

## Тема 13

Преобразуване между типове. Множествено наследяване. Диамантен проблем.  
Виртуално наследяване.

### Type casting.

Можем да извършваме преобразуване между типове по следните начини:

#### I начин:

`const_cast <желан тип> (израз)`  
тип, в който израз, който  
ще преобразуваме ще преобразуваме

Чрез този метод можем да премахнем константността на някой указател.  
Този указател трябва да сочи към обект, който някога не е бил константен.

#### Пример:

```
int N = 10;  
const int* ptr = &N;  
int* ptr2 = const_cast<int*>(ptr);  
(*ptr2) = 30;  
std::cout << N; // 30
```

#### Пример за странно поведение:

```
const int N = 10;  
const int* ptr = &N;  
int* ptr2 = const_cast<int*>(ptr);  
(*ptr2) = 30;  
std::cout << N;
```

В този случай може да се отпечата както 10, така и 30 (undefined behavior).

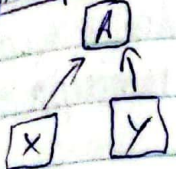
Причината за това е, че указателът сочи към обект, който винаги е бил константен. Не е константен обект, който има своя памет и ние не можем да го променим. Трябва обектът някога да не е бил константен, за да може да се промени в паметта.



### I начин:

`dynamic_cast <желан тип> (израз);`  
`dynamic_cast` прави преобразуване, но заедно с това има и runtime проверка за валидност.

#### Пример:

 `A* ptr = new X();`  
`X* ptr2 = dynamic_cast<X*> (ptr); // OK`  
`Y* ptr3 = dynamic_cast<Y*> (ptr); // nullptr`  
Runtime ще получим грешка <sup>за второто преобразуване</sup>, защото `ptr` е от тип `X*` и не е възможен преобразуването му към `Y*`.

Що се отнася до референции това не работи така.

#### Пример:

`X obj;`  
`X& ref = obj;`  
`dynamic_cast<X> (ref);`

В случая не може да ни върне `nullptr`, защото искаме да ни върне обект, а този обект няма неутрална стойност, което довежда до хвърляне на изключение `std::bad_cast`.

Съответно, когато работим по този начин, внесо проверка за `nullptr` ще имаме `try-catch` блок, който ще обработи грешката, ако възникне.

### II начин:

`static_cast <желан тип> (израз);`  
`static_cast` прави преобразуване, но не прави проверка за валидност. Тъси, ако кажем `X*` в `Y*` ще получим грешка. `static_cast` е по-бърз от `dynamic_cast`, но за да избегнем грешки, го използваме само, когато сме сигурни за типа.

### III начин:

`reinterpret_cast <желан тип> (израз);`  
Чрез `reinterpret_cast` можем да преобразуваме от всеки тип във всеки тип, което понякога може да доведе до нежелано поведение и загуба на данни. Затова използваме в крайен случай.

Пример за употреба:  
писане и четене от бинарни файлове



## V начин:

c-style cast

Пример:

$B^* ptr;$

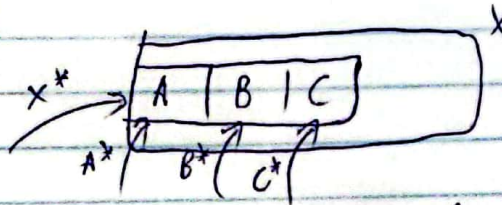
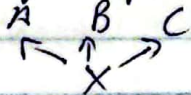
$A^* ptr2 = (A^*) ptr;$

Начин на работа: Опита първо да направи static-cast. Ако не успее, опита с dynamic-cast. Ако и това не успее, опита с reinterpreted-cast. C-style cast е базиран на тези три типа кастове.

Множествено наследяване

Множественото наследяване позволява на "деиски" клас да наследява повече от един родителски клас

Пример:



можем да правим такава указател

Голямата гетворка при множествено наследяване.

Разписва се само, ако има динамична памет в класа-наследник.

В конструкторите на обекти от тип X е важно да се погрижим за извикване на конструкторите на базовите класове.

$X(const X \& other): A(other), B(other), C(other)$

{

copyFrom(other); // Грими се само за член-данните на X

}

При дефиниране на оператор= също трябва да се погрижим за извикване на операторите на базовите класове.



```
x& operator = (const X& other)
```

```
{ if (this != &other)
```

```
{ A::operator = (other);
```

```
B::operator = (other);
```

```
C::operator = (other);
```

```
free();
```

```
copy from (other); }
```

```
return *this;
```

```
}
```

При деструкції на ~~после~~ закриваючій скобці се викликає деструкція на C, B, A (не се викликає екземпляр се викликає об'єкти на рега в тойто са декларирани при наслідванні)

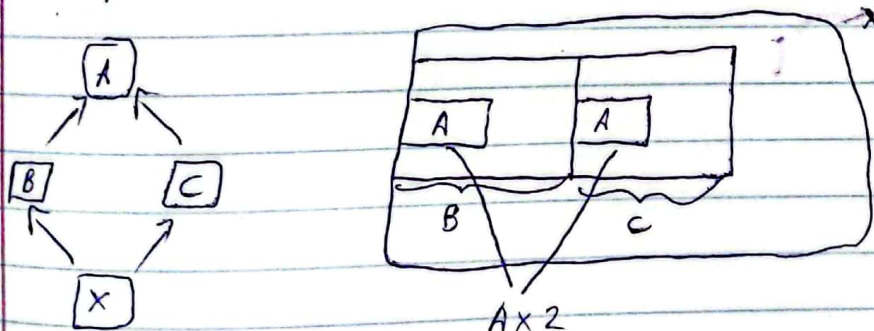
```
~X()
```

```
{
```

```
free();
```

```
} // ~C(), ~B(), ~A()
```

При множинному наслідванні виникає проблема, известна як "діамантний проблем"



Във всеки от класове B и C ~~существува~~ екземпляр на клас A, защото те по-отделно го наслідват. След като класът X наслідва класове B и C, той получава 2 екземпляри на клас A и това порозумява двукратността, известна като "діамантний проблем". На крайго той представлява



многократно наследяване на базов клас - проблема се не еднозначност при използване. В класа X има 2 инстанции на класа A, но ние искаме характеристиките на A да бъдат наследени само веднъж.

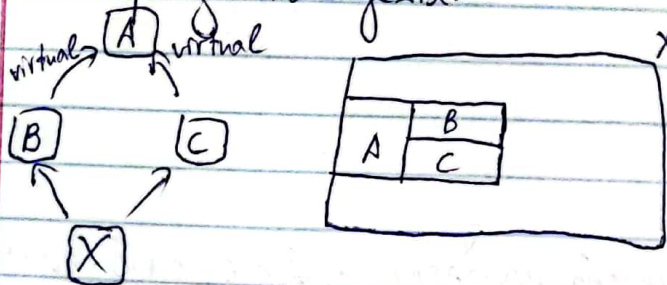
Преодоляването на големия част от недостатъците на многократното наследяване на клас се осъществява чрез използването на т. нар. виртуални основни класове. Те дава възможност за "поделение" на компонентите на основните класове - създава се само едно техно копие. Декларира се като в декларацията на производния клас се употреби и ключовата дума virtual

```
class B: virtual A { ... }  
class C: public virtual A { ... }
```

Конструкторите на параметри на виртуалните класове трябва да се извикват от конструкторите на всички класове, които са техни наследници, а не само от конструкторите на прехите им наследници

```
X(): B(...), C(...), A(...) { }
```

Илюстрация на идеята:



Виртуалното наследяване се припъи всеки наследник да има една инстанция на клас A