Instrukcja 2

Programowanie obiektowe 1

Część A

Dziedziczenie i polimorfizm

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.

1 Wprowadzenie

Polimorfizm to, obok dziedziczenia, najważniejszy mechanizm programowania obiektowego. Pozwala on na tworzenie wielu obiektów posiadających tę samą funkcjonalność ale różniącą się sposobem działania. Przykładem mogą być figury geometryczne, które dostarczają metody obliczającej pole powierzchni. Możemy obliczyć pole powierzchni każdej figury, ale w zależności od rodzaju figury obliczenie go będzie wyglądało inaczej.

```
#include <iostream>

using namespace std;

class Figura{
public:
  float pole(){
    return 0;
  }

};

class Prostokat : public Figura{
private:
  float a, b;
public:
  Prostokat(float a, float b){
    this->a = a;
    this->b = b;
  }

  float pole(){
    return a*b;
  }
};


class Kwadrat : public Figura{
private:
  float a;
public:
  Kwadrat(float a){
    this->a = a;
  }
  float pole(){
    return a*a;
  }
};

class Kolo : public Figura{
private:
  float r;
public:
  Kolo(float r){
```

```
    this->r = r;
  }

  float pole(){
    return 3.14*r*r;
  }
};

int main(int argc, char* argv[]){
Kolo kolo(3);
Prostokat protokat(2,4);
Kwadrat kwadrat(2);

cout << "Pole kola = " << kolo.pole() << endl;
cout << "Pole prostokata = " << protokat.pole() << endl;
cout << "Pole kwadratu = " << kwadrat.pole() << endl;

}
```

Rezultatem programu będzie:

```
Pole kola = 28.26
Pole prostokata = 8
Pole kwadratu = 4
```

Natomiast w przypadku uruchomienia kodu:

```
Figura figury[3] = {Kolo(1), Kwadrat(2), Prostokat(2,3)};
for (int i = 0; i < 3; i++)
  cout << "Pole figury " << i << " = " << figury[i].pole() << endl;
```

Rezultatem programu będzie:

```
Pole figury 0 = 0
Pole figury 1 = 0
Pole figury 2 = 0
```

Ponieważ wykorzystana została tablica obiektów klasy Figura (klasy bazowej) wszystkie figury traktowane są jako klasa bazowa. W związku z tym wykonywana jest metoda klasy bazowej, która zwraca za każdym razem wartość zero a nie odpowiednia metoda z klasy pochodnej. Dzieje się tak dlatego, że na etapie kompilacji kompilator ustawia na stałe adres wykonywanej metody na podstawie typu jakiego używamy (tzw. wczesne wiązanie metod).

Jeśli chcemy aby wykonywana została wykonywana metoda właściwego obiektu musimy odroczyć ustawienie adresu do metody do czasu wykonywania programu (tzw. późne wiązanie). W języku C++ wykonujemy to zadanie za pomocą słowa kluczowego virtual.

```
class Figura{
public:
  virtual float pole(){
    return 0;
  }

};

class Prostokat : public Figura{
private:
  float a, b;
public:
  Prostokat(float a, float b){
    this->a = a;
    this->b = b;
  }

  float pole(){
    return a*b;
  }
};


class Kwadrat : public Figura{
private:
  float a;
public:
  Kwadrat(float a){
    this->a = a;
  }
  float pole(){
    return a*a;
  }
};

class Kolo : public Figura{
private:
  float r;
public:
  Kolo(float r){
    this->r = r;
  }

  float pole(){
    return 3.14*r*r;
  }
};
```

W tym przypadku deklarujemy, że metoda pole() jest metodą wirtualną, więc adres metody do wykonania będzie obliczany nie na etapie kompilacji ale na etapie działania programu (dynamicznie). Dzięki temu będzie mogła zostać wykonany za każdym razem odpowiedni kod.

Wykonanie następującego kodu (UWAGA!: Należy zwrócić uwagę, że tym razem użyto wskaźników na obiekty a nie samych obiektów!):

```
Figura* figury[3] = { new Kolo(1), new Kwadrat(2), new Prostokat(2,3)};
for (int i = 0; i < 3; i++)
  cout << "Pole figury " << i << " = " << figury[i]->pole() << endl;
```

spowoduje w rezultacie:

```
Pole figury 0 = 3.14
Pole figury 1 = 4
Pole figury 2 = 6
```

Instrukcja 2

Programowanie obiektowe 1

Część B

Strumienie

Programowanie w języku C - Adam Krechowicz, Daniel Kaczmarski

1 Strumienie

W języku C++ pliki obsługiwane są za pomocą *strumieni*. Strumień pozwala na sekwencyjny dostęp do pliku. Elementy, które jako pierwsze zostają zapisane do strumienia jako pierwsze będą z niego odczytane. Strumienie wykorzystywane są także do obsługi standardowego wejścia/wyjścia (std::cin, std::cout, std::cerr, std::clog). Zapisanie danych do strumienia odbywa się za pomocą operatora <<, natomiast odczytanie danych ze strumienia za pomocą operatora >>. Operatory te zwracają strumień na rzecz którego zostały wywołane dlatego możliwe jest np jednoczesne wysłanie kilku elementów do strumienia. Przykład:

```
int i = 5;
std::cout << "Tekst" << i << "Inny tekst" << std:: end;
```

2 Pliki

Strumienie do obsługi plików zdefiniowane są w pliku nagłówkowym fstream. W pliku tym zdefiniowane są następujące strumienie:

- std::ifstream – strumień do odczytu pliku
- std::ofstream – strumień do zapisu pliku
- fstream – strumień do odczytu i zapisu pliku

2.1 Zapis do pliku tekstowego

```
#include <iostream>
#include <fstream>

int main(){

std::ofstream file;
file.open("plik.txt");
file << "Hello World";
file.close();

return 0;
}
```

2.2 Odczyt pliku tekstowego

```
#include <iostream>
#include <fstream>

int main(){

std::string linia;
std::ifstream plik;
plik.open("plik.txt");

if (plik.is_open()){
        while(getline(plik, linia))
                std::cout << linia << std::endl;
        plik.close();

} else std::cout << "Nie udalo sie otworzyc pliku" << std::endl;

return 0;
}
```

2.3 Zapis pliku binarnego

```
#include <iostream>
#include <fstream>

int main(){

int i = 3;
float f = 4.6f;
long l = 30000001;

std::ofstream plik;
plik.open("plik.abc", std::ios::binary);
plik.write(reinterpret_cast<char*>(&i), sizeof(i));
plik.write(reinterpret_cast<char*>(&f), sizeof(f));
plik.write(reinterpret_cast<char*>(&l), sizeof(l));
plik.close();

return 0;
}
```

2.4 Odczyt pliku binarnego

```
#include <iostream>
#include <fstream>

int main(){

int i;
float f;
long l;

std::ifstream plik;
```

```
plik.open("plik.abc", std::ios::binary);
plik.read(reinterpret_cast<char*>(&i), sizeof(i));
plik.read(reinterpret_cast<char*>(&f), sizeof(f));
plik.read(reinterpret_cast<char*>(&l), sizeof(l));
plik.close();
std::cout << i << "," << f << "," << l << std::endl;

return 0;
}
```

Pierwsze parametry metod read i write to wskaźniki na adres w pamięci gdzie znajdują się dane do zapisania. Dlatego aby przesłać podstawowe typy muszą one wcześniej zostać rzutowane na typ char*.

2.5 Metody strumieni

- open() – otwiera plik do zapisu lub odczytu. Jako drugi parametr można podać opcjonalne parametry:
  - std::ios::binary – otwieramy plik binarny, jeśli brak parametru to plik tekstowy
  - std::ios::in – otwieramy plik do odczytu, jeśli otwieramy strumień ifstream to parametr ten można pominąć
  - std::ios::out – otwieramy plik do zapisu, jeśli otwieramy strumień ofstream to parametr ten można pominąć
  - std::ios::ate – otwieramy plik i przesuwamy się na koniec pliku w celu dopisania zawartości (można zmienić pozycję wskaźnika)
  - std::ios::app – otwieramy plik tylko i wyłącznie do dopisywania zawartości (nie można zmienić pozycji wskaźnika w celu nadpisania)
  - std::ios::trunc – otwieramy plik wymazując jego poprzednią zawartość
- close() – zamyka plik
- is_open() – sprawdza czy plik został prawidłowo otworzony
- read() – odczytuje dane z pliku
- write() – zapisuje dane do pliku
- getline() – pobiera całą linię z pliku
- tellg(), tellp() – zwraca aktualną pozycję w pliku odpowiednio do pobierania i zapisywania

- seekg(), seekp() – ustawia aktualną pozycję w pliku odpowiednio do pobierania i zapisywania
- eof() – sprawdza czy osiągnięto koniec pliku
- fail() – sprawdza czy operacja zakończyła się niepowodzeniem
- bad() – sprawdza czy operacja zakończyła się problemem uniemożliwiającym dalsze operacje na strumieniu
- good() – sprawdza czy strumień jest prawidłowy

3 Strumienie łańcuchów znaków

W programach pisanych w C++ przydatny może być także strumienie operujące na łańcuchach znaków zdefiniowane w <sstream>:

- istringstream – wejściowy strumień łańcucha znaków
- ostringstream – wyjściowy strumień łańcucha znaków
- stringstream – wejściowo/wyjściowy strumień łańcucha znaków

Strumień ostringstream może być przydatny np. do zamiany określonych typów danych na postać łańcucha znaków:

```
std::ostringstream oss;
oss << "Hello" << i << ";" << 1;
std::cout << oss.str() << std::endl
```

Strumień istringstream może być przydatny np. do pobrania określonych wartości z łańcucha znaków:

```
std::string str = "1 2 3";
std::istringstream iss(str);
int a1, a2, a3;
iss >> a1 >> a2 >> a3;
std::cout << a1 << "," << a2 << "," << a3 << std::endl;
```

Zadania do wykonania:

1. Przetestować dziedziczenie z różnymi operatorami widoczności.
2. Napisać program z wykorzystaniem kolekcji typu vector roślin o różnych klasach.
3. Napisać program z wykorzystaniem hierarchii klas, które wykorzystują wielodziedziczenie.
4. Sprawdzić możliwość wykorzystania metod wirtualnych na tablicy obiektów.
5. Sprawdzić możliwość wykorzystania metod wirtualnych na tablicy wskaźników na obiekty.
6. Zapisać do pliku tekstowego linie pobrane od użytkownika do czasu aż wprowadzi określony ciąg znaków.
7. Wprowadzić do pliku tekstowego nazwy dni tygodnia, a następnie odczytać cały plik tekstowy linia po linii.
8. Zapisać 10 losowych liczb do pliku binarnego, a następnie odczytać cały plik binarny oraz utworzyć jego histogram.