Programowanie Obiektowe

Laboratorium 2 - podstawy zarządzania obiektami w języku C++

Przygotowanie do zajęć w domu

Ćwiczenia dotyczą zagadnień związanych z obiektami w C++. Poruszone zostaną m.in. tematy dotyczące dynamicznej alokacji pamięci w klasach, kopiowania obiektów, przekazywania parametrów przez wartość i przez referencję. Dodatkowo zilustrowane zostały funkcje zaprzyjaźnione na przykładzie operatora << zastosowanego do wypisywania danych na strumień (np. cout).

Zadanie 1. Przeanalizuj poniższy program.

```
#include <iostream>
using namespace std;
class Vektor3d {
	double _liczby[3]; //pole z tablicą liczb
public:
	Vektor3d (double x1,double x2,double x3){ //konstruktor
		_liczby[0]=x1;
		_liczby[1]=x2;
		_liczby[2]=x3;
	}
	void getCoefs(double liczby[]){ //metoda publiczna
		liczby[0]=_liczby[0];
		liczby[1]=_liczby[1];
		liczby[2]=_liczby[2];
	}
};
void wypisz(double tab[],int rozmiar){
	for(int i=0;i<rozmiar;i++)
		cout << tab[i]<<'\t';
	cout <<endl;
}
int main(){
	Vektor3d v1(1,2,3); //utworzenie obiektu
	cout << sizeof v1 <<endl;
	double l[3];
	v1.getCoefs(l);   //wywołanie metody
	wypisz(l,3);
	return 0;
}
```

Zwróć uwagę na rozmiar obiektu v1. Czy wiesz skąd on się bierze? Zwróć uwagę kiedy wywołuje się konstruktor. Możesz przeanalizować przebieg programu za pomocą debugera (lub poprzez dodanie instrukcji w rodzaju cout<<”Komunikat”;)

Klasa Vektor3d zawiera pole typu tablicowego o trzech elementach. Cała tablica zawarta jest wewnątrz obiektu Vektor3d. Na obiektach należących do klasy wykonywać można operacje bazując na ich domyślnej implementacji dostarczanej przez kompilator. Do tych operacji należą m.in. kopiowanie (tworzenie nowych obiektów przez kopiowanie) oraz przypisywanie jednego obiektu do drugiego obiektu.

Ilustruje to rozbudowana wersja funkcji main() w kolejnym przykładzie.

Przykład 1. Ilustracja kopiowania oraz przypisywania wartości obiektów

```
int main(){
	Vektor3d v1(1,2,3); //utworzenie obiektu
	cout << sizeof v1 <<endl;
	Vektor3d v2(4,5,6); //utworzenie obiektu
	Vektor3d v1Kopia1(v1); //utworzenie obiektu - konstruktor kopiujący
	Vektor3d v1Kopia2 = v1; //utworzenie obiektu - konstruktor kopiujący
	double l[3];
	v1.getCoefs(l);   //wywołanie metody
	wypisz(l,3);
	v2.getCoefs(l);   //wywołanie metody
	wypisz(l,3);
	v1 = v2; //przypisanie wartości
	v1.getCoefs(l);   //wywołanie metody
	wypisz(l,3);
	v1Kopia1.getCoefs(l);   //wywołanie metody
	wypisz(l,3);
	v1Kopia2.getCoefs(l);   //wywołanie metody
	wypisz(l,3);
	return 0;
}
```

W powyższym przykładzie wykorzystano zarówno kopiowanie obiektu przy tworzeniu (konstruktor kopiujący) jak również przypisywanie wartości obiektu do drugiego.

Konstruktor kopiujący – specjalny konstruktor, który tworzy nowy obiekt na podstawie innego obiektu tego samego typu.

Możliwości w tym zakresie biorą się z faktu, że język C++ zapewnia domyślne mechanizmy kopiowania pól z jednego obiektu do drugiego podczas operacji przypisania i konstrukcji obiektu na bazie innego obiektu tego samego typu.

Sytuacja komplikuje się gdy kopiowanie wartości poszczególnych pól należących do obiektów nie jest wystarczające. Ilustruje to poniższy przykład.

Przykład 2. Klasa wykorzystując a dynamiczną alokację pamięci

```
#include <iostream>
using namespace std;
class VektorNd {
	double *_liczby; //pole ze wskaźnikiem liczb
	int _wymiar; //wymiar wektora
public:
	VektorNd (const double liczby[],int wymiar){ //konstruktor
		_liczby = new double[wymiar];
		_wymiar = wymiar;
		for(int i=0;i<_wymiar;i++)
			_liczby[i] = liczby[i];
	}
	~VektorNd(){
		delete[] _liczby;
	}
	void getCoefs(double liczby[]){ //metoda publiczna
		for(int i=0;i<_wymiar;i++)
			liczby[i] = _liczby[i];
	}
};
void wypisz(double tab[],int rozmiar){
	for(int i=0;i<rozmiar;i++)
		cout << tab[i]<<'\t';
	cout <<endl;
}
int main(){
	const double wartosci[] = {1,2,3};
	VektorNd v1(wartosci,3); //utworzenie obiektu
	cout << sizeof v1 <<endl;
	double l[3];
	v1.getCoefs(l);
	wypisz(l,3);
	return 0;
}
```

W tym przypadku klasa VektorNd zawiera pole będące wskaźnikiem na tablicę liczb typu double. Dodatkowo zawiera pole przechowujące wymiar wektora. Utworzenie i wypełnienie odpowiedniej przestrzeni w pamięci realizowane jest przez konstruktor. Ze względu na dynamiczną alokację pamięci konieczne jest jej zwolnienie w destruktorze.

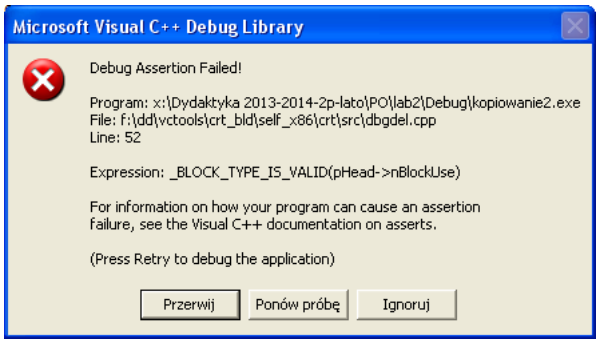
Używane w przypadku klasy Vektor3d operatory przypisania i konstrukcja obiektu na podstawie kopiowania tym razem powodują kłopoty. W celu zademonstrowania problemów wynikających z tego rozwiązania dodano metodę setCoef oraz komunikat do destruktora.

Przykład 3. Ilustracja problemów przy przypisaniu wartości w przypadku pól wskaźnikowych.

```
class VektorNd {
	[.....................................................]
	~VektorNd(){
		cout << "zwalniam pamiec pod adresem: " << _liczby <<endl;
		delete[] _liczby;
	[.....................................................]
	}
	void setCoef(double wartosc,int index){
		_liczby[index] = wartosc;
	}
};
	[.....................................................]
int main(){
	const double wartosci[] = {1,2,3};
	const double wartosci2[] = {4,5,6};
	VektorNd v1(wartosci,3); //utworzenie obiektu
	VektorNd v2(wartosci2,3); //utworzenie drugiego obiektu
	double l[3];
	v1.getCoefs(l);
	wypisz(l,3);
	v2.getCoefs(l);
	wypisz(l,3);
	v1 = v2;		//kłopotliwe przypisanie
	v1.getCoefs(l);
	wypisz(l,3);
	v2.getCoefs(l);
	wypisz(l,3);
	v1.setCoef(100.0,0);	//ustawienie wartości w obiekcie v1
	v1.getCoefs(l);
	wypisz(l,3);
	v2.getCoefs(l);
	wypisz(l,3);
	return 0;
}
```

Po uruchomieniu tak zmodyfikowanego programu można zaobserwować dwa problemy:

1. Zmiana wartości wektora w jednym obiekcie (metodą setCoef) powoduje zmianę wartości również w drugim obiekcie.
2. Destruktory obiektów v1 oraz v2, które zostają uruchomione na koniec programu próbują zwolnić tę samą pamięć. Ten problem może objawić się komunikatem w postaci:



Wszystko spowodowane jest instrukcją kłopotliwego przypisania. Podobne objawy pojawiają się w przypadku próby użycia konstrukcji przez kopiowanie.

Zadanie 2. Sprawdź co dzieje się w przypadku konstrukcji obiektu na podstawie kopiowania.

Reasumując, w języku C++ może występować kłopot w przypadku gdy w klasie pola są typu wskaźnikowego. Występuje on szczególnie gdy stosuje się operatory przypisania i konstruktory kopiujące. Można sobie z tym poradzić na różne sposoby, m.in.:

1. Nie używać kłopotliwych operatorów i konstruktorów kopiujących.
2. Zapewnić niestandardową implementację tych mechanizmów.

O ile pierwszy sposób jest bardzo drastyczny to warto go zastosować szczególnie gdy programista jest niedoświadczony lub nie wie jak dobrze zaimplementować pożądane mechanizmy.

Jednym z rozwiązań drugiego rodzaju jest samodzielna implementacja konstruktora kopiującego (w celu zapewnienia możliwości tworzenia obiektu przez kopiowanie) jak również samodzielna implementacja operatora przypisania (w celu zapewnienia możliwości przypisywania wartości z jednego obiektu do drugiego) – przeciążenie operatora.

Przykład 4. Implementacja konstruktora kopiującego oraz operatora przypisania.

```
class VektorNd {
	[.....................................................]
	VektorNd(const VektorNd& oryginal){  //konstruktor kopiujący
		_liczby = new double[oryginal._wymiar];
		_wymiar = oryginal._wymiar;
		for(int i=0;i<_wymiar;i++)
			_liczby[i] = oryginal._liczby[i];
	}
	[.....................................................]
	VektorNd& operator=(const VektorNd& right){ //implementacja operatora przypisania
		if(_wymiar!=right._wymiar){   //gdy zgodne wymiary nie potrzeba realokować
pamięci
			delete[] _liczby;
			_liczby = new double[right._wymiar]; //alokacja pamięci dla nowego
wymiaru
			_wymiar=right._wymiar;
		}
		for(int i=0;i<_wymiar;i++)
			_liczby[i] = right._liczby[i];
		return *this;
	}
	[.....................................................]
};
	[.....................................................]
int main(){
	const double wartosci[] = {1,2,3};
	const double wartosci2[] = {4,5,6,7,8,9};
	VektorNd v1(wartosci,3); //utworzenie obiektu
	VektorNd v2(v1);              //utworzenie drugiego obiektu na bazie pierwszego
	VektorNd v3(wartosci2,6);     //utworzenie trzeciego wektora z 6 wymiarami
	v2.setCoef(4.0,0);            //ustawienie wartości w obiekcie v2
	v2.setCoef(5.0,1);            //ustawienie wartości w obiekcie v2
	double l[6];
	v1.getCoefs(l);
	wypisz(l,3);
	v2.getCoefs(l);
	wypisz(l,3);
	v1 = v2;              //już nie kłopotliwe przypisanie
	v1.getCoefs(l);
	wypisz(l,3);
	v2.getCoefs(l);
	wypisz(l,3);
	v1.setCoef(100.0,0);  //ustawienie wartości w obiekcie v1
	v1.getCoefs(l);
	wypisz(l,3);
	v2.getCoefs(l);
	wypisz(l,3);
	v2 = v3;       //przypisanie wektorów o innej liczbie wymiarów
	v2.getCoefs(l);
	wypisz(l,6);
	return 0;
}
```

Warto zwrócić uwagę, że zarówno operator przypisania jak również konstruktor kopiujący w swoich deklaracjach przyjmują jako parametr referencję na obiekt macierzystego typu.

```
VektorNd(const VektorNd& oryginal){ //konstruktor kopiujący


VektorNd& operator=(const VektorNd& right){ //implementacja operatora
przypisania
```

W przypadku operatora przypisania referencja ta jest odwołaniem do obiektu występującego po prawej stronie operatora przypisania, natomiast w przypadku konstruktora kopiującego jest to odniesienie do obiektu który jest kopiowany.

Dodatkowym elementem jest zwracanie przez operator przypisania wyrażenia *this. Jest to wymagane ze względu na składnię języka która umożliwia między innymi operacje typu:

```
a = b = c;
```

Generalnie istnieje kilka wariantów deklaracji zarówno dla konstruktora kopiującego, jak również dla operatora przypisania. Szczegóły na ten temat można znaleźć m.in. w:

http://www.cplusplus.com/articles/y8hv0pDG/

Podsumowując, wyodrębnia się dwie podstawowe kategorie w zakresie kopiowania obiektów:

a. kopiowanie płytkie
b. kopiowanie głębokie

W języku C++ kopiowanie płytkie jest zapewnione przez domyślny konstruktor kopiujący. Chcąc zapewnić kopiowanie głębokie można zaimplementować własny konstruktor kopiujący. Podobna sytuacja dotyczy przypisywania wartości.

Poza sytuacją gdy wprost wywołujemy konstruktor kopiujący występują jeszcze sytuacje gdy dzieje się to nie wprost. Jest szczególnie ważne żeby programista zdawał sobie z tego sprawę.

Niejawne wywołanie konstruktora kopiującego odbywa się podczas przekazywania parametrów przez wartość do funkcji lub metody. Ilustruje to przykład poniżej.

Przykład 5. Różnica pomiędzy przekazywaniem parametru przez wartość, a przez referencję

```
class VektorNd {
	[.....................................................]
	~VektorNd(){
		cout << "zwalniam pamiec pod adresem: " << this <<endl;
		delete[] _liczby;
	}
	[.....................................................]
	double getCoef(int index){ //pomocznicza metoda
		return _liczby[index];
	}
	int getDim(){ 		//pomocznicza metoda
		return _wymiar;
	}
	[.....................................................]
};
	[.....................................................]
void wypiszV1(VektorNd v){
	cout << "|----------BEGIN-----------------------------|"<<endl;
	cout << "Jestem wektorem pod adresem: " << &v <<endl;
	int wymiar = v.getDim();
	cout << "Mój wymiar to: " << wymiar << endl;
	cout << "A elementy:"<<endl;
	for(int i=0;i<wymiar;i++)
		cout <<"["<<i<<"] = \t" << v.getCoef(i)<<endl;
	cout << "|-----------END-------------------------------|"<<endl;
}
void wypiszV2(VektorNd& v){
	[...........TAKIE SAMO CIAŁO JAK wypiszV1()...........]
}
int main(){
	const double wartosci[] = {1,2,3};
	VektorNd v1(wartosci,3); //utworzenie obiektu
	wypiszV1(v1);
	wypiszV2(v1);
}
```

W powyższym przykładzie porównano dwie funkcje wypiszV1 oraz wypiszV2. Pierwsza z nich jako parametr przyjmuje wartość wektora natomiast druga referencję. Różnice w działaniu można zaobserwować m.in. w innym adresie pod jakim znajduje się wektor (adres jest raportowany przez funkcję instrukcją cout<<).

Jedyna różnica pomiędzy funkcjami występuje w deklaracji typu parametru. Jednak efektywność drugiej funkcji jest znacznie wyższa gdyż nie następuje niejawne kopiowanie i utworzenie de facto dodatkowego obiektu (obiektu tymczasowego).

Zadanie 3. Sprawdź, że w przypadku funkcji wypiszV1 rzeczywiście następuje wywołanie konstruktora kopiującego.

Sprawdź również, że w przypadku funkcji wypiszV2 nie występuje wywołanie konstruktora kopiującego.

Zadanie 4. Przenieś implementację funkcji wypiszV2 do klasy VektorNd jako metody.

Przy okazji uogólnić funkcję wypisz aby umożliwiała wypisywanie wektora na dowolny strumień wyjściowy, a nie tylko na standardowe wyjście.

```
class VektorNd {
[................................................]
      void wypisz(ostream & out = cout){
            out << "|-------------------------BEGIN-------------------------
------|"<<endl;
            out << "Jestem wektorem pod adresem: " << this <<endl;
            out << "Mój wymiar to: " << _wymiar << endl;
            out << "A elementy:"<<endl;
            for(int i=0;i<_wymiar;i++)
                  out <<"["<<i<<"] = \t" << _liczby[i]<<endl;
            out << "|-------------------------END---------------------------
------|"<<endl;
      }
};
int main(){
      const double wartosci[] = {1,2,3};
      VektorNd v1(wartosci,3);
      v1.wypisz();
      VektorNd v2(v1);
      v2.wypisz();
      v2.setCoef(5,0);
      v1.wypisz();
      v2.wypisz();
}
```

Przeniesienie funkcji do wewnątrz klasy w znacznym stopniu poprawia projekt systemu z punktu widzenia obiektowości. Nie ma już potrzeby jawnego przekazywania wypisywanego obiektu jako parametru funkcji. W implementacji pojawił się wskaźnik **this** równy adresowi bieżącego obiektu.

Funkcja wypisz została poprawiona jeszcze pod jednym względem. Otrzymała parametr typu referencyjnego:

```
ostream & out = cout
```

W implementacji zamiast znanej konstrukcji cout<< stosuje się out<<. Nie jest to literówka. Obiekt cout jest szczególnym egzemplarzem klasy zgodnej z ostream.

Parametr out ma zadeklarowaną wartość domyślną cout odpowiadającą standardowemu strumieniowi wyjściowemu. Programista może zrezygnować z podawania czegokolwiek wówczas informacja zostaje wypisana na standardowe wyjście (najczęściej na ekran). Istnieje możliwość wywołania z parametrem inny niż domyślny w celu wypisania informacji na inny strumień.

Pliki w języku C++ mogą być otwierane i dostępne w postaci strumieni. Ilustruje to kolejny przykład:

```
#include <iostream>
#include <fstream>
using namespace std;
class VektorNd {
[.....................................................]
};
int main(){
      const double wartosci[] = {1,2,3};
      VektorNd v1(wartosci,3);
      ofstream file("C:\\plik.txt");
      v1.wypisz(file);
}
```

Konieczne jest włączenie pliku nagłówkowego fstream. Plik jest otwierany i następuje do niego zapis. Zamknięcie pliku odbywa się automatycznie podczas destrukcji obiektu file. Obiekt file jest zmienną automatyczną wewnątrz funkcji main zatem wyjście z funkcji spowoduje destrukcję obiektu file i w konsekwencji zamknięcie pliku.

Istnieje możliwość aby obiekty naszej klasy umożliwiały zastosowanie składni znanej z instrukcji cout<<. W tym celu należy utworzyć implementację dla operatora << (przesunięcia bitowego). W przypadku tego operatora rozróżnia się lewy operand, prawy operand oraz wartość zwracaną (podobnie zresztą jak w przypadku operatora przypisania i większości operatorów dwuargumentowych).

Operator taki musi być zaimplementowany jako funkcja na zewnątrz naszej klasy. Wynika to stąd, że lewy operand nie jest obiektem naszej klasy. Aby jednak funkcja zewnętrzna miała dostęp do składników niepublicznych naszej klasy konieczne jest nadanie jej statusu zaprzyjaźnionej (friend). Ilustruje to kolejny przykład.

```
class VektorNd {
[.....................................................]
private: //modyfikator dodany żeby zademosntrować ideę friend
      void wypisz(ostream & out = cout){
 [.....................................................]
      } //koniec metody wypisz
//wewnątrz klasy należy zadeklarować fakt zaprzyjaźnienia
      friend ostream& operator<<(ostream & left,VektorNd& right);
};// koniec VektorNd

ostream& operator<<(ostream & left,VektorNd& right){
      right.wypisz(left);     //wolno wywołać gdyż operator jest friend
      return left;
}
int main(){
      const double wartosci[] = {1,2,3};
      VektorNd v1(wartosci,3);
      ofstream file("C:\\plik.txt");
      file << v1;
      //v1.wypisz();  //nie wolno gdyż funkcja wypisz jest prywantna
}
```

Zadanie 5. Sprawdź że usunięcie deklaracji

```
friend ostream& operator<<(ostream & left,VektorNd& right);
```

z wewnątrz klasy VektorNd spowoduje błąd kompilacji związany z prywatnym charakterem funkcji wypisz.

Przykładowe zadania na zajęcia

Zadanie 1. (3p) Utwórz klasę będącą tablicą wektorów. Wraz z głębokim kopiowaniem tych wektorów.

Porównaj dwie koncepcje:

```
class VectorOfVectors {
      VektorNd vektory[N];
      int size;
}

class VectorOfVectors {
      VektorNd *vektory;
      int size;
}
```

Zaimplementuj konstruktory kopiujące, destruktory, operatory przypisania i operatory wypisania na strumień.

Uwaga:
W pierwszej koncepcji ponieważ pole `vektory[N]` powoduje utworzenie N obiektów klasy `VektorNd` wewnątrz obiektu klasy `VectorOfVectors` konieczne jest aby w klasie `VektorNd` obecny był konstruktor bezparametrowy. Domyślny konstruktor bezparametrowy nie zostaje dołączony do klasy gdy występuje w niej jakikolwiek inny konstruktor.
Należy utworzyć domyślny konstruktor bezparametrowy aby utworzył on "zerowy obiekt" - czyli taki, który jest logicznie poprawny ale nie zawiera żadnych specyficznych informacji.

Zadanie 2. (1p) Zaimplementuj mechanizm wczytywania wektorów ze strumienia za pomocą operatora >>. Przetestuj działanie operatora dla strumienia plikowego i standardowego strumienia wejściowego (cin).

Funkcja **main** powinna zawierać:

```
//////////////////////////

VectorOfVectors2 vof;

cin >> vof;

cout <<vof;

//////////////////////////
```

Zadanie 3. (1p) Skorzystaj z informacji pod adresem:

http://www.cplusplus.com/articles/y8hv0pDG/

i zastosuj idiom kopiuj-zamień (ang. "copy-swap") dla operatora przypisania.

W przeciwieństwie do standardowej operacji przepisania wartości z jednego obiektu do drugiego, idiom "kopiuj-zamień" sprowadza się do wykonania następujących kroków:

krok **kopiuj**

a) zaalokowanie pamięci dla nowego, tymczasowego obiektu oraz przypisanie mu nowych wartości za pomocą konstruktora kopiującego,

krok **zamień**

b) zamainy składowych w nowoutworzonym obiekcie tymczasowym ze składowymi w dotychczasowym obiekcie (tym po lewej stronie operatora przypisania)

Konstruktor obiektu tymczasowego zarezerwuje pamięć która następnie zostanie użyta dla nowych składowych obiektu po lewej stronie operatora przypisania. Jeśli konstruktor nie powiedzie się wówczas obiekt do którego przypisujemy nie zostanie zmieniony (gdyż nie dojdziemy do fazy zamień).

Destruktor obiektu tymczasowego zwolni pamięć zajmowaną dotychczas przez składowe obiektu po lewej stronie operatora przypisania (pamięć zaalokowana dla składowych obiektu tymczasowego zostanie zwolniona dopiero podczas destrukcji obiektu znajdującego się po lewej stronie operatora przypisania.