Wrocław 2024.11.13

Autorzy: Michał Przewoźniczek, Marcin Jodłowiec, Marcin Komarnicki i Artur Zawisza

Techniki Efektywnego Programowania – zadanie 4
Klasy szablonowe

Szablony pozwalają na wielokrotne wykorzystanie tego samego kodu. Jest to wygodne, gdy tworzymy funkcje i klasy, które mają służyć np. do przechowywania jakichś wartości. Typową i często używanym szablonem klasy jest `vector`. Tej klasy można użyć do przechowywania dowolnych typów, jednak przechowywany typ należy określić tworząc `vector`. Istotne jest, że logika operacji wykonywanych przez `vector` nie jest zależna od przechowywanego typu. Może więc zostać napiasana bez znajomości przechowywanego typu.

W C++ poza szablonami klas można deklarować również szablony funkcji, w tym ćwiczeniu skupimy się jednak wyłącznie na klasach. Przykład szablonu, który tworzy tablicę typu `T`, znajduje się poniżej.

Deklaracja szablonu, typ `T` jest parametrem typowym. Jego zdefiniowanie jest wymagane w momencie, w którym nasz program chce użyć klasy `CTable`.

```
template< typename T > class CTable
{
public:
	CTable() { i_size = 0; pt_table = NULL; }
	~CTable() { if (pt_table != NULL)  delete  [] pt_table; }

	bool  bSetLength(int iNewSize);

	T*  ptGetElement(int iOffset);
	bool  bSetElement(int iOffset, T tVal);

private:
	int i_size;
	T *pt_table;
};//template< typename T > class CTable
```

W przypadku typów szablonowych w C++, przez wgląd na błędy linkera, implementację metod zamieszcza się zazwyczaj w plikach nagłówkowych pod deklaracją klasy.Wynika to z faktu, że kompilator musi wiedzieć jak wygenerować daną funkcję dla konkretnego typu `T`. W przypadku klasy `CTable`, definicje metod będą następujące.

```
template <typename T>
bool  CTable<T>::bSetLength(int iNewSize)
{
	if (iNewSize <= 0)  return(false);

	T *pt_new_table;
	pt_new_table = new T[iNewSize];

	if (pt_table != NULL)
	{
		int  i_min_len;
		if (iNewSize < i_size)
			i_min_len = iNewSize;
		else
			i_min_len = i_size;

		for (int ii = 0; ii < i_min_len; ii++)
			pt_new_table[ii] = pt_table[ii];

		delete[] pt_table;
	}//if (pt_table != NULL)

	pt_table = pt_new_table;
	return(true);
}//bool  CTable<T>::bSetLength(int iNewSize)


template <typename T>
T*  CTable<T>::ptGetElement(int iOffset)
{
	if ((0 <= iOffset) && (iOffset < i_size))  return(NULL);

	return(&(pt_table[iOffset]));
}//T*  CTable<T>::ptGetElement(int iOffset)


template <typename T>
bool  CTable<T>::bSetElement(int iOffset, T tVal)
{
	if ((0 <= iOffset) && (iOffset < i_size))  return(false);

	pt_table[iOffset] = tVal;

	return(true);
}//bool  CTable<T>::bSetElement(int iOffset, T tVal)
```

Proszę zwrócić uwagę, że działanie klasy `CTable` jest takie samo, bez względu na to, czy typem `T` będzie `int`, `double`, klasa, lub wskaźnik (pojedynczy lub wielokrotny).

Użycie klasy CTable może być następujące.

```
int i_template_test()
{
	CTable<int>  c_tab_int;
	CTable<double *>  c_tab_double_point;

	c_tab_int.bSetLength(10);
	c_tab_int.bSetElement(1, 22);
	int  i_val = *(c_tab_int.ptGetElement(1));

	double  d_my_doub = 5;
	c_tab_double_point.bSetLength(2);
	c_tab_double_point.bSetElement(1, &d_my_doub);
	**(c_tab_double_point.ptGetElement(1)) = 5;
}//int i_template_test()
```

Tworzenie metod dedykowanych dla danego typu szablonowego

Czasami chcielibyśmy, żeby niektóre typy szablonowe działały w inny sposób, w zależności od tego jaki jest typ parametru. Specjalizację metod uzyskuje się w następujacy sposób. Do klasy `CTable` dodajemy metodę `sGetKnownType`.

```
template< typename T > class CTable
{
public:
	CTable() { i_size = 0; pt_table = NULL; }
	~CTable() { if (pt_table != NULL)  delete  [] pt_table; }

	bool  bSetLength(int iNewSize);

	T*  ptGetElement(int iOffset);
	bool  bSetElement(int iOffset, T tVal);

	CString  sGetKnownType();
private:
	int i_size;
	T *pt_table;
};//template< typename T > class CTable
```

Implementacja metody `sGetKnownType` wygląda następująco.

```
template <typename T>
CString  CTable<T>::sGetKnownType()
{
	CString  s_type = "Unknown";
	return(s_type);
}//CString  CTable<T>::sGetKnownType()
template <>
CString  CTable<int>::sGetKnownType()
{
	CString  s_type = "INT";
	return(s_type);
}//CString  CTable<int>::sGetKnownType()
```

Przykład użycia:

```
int i_template_test()
{
	CTable<int>  c_tab_int;
	CTable<double *>  c_tab_double_point;

	c_tab_int.bSetLength(10);
	c_tab_int.bSetElement(1, 22);
	int  i_val = *(c_tab_int.ptGetElement(1));

	double  d_my_doub = 5;
	c_tab_double_point.bSetLength(2);
	c_tab_double_point.bSetElement(1, &d_my_doub);
	**(c_tab_double_point.ptGetElement(1)) = 5;

	CString  s_type;
	s_type = c_tab_int.sGetKnownType();
	s_type = c_tab_double_point.sGetKnownType();
}//int i_template_test()
```

Jeżeli metoda `sGetKnownType` zostanie użyta tak, jak powyżej, to dla dowolnego typu szblonowego innego niż `int`, `sGetKnownType` zwróci `"Unknown"`, a dla typu `int` zwróci `"INT"`. Specjalizacji może być dowolnie dużo.

Zachowanie kompilatora w przypadku napotkania klas szablonowych.
Szablon klasy należy traktować jak *przepis* na zrobienie klasy. Na przykład, w momencie kompilacji powyższego programu, kompilator stwierdzi, że CTable będzie używany z dwoma typami - int i double*. Na podstawie definicji szablonu kompilator stworzy dwie, niezależne od siebie implementacje – klasy szablonowe. W pierwszej z nich zamiast T użyty będzie typ int, a w drugiej zostanie użyty typ double*.

Zadanie
UWAGI:

1. **Pisząc własny program można użyć innego nazewnictwa niż to przedstawione w treści zadania i w przykładach. Należy jednak użyć jakiejś spójnej konwencji kodowania, zgodnie z wymaganiami kursu.**
2. **Nie wolno używać wyjątków (jest to jedynie przypomnienie, wynika to wprost z zasad kursu).**
3. **Wolno używać wyłącznie komend ze standardu C++98**

Oprogramuj szablon klasy CResult, który przechowywać będzie wynik dowolnej metody lub błędy napotkane podczas jej wykonania. Wykonaj następujące ćwiczenia.

1. Oprogramuj klasę CError, która przechowywać będzie co najmniej opis błędu.

2. Oprogramuj szablon CResult zgodnie z poniższą specyfikacją. Jest to minimalny zakres wymagań, który szablon CResult musi spełniać. Wszelkie metody pomocnicze są dozwolone.

```
template <typename T, typename E>
class CResult
{
public:
	CResult(const T& cValue);
	CResult(E* pcError);
	CResult(vector<E*>& vErrors);
	CResult(const CResult<T, E>& cOther);

	~CResult();

	static CResult<T, E> cOk(const T& cValue);
	static CResult<T, E> cFail(E* pcError);
	static CResult<T, E> cFail(vector<E*>& vErrors);

	CResult<T, E>& operator=(const CResult<T, E>& cOther);

	bool bIsSuccess();

	T cGetValue();
	vector<E*>& vGetErrors();

private:
	T *pc_value;
	vector<E*> v_errors;
};
```

W powyższym fragmencie kodu, `T` określa zwracany typ, natomiast `E` jest typem błędu. Konstruktory `CResult(const T& cValue)` i `CResult(E* pcError)` pozwalają na implementację następującego fragmentu kodu:

```
CResult<double, CError> eDivide(double dDividend, double dDivisor)
{
	if (dDivisor == 0)
	{
```

```
		return new CError("cannot divide by zero");
	}

	return dDividend / dDivisor;
}
```

w którym w wygony sposób zwracamy jako wynik metody bezpośrednio `new CError("cannot divide by zero")` lub `dDividend / dDivisor`. Nie musimy opakowywać ich w obiekt klasy `CResult<double, CError>`.

W przypadku gdy jako `E` podamy typ, którego nazwa nie sugeruje, że zwracamy błąd

```
const int iERROR_CODE_DIVISION_BY_ZERO = 123;

CResult<double, int> eDivide(double dDividend, double dDivisor)
{
	if (dDivisor == 0)
	{
		return new int(iERROR_CODE_DIVISION_BY_ZERO);
	}

	return dDividend / dDivisor;
}
```

to prowadzić to może do sytuacji, w której trudno rozpoznać czy zwracamy poprawny wynik czy też błąd. W celu uniknięcia takich niejednoznaczności należy zaimplementować m.in. statyczne metody `static CResult<T, E> cOk(const T& cValue)` i `static CResult<T, E> cFail(E* pcError)`. Pozwalają one na implementację następującego fragmentu kodu.

```
CResult<double, CError> eDivide(double dDividend, double dDivisor)
{
	if (dDivisor == 0)
	{
		return CResult<double, CError>::cFail(new CError("cannot divide by
zero"));
	}

	return CResult<double, CError>::cOk(dDividend / dDivisor);
}
```

Zakładamy, że klasa `T` posiada poprawnie zaimplementowany konstruktor kopiujący. Pojedynczy obiekt szablonu `CResult<T, E>` może przechowywać wiele wskaźników do obiektów klasy `E`. Wykorzystanie wskaźników umożliwia przechowywanie obiektów różnych klas błędów, pod warunkiem, że ich klasą bazową jest klasa `E`.

3. W przypadku gdy będziemy chcieli użyć `CResult<T, E>` jako typu zwracanego przez metodę, która według założeń ma nic nie zwracać to intuicyjnym wyborem wydaje się być typ `CResult<void, E>`. W obecnej implementacji utworzenie obiektu szablonu `CResult<void, E>` jest niemożliwe m.in. ze względu na konstruktor `CResult(const T& cValue)`. Oprogramuj specjalizację całego szablonu `CResult<T, E>` zgodnie z poniższą

specyfikacją. Jest to minimalny zakres wymagań, który szablon `CResult<void, E>` musi spełniać. Wszelkie metody pomocnicze są dozwolone.

```
template <typename E>
class CResult<void, E>
{
public:
	CResult();
	CResult(E *pcError);
	CResult(vector<E*>& vErrors);
	CResult(const CResult<void, E>& cOther);

	~CResult();

	static CResult<void, E> cOk();
	static CResult<void, E> cFail(E* pcError);
	static CResult<void, E> cFail(vector<E*>& vErrors);

	CResult<void, E>& operator=(const CResult<void, E>& cOther);

	bool bIsSuccess();

	vector<E*>& vGetErrors();

private:
	vector<E*> v_errors;
};
```

4. Wykorzystaj obiekty szablonu `CResult<T, E>` do próby stworzenia drzewa z poprzedniej listy na podstawie wyrażenia podanego przez użytkownika. Załóż, że niepoprawne wyrażenie nie wymaga zwrócenia jakiegokolwiek drzewa.

5. Oprogramuj klasę, która zapisywać będzie stan obiektu szablonu `CResult<T, CError>` do pliku. Dla dowolnego `T` innego niż `CTree*`, należy zapisać jedynie opisy błędów (o ile jakiekolwiek błędy wystąpiły). Dla typu `CTree*` należy zapisać opisy błędów lub drzewo w postaci prefiksowej.

Zalecana literatura

Jerzy Grębosz „Symfonia C++”, Wydawnictwo Edition, 2000.
Wykład
Materiały możliwe do znalezienia w Internecie