Laboratorium 5 – dziedziczenie

Zadanie 1

Zdefiniuj klasę Zwierze. Powinna ona zawierać:

- prywatne pole mImie typu `string`
- konstruktor ustawiający imię zwierzęcia (imię jest parametrem konstruktora)
- metodę `ustawImie(string imie)` zmieniającą imię zwierzęcia
- stałą metodę `naLancuch()`, która zwraca stałą referencję do pola imię.

W funkcji `main()`

- stwórz i wyświetl (używając metody `naLancuch()`) obiekt klasy `Zwierze`.
- stwórz stałą referencję do obiektu klasy zwierzę i wyświetl imię zwierzęcia używając tej referencji. Co się dzieje gdy metoda naLancuch() nie będzie stała?

Dziedziczenie

W programowaniu obiektowym bardzo często wykorzystuje się dziedziczenie. Umożliwia ono stworzenie klasy, która zawiera elementy wspólne obiektów a następnie klasy potomnych implementujących tylko dodatkowe szczegóły. Klasy potomne przejmują od klasy nadrzędnej wszystkie pola i metody. Dzięki temu implementacja wszystkich potrzebnych funkcji jest krótsza i łatwiejsza w konserwacji (jeżeli trzeba naprawić błąd dotyczący części wspólnej – wystarczy poprawić błąd w klasie bazowej).

W języku C++ dla pól i metod używa się specyfikatorów dostępu (`private`, `protected` i `public`). Pola i metody prywatne nie są dostępne w klasie potomnej. W przypadku dziedziczenia pola określa się również specyfikator dostępu dla całej klasy nadrzędnej. Klasa może dziedziczyć ze specyfikatorem: publicznym (`public`), chronionym (`protected`) lub prywatnym (`private`). Specyifkator powoduje ograniczenie poziomu dostępu np. jeżeli specyfikator to `protected` wtedy wszystkie pola/metody pochodzące z klasy bazowej będą miały specyfikator `protected` lub `private` w klasie pochodnej. Innymi słowy pola/metody publiczne z klasy bazowej staną się chronione. Najczęściej jednak przy dziedziczeniu używa się specyfikatora `public`.

Jeżeli zachodzi konieczność przekazania parametrów do konstruktora klasy bazowej można użyć listy inicjalizacyjnej. Można na niej po dwukropu wymienić: konstruktor klasy nadrzędnej wraz z parametrami, pola klasy wraz z wartościami początkowymi.

```
class Klasa
{
private:
	int mPole1;
protected:
	int mPole2;
public:
	Klasa(int parametr1)
	{
		mPole1 = parametr1;
		mPole2 = 0;
	}
};

class KlasaPotomna: public Klasa
{
private:
```

```
	int mPole3;
public:
                        //lista inicjalizacyjna - wywołanie konstruktora Klasy
	KlasaPotomna(int parametr1, int parametr3) : Klasa(parametr1)
	{
		//mPole1 = 5  //na to kompilator nie pozwoli
		mPole2 = 5; //OK
		mPole3 = parametr3;
	}
};
```

Zadanie 2

Stwórz klasę `Pies`, która

- dziedziczy (publicznie) po klasie `Zwierze`
- posiada konstruktor ustawiający imię psa, konstruktor powinien korzystać z konstruktora klasy `Zwierze`.
- posiada dwie funkcje `szczekaj()` oraz `dajLape()`, które wyświetlają odpowiednie napisy.

W funkcji main()

- utwórz obiekt klasy `Pies`
- wyświetl nowy obiekt
- `wywołaj funkcje szczekaj() oraz dajLape()`

Co się stanie gdy specyfikatorem dziedziczenia będzie `private`?

Funkcje wirtualne i polimorfizm

Zadanie 3

W klasie `Zwierze`

- dodaj stałą metodę `string dajRodzajZwierzecia()`, która zwraca napis „`Zwierzątko: `„
- zmodyfikuj metodę `naLancuch()` tak aby korzystała z nowej metody i zwracała napis „Zwierzątko: imię zwierzątka”. Dodatkowo zmodyfikuj metodę tak aby zwracała nowy łańcuch a nie referencję do niego.

W klasie `Pies`

- dodaj stałą metodę `string dajRodzajZwierzecia()`, która zwraca napis „`Pies: `„

Jaki jest efekt uruchomienia programu?

Czasami zdarzają się sytuacje, w których zadanie, które metoda ma wykonać jest dość ogólne, jednakże istnieje kilka wariantów tego zadania różniących się drobnymi szczegółami. Tak jak w przykładzie powyżej (tam jedyną różnicą jest rodzaj zwierzęcia). Aby uniknąć tworzenia kilku różnych implementacji zadania w klasach pochodnych można wykorzystać metody wirtualne. Różnica między „zwykłą” metodą a metodą wirtualną polega na tym, że wywoływane metody wirtualne zawsze pochodzą z rzeczywistej klasy obiektu (niezależnie od tego gdzie zostały wywołane), natomiast metody „zwykłe” pochodzą z klasy do której mamy referencję/wskaźnik/zmienną w momencie wywołania (dotyczy to również wywołań wewnątrz klasy).

```
class Klasa
{
public:
	void metoda()
	{
		metodaWirtualna();
	}
	virtual void metodaWirtualna();
```

```
};

class KlasaPotomna: public Klasa
{
public:
	void metodaWirtualna();
};
```

Zadanie 4

Zmodyfikuj klasę `Zwierze` tak aby w przypadku obiektu klasy `Zwierze` pojawiał się napis „`Zwierzątko: imię zwierzątka`” a w przypadku obiektu klasy Pies napis: „`Pies: imię psa`”. Jak można osiągnąć ten cel za pomocą metod wirtualnych?
Dodatkowo w funkcji `main()`:

- Utwórz referencje `Zwierze& refZwierzatko` oraz `Zwierze& refPies` i zainicjuj je za pomocą odpowiednich obiektów.
- Za pomocą referencji wywołaj metody `naLancuch()` w poszczególnych obiektach
- Utwórz dwa wskaźniki (`Zwierze*`) i przeprowadź analogiczną próbę

Co się stanie gdy usunie się słowo kluczowe `virtual`?

Rzutowanie dynamic_cast<>()

Z hierarchią dziedziczenia wiąże się czwarty rodzaj rzutowania (niewprowadzony wcześniej na zajęciach) – mianowicie dynamic_cast<>(). Pozwala ono na rzutowanie wskaźników i referencji w dół hierarchii dziedziczenia – tzn. na rzutowanie wskaźnika/referencji do obiektu klasy nadrzędnej na wskaźnik/referencję do jednej z klas pochodnych. W przypadku gdy takie rzutowanie jest nieprawidłowe wskaźniki przyjmują wartość NULL a dla referencji zgłaszany jest wyjątek std::bad_cast (zdefiniowany w pliku nagłówkowym typeinfo.

Zadanie 5

W funkcji `main()`:

- Utwórz wskaźniki `Pies* wskPies2` oraz `Pies* wskPies3`. Spróbuj przypisać do nich wskaźniki z zadania 4.
- Utwórz referencje `Pies& refPies2` oraz `Pies& refPies3` i spróbuj przypisać do nich referencje z zadania 4.

Zadanie 6

Utwórz klasę `SchroniskoDlaZwierzat`:

- dane w są przechowywane w tablicy wskaźników `Zwierze*`
- pojemność schroniska to 100 miejsc (tablica statyczna)
- posiada publiczną metodę `dajZwierze(numer)` zwracającą wskaźnik (typu `Zwierze*`) do zwierzęcia o zadanym numerze.

W funkcji `main()`

- utwórz obiekt `schronisko`
- utwórz kilka psów i umieść je w schronisku
- spowoduj aby psy w schronisku szczekały (użyj funkcji `dajZwierze()`)
- wyświetl imiona psów ze schroniska.

Dziedziczenie wielokrotne i czyste funkcje wirtualne

W C++ klasa może dziedziczyć po wielu klasach bazowych. Tak jak w przypadku dziedziczenia jednokrotnego klasa pochodna przejmuje wszystkie pola i metody klas nadrzędnych. W związku z

tym może wystąpić problem niejednoznaczności nazw – klasy bazowe mogą mieć pole o takiej samej nazwie. Aby rozwiązać ten problem należy użyć operatora `::` i podać przed nim nazwę klasy bazowej zawierającej właściwe pole.

Z dziedziczeniem wielokrotnym łączy się zagadnienie czystych funkcji wirtualnych. Ich cechą charakterystyczną jest to, że nie posiadają implementacji. Może to wydawać się dziwne, ale takie metody są często stosowane do tworzenia tzw. *interfejsów*. Interfejs to po prostu zbiór prototypów metod (określa ich nazwy, parametry i ich typy). Klasa zawierająca czystą metodę wirtualną jest klasą abstrakcyjną. Klasa abstrakcyjna może oczywiście zawierać pola. Typowym zastosowaniem klasy abstrakcyjnej jest tworzenie klas pochodnych. Klasy pochodne powinny implementować metody z klasy abstrakcyjnej. Dzięki takiemu podejściu nazwy i parametry metod w klasach pochodnych są jednolite.

```
class Klasa
{
protected:
	int mPole;
	//pozostała treść klasy...
};

class Klasa2
{
protected:
	int mPole;
public:
	//nie ma implementacji – to że metoda jest czysta
	//zaznacza się za pomocą =0
	virtual void metodaCzystaWirtualna()=0;
};

//dziedziczy po dwóch klasach
class KlasaPotomna: public Klasa, public Klasa2
{
public:
	//w którejś z klas pochodnych trzeba stworzyć implementację czystej
	//metody wirtualnej → dzięki temu już nie będzie czysta a tylko wirtualna
	void metodaCzystaWirtualna()
	{
		//odwołanie do pola o tej samej nazwie
		cout<<Klasa::mPole<<Klasa2::mPole<<endl;
		//reszta implementacji
	}
};
```

Zadanie 7

Stwórz klasę `FunkcjeZyciowe`, która:

- zawiera metody `jedz(iloscPozywienia)`, `pij(iloscWody)`, `spij(czas)`, `bawSie(czas)`. Metody mają być czystymi metodami wirtualnymi.

W klasie `Zwierze`:

- dodaj chronione pola `glod`, `pragnienie`, `zmeczenie`, `humor` których wartość początkowa wynosi 0
- zmodyfikuj funkcję `naLancuch()` tak aby wyświetlała wartości nowych pól

W klasie `Pies`:

- zmodyfikuj klasę pies tak aby dziedziczyła również po klasie FunkcjeZyciowe
- zaimplementuj funkcje `jedz()`, `pij()`, `spij()` tak aby zmniejszały wartości pól opisujących głód, pragnienie, zmęczenie i humor
- zaimplementuj funkcję `bawSie(czas)`, która w zależności od czasu zwiększa głód,

pragnienie, zmęczenie oraz humor psa

Utwórz klasę `Kot`:

- która dziedziczy po klasach: `Zwierze` i `FunkcjeZyciowe`
- implementuje metodę `miaucz()` oraz `myjSie()`
- implementuje funkcję `dajRodzajZwierzecia()`, która zwraca napis `„Kot: „`,

W funkcji `main()`

- użyj funkcji `bawSie()`, `jedz()`, `pij()` oraz `spij()` dla obiektu klasy Pies
- po każdym wywołaniu funkcji wyświetl obiekt klasy `Pies`
- utwórz obiekt klasy `Kot` i przypisz go do referencji `FunkcjeZyciowe& funkcje`
- korzystając z referencji funkcje wywołaj metody `bawSie(), jedz(), pij()` oraz `spij()`
- wyświetl obiekt, na który wskazuje referencja `funkcje (dynamic_cast<>())`
- utwórz kolejną referencję typu `FunkcjeZyciowe&` i przypisz do niej obiekt klasy `Pies`. Następnie powtórz dwa poprzednie punkty
- Utwórz obiekt klasy `FunkcjeZyciowe`.