Wrocław 2020.10.13

Autor: Michał Przewoźniczek

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 używaną klasą szablonową 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 klasami szablonowymi można deklarować również funkcje szablonowe, w tym ćwiczeniu skupimy się jednak wyłącznie na klasach. Przykład klasy, która tworzy tablicę typu szablonowego `T`, znajduje się poniżej.

Deklaracja klasy, typ `T` jest typem szablonowym. 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 błedy llinkera implementację metod zamieszcza się zazwyczaj w plikach nagłówkowych pod deklaracją klasy. 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 szablonowy. 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.
Klasę szablonową należy traktować jak *przepis* na zrobienie klasy. Na przykład, w momencie kompilacji powyższego programu, kompilator stwierdzi, że klasa CTable będzie używana z dwoma typami int i double*. Na podstawie definicji klasy szablonowej kompilator stworzy dwie niezależne od siebie implementacje. 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**

Zamień klasy oprogramowane w ramach poprzedniego ćwiczenia na klasy szablonowe. Jeśli chodzi o działanie drzew, to pozwól, żeby drzewo mogło działać co najmniej dla typów `int`, `double` i std:string (można wykorzystać inną klasę do obsługi stringów jeśli się chce). Typ szablonowy wpływa m.in. na sposób przeprowadzania działań (dla typów liczbowych działanie jest oczywiste, ale dla typu string już nie), oraz na sposób wczytywania danych do drzewa. Wymóg wprowadzenia szablonów dotyczy również klas (lub funkcji) implementujących interfejs.

W przypadku drzew z typem szablonowym string należy przyjąć, że:

1. Przy wczytywaniu wartości string są wzięte w cudzysłów. A więc wyrażenie:
   *+ * „ala” „ma” kota*
   Jest rónoważne zapisowi:
   *(„ala” * „ma”) + kota*, gdzie
   kota to nazwa zmiennej, a *„ala”* i *„ma”* to wartości

2. Funkcjonalność operatorów:
   \+ - konkatencja dwóch stringów
   \- usunięcie z odjemnej ostatniego podstringu, który jest odjemnikiem, o ile występuje, czyli:
   „alaxalaxala” – „ala” = „alaxalax”
   „alaxalaxala” – „kot” = „alaxalaxala”

   * - mnożenie należy zaimplementować w następujący sposób. W lewostronnym argumencie mnożenia znajdujemy pierwszy znak drugiego argumentu i w każde takie miejsce wstawiamy pozostałe znaki drugiego argumentu. Na przykład:

   „alaxalaxala” * „xABC” = „alaxABCalaxABCala”
   „alaxalaxala” * „RDG” = „alaxalaxala”
   „alaxalaxala” * „” = „alaxalaxala”

   / - dzielenie stringów należy wykonać w ten sposób, że w dzielnej odnajdujemy wszystkie wystąpienia dzielnika i zostawiamy tylko pierwszy znak dzielnika. Na przykład:

   „alaxABCalaxABCala” / „xABC” = „alaxalaxala”
   „alaxABCalaxABCala” / „ZZZ” = „alaxABCalaxABCala”
   „alaxABCalaxABCala” / „” = „alaxABCalaxABCala”

Zalecana literatura

Jerzy Grębosz „Symfonia C++”, Wydawnictwo Edition, 2000.

Wykład

Materiały możliwe do znalezienia w Internecie