Laboratorium 4 – konstruktory i destruktory

Metody stałe

Często istnieje potrzeba zapewnienia kontrolowanego dostępu do pól klasy. Stosuje się do tego metody dostępowe (często po angielsku nazywane getters/setters). Aby umożliwić odczyt pól klasy, równocześnie blokując możliwość ich niekontrolowanej zmiany na zewnątrz obiektu przydatne są metody stałe często (w przypadku gdy pole samo jest obiektem) łączone ze stałymi referencjami.

```
class Klasa
{
private:
	typ mPole;
public:
	//nie wszystkie stałe metody zwracają stałą referencję
	const typ& stalaMetoda() const
	{
		//tak nie można bo metoda jest stała
		//mPole = nowa_wartosc;

		//zwracamy stałą referencję – więc nie będzie można pola
		//zmienić na zewnątrz
		return mPole;
	}

};
```

Metoda stała „zapewnia kompilator”, że stan obiektu nie zostanie zmodyfikowany. Dzięki temu możliwe jest wywoływanie takich metod za pośrednictwem stałych referencji.

```
Klasa obiekt;
const Klasa& stRef=obiekt;
typ zmienna=stRef.stalaMetoda();
```

Konstruktory i destruktory

Konstruktor ma za zadanie utworzyć obiekt – przygotować go do działania. Jego nazwa jest taka sama jak nazwa klasy. Nie posiada on żadnego typu. Klasa może posiadać wiele różnych konstruktorów, które muszą różnić się parametrami.

Analogiczny do konstruktora jest destruktor jego nazwa rozpoczyna się od znaku „~” a dalsza część nazwy jest taka sama jak nazwa klasy. Zadaniem destruktora jest likwidacja obiektu.

Zarówno konstruktory jak i destruktory wywoływane są automatycznie w momencie tworzenia/likwidowania obiektów. Samodzielne wywoływanie destruktora JEST BŁĘDEM! (w 99,99% przypadków).

Treść konstruktora/destruktora może być zdefiniowana w zarówno w definicji klasy:

```
class Klasa
{
public:
	Klasa()
	{
		//treść konstruktora
		//tworzenie obiektu - np zajęcie pamięci
	}

	Klasa(int param1, int param2)
```

```
	{
		//reszta konstruktora
		//tworzenie obiektu - np zajęcie pamięci
	}

	~Klasa()
	{
		//treść destruktora - sprzątanie
		//np zwolnienie pamięci
	}
};
```

jak i poza nią:

```
class Klasa
{
public:
	Klasa();
	Klasa(int param1, int param2);
	~Klasa();
};

Klasa::Klasa()
{
	//reszta konstruktora
	//tworzenie obiektu - np zajęcie pamięci
}

Klasa::Klasa(int param1, int param2)
{
	//reszta konstruktora
	//tworzenie obiektu - np zajęcie pamięci
}

Klasa::~Klasa()
{
	//treść destruktora - sprzątanie
	//np zwolnienie pamięci
}
```

Lista inizjalizacyjna

W przypadku konstruktorów przydatna jest lista inicjalizacyjna (elementy wymienione po „:” po nagłówku konstruktora). Pozwala ona na szybkie ustawienie wartości początkowych pól. Pozwala również na przekazywanie parametrów do konstruktorów klas nadrzędnych (laboratorium 5). Elementy na liście inicjalizacyjnej warto ustawiać w takiej kolejności w jakiej występują w klasie – w innym wypadku kompilator będzie wyświetlał ostrzeżenia.

Listy inizjalizacyjne mogą się pojawiać zarówno w definicji klasy

```
class Klasa
{
private:
	int mPole1, mPole2;
public:
	Klasa() :
			mPole1(0), mPole2(0)
	{
		//treść konstruktora
		//tworzenie obiektu - np zajęcie pamięci
	}
```

```
	Klasa(int param1, int param2) :
			mPole1(param1), mPole2(param1)
	{
		//reszta konstruktora
		//tworzenie obiektu - np zajęcie pamięci
	}
};
```

jak i poza nią (w przypadku gdy konstruktor jest zdefiniowany poza definicją klasy):

```
Klasa::Klasa(int param1, int param2) :
		mPole1(param1), mPole2(param1)
{
	//reszta konstruktora
	//tworzenie obiektu - np zajęcie pamięci
}
```

Inicjalizacja pól klasy

Poza listą inicjalizacyjną do ustawienia wartości początkowych pól można użyć następujących sposobów. Niektóre wymagają włączenia obsługi standardu C++11 w kompilatorze.

```
class Inicjalizacja {
public:
    int a;   //brak inicjalizacji
    int b{}; //inicjalizacja wartościowa - konstruktor domyślny lub zerowanie
    double c[10]{};
    double d=1.0d; //inicjalizacja kopiująca
    double e[3]={1.0,2.0,3.0};
    string f="f";
    string g=f;
    double h[3]{1.0,2.0,3.0}; //inicjalizacja listowa - uzupełnienie listą wartości
    string i{'a','b','c'};
    char j[3]={'a','b','c'}; //inicjalizacja zagregowana
    string k[2]={string("abc"),string("def")};
};
```

Konstruktor kopiujący

Ważnym typem konstruktorów jest konstruktor kopiujący. Pozwala on na szybkie stworzenie kopii obiektu. Jego parametrem jest często stała referencja do obiektu kopiowanego. Warto zwrócić uwagę, że w konstruktor kopiujący może się bezpośrednio odnosić do pól prywatnych – tworzonego i kopiowanego obiektu .

```
class Klasa
{
private:
	int mPole1, mPole2;
public:
	Klasa();

	Klasa(const Klasa& kopiowanyObiekt) :
			mPole1(kopiowanyObiekt.mPole1)
	{
		mPole2 = kopiowanyObiekt.mPole2;
		//tworzenie kopii oryginału bez zmieniania jego stanu
	}

};
```

Użycie konstruktorów

Odpowiednie konstruktory zostaną wywołane w następujących przypadkach:

```
Klasa o; //konstruktor bez parametrów
Klasa o2(1,2); //z parametrami
Klasa o3(o); //konstruktor kopiujący o3 jest kopią o
```

Zadanie 1

Stwórz klasę `Tablica2D`, która implementuje dynamiczną tablicę dwuwymiarową. Klasa ma posiadać:

- prywatne pola przechowujące: szerokość, wysokość i nazwę tablicy
- wskaźnik do elementu typu `int` (w nim zostanie zapisany wskaźnik do początku jednowymiarowej tablicy dynamicznej)
- konstruktor o trzech parametrach: szerokość, wysokość i nazwa tablicy. Konstruktor ma ustawiać wartości pól oraz zajmować pamięć na jednowymiarową tablicę dynamiczną za pomocą operatora `new`
- konstruktor kopiujący
- metodę pozwalającą na zmianę nazwy tablicy
- metodę pozwalającą na ustawienie elementu (nadanie mu wartości). Jej parametrami mają być: wiersz, kolumna oraz wartość
- stałą metodę pozwalającą na odczytanie elementu z tablicy (parametry: wiersz, kolumna)
- stałą metodę `Tablica2D dodaj(const Tablica2D& dodawana) const,` która do tablicy (obiektu) na rzecz którego została wywołana doda tablicę przekazaną jako parametr, a wynik zwróci w nowym obiekcie. Nazwą tablicy wynikowej jest automatycznie ustawiana na „wynik dodawania”
- analogiczną do metody `dodaj()` stałą metodę `mnoz()`
- stałą metodę `ostream& wyswietl(ostream& strumien) const,` która do strumienia przekazanego jako parametr wysyła zawartość tablicy (nazwę i wartości). Strumień do którego wysłano dane jest również zwracany jako wynik metody – dzięki temu można użyć zapisu: `tab1.wyswietl(cout)<<endl;` (dopisanie `endl` do strumienia)
- destruktor zwalniający pamięć zajętą dynamicznie (operator `delete[]`)

We wszystkich konstruktorach i destruktorze umieść wyświetlanie komunikatów, które pozwolą stwierdzić, który z nich i kiedy został wywołany.