C++/Dziedziczenie

Wstęp - Co to jest dziedziczenie

Często podczas tworzenia klasy napotykamy na sytuację, w której klasa ta powiększa możliwości innej klasy, nierzadko precyzując jednocześnie jej funkcjonalność. Dziedziczenie daje nam możliwość wykorzystania nowych klas w oparciu o stare klasy. Nie należy jednak traktować dziedziczenia jedynie jako sposobu na współdzielenie kodu między klasami. Dzięki mechanizmowi rzutowania możliwe jest interpretowanie obiektu klasy tak, jakby był obiektem klasy z której się wywodzi. Umożliwia to skonstruowanie szeregu klas wywodzących się z tej samej klasy i korzystanie w przejrzysty i spójny sposób z ich wspólnych możliwości. Należy dodać, że dziedziczenie jest jednym z czterech elementów programowania obiektowego (obok abstrakcji, enkapsulacji i polimorfizmu).

Klasę z której dziedziczymy nazywamy **klasą bazową**, zaś klasę, która po niej dziedziczy nazywamy **klasą pochodną**. Klasa pochodna może korzystać z funkcjonalności klasy bazowej i z założenia powinna rozszerzać jej możliwości (poprzez dodanie nowych metod, lub modyfikację metod klasy bazowej).

Składnia

Składnia dziedziczenia jest bardzo prosta. Przy definicji klasy należy zaznaczyć po których klasach dziedziczymy. Należy tu zaznaczyć, że C++ umożliwia *Wielodziedziczenie*, czyli dziedziczenie po wielu klasach na raz. Jest ono opisane w rozdziale Dziedziczenie wielokrotne.

```
   class nazwa_klasy :[operator_widocznosci] nazwa_klasy_bazowej,
[operator_widocznosci] nazwa_klasy_bazowej ...
   {
        definicja_klasy
   };
```

operator_widoczności może przyjmować jedną z trzech wartości: **public**, **protected**, **private**. Operator widoczności przy klasie, z której dziedziczymy pozwala ograniczyć widoczność elementów publicznych z *klasy bazowej*.

- **public** - oznacza, że dziedziczone elementy (np. zmienne lub funkcje) mają taką widoczność jak w klasie bazowej.

  public $\Rightarrow$ *public*

  protected $\Rightarrow$ *protected*

  private $\Rightarrow$ *brak dostępu w klasie pochodnej*

- **protected** - oznacza, że elementy publiczne zmieniają się w chronione.

  public $\Rightarrow$ *protected*

  protected $\Rightarrow$ *protected*

  private $\Rightarrow$ *brak dostępu w klasie pochodnej*

- **private** - oznacza, że wszystkie elementy klasy bazowej zmieniają się w prywatne.

  public $\Rightarrow$ *private*

  protected $\Rightarrow$ *private*

  private $\Rightarrow$ *brak dostępu w klasie pochodnej*

- **brak operatora** - oznacza, że niejawnie (domyślnie) zostanie wybrany operator **private**..

  public $\Rightarrow$ *private*

  protected $\Rightarrow$ *private*

private $\Rightarrow$ *brak dostępu w klasie pochodnej*

Dostęp do elementów klasy bazowej można uzyskać jawnie w następujący sposób:

```
[klasa_bazowa::...]klasa_bazowa::element
```

Zapis ten umożliwia dostęp do elementów klasy bazowej, które są "przykryte" przez elementy klasy nadrzędnej (mają takie same nazwy jak elementy klasy nadrzędnej). Jeżeli nie zaznaczymy jawnie o który element nam chodzi kompilator uzna że chodzi o element klasy nadrzędnej, o ile taki istnieje (przeszukiwanie będzie prowadzone w głąb aż kompilator znajdzie "najbliższy" element).

Przykład 1

Definicja i sposób wykorzystania dziedziczenia

Najczęstszym powodem korzystania z dziedziczenia podczas tworzenia klasy jest chęć sprecyzowania funkcjonalności jakiejś klasy wraz z implementacją tej funkcjonalności. Pozwala to na rozróżnianie obiektów klas i jednocześnie umożliwia stworzenie funkcji korzystających ze wspólnych cech tych klas. Załóżmy że piszemy program symulujący zachowanie zwierząt. Każde zwierze powinno móc jeść. Tworzymy odpowiednią klasę:

```
class Zwierze
{
     public:
          Zwierze();
          void jedz();
};
```

Następnie okazuje się, że musimy zaimplementowac klasy *Ptak* i *Ryba*. Każdy ptak i ryba jest zwierzęciem. Oprócz tego ptak może latać, a ryba płynąć. Wykorzystanie dziedziczenia wydaje się tu naturalne.

```
class Ptak : public Zwierze
{
     public:
          Ptak();
          void lec();
};

class Ryba : public Zwierze
{
     public:
          Ryba();
          void plyn();
};
```

Co istotne tworząc takie klasy możemy wywołać ich metodę pochodzącą z klasy *Zwierze*:

```
Ptak ptak;
ptak.jedz(); //metoda z klasy Zwierze
ptak.lec(); //metoda z klasy Ptak

Ryba *ryba=new Ryba();
ryba->jedz(); //metoda z klasy Zwierze
```

```
ryba->plyn(); //metoda z klasy Ryba
```

Możemy też zrzutować obiekty klasy *Ptak* i *Ryba* na klasę *Zwierze*:

```
Ptak *ptak=new Ptak();
Zwierze *zwierze;
zwierze=ptak;
zwierze->jedz();

Ryba ryba;
((Zwierze)ryba).jedz();
```

Jeżeli tego nie zrobimy, a rzutowanie jest potrzebne, kompilator sam wykona rzutowanie niejawne:

```
Zwierze zwierzeta[2];
zwierzeta[0]=Ryba(); //rzutowanie niejawne
zwierzeta[1]=Ptak(); //rzutowanie niejawne
for (int i=0; i<2; ++i)
     zwierzeta[i].jedz();
```

Elementy chronione - operator widoczności *protected*

Sekcja protected klasy jest ściśle związana z dziedziczeniem - elementy i metody klasy, które się w niej znajdują, mogą być swobodnie używane w klasie dziedzicznej ale poza klasą dziedziczną i klasą bazową nie są widoczne.

Elementy powiązane z dziedziczeniem

Chciałbym zwrócić uwagę na inne, bardzo istotne elementy dziedziczenia, które są opisane w następnych rozdziałach tego podręcznika, a które mogą być wręcz niezbędne w prawidłowym korzystaniu z dziedziczenia (przede wszystkim Funkcje wirtualne).

Funkcje wirtualne

Przykrywanie metod, czyli definiowanie metod w klasie pochodnej o nazwie i parametrach takich samych jak w klasie bazowej, ma zwykle na celu przystosowanie metody do nowej funkcjonalności klasy. Bardzo często wywołanie metody klasy bazowej może prowadzić wręcz do katastrofy, ponieważ nie bierze ona pod uwagę zmian miedzy klasą bazową a pochodną. Problem powstaje, kiedy nie wiemy jaka jest klasa nadrzędna obiektu, a chcielibyśmy żeby zawsze była wywoływana metoda klasy pochodnej. W tym celu język C++ posiada **funkcje wirtualne**. Są one opisane w rozdziale Funkcje wirtualne.

Wielodziedziczenie - czyli dziedziczenie wielokrotne

Język C++ umożliwia dziedziczenie po wielu klasach bazowych na raz. Proces ten jest opisany w rozdziale Dziedziczenie wielokrotne.

Przykład 2

```
#include <iostream>

class Zwierze
{
   public:
```

```
        Zwierze()
        { }

        void jedz( )
        {
            for ( int i=0; i<10; ++i )
               std::cout << "Om Nom Nom Nom\n";
        }

        void pij( )
        {
            for ( int i=0; i<5; ++i )
               std::cout << "Chlip, chlip\n";
        }

        void spij( )
        {
            std::cout << "Chrr...\n";
        }
};

class Pies : public Zwierze
{
   public:
     Pies()
     { }

     void szczekaj()
     {
        std::cout << "Hau! hau!...\n";
     }

     void warcz()
     {
        std::cout << "Wrrrrrr...\n";
     }
};

...
```

Za pomocą

```
...
class Pies : public Zwierze
{
...
```

utworzyliśmy klasę Psa, która dziedziczy klasę Zwierze. Dziedziczenie umożliwia przekazanie zmiennych, metod itp. z jednej klasy do drugiej. Możemy funkcję main zapisać w ten sposób:

```
...
int main()
{
   Pies burek;
   burek.jedz();
   burek.pij();
   burek.warcz();
   burek.pij();
   burek.szczekaj();
   burek.spij();
   return 0;
}
```

Źródła i autorzy artykułu

C++/Dziedziczenie *Źródło*: http://pl.wikibooks.org/w/index.php?oldid=156456 *Autorzy*: Derbeth, DrJolo, Felix, Kj, Lethern, Maricn24, MonteChristof, Mythov, Piotr, Savh, Steeve, Teceha, Uniq, Wutsje, 41 anonimowych edycji

Licencja