dziedziczenie klas

Jednym z najważniejszych powodów, dla którego programowanie obiektowe zyskało taką popularność, jest znaczący postęp w kwestii ponownego wykorzystywania raz napisanego kodu oraz rozszerzania i dostosowania go do własnych potrzeb. Możliwość ta istnieje dzięki mechanizmowi dziedziczenia klas. Człowiek jest istotą, która bardzo lubi posiadać uporządkowany i usystematyzowany obraz świata. Wprowadzanie porządku i pewnej hierarchii do postrzeganych zjawisk i przedmiotów jest dla nas rzeczą naturalną. Chyba najlepiej przejawia się to w klasyfikacji biologicznej. Widząc na przykład psa wiemy przecież, że nie tylko należy on do gatunku zwanego psem domowym, lecz także do gromady znanej jako ssaki (wraz z końmi, słoniami, lwami, małpami i ludźmi). Te z kolei, razem z gadami, ptakami czy rybami należą do kolejnej, znacznie większej grupy organizmów zwanych po prostu zwierzętami. Nasz pies jest zatem **jednocześnie** psem domowym, ssakiem i zwierzęciem. Gdyby był obiektem w programie, wtedy musiałby należeć aż do trzech klas naraz. Byłoby to oczywiście niemożliwe, jeżeli wszystkie miałyby być wobec siebie równorzędne. Tutaj jednak tak nie jest: występuje między nimi hierarchia, jedna klasa pochodzi od drugiej. Zjawisko to nazywamy właśnie dziedziczeniem. Dziedziczenie (ang. *inheritance*) to tworzenie nowej klasy na podstawie jednej lub kilku istniejących wcześniej klas bazowych. Wszystkie klasy, które powstają w ten sposób (nazywamy je **pochodnymi** lub **potomnymi**), posiadają pewne elementy wspólne. Części te są dziedziczone z klas **bazowych**, gdyż tam właśnie zostały zdefiniowane. Ich zbiór może jednak zostać **poszerzony** o pola i metody specyficzne dla klas pochodnych. Będą one wtedy współistnieć z tymi pochodzącymi od klas bazowych, ale mogą oferować dodatkową funkcjonalność. Tak w teorii wygląda idea dziedziczenia w programowaniu obiektowym. Teraz przyjrzymy się, jak w praktyce może wyglądać jej zastosowanie. Powróćmy więc do naszego przykładu ze zwierzętami. Chcąc stworzyć programowy odpowiednik zaproponowanej hierarchii, musielibyśmy zdefiniować najpierw odpowiednie **klasy bazowe**. Następnie ich pola i metody zostaną odziedziczone w **klasach pochodnych** i dodane zostaną nowe, właściwe tylko im. Powstałe klasy same mogłyby być potem bazami dla kolejnych, jeszcze bardziej wyspecjalizowanych typów. Wracając do klasyfikacji biologicznej, wszystkie występujące w niej klasy wywodzą się z jednej, nadrzędnej wobec wszystkich: jest nią naturalnie klasa *Zwierzę*. Dziedziczy z niej każda z pozostałych klas - **bezpośrednio**, jak *Ryba*, *Ssak* oraz *Ptak*, lub **pośrednio** - jak *Pies domowy*. Tak oto tworzy się kilkupoziomowa klasyfikacja oparta na mechanizmie dziedziczenia. Klasę pochodną tworzymy tylko wówczas, gdy chcemy opisać bardziej wyspecjalizowane obiekty klasy bazowej. Oznacza to, że każdy obiekt klasy pochodnej jest obiektem klasy bazowej. Nie możemy, na przykład, za pomocą dziedziczenia klas utworzyć klasę `Komputer` z klasy `Samolot` gdyż `Komputer` nie jest bardziej szczególnym przypadkiem `Samolotu`. Mając ogólne informacje o dziedziczeniu klas, możemy teraz zobaczyć, jak ta idea została przełożona na nasz język C++. Przykładowa klasa bazowa:

```
class A
{
private:
  int m_value1;
public:
  int m_value3;
};
```

Przykładowa podklasa (zwróć uwagę na sposób zapisu, że klasa `B` dziedziczy z klasy `A`)

```
class B : public A {
private:
 int m_value4;
public:
 int m_value6;
void m();
};
```

Implementacja metody `m()`, zwróć uwagę na użycie operatora zakresu `::`

```
void B::m(){
  m_value2 = 2;
  m_value4 = m_vlaue5 = m_value6 = 4;
 }
```

W podklasach można deklarować składowe o takiej samej nazwie jak w nadklasach:

```
class C {
 public:
  int m_a;
  void m();
};

class D : public C {
 public:
```

```
  int m_a;
  void m();
};
```

Kompilator zawsze będzie w stanie rozróżnić, o którą składową chodzi:

```
void C::m(){
 m_a = 1;  // Składowa klasy C
}

void D::m(){
 m_a = 2;  // Składowa klasy D
}
```

Konstruktor klasy pochodnej powinien wywoływać odpowiedni konstruktor klasy bezpośrednio nadrzędnej

```
class A
{
  int m_value_in_a;
public:
   A(int value) : m_value_in_a(value) {}
};

class B : public A
{
  int m_value_in_b;
public:
   B(int value_a, int value_b) : A(value_a), m_value_in_b(value_b) {}
};
```

Bardzo ważne jest by pamiętać, że dziedziczenie klas można użyć tylko wtedy gdy klasa potomna jest bardziej szczegółowym przypadkiem klasy rodzica. Dla przykładu, uzasadnionym jest użycie dziedziczenia by na podstawie klasy `Zwierzę` stworzyć klasę `Pies` gdyż `Pies` jest bardziej szczegółowym przypadkiem klasy `Zwierzę`. Dla porównania, użycie dziedziczenia by stworzyć klasę `Kot` z klasy `Pies` jest dużym błędem ponieważ `Kot` nie jest bardziej szczegółowym przypadkiem klasy `Pies`. Należy również pamiętać, że we wszystkich przypadkach gdzie można użyć klasy rodzica można zamiast niej użyć klasy potomnej. Na przykład, jeśli w kontenerze przechowujemy obiekty klasy `Zwierzę` to możemy w nim umieścić również obiekty klasy potomnej `Pies`. Zalety dziedziczenia: po pierwsze, jawne wyrażanie zależności między klasami (pojęciami) - dla przykładu, możemy jawnie zapisać, że każdy prostokąt jest równoległobokiem, zamiast tworzyć dwa opisy różnych klas. Po drugie, unikanie ponownego pisania tych samych fragmentów programu (ang. *code reuse*).

wyjątki

Wyjątki są mechanizmem pozwalającym obsłużyć różne sytuacje wyjątkowe, tzn. takie, które powinny się zdarzać relatywnie rzadko. Zazwyczaj jest to obsługa błędu, choć nie oznacza to wcale, że to miałby być to jakiś wyjątkowy błąd - może być to błąd zupełnie spodziewany i oczekiwany. W języku C++ wyjątek (ang. *exception*) to obiekt pewnej klasy. Działanie mechanizmu wyjątków polega na tym, że pewien fragment kodu próbuje się (ang. *try*) wykonać. W razie wystąpienia sytuacji wyjątkowej, rzuca się (ang. *throw*) wyjątek, który następnie powinien zostać przechwycony (ang. *catch*). Wygenerowanie (zgłoszenie) wyjątku powoduje przerwanie wykonywania sprawiającego problemy kodu i przejście do obsługi sytuacji problematycznej. Obsługa ta może znajdować się w innym miejscu kodu. Wyjątkiem rzuca się przy pomocy instrukcji `throw`, po której podajemy obiekt,który posłuży do przeniesienia informacji do odpowiedniej dla niego klauzuli `catch`. Mamy do dyspozycji wiele wyjątków standardowych zdefiniowanych w pliku nagłówkowym `exception`. Można również tworzyć własne klasy reprezentujące różnego rodzaju wyjątki. Po dotarciu wyjątku do odpowiedniej klauzuli `catch`, wykonuje się procedura jego obsługi. Kiedy ta procedura się zakończy, wykonuje się kod znajdujący się za obszarem `try/catch`.

```
int main()
{
double x;
cin>>x;
try
{
   if ( x < 0 ) throw "cannot compute square root of a negative value";
   cout << sqrt( x );
```

```
}
catch (const char* ex) // przechwyć wyjątek typu const char*
{
   cerr << ex << endl;
}
catch (int ex) {} // przechwyć wyjątek typu int
catch (...) {} // przechwyć wyjątek dowolnego typu
```

1. Stwórz klasę o nazwie `Person` reprezentującą osobę. Klasa powinna zawierać: pola reprezentujące imię i nazwisko, pole reprezentujące numer PESEL, odpowiedni konstruktor z wartościami domyślnymi oraz gettery. Stwórz w klasie metodę `string to_string()`, która zwróci reprezentację obiektu w postaci stringu.

2. Stwórz klasę `Student` reprezentującą studenta i która dziedziczy z klasy `Person`. Klasa posiada pola reprezentujące numer indeksu oraz kierunek studiów. Stwórz odpowiedni konstruktor w klasie. Stwórz w klasie metodę `string to_string()`, która zwróci reprezentację obiektu w postaci stringu a następnie stwórz operator wyjścia dla klasy `Student`. Sprawdź czy w metodzie `to_string()` klasy `Student` masz dostęp do prywatnych pól klasy `Person`.

3. W funkcji `main` stwórz obiekty klasy `Person` i `Student`. Wypisz na ekranie stan każdego z tych obiektów posługując się metodą `to_string()`. Sprawdź czy z obiektu klasy `Student` masz dostęp do prywatnych i publicznych składowych klasy `Person` i `Student`.

4. Stwórz wektor obiektów klasy `Person` i dodaj do niego kilka obiektów klasy `Student`. Wypisz na ekranie stan każdego obiektu w wektorze za pomocą metody `to_string()`. Zwróć, uwagę na to, która metoda `to_string()` została wywołana: z klasy `Person` czy `Student`.

5. Stwórz wektor wskaźników do obiektu klasy `Person`. Dodaj do wektora obiekty klasy `Student`. Aby utworzyć obiekt i uzyskać do niego wskaźnik należy posłużyć się operatorem `new`, np. `new Student(...)`. Wypisz na ekranie stan każdego obiektu w wektorze za pomocą metody `to_string()`. Zwróć, uwagę na to, która metoda `to_string()` została wywołana: z klasy `Person` czy z klasy `Student`. Następnie poprzedź metodę `to_string()` w klasie `Person` słowem kluczowym `virtual` i ponownie uruchom program. Ponownie zaobserwuj, która metoda `to_string()` została wywołana.

6. Napisz funkcję która rozwiązuje równanie kwadratowe $ax^2+bx+c=0$. Wynik powinien być zwracany za pomocą argumentów funkcji. W przypadku gdy równanie nie posiada rozwiązań w zbiorze liczb rzeczywistych, funkcja powinna zgłaszać wyjątek “Equation has no real solution”. W funkcji `main` poproś użytkownika o wprowadzenie współczynników a, b, c a następnie wypisz na ekranie rozwiązania równania. Przechwyć wyjątek który może być wyrzucony przez tą funkcję. Zmień typ przechwytywanego wyjątku i uruchom ponownie program. Zaobserwuj co się dzieje w przypadku nie przechwyconego wyjątku.

7. W funkcji `main` wykonaj następujące zadania:

    a. Wczytaj jedną linię ze standardowego wejścia.

    b. Przekształć wczytany string na liczbę typu int używając funkcji `std::stoi`.

    c. przekształć tę liczbę na string w formacie szesnastkowym. Posłuż się obiektem klasy `std::stringstream`. Wypisz na ekranie wynik konwersji.

    d. Przeczytaj w dokumentacji (http://www.cplusplus.com/reference/string/) jakie wyjątki mogą być wyrzucone przez funkcję `std::stoi`. Dodaj kod który przechwytuje te wyjątki i wypisuje do strumienia `std::cerr` odpowiedni komunikat.

    e. Wprowadź z klawiatury wartości, które spowodują wyrzucenie przez funkcję stoi odpowiednich wyjątków.

8. Stwórz klasę o nazwie `StaticArray`, która reprezentuje statyczną tablicę 16 nieujemnych liczb całkowitych. Następnie utwórz klasy `InvalidIndex` oraz `InvalidValue` reprezentujące odpowiednie wyjątki. Klasy te

powinny zawierać metodę `const char* what()`, która zwraca tekstową reprezentację wyjątku. W klasie `StaticArray` powinny być zaimplementowane:

a. funkcja `int& at( int index )`, która zwraca wartość w tablicy pod wskazanym indeksem. Jeśli przekazany do funkcji indeks jest poza granicami tablicy to funkcja powinna zgłosić wyjątek `InvalidIndex`.

b. funkcja void `set_item( int index, int value )`, która ustawia w tablicy przekazaną wartość pod wskazanym indeksem. Funkcja powinna zgłosić wyjątek `InvalidIndex`, gdy indeks jest poza rozmiarami tablicy lub wyjątek `InvalidValue`, gdy podana wartość jest mniejsza niż zero.

W funkcji `main` utwórz obiekt klasy `StaticArray`, a następnie użyj funkcji `at()` oraz `set_item()` w taki sposób aby zgłosiły one wyjątki.