1 Wprowadzenie

1. Dziedziczenie jest jednym z najistotniejszych elementów obiektowości.
2. Dziedziczenie umożliwia pogodzenie dwóch sprzecznych dążeń:
   (a) Raz napisany, uruchomiony i przetestowany program powinien zostać w niezmienionej postaci.
   (b) Programy wymagają stałego dostosowywania do zmieniających się wymagań użytkownika, sprzętowych itp..
3. Dziedziczenie umożliwia tworzenie hierarchii klas.
4. Klasy odpowiadają pojęciom występującym w świecie modelowanym przez program. Hierarchie klas pozwalają tworzyć hierarchie pojęć, wyrażając w ten sposób zależności między pojęciami.
5. Klasa pochodna (podklasa) dziedziczy po klasie bazowej (nadklasie). Klasę pochodną tworzymy wówczas, gdy chcemy opisać bardziej wyspecjalizowane obiekty klasy bazowej. Oznacza to, że każdy obiekt klasy pochodnej jest obiektem klasy bazowej. ...

Zalety dziedziczenia:

- Jawne wyrażanie zależności między klasami (pojęciami). Np. możemy jawnie zapisać, że każdy kwadrat jest prostokątem, zamiast tworzyć dwa opisy różnych klas.
- Unikanie ponownego pisania tych samych fragmentów programu (ang. reuse).

2 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).

3 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.

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

[operator widocznosci] 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.

$$public \Rightarrow public$$

$$protected \Rightarrow protected$$

$$private \Rightarrow \text{brak dostępu w klasie pochodnej}$$

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

$$public \Rightarrow protected$$

$$protected \Rightarrow protected$$

$$private \Rightarrow \text{brak dostępu w klasie pochodnej}$$

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

$$public \Rightarrow private$$

$$protected \Rightarrow private$$

$$private \Rightarrow \text{brak dostępu w klasie pochodnej}$$

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

$$public \Rightarrow private$$

$$protected \Rightarrow private$$

$$private \Rightarrow \text{brak dostępu w klasie pochodnej}$$

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).

4 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 zaimplementować 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();
```

```
 #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;
}
```

5 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.

6 Dziedziczenie wielorakie

W C++ klasa może dziedziczyć z wielu klas bazowych. Jest to narzędzie pozwalające na tworzenie skomplikowanych hierarchii dziedziczenia i daje spore możliwości programistyczne. Z drugiej jednak strony, zbytnie skomplikowanie struktury klas dziedziczących prowadzi wtedy do trudnego do opanowania i modyfikowania kodu. W zasadzie dziedziczenie wielorakie należy zatem unikać, jeśli nie jest niezbędne.

Definiując klasę dziedziczącą z wielu klas wymieniamy je na liście dziedziczenia po kolei, oddzielając przecinkami. Dla każdej z nich z osobna podajemy specyfikator dostępu (private, protected lub public). Opuszczenie tego specyfikatora jest równoważne podaniu specyfikatora private dla klas, a public dla struktur. Wszystkie klasy występujące na liście dziedziczenia muszą być kompilatorowi znane, - nie wystarczy deklaracja zapowiadająca.

```
class A { ... };
class B: public A { ... };  //dziedziczy z A
class C: public A, public B { ... };  //C dziedziczy z A i B
class D: public B, public C { ... };  //D dziedziczy z B i C
```

Uwaga!!! Klasa C otrzyma dwukrotnie klasę A. Klasa D otrzyma dwukrotnie klasę B oraz trzykrotnie klasę A.

7 Zadania do wykonania

1. Stworzyć hierarchię klas dla klasyfikacji roślin
2. Przetestować dziedziczenie z różnymi operatorami widoczności
3. Sprawdzić możliwość rzutowania obiektu klasy na obiekt klasy bazowej i klasy pochodnej
4. Stworzyć tablicę okiektów roślin o różnych klasach

5. Stworzyć kolekcję typu vector roślin o różnych klasach
6. Stworzyć program pozwalający na dodawanie, odczytywanie, usuwanie i modyfikację różnych roślin
7. Zaproponować i stworzyć hierarchię klas, które wykorzystują wielodziedziczenie