразу:

```
typeid (expression)
```

Цей оператор повертає посилання на постійний об'єкт типу type_info, який визначений у стандартному заголовку <typeinfo>. Значення, повернене typeid, можна порівняти з іншим значенням, поверненим typeid за допомогою операторів == і! =, Або може служити для отримання нульової закінченої

послідовності символів, що представляє тип даних або ім'я класу, використовуючи його name () член.

```
// typeid
#include <iostream>
#include <typeinfo>
using namespace std;

int main () {
  int * a,b;
  a=0; b=0;
  if (typeid(a) != typeid(b))
  {
    cout << "a and b are of different types:\n";
    cout << "a is: " << typeid(a).name() << '\n';
    cout << "b is: " << typeid(b).name() << '\n';
  }
  return 0;
}
```

Результат:

```
a and b are of different types:
a is: int *
b is: int
```

Коли typeid застосовується до класів, typeid використовує RTTI для відстеження типу динамічних об'єктів. Коли typeid застосовується до виразу, тип якого є поліморфним класом, результатом є тип найбільш похідного завершеного об'єкта:

```
// typeid, polymorphic class
#include <iostream>
#include <typeinfo>
#include <exception>
using namespace std;

class Base { virtual void f(){} };
class Derived : public Base {};

int main () {
  try {
    Base* a = new Base;
    Base* b = new Derived;
    cout << "a is: " << typeid(a).name() << '\n';
    cout << "b is: " << typeid(b).name() << '\n';
    cout << "*a is: " << typeid(*a).name() << '\n';
    cout << "*b is: " << typeid(*b).name() << '\n';
  } catch (exception& e) { cout << "Exception: " << e.what() << '\n'; }
```

```
  return 0;
}
```

Результат:

```
a is: class Base *
b is: class Base *
*a is: class Base
*b is: class Derived
```

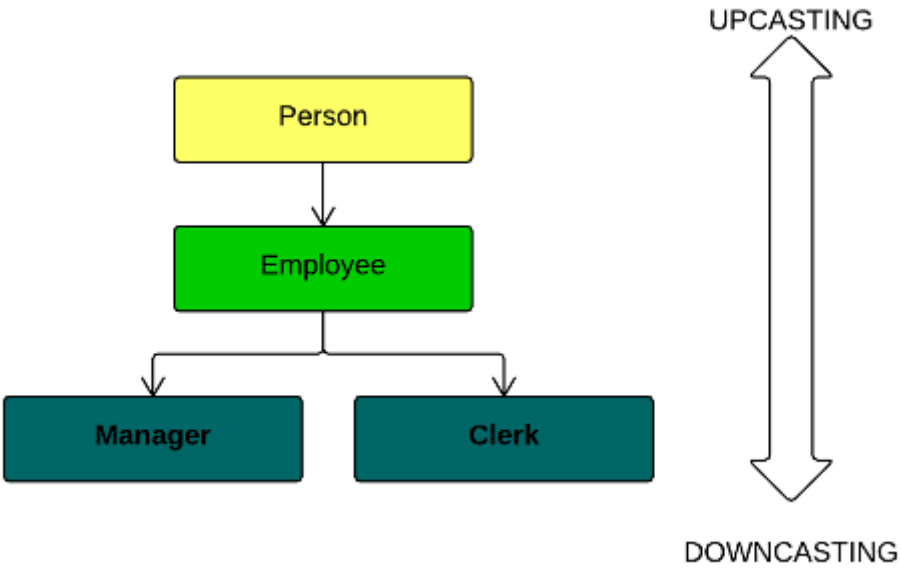
Примітка: рядок, що повертається ім'ям члена type_info, залежить від конкретної реалізації вашого компілятора та бібліотеки. Це не обов'язково проста рядок із типовим ім'ям типу, як у компіляторі, який використовується для отримання цього виводу.

Зверніть увагу, як тип, який typeid вважає для вказівників, є самим типом вказівника (і a, і b є класом типу Base *). Однак, коли typeid застосовується до об'єктів (типу * a і * b), typeid дає їх динамічний тип (тобто тип їх найбільш похідного повного об'єкта).

Якщо typeid type оцінює - це вказівник, якому передує оператор dereference (*), і цей вказівник має нульове значення, typeid видає виняток bad_typeid.

Перетоворення типів вниз та вверх

Перетворення вниз та вверх є важливою частиною C ++. Вони дають можливість будувати складні програми з простим синтаксисом. Цього можна досягти, використовуючи **поліморфізм**.
C ++ дозволяє похідному вказівнику класу (або посиланню) трактуватись як вказівник базового класу. Це перетворення вверх.
Переторення вниз – це протилежний процес, який полягає в перетворенні вказівника (або посилання) базового класу на похідний вказівник класу.
Перетворення та посилання вниз не слід розуміти як просту трансляцію різних типів даних. Це може призвести до великої плутанини.
У цій темі ми використаємо таку ієрархію класів:



Як бачите, менеджер та службовець - обидва працівники. Що це означає? Це означає, що класи Manager та Clerk успадковують властивості класу Employee, який успадковує властивості класу Person.
Наприклад, нам не потрібно вказувати, що і менеджер, і службовця ототожнюються за ім'ям та прізвищем, мають зарплату; ви можете показати інформацію про них і додати бонус до їх зарплати. Ми повинні вказати ці властивості лише один раз у класі Співробітник:
У той же час класи менеджерів та діловодів різні. Менеджер бере комісійну винагороду за кожен контракт, а Клерк має інформацію про свого менеджера:

```
#include <iostream>
using namespace std;

class Person
{
	//content of Person
};


class Employee:public Person
{
public:
	Employee(string fName, string lName, double sal)
	{
		FirstName = fName;
		LastName = lName;
		salary = sal;
	}
```

```
	string FirstName;
	string LastName;
	double salary;
	void show()
	{
		cout << "First Name: " << FirstName << " Last Name: " << LastName
<< " Salary: " << salary<< endl;
	}
	void addBonus(double bonus)
	{
		salary += bonus;
	}
};

class Manager :public Employee
{
public:
	Manager(string fName, string lName, double sal, double comm)
:Employee(fName, lName, sal)
	{
		Commision = comm;
	}
	double Commision;
	double getComm()
	{
		return Commision;
	}
};

class Clerk :public Employee
{
public:
	Clerk(string fName, string lName, double sal, Manager* man)
:Employee(fName, lName, sal)
	{
		manager = man;
	}
	Manager* manager;
	Manager* getManager()
	{
		return manager;
	}
};

void congratulate(Employee* emp)
```

```
{
        cout << "Happy Birthday!!!" << endl;
        emp->addBonus(200);
        emp->show();
};

int main()
{
   //pointer to base class object
   Employee* emp;

   //object of derived class
   Manager m1("Steve", "Kent", 3000, 0.2);
   Clerk c1("Kevin","Jones", 1000, &m1);

   //implicit upcasting
   emp = &m1;

   //It's ok
   cout<<emp->FirstName<<endl;
   cout<<emp->salary<<endl;

   //Fails because upcasting is used
   //cout<<emp->getComm();

   congratulate(&c1);
   congratulate(&m1);

   cout<<"Manager of "<<c1.FirstName<<" is "<<c1.getManager()->FirstName;
}
```

Менеджер і службовець - це завжди працівники. Більше того, працівник - це людина. Тому менеджер та службовець теж є особами. Ви повинні це зрозуміти, перш ніж ми почнемо вчитися оновленню та пониженню мовлення.
І перетворення, і зникнення не змінюють об'єкт сам по собі. Під час використання ви просто "позначаєте" об'єкт різними способами.

Upcasting

Upcasting - це процес обробки вказівника або посилання на похідний об'єкт класу як вказівник базового класу. Не потрібно здійснювати перетворення вручну. Вам просто потрібно призначити похідний вказівник класу (або посилання) на вказівник базового класу:

```
//pointer to base class object
Employee* emp;
//object of derived class
Manager m1("Steve", "Kent", 3000, 0.2);
//implicit upcasting
emp = &m1;
```

Коли ви використовуєте перетворення, об'єкт не змінюється. Тим не менше, коли ви обновлюєте об'єкт, ви матимете доступ до лише функцій-членів та членів даних, визначених у базовому класі:

```
//It's ok
emp->FirstName;
emp->salary;
//Fails because upcasting is used
emp->getComm();
```

Приклад використання перетворення

Однією з найбільших переваг перетворення є можливість запису загальних функцій для всіх класів, що походять з одного базового класу. Подивіться на приклад:

```
void congratulate(Employee* emp)
{
	cout << "Happy Birthday!!!" << endl;
	emp->show();
	emp->addBonus(200);
};
```

Ця функція буде працювати з усіма класами, що походять від класу Співробітник. Коли ви називаєте його об’єктами типу Manager та Person, вони будуть автоматично перенесені на клас Employee:

```
//automatic upcasting
congratulate(&c1);
congratulate(&m1);
```

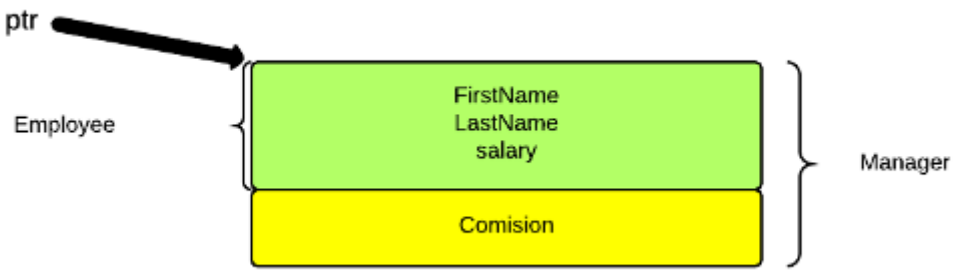
Результат роботи програми:

```
Happy Birthday!!!
First Name: Kevin Last Name: Jones
Happy Birthday!!!
First Name: Steve Last: Name Kent
```

Макет пам'яті

Як відомо, похідний клас розширює властивості базового класу. Це означає, що похідний клас має властивості (члени даних та функції членів) базового

класу та визначає нових членів даних та функцій членів.
Подивіться на макет пам'яті класів Співробітник та Менеджер:



Звичайно, ця модель - спрощений вигляд компонування пам'яті для об'єктів. Однак це означає, що коли ви використовуєте вказівник базового класу для наведення об'єкта похідного класу, ви можете отримати доступ лише до елементів, визначених у базовому класі (зелена зона). Елементи похідного класу (жовта область) недоступні при використанні вказівника базового класу.

Downcasting

Downcasting - протилежний процес перетворення. Він перетворює вказівник базового класу у похідний вказівник класу. Спускання каналів потрібно робити вручну. Це означає, що вам потрібно вказати чіткий тип передачі. Даудаустінг не є безпечним, як перетворення. Ви знаєте, що похідний об'єкт класу завжди може розглядатися як об'єкт базового класу. Однак навпаки не вірно. Наприклад, менеджер - це завжди людина; Але людина не завжди є менеджером. Це може бути і чиновник.
Ви повинні використовувати явний амплуа для зниження мовлення:

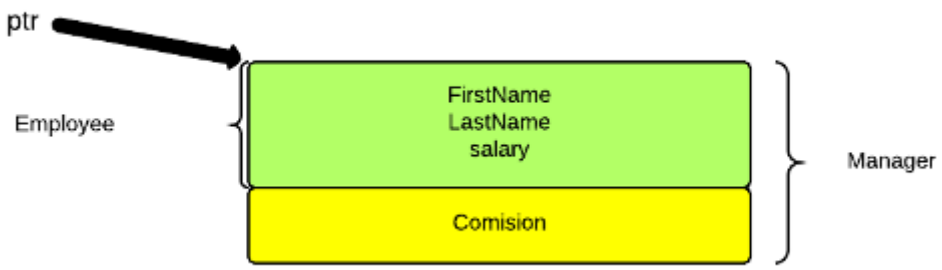
```
//pointer to base class object
Employee* emp;
//object of derived class
Manager m1("Steve", "Kent", 3000, 0.2);
//implicit upcasting
emp = &m1;
//explicit downcasting from Employee to Manager
Manager* m2 = (Manager*)(emp);
```

Цей код збирається та працює без будь-яких проблем, оскільки emp вказує на об'єкт класу Manager.
Що станеться, якщо ми спробуємо схилити вказівник базового класу, який вказує на об’єкт базового класу, а не на об'єкт похідного класу? Спробуйте скласти і запустити цей код:

```
Employee e1("Peter", "Green", 1400);
//try to cast an employee to Manager
Manager* m3 = (Manager*)(&e1);
```

cout << m3->getComm() << endl;

Об'єкт e1 не є об'єктом класу Manager. Він не містить жодної інформації про комісію. Ось чому така операція може дати неочікувані результати.
Подивіться на макет пам'яті ще раз:



Якщо ви спробуєте скинути вказівник базового класу (Employee), який насправді не вказує на об'єкт похідного класу (Manager), ви отримаєте доступ до пам'яті, яка не має жодної інформації про похідний об'єкт класу (жовта область). Це головна небезпека зриву.
Ви можете використовувати безпечний склад, який допоможе вам дізнатися, чи можна один тип правильно перетворити на інший. Для цього використовуйте динамічний акторський склад.

dynamic_cast

dynamic_cast - це оператор, який безпечно перетворює один тип в інший. У випадку, якщо розмова можлива і безпечна, вона повертає адресу об'єкта, який перетворюється. В іншому випадку він повертає nullptr.
Синтаксис:

dynamic_cast<new_type> (object)

Якщо ви хочете використовувати динамічний амплуа для пониження мовлення, базовий клас повинен бути поліморфним - він повинен мати принаймні одну віртуальну функцію. Змініть особу базового класу, додавши віртуальну функцію:

virtual void foo() {}

Тепер ви можете використовувати downcasting для перетворення вказівників класу Employee у вказівники похідних класів.

Employee e1("Peter", "Green", 1400);
Manager* m3 = dynamic_cast<Manager*>(&e1);
if (m3)

```
        cout << m3->getComm() << endl;
else
        cout << "Can't  cast from Employee to Manager" << endl;
```

У цьому випадку динамічний каст повертає nullptr. Тому ви побачите попереджувальне повідомлення.