Języki i paradygmaty programowania 2

Laboratorium 6. Dziedziczenie. Lista inicjalizacyjna konstruktora. Funkcje wirtualne. Abstrakcyjne klasy bazowe. Dziedziczenie, a dynamiczny przydział pamięci.

Jednym z głównym celów programowania obiektowego jest powtórne wykorzystanie kodu. Pracując na kolejnymi projektami chcielibyśmy wykorzystać elementy z projektów poprzednich. Nie musimy tracić czasu na pisanie go na nowo, na powtórne testy.

Załóżmy, że wykupiliśmy jakąś biblioteczkę do języka C np. do obsługi bazy danych. Dostaliśmy dll i instrukcję korzystania. Jak możemy zmodyfikować tą biblioteczkę? Będzie to niemożliwe bez dostępu do kodu źródłowego. Nawet mając taki dostęp nigdy nie mamy pewności czy nasza zmiana nie zepsuje czegoś co już działa.

W języku C++ mamy lepszy sposób na modyfikacje/poszerzenie istniejącego już kodu -> dziedziczenie. Dziedziczymy nową klasę(klasa pochodna) po klasie istniejącej(nazywanej bazową).

Dzięki dziedziczeniu możemy:

a) Dodać nowe metody do istniejącej klasy
b) Dodać nowe pola do klasy
c) Zmieniać działanie istniejących już metod.

Nie potrzebujemy kodu klasy bazowej – wystarczą pliki nagłówkowe i skompilowany kod biblioteki.

1. Przygotuj klasę Person zawierają name i surname jako string, a także age jako int. Niech to będą pola prywatne. Umieść nagłówek klasy w pliku .h a implementację w pliku .cpp. Przygotuj konstruktor domyślny, który nadaje imię: Jan, nazwisko: Kowalski i wiek 18. Napisz metodę void display() wyświetlającą informacje o osobie.

   Przygotuj konstruktor sparametryzowany zawierający te 3 pola. Dokonaj jego implementacji z wykorzystaniem listy inicjalizacyjnej wyglądającej w następujący sposób:

```
Person::Person(string name, string surname, int age) :name(name),
surname(surname), age(age) {}
```

2. Dziedziczenie to przykład relacji jest czymś.

Spróbujmy wykorzystać ten mechanizm do przygotowania klasy Student. Student jest człowiekiem więc możemy przygotować klasę pochodną Student dziedziczącą po klasie bazowej Person.

```
class Student : public Person
{

};
```

Dwukropek oznacza dziedziczenie. W tym przypadku jest to dziedziczenie publiczne.

To oznacza, że:

a) publiczne składowe oraz metody klasy bazowej stają się publicznymi składowymi klasy pochodnej
b) prywatne składowe oraz metody klasy bazowej są dostępne w klasie pochodnej wyłącznie z poziomu publicznych i chronionych metod klasy bazowej.

Dopiszmy do klasy student pola identificationNumber będący stringiem (będzie to numer albumu) oraz faculty(wydział) jako string. Niech te pola będą prywatne.

Dodatkowo klasa pochodna potrzebuje własnego konstruktora. Ten konstruktor nie może wyglądać tak:

```
Student::Student(string name, string surname, int age, string identificationNumber,
string faculty)
{
    this->name = name; //nie mamy dostępu do składowej prywatnej !
    this->surname = surname;//nie mamy dostępu do składowej prywatnej !
    this->age = age;//nie mamy dostępu do składowej prywatnej !
    this->identificationNumber = identificationNumber;
    this->faculty = faculty;
}
```

Nie mamy dostępu do składowych prywatnych. Musimy posłużyć się konstruktorem bazowym i napisać coś takiego:

```
Student::Student(string name, string surname, int age, string identificationNumber,
string faculty):Person(name,surname,age)
{
    this->identificationNumber = identificationNumber;
    this->faculty = faculty;
}
```

Jeżeli pominiemy ten fragment:

```
:Person(name,surname,age)
```

To automatycznie zostanie wywołany konstruktor domyślny klasy Person.

Generalnie najpierw zawsze zostaje wykonany konstruktor klasy bazowej, potem pochodnej. Jeżeli chodzi o destruktory jest na odwrót.

Przygotuj implementacje destruktorów w klasie bazowej oraz pochodnej i zweryfikuj to.

Spróbuj przygotować metodę void getInformation() gdzie wyświetlimy wszystkie informacje o studencie.

Powinna wyglądać mniej więcej tak:

```
void Student::getInformation()
{
    cout << name << endl;
}
```

Czy zadziała ona poprawnie ?

Niestety nie. Nie mamy dostępu do składowych prywatnych w klasie pochodnej. Musimy w klasie bazowej przygotować publiczne metody getName, getSurname oraz getAge.

Publiczne metody klasy bazowej możemy wykorzystać zarówno dla obiektów klasy bazowej jak i pochodnej. Przetestuj przykład:

```
Person p("Jan", "Nowak", 13);
p.display();
Student s("Jan", "Nowak", 13,"111","FMI");
s.display();//możemy skorzystać bo jest to publiczna metoda klasy Person
s.getInformation();//metoda klasy pochodnej
//p.getInformation();// niepoprawne - klasa bazowa nie zna metod klasy pochodnej
```

3. Relacja pomiędzy klasą bazową, a pochodną.

Jednym z najważniejszych powiązań jest możliwość wskazywania wskaźnikiem klasy bazowej na klasę pochodną.

Dzięki temu możemy wywoływać publiczne metody klasy bazowej oraz oczywiście nie mamy dostępu do metod klasy pochodnej.

```
Student s("Jan", "Nowak", 13, "111", "FMI");

Person & pRef = s;
pRef.display();

Person * pPoint = &s;
pPoint->display();
```

Nie możemy jednak zrobić na odwrót.

Spróbujmy przygotować funkcje nie należącą do żadnej klasy:

```
void Show(const Person &p)
{
      cout << p.getAge(); //pamiętaj, że funkcje getAge musi mieć const pilnującego
niezmienność wskaźnika this
    }
```

Wywołajmy ją:

```
Student s("Jan", "Nowak", 13, "111", "FMI");
Person p("Jan", "Nowak", 13);
Show(s);
Show(p);
```

Jak widzimy, możemy wykorzystać tą metodę zarówno dla obiektów klasy bazowej jak i pochodnej.

Należy pamiętać, że w przypadku dynamicznej alokacji pamięci należy dodatkowo przygotować konstruktor kopiujący.

Rozpatrzmy jeszcze jedno zagadnienie:

```
Student s("Jan", "Nowak", 13, "111", "FMI");
Person p;
p = s;
p.display();
```

W tym przypadku niejawnie wykonuje się operator= dla klasy Person. Ponownie w przypadku dynamicznej alokacji pamięci należało by go w pełni poprawnie zaimplementować.

4. Relacje jest czymś i relacja zawierania.

Dziedziczenie reprezentuje relację jest czymś, np.:

```
class Owoc
{
public:
    string kolor;
    double waga;
};

class Jablko : public Owoc
{

};
```

Przykład relacji zawierania – śniadanie zawiera jabłko. Spróbujmy dziedziczyć:

```
class Sniadanie : public Jablko
{
    // problem -> przecież śniadanie nie ma jednego koloru
};
```

Zupełnie bez sensu. Poprawnie powinniśmy przygotować implementację tej klasy następująco:

```
class Sniadanie
{
private:
    Jablko j;
};
```

Albo już w ogóle pomyśleć bardzo przyszłościowo:

```
class Sniadanie
{
private:
    Owoc* owoc;//jeszcze nie wiemy co to będzie na śniadanie, zdecydujemy na etapie wykonania(w sklepie), a nie kompilacji
};
```

Jedna z najważniejszych zasad myślenia obiektowego mówi, że kod powinien być zamknięty na modyfikacje, ale otwarty na rozwój. Teraz daliśmy sobie szansę na różnorodne śniadanie ☺

5. Polimorficzne dziedziczenie publiczne.

Wróćmy do klasy Person. Mieliśmy tam metodę display(). W klasie Student mamy więcej pól. Co powinniśmy zrobić ? Napisać metodę display2() ? Oczywiście, że nie. Z pomocą przychodzi polimorficzne dziedziczenie publiczne.

Polega ono na tym, że metody powinny inaczej zachowywać się dla obiektów klasy pochodnej, a inaczej dla obiektów klasy bazowej. Dwa mechanizmy z tym związane:

a) Ponowne definiowanie metody klasy bazowej w klasie pochodnej
b) Używanie metod wirtualnych

Przygotuj metodę void display() w klasie student wyświetlającą wszystkie właściwości.

Powinna wyglądać mniej więcej tak:

```
void Student::display()
{
    cout << getName() << " " << getSurname() << " " << getAge() <<
          " "<<identificationNumber<<" "<<faculty<<endl;
}
```

Wywołaj teraz następujący fragment:

```
    Person p("Jan", "Nowak", 13);
    p.display();
    Student s("Jan", "Nowak", 13, "111", "FMI");
    s.display();
```

Wróćmy teraz jednak do przykładu z referencją:

```
    Student s("Jan", "Nowak", 13, "111", "FMI");
    Person & pRef = s;
    pRef.display();
```

Wskaźnik klasy bazowej(czy też referencja) uruchomi metodę z klasy bazowej ! Jeżeli chcemy skorzystać z metody klasy pochodnej musimy do nagłówka metody display w klasie Person dodać słówko kluczowe virtual:

```
virtual void display();
```

Nie ma konieczności dodawania słówka virtual w klasie pochodnej, ale można to zrobić. Nie dodajemy też słówka virtual przy implementacji metody.

Przygotuj metodę string getClassName w klasie Person, która zwraca ciąg „Person” oraz metodę o dokładnie takiej samej nazwie getClassName w klasie Student.

Następnie przygotuj w klasie Student będą void displayBaseClassInformation() i wywołaj metodę z klasy bazowej. Implementacja powinna być następująca:

```
void Student::displayBaseClassInformation()
{
	cout << Person::getClassName(); //operatorem zasięgu mówimy, którą metodę chcemy wywołać
    }
```

6. Destruktory wirtualne.

Przygotuj w klasie Person oraz Student destruktory. Niech wypisują one informacje: „Destruktor klasy Person” oraz „Destruktor klasy Student”

```
Person *persons[3]; //możemy tutaj trzymać obiekty klasy Person jak i Student
persons[0] = new Student("Jan", "Nowak", 13, "111", "FMI");
persons[1] = new Student("Jan", "Kowalski", 13, "111", "FMI");
persons[2] = new Student("Jan", "Małysz", 13, "111", "FMI");

for (int i = 0; i < 3; i++)
{
	delete persons[i];
}
```

A teraz dopisz słówko virtual przy destruktorze klasy Person i porównaj różnicę. Destruktory wirtualne zapewniają poprawność kolejność wywoływania destruktorów – najpierw te klasy pochodnej, a potem bazowej.

7. Wiązanie statyczne oraz dynamiczne.

Przeciążenie funkcji, operatorów -> decyzja, której funkcji użyć na etapie kompilacji.

Funkcje wirtualne -> decyzja na etapie wykonania programu.

Przekształcenie referencji(wskaźnika) klasy pochodnej na bazową nazywa się rzutowaniem w górę i można stosować go w publicznym dziedziczeniu bez konieczności rzutowania. Poniższy fragment kodu jest w pełni poprawny:

```
Student s("Jan", "Nowak", 13, "111", "FMI");
Person & pRef = s;
```

Przekształcenie odwrotne wymaga jawnego rzutowania (rzutowanie w dół). Bardzo często też nie będzie miało żadnego zastosowania. W klasy pochodnej pojawiły się metody nieznane w klasie bazowej. Przetestuj poniższy fragment kodu:

```
Person p("Jan", "Nowak", 13);
Student *s =  (Student *)&p;
s->displayBaseClassInformation();
```

Dlaczego nie wszystko jest wirtualne ? Jest to kwestia po pierwsze wydajności, a po drugie niektórych aspektów klas bazowych nie powinno się modyfikować.

Każdy obiekt klasy zawiera ukrytą tablicę funkcji wirtualnych(vtbl virtual function table). Musimy ją trzymać w pamięci, dodatkowo przed wywołaniem metody musimy sprawdzać w tej tablicy, którą metodę należy wywołać.

Dodatkowa uwaga: konstruktory nie mogą być wirtualne. Funkcje zaprzyjaźnione również nie mogą być wirtualne – nie są składowymi klasy.

8. Kontrola dostępu – poziom protected.

Do tej pory korzystaliśmy wyłącznie z modyfikatorów public oraz private. Protected jest bardzo podobny do private. Jedyna różnica polega na tym, że w klasie pochodnej mamy bezpośredni dostęp do składowych protected, tak jakby były publiczne.

9. Abstrakcyjne klasy bazowe.

Postawmy sobie za zadanie opracowanie programu umożliwiającego obliczanie pól oraz obwodów figur. Niech to będą koło, kwadrat oraz prostokąt.

Nie znając dziedziczenia przygotowalibyśmy trzy klasy. Znając dziedziczenie spróbowalibyśmy przygotować klasę Figure, a potem dziedziczyć po niej tworząc klasy koło, kwadrat, prostokąt.

Spróbujmy więc:

```
class Figure {
private:
	double field;
	int a; //bok tylko czego?
	int b; // drugi bok dla ewentualnego prostokąta ?
	int r; //promień koła ?
public:
	void computeField() { co tutaj będzie ???}
};

class Circle : Figure
{

};
```

Rozwiązanie jak widać ma kilka niedogodności. C++ przygotował na tą okoliczność abstrakcyjne klasy bazowa. To takie klasy, które zawierają przynajmniej jedną funkcję czysto wirtualną czyli bez implementacji. Nasze nowe podejście będzie prezentować się następująco:

```
class Figure {
private:
	double field;
public:
	virtual void computeField() = 0; //funkcja czysto wirtualna
	void setField(double field) { this->field = field; }
	void displayField() { cout << field << endl; }
};
```

```
class Circle : public Figure
{
private:
	double r;
public:
	virtual void computeField();
	Circle();
	Circle(double r) { this->r = r; }
};

void Circle::computeField()
{
	double f = 2 * M_PI*r*r;
	setField(f);
}
```

Pamiętaj by dokonać includów po „pch.h”:

```
#define _USE_MATH_DEFINES
#include <cmath>
```

Przetestuj poniższy kod:

```
	Circle c(4.5);
	Figure * f = &c;
	f->computeField();
	f->displayField();
```

Dokonaj implementacji klas dla kwadratu oraz prostokąta.

10. Dziedziczenie a dynamiczny przydział pamięci.

Wróćmy do przykładu z laboratorium nr 6 – klasy MyString. Mamy w niej zaimplementowany konstruktor kopii, przeciążony operator przypisania oraz destruktor.

Rozszerzymy możliwości tej klasy z wykorzystaniem dziedziczenia. Utwórzmy klasę ExtendedMyString.

a) W pierwszym wariancie dodajmy do klasy jedno pole: int number;

W związku z tym czy musimy przygotować jakieś specjalne metody ? Odpowiedź brzmi nie. Rozpatrzmy dokładniej ten przypadek.

- Destruktor – nie potrzebujemy definiować destruktora. Kompilator utworzy destruktor domyślny, który wywołuje destruktor klasy bazowej.

- Konstruktor kopiujący – nie musimy przygotować nowego konstruktora bo nasze nowe pole jest typu prostego, a jeżeli chodzi o kopiowanie składowych klasy bazowej – wywołany zostanie konstruktor kopiujący klasy bazowej i wykona tą operacje

- Operator przypisania – podobnie jak w przypadku konstruktora kopii wywołany zostanie operator przypisania z klasy bazowej

b) Drugi przypadek – klasa pochodna też zawiera składowe alokowane dynamicznie

Załóżmy, że chcemy w tej klasie przechowywać dwa ciągi znaków. Dodajmy więc dodatkową składową char *secondText.

Teraz musimy zdefiniować w klasie pochodnej wszystkie wyżej wymienione metody.

Przygotujmy jeszcze konstruktor umożliwiający tworzenie nam takich obiektów:

```
ExtendedMyString::ExtendedMyString(const char* s, const char* s2) : MyString(s)
{
    secondText = new char[strlen(s2) + 1];
    strcpy_s(secondText, strlen(s2) + 1, s2);
}
```

Teraz zdefiniujmy destruktor:

```
ExtendedMyString::~ExtendedMyString()
{
    delete[] secondText;
}
```

Czas na konstruktor kopii. Musimy skopiować zawartość zmiennej secondText oraz wywołać konstruktor kopii z klasy bazowej. Powinien on wyglądać w sposób następujący:

```
ExtendedMyString::ExtendedMyString(const ExtendedMyString&s) :MyString(s)
{
    cout << "konstruktor kopii extendedstring" << endl;
    secondText = new char[strlen(s.secondText)+1];
    strcpy_s(secondText, strlen(s.secondText) + 1, s.secondText);
}
```

Przypomnijmy sobie. Konstruktor kopii używany jest, gdy:

- Kiedy nowy obiekt inicjalizowany jest za pomocą obiektu tej samej klasy

- Kiedy obiekt jest przekazywany do funkcji przez wartość

- Kiedy funkcja zwraca obiekt przez wartość

- Kiedy kompilator tworzy obiekt tymczasowy

Ostatni etap to przygotowanie przeciążonego operatora przypisania. Jego implementacja powinna wyglądać następująco:

```
ExtendedMyString & ExtendedMyString::operator=(const ExtendedMyString& st)
{
    if (this == &st)
    {
            return *this;
    }
    MyString::operator=(st); //przypisanie składowych z klasy bazowej
    delete[] secondText;
    secondText = new char[strlen(st.secondText) + 1];
    strcpy_s(secondText, strlen(st.secondText) + 1, st.secondText);
    return *this;
}
```

11. Podsumowanie rodzajów dziedziczenia.

Właściwość	Dziedziczenie publiczne	Dziedziczenie chronione	Dziedziczenie prywatne
Składowa publiczna staje się	Składową publiczną klasy pochodnej	Składową chronioną klasy pochodnej	Składową prywatną klasy pochodnej
Składowa chroniona staje się	Składową chronioną klasy pochodnej	Składową chronioną klasy pochodnej	Składową prywatną klasy pochodnej
Składowa prywatna staje się	Dostępna tylko poprzez interfejs klasy bazowej	Dostępna tylko poprzez interfejs klasy bazowej	Dostępna tylko poprzez interfejs klasy bazowej
Niejawne rzutowanie w górę	Tak	Tak, ale tylko wewnątrz klasy pochodnej	Nie

12. Podsumowanie metod klasy:

Funkcja	Może być dziedziczona	Metoda klasy czy funkcja zaprzyjaźniona	Generowana domyślnie	Może być wirtualna	Może mieć zwracany typ
Konstruktor	Nie	Metoda	Tak	Nie	Nie
Destruktor	Nie	Metoda	Tak	Tak	Nie
=	Nie	Metoda	Tak	Tak	Tak
&	Tak	Obie	Tak	Tak	Tak
Konwersja	Tak	Metoda	Nie	Tak	Nie
()	Tak	Metoda	Nie	Tak	Tak
[]	Tak	Metoda	Nie	Tak	Tak
->	Tak	Metoda	Nie	Tak	Tak
op=	Tak	Obie	Nie	Tak	Tak
new	Tak	Metoda statyczna	Nie	Nie	void *
delete	Tak	Metoda statyczna	Nie	Nie	void
Inne operatory	Tak	Obie	Nie	Tak	Tak
Inne metody	Tak	Metoda	Nie	Tak	Tak
Funkcje zaprzyjaźnione	Nie	Zaprzyjaźniona	Nie	Nie	Tak