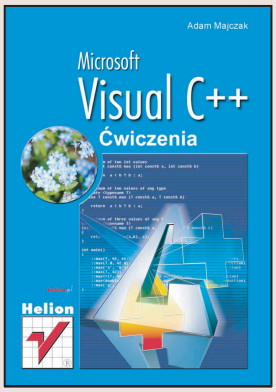
MS Visual C++. Ćwiczenia



Autor: Adam Majczak
ISBN: 83-7361-206-8
Format: B5, stron: 122

Visual C++ to popularne środowisko RAD (szybkiego tworzenia aplikacji). Dzięki intuicyjnej obsłudze jest ono nie tylko wygodnym narzędziem dla profesjonalistów tworzących zaawansowane aplikacje dla Windows, ale także bezpiecznym „polem doświadczalnym" dla wszystkich tych, którzy chcą spróbować swoich sił w programowaniu w C++.

Niniejsza książka stanowi praktyczne uzupełnienie podręcznika lub kursu programowania w C++. Może być dla Ciebie zbiorem zadań wykonywanych jako indywidualne prace domowe, może być także potraktowana jako zbiór „podpowiedzi" pokazujący, jak zawodowcy rozwiązują niektóre typowe problemy występujące przy programowaniu w C++ i Visual C++. Więcej w niej praktycznych zadań dla programisty niż teoretycznych rozważań o niuansach sztuki programowania w C++.

Książka opisuje:

- Aplikacje konsolowe ANSI C++
- Programowanie sekwencyjne w Visual C++
- Konstruowanie aplikacji zdarzeniowych i obiektowych
- Wyprowadzanie danych w trybie graficznym
- Korzystanie z szablonów
- Obsługę wyjątków
- Tworzenie aplikacji w środowisku graficznym
- Stosowanie biblioteki klas MFC w Visual C++
- Wprowadzenie do MS VisualStudio.NET

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

Rozdział 5.

Szablony i obsługa wyjątków

Podobnie jak w przypadku bibliotek klas (MFC, OWL, itp.) i bibliotek komponentów wizualnych (np. VCL), producenci kompilatorów C++ często dodają do IDE biblioteki szablonów (templates). W Visual C++ jest to biblioteka ATL (*Active Template Library*), popularną i szeroko dostępną wersją jest STL (*Standard Templates Library*). Te dość złożone zagadnienia wymagają przed przystąpieniem do ćwiczeń krótkiego wyjaśnienia i wprowadzenia.

Wprowadzenie

Wykorzystanie szablonów i obsługa sytuacji wyjątkowych EH (*Exception Handling*) to kolejne dwa elementy, które silnie wpływają na styl programowania i praktyczny sposób konstruowania aplikacji Visual C++. Zrozumienie tych zagadnień jest niezbędne do świadomego i umiejętnego korzystania z możliwości Visual C++. Mechanizmy te najłatwiej wyjaśnić i zademonstrować w najprostszych aplikacjach, dlatego to ostatni rozdział, w którym będziemy się jeszcze posługiwać znakowymi aplikacjami konsoli. W następnych rozdziałach, posiadając już w spory zasób wiedzy, umiejętności i praktycznych doświadczeń w pracy z Visual C++, wykorzystamy to wszystko, by nasze aplikacje umiejętnie wykorzystywały możliwości środowiska Windows.

Obsługa sytuacji wyjątkowych w C++

W C++ wyjątek lub sytuacja wyjątkowa to w istocie obiekt, który jest przekazywany (przy okazji wraz z pewnymi informacjami) z tego obszaru w kodzie, gdzie wystąpił problem do tego obszaru w kodzie, który zajmuje się „pokojowym rozwiązaniem" tego

problemu. Określenie „pokojowe rozwiązanie" oznacza tu przemyślaną obsługę sytuacji konfliktowej typu: brak pamięci operacyjnej, nie istnieje potrzebny plik dyskowy, próba dzielenia przez zero, itp., która w normalnych warunkach mogłaby spowodować przerwanie działania programu i, co groźniejsze, często utratę danych trudnych do odzyskania i odtworzenia.

Rodzaj wyrażenia (obiektu — wyjątku) może decydować o tym, która część kodu podejmie próbę rozwiązania konfliktowej sytuacji. Zawartość tak „przerzucanego obiektu-wyjątku" (ang. *throw-n object*) może decydować o sposobie powrotu do dalszej pracy aplikacji i o sposobie informowania użytkownika o wystąpieniu i obsłudze sytuacji wyjątkowej. Podstawowe zasady logiki obsługi wyjątków sprowadzają się do następujących działań:

1. Zidentyfikowanie tych fragmentów kodu, które potencjalnie mogą spowodować wystąpienie sytuacji wyjątkowej i ujęcie ich w bloki typu *try* (spróbuj).

```
try
{
  // Kod zawierajacy ryzykowne operacje
...
}
```

2. Utworzenie bloków obsługi (dosł.: *catch* — przechwyć) przewidywanych, możliwych sytuacji wyjątkowych.

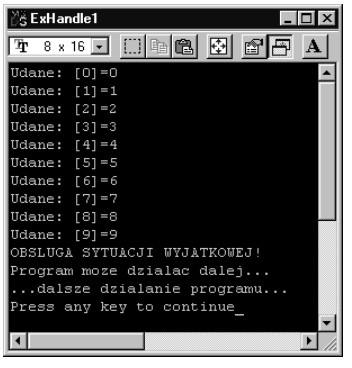
```
catch( OutOfMemory )
{
   // cos robimy, by uratować aplikację...
...
}
catch( FileNotFound )
{
   // robimy cos innego...
...
}
// ... itd. ...
```

Z technicznego punktu widzenia bloki `catch` powinny w kodzie występować bezpośrednio po bloku `try`. Sterowanie zostanie przekazane do bloku `catch` tylko wtedy, gdy przewidywana sytuacja wyjątkowa rzeczywiście wystąpi.

Przykładowy kod pokazany poniżej ilustruje praktyczne zastosowanie najprostszego, „pustego" obiektu-wyjątku przeznaczonego do obsługi typowej sytuacji konfliktowej — próby zapisania do macierzy elementu, dla którego nie ma przewidzianego miejsca (spoza dopuszczalnego zakresu indeksu-offsetu), co może w praktyce zagrażać przerwaniem pracy aplikacji i utratą danych. Listing 5.1 i rysunek 5.1 przedstawiają prosty przykład obsługi wyjątku.

Listing 5.1. *Obsługa wyjątku — ExHandle1.CPP*

```
#include <iostream.h>
const int DefaultSize = 5;
class ZaDuzo {};   // definicja klasy dla obiektu - wyjatku
class Array
{
private:
int *Pointer;   // Dane prywatne:
```



Rysunek 5.1. *Obsługa sytuacji wyjątkowej w działaniu. Aplikacja konsoli Visual C++ o nazwie projektu ExHandle1*

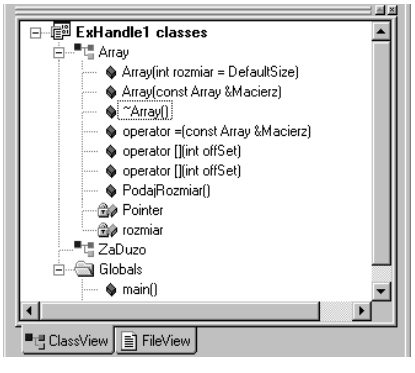
```
int  rozmiar;   // wskaznik i rozmiar macierzy
public:
Array(int rozmiar = DefaultSize);      // Dwa konstruktory
Array(const Array &Macierz);
~Array() { delete [] Pointer;}
Array& operator=(const Array&);               // Przeciazone operatory
int& operator[](int offSet);
const int& operator[](int offSet) const;
int PodajRozmiar() const { return rozmiar; }  // Zwykla metoda
friend ostream& operator<< (ostream&, const Array&);
};
Array::Array(int size) : rozmiar(size)
{
Pointer = new int[size];
for (int i = 0; i<size; i++) Pointer[i] = 0;
}
Array& Array::operator=(const Array &Macierz)
{
if (this == &Macierz)
return *this;
delete [] Pointer;
rozmiar = Macierz.PodajRozmiar();
Pointer = new int[rozmiar];
for (int i=0; i<rozmiar; i++)
Pointer[i] = Macierz[i];
return *this;
}
Array::Array(const Array &Macierz)
{
rozmiar = Macierz.PodajRozmiar();
Pointer = new int[rozmiar];
for (int i = 0; i<rozmiar; i++)
Pointer[i] = Macierz[i];
}
int& Array::operator[](int offSet)
{
int size = PodajRozmiar();
if (offSet >= 0 && offSet < PodajRozmiar())
  return Pointer[offSet];
else
  {
         throw ZaDuzo();   // "rzucany" obiekt - wyjatek
  return Pointer[offSet];
  }
```

```
}
const int& Array::operator[](int offSet) const
{
int mysize = PodajRozmiar();
if (offSet >= 0 && offSet < PodajRozmiar())
return Pointer[offSet];
throw ZaDuzo();
return Pointer[offSet];
}
ostream& operator<< (ostream& output, const Array& theArray)
{
for (int i = 0; i<theArray.PodajRozmiar(); i++)
output << "[" << i << "] " << theArray[i] << endl;
return output;
}

int main()
{
Array intArray(10);
try
{
for (int j = 0; j< 100; j++)
  {
  intArray[j] = j;
  cout << "Udane: [" << j << "]=" << j << endl;
  }
}
catch (ZaDuzo)          // "Przechwycenie" obiektu - wyjatku
  {
  cout << "OBSLUGA SYTUACJI WYJATKOWEJ!\n";
  }
cout << "Program moze dzialac dalej..." << endl;
cout << "...dalsze dzialanie programu..." << endl;
return 0;
}
```

To dobra okazja, by sprawdzić, jak na zakładce *ClassView* w panelu struktury projektu Visual C++ prezentowana jest hierarchia klas w przykładowej aplikacji (rysunek 5.2).

Rysunek 5.2.
Hierarchia klas na zakładce ClassView w panelu struktury Visual C++



Powyższy przykładowy prosty kod przy okazji demonstruje przeciążanie operatorów przy użyciu metod oraz funkcji zewnętrznej kategorii *friend*.

Konstruowanie i stosowanie szablonów

Szablony wymagają zastosowania słowa kluczowego C++ `template`. Szablony można stosować w odniesieniu do funkcji. Oto prosty przykład parametryzacji algorytmu sortowania bąbelkowego przy użyciu szablonu

```
template <class V>
void VectorBubleSort(V wektor[], int rozmiar)
```

dla jednowymiarowej macierzy (*wektora*) stanowiącej argument funkcji sortującej.

Listing 5.2. *Konstrukcja i zastosowanie prostego szablonu: TemplateDemo1.CPP*

```
// TeplateDemo – szablon uzyty na liscie argumentów funkcji
#include <iostream.h>

template <class V>
void VectorBubleSort( V wektor[], int rozmiar )
{
  for(int i=0; i<(rozmiar-1); i++)
         for(int j=(rozmiar-1); i<j; j--)
                if( wektor[j-1] > wektor[j] )
                {
                        V tymczasowa = wektor[j];
                        wektor[j] = wektor[j-1];
                        wektor[j-1] = tymczasowa;
                }
}

main()
{
int Numerki[7] = {1, 3, 5, 4, 2, 0, 6};
char Literki[5] = {'d', 'A', 't', 'Z', 'a'};

VectorBubleSort(Numerki, sizeof(Numerki)/sizeof(Numerki[0]));
VectorBubleSort(Literki, sizeof(Literki)/sizeof(Literki[0]));
for(int i=0; i<sizeof(Numerki)/sizeof(Numerki[0]); i++)
cout << "Num[" << i << "]=" << Numerki[i] << endl;
for(i=0; i<sizeof(Literki)/sizeof(Literki[0]); i++)
cout << "Lit[" << i << "]=" << Literki[i] << endl;
return 0;
}
```

Szablony zastosowane w odniesieniu do klas powodują utworzenie rodziny klas. Kompilator może następnie wygenerować samodzielnie nową klasę na podstawie zadanego szablonu. A oto prosty przykład kolejkowania z zastosowaniem szablonu wobec własnej klasy `BUFOR`.

Listing 5.3. *Drugi szablon: TemplateDemo2.CPP*

```
#include <iostream.h>
const DefaultSize = 5;
template <class K>
class BUFOR
{
```

```
public:
BUFOR(int Rozmiar = DefaultSize)
  { Licznik = 0; Pointer = new K[Rozmiar]; }
~BUFOR()
  { delete [] Pointer; }
void DoBufora( K zmienna);
K ZBufora();

protected:
  int Licznik;
  K *Pointer;
};

template <class K>
void BUFOR<K>::DoBufora(K szablon)
{
  Pointer[Licznik++] = szablon;
}

template <class K>
K BUFOR<K>::ZBufora(void)
{
  return Pointer[--Licznik];
}

main()
{
  BUFOR<int> B(5);
  B.DoBufora(1);
  B.DoBufora(2);
  B.DoBufora(3);
  cout << B.ZBufora() << ' ' << B.ZBufora() << ' '
     << B.ZBufora() << endl;
  BUFOR<double> D(4);
  D.DoBufora(1.11);
  D.DoBufora(2.12);
  D.DoBufora(3.13);
  cout << D.ZBufora() << ' ' << D.ZBufora() << ' '
     << D.ZBufora() << endl;
return 0;
}
```

Przy okazji przykład ilustruje zastosowanie operatorów new i delete. Bufory tworzone z użyciem szablonu *K* mogą mieć różną wielkość i zawierać elementy różnych typów. Ta „typologiczna" elastyczność powoduje, że szablony są często stosowane w praktyce do tworzenia klas i obiektów -pojemników (ang. *container classes*).

Ćwiczenie 5.1.

Stan wyjściowy: poprzedni projekt zamknięty.

Aby przetestować działanie obsługi wyjątków, należy wykonać kolejno wyszczególnione poniżej czynności.

1. Otwieramy nowy projekt aplikacji konsoli *ExHandle1*.
2. Wprowadzamy kod przykładu pierwszego do pliku *ExHandle.CPP* z listingu 5.1.

3. Uruchamiamy i sprawdzamy poprawność działania aplikacji.
4. Zamykamy projekt poleceniem *File/Close Workspace*.

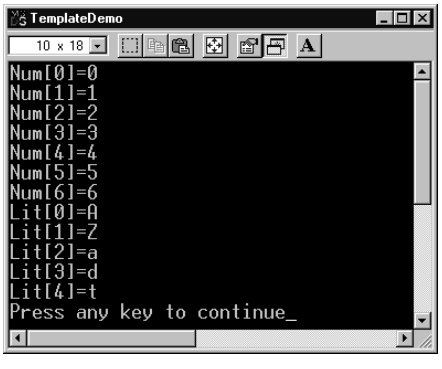
Ćwiczenie 5.2.

Stan wyjściowy: poprzedni projekt zamknięty (stan wyjściowy IDE).

Aby przetestować działanie obsługi szablonów, należy wykonać kolejno wyszczególnione poniżej czynności.

1. Otwieramy nowy projekt aplikacji konsoli *TemplateDemo1*.
2. Wprowadzamy kod przykładowy do pliku *TemplateDemo1.CPP* z listingu 5.2.
3. Uruchamiamy i sprawdzamy poprawność działania aplikacji (rysunek 5.3).

Rysunek 5.3.
Sortowanie z zastosowaniem szablonu <class V>



4. Zamykamy projekt poleceniem *File/Close Workspace*.

Ćwiczenie 5.3.

Stan wyjściowy: poprzedni projekt zamknięty (stan wyjściowy IDE).

Aby przetestować działanie kolejnego (drugiego) wariantu obsługi szablonów, należy wykonać kolejno wyszczególnione poniżej czynności.

1. Otwieramy nowy projekt aplikacji konsoli *TemplateDemo2*.
2. Wprowadzamy kod przykładowy do pliku *TemplateDemo2.CPP* z listingu 5.3.
3. Uruchamiamy i sprawdzamy poprawność działania aplikacji (rysunek 5.4).

Rysunek 5.4.
Bufor z zastosowaniem szablonów w działaniu

4. Zamykamy bieżący projekt poleceniem *File/Close Workspace*.

Podsumowanie

W ćwiczeniach z tego rozdziału prześledziliśmy ogólne zasady konstrukcji kodów, działanie prostych aplikacji konsoli wykorzystujących obsługę sytuacji wyjątkowych za pomocą obiektów i proste szablony użyte w odniesieniu do funkcji i klas.