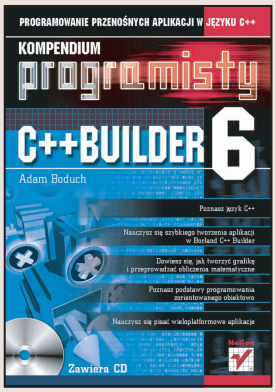
C++Builder. Kompendium programisty



Autor: Andrzej Daniluk
ISBN: 83-7361-028-6
Format: B5, stron: 496

Język C++ od czasu jego zdefiniowania przez Bjarne Stroustrupa był kilkakrotnie uaktualniany w latach 80. i 90. XX wieku. Chociaż C++Builder nie jest dokładnym odzwierciedleniem standardu ANSI języka C++, to obecna jego wersja przygotowana przez firmę Borland jest stabilna i zgodna z oczekiwaniami programistów. Borland C++Builder stanowi połączenie nowoczesnego języka programowania, jakim jest C++, biblioteki komponentów wizualnych VCL/CLX oraz zintegrowanego środowiska programisty IDE.

„C++Builder. Kompendium programisty" omawia podstawy programowania w C++ ze szczególnym uwzględnieniem możliwości oferowanych przez kompilator Borland C++Builder. Poznasz więc nie tylko sam język, ale nauczysz się pisać w nim wydajne i przenośne aplikacje, działające zarówno w środowisku linuksowym, jak i w Windows.

Omówione zagadnienia obejmują:

- Opis zintegrowanego środowiska programisty C++Buildera
- Podstawy języka C++
- Wczesne oraz późne wiązanie — wstęp do programowania zorientowanego obiektowo
- Możliwości C++Buildera w zakresie posługiwania się tablicami różnego typu
- Zaawansowane operatory rzutowania typów
- Informacje czasu wykonania
- Obsługę wyjątków
- Obsługę plików
- Łańcuchy ANSI
- Zmienne o typie modyfikowalnym w czasie wykonywania programu
- Funkcje FPU i systemowe
- Elementy wielowątkowości — wykorzystanie C++ oraz C++Buildera w nowoczesnych, wielowątkowych systemach operacyjnych
- Liczby pseudolosowe i konwersje wielkości liczbowych
- Wprowadzenie do grafiki
- Komponentowy model C++Buildera i biblioteki DLL
- Biblioteka CLX – projektowanie aplikacji przenośnych pomiędzy systemami operacyjnymi Windows i Linux

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

Spis treści

5.

Zaawansowane operatory rzutowania typów

Oprócz tradycyjnych operatorów rzutowania typów zaczerpniętych z języka C, w C++ zdefiniowano dodatkowo cztery operatory rzutowania typów o nazwach: `static_cast`, `dynamic_cast`, `const_cast` oraz `reinterpret_cast`. Ogólne postaci omawianych operatorów wyglądają następująco:

```
static_cast< Typ > (arg)
dynamic_cast< Typ > (arg)
const_cast< Typ > (arg)
reinterpret_cast< Typ > (arg)
```

Słowo `Typ` oznacza docelowy typ rzutowania, zaś `arg` (argument) — obiekt, któremu przypisujemy nowy typ danych.

Operator static_cast

Operator `ststic_cast` wykonuje tzw. *rzutowanie niepolimorficzne*, co między innymi oznacza możliwość wykonania każdej standardowej konwersji typów bez konieczności sprawdzania jej poprawności.

Jako prosty przykład wykonania niepolimorficznego rzutowania typów rozpatrzymy prostą funkcję typu `int` z argumentem typu `void*`, której jedynym celem będzie zamiana małej litery na dużą. Ponieważ dane typu `void*` nie mogą wskazywać na konkretne obiekty, należy wykorzystać odpowiedni operator rzutowania typów, tak jak pokazano to na listingu 5.1, gdzie przedstawiono kod głównego modułu projektu *Projekt_R5_01.bpr*.

W omawianym przykładzie dzięki operatorowi static_cast uzyskamy wskaźnik let do typu int w funkcji capital() ze wskaźnika letter do zupełnie innego typu bazowego będącego jej argumentem.

Listing 5.1. *Rzutowanie wskaźnika i jego dereferencja*

```
#include <vcl.h>
#pragma hdrstop
#include "Unit_R5_01.h"
#pragma package(smart_init)
#pragma resource "*.dfm"

TForm1 *Form1;

int __fastcall capital(void* letter)
{
  // rzutowanie wskaźnika na prawidłowy typ danych
  int* let = static_cast<int*>(letter);
  // dereferencja wskaźnika
  return toupper(*let);
}
//---------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
}
//---------------------------------------------------------
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  Canvas->Font->Color = clBlue;
  Canvas->Font->Height = 50;
  Canvas->TextOut(ClientHeight / 3, ClientWidth / 3,
                  (char)(capital("a")));
}
//---------------------------------------------------------
```

Powtórne wykorzystanie operatora dostępu pośredniego * wraz z parametrem aktualnym funkcji toupper() oznacza w praktyce dereferencję wskaźnika let, gdyż parametrami funkcji toupper() nie mogą być dane wskazujące.

Operator dynamic_cast

Operator dynamic_cast wykonuje rzutowanie czasu wykonywania, co oznacza, iż poprawność tej operacji zawsze sprawdzania jest w trakcie działania programu. W przypadku, gdy operacja rzutowania typów nie jest możliwa do zrealizowania, całość wyrażenia, w którym występuje omawiany operator, przyjmuje wartość zerową.

Ponieważ operator dynamic_cast wykonuje rzutowanie czasu wykonywania, należy go używać głównie do przekształcania typów polimorficznych. Oznacza to, iż np. w przypadku, gdy pewna klasa polimorficzna Pochodna jest klasą potomną innej klasy polimorficznej Bazowa, to, posługując się operatorem dynamic_cast, zawsze można przekształcić wskaźnik *P do typu Pochodna na wskaźnik *B do typu Bazowa.

Jako przykład rozpatrzmy klasę abstrakcyjną `Vehicle` oraz dziedziczące po niej dwie klasy polimorficzne `Lorry` oraz `Tir`. W klasie bazowej `Vehicle` zadeklarujemy czystą funkcją wirtualną `show()`, której kolejne implementacje zostaną umieszczone w polimorficznych klasach pochodnych, tak jak pokazano to na listingu 5.2.

Listing 5.2. *Kod modułu R5_02.h zawierającego definicje klas Vehicle, Lorry oraz Tir*

```
#ifndef R5_02H
#define R5_02H

#define NUM_VEHICLEES 3
//---------------------------------------------------------
class Vehicle
{
  public:
    __fastcall Vehicle(){

    Form1->Memo1->Lines->Add
           ("Tworzenie egzemplarza klasy Vehicle");
    }
    // czysta funkcja wirtualna show() jest zadeklarowana
    // w klasie bazowej, ale nie posiada w niej swojej
    // implementacji
    virtual void __fastcall show()=0;
};
//---------------------------------------------------------
class Lorry: public Vehicle
{
  public:
   __fastcall Lorry(){

    Form1->Memo1->Lines->Add
           ("Tworzenie egzemplarza klasy Lorry");
  }
  virtual void __fastcall show(){
    Form1->Memo1->Lines->Add
           ("Drukowanie danych egzemplarza klasy Lorry");
  }
};
//---------------------------------------------------------
class Tir: public Vehicle
{
  public:
  __fastcall Tir(){
   Form1->Memo1->Lines->Add
          ("Tworzenie egzemplarza klasy Tir");
  }
   virtual void __fastcall show(){
   Form1->Memo1->Lines->Add
          ("Drukowanie danych egzemplarza klasy Tir");
  }
};
//---------------------------------------------------------
#endif
//---------------------------------------------------------
```

W funkcji obsługi zdarzenia `Button1Click()` w głównym module *Unit_R5_02.cpp* projektu *Projekt_R5_02.bpr* zadeklarujemy dwa egzemplarze klasy `Lorry` oraz jeden klasy `Tir`. Następnie zainicjujemy odpowiednimi wartościami tablicę wskaźników `*V` do klasy `Vehicle` ogólnie reprezentującej pojazdy. Pojazdy zwane ciężarówkami odszukamy, wykonując w pętli `for()` operację polimorficznego rzutowania typu `Vehicle` reprezentowanego przez kolejne elementy tablicy wskaźników `V` na typ `Lorry`, tak jak pokazano to na listingu 5.3.

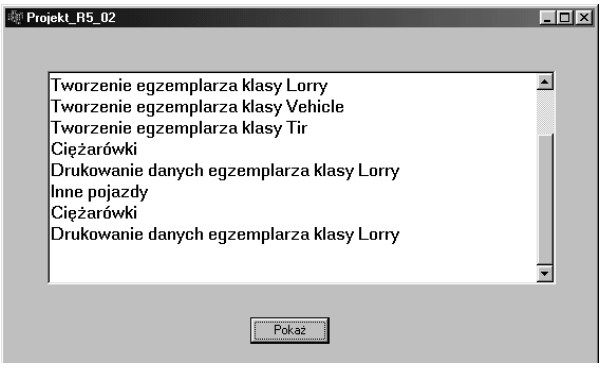
Listing 5.3. *Kod źródłowy głównego modułu Unit_R5_02.cpp projektu Projekt_R5_02.bpr*

```
#include <vcl.h>
#pragma hdrstop
#include "Unit_R5_02.h"
#include "R5_02.h"
#pragma package(smart_init)
#pragma resource "*.dfm"
TForm1 *Form1;
//---------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
}
//---------------------------------------------------------
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  // deklaracja egzemplarzy klas Lorry i Tir
  Lorry L1, L2;
  Tir T;
  // inicjuje tablicę pojazdów
  Vehicle *V[NUM_VEHICLEES];
  V[0] = &L1;
  V[1] = &T;
  V[2] = &L2;

  // odszukuje ciężarówki
  for(int i=0; i < NUM_VEHICLEES; i++)
    {
      Lorry *ll = dynamic_cast<Lorry*>(V[i]);
      if (ll)// lub if (ll != FALSE)
        {
         Memo1->Lines->Add("Ciężarówki");
         ll->show();
        }
        else
          Memo1->Lines->Add("Inne pojazdy");
    }
}
//---------------------------------------------------------
void __fastcall TForm1::FormClose(TObject *Sender,
                                  TCloseAction &Action)
{
  Action = caFree;
}
//---------------------------------------------------------
```

Śledząc powyższe zapisy, musimy zauważyć, iż użycie operatora `dynamic_cast` pozwoliło na odszukanie wszystkich istniejących wskaźników do klasy reprezentującej ciężarówki. W przypadku, gdy wartość wyrażenia:

```
Lorry *ll = dynamic_cast<Lorry*>(V[i]);
```

będzie równa zeru (przyjmie wartość fałszywą), oznaczać to będzie, iż operacja rzutowania nie jest możliwa do przeprowadzenia. Fakt ten nie powinien nas zbytnio dziwić chociażby z tego względu, że pojazdy zwane ciężarówkami nie zawsze są popularnymi „tirami", zaś „tiry" z reguły bywają ciężarówkami, co widoczne jest w działaniu naszej aplikacji pokazanej na rysunku 5.1 oraz co zostało wyraźnie określone w hierarchii klas przedstawionych w listingu 5.2.



Rysunek 5.1. *Rezultat wykonania aplikacji projektu Projekt_R5_02.bpr*

Operator const_cast

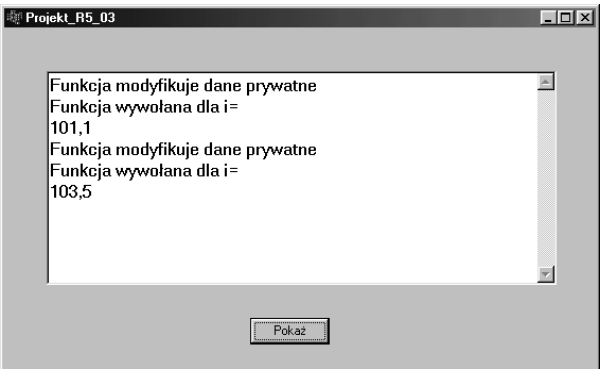
Operator ten wykorzystywany jest głównie w celu przedefiniowywania danych ze specyfikatorami `const` oraz `volatile` oraz w celu zmiany atrybutu `const` wybranej zmiennej. Stosując operator `const_cast`, należy pamiętać, iż typ docelowy modyfikowanej zmiennej musi odpowiadać jej typowi bazowemu. Na listingu 5.4 zaprezentowano kod źródłowy głównego modułu projektu *Projekt_R5_03.bpr*, gdzie zdefiniowano klasę `Bazowa` z funkcją składową `func1()` określoną za pomocą atrybutu `const` oraz przeładowaną względem `func1()` funkcję `func2()`.

Listing 5.4. *Przykład praktycznego wykorzystania operatora const_cast*

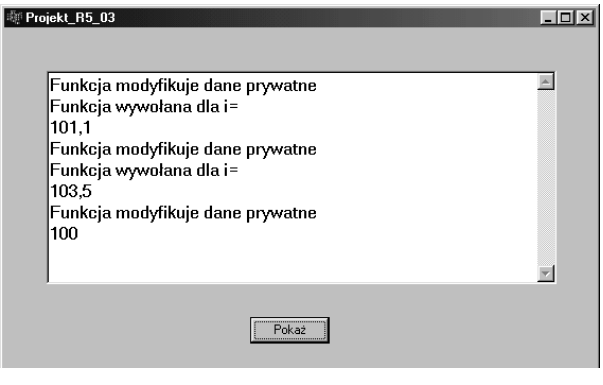
```
#include <vcl.h>
#pragma hdrstop
#include "Unit_R5_03.h"
#pragma package(smart_init)
#pragma resource "*.dfm"

TForm1 *Form1;

class Bazowa {
private:
   int number;
```

```
public:
   __fastcall Bazowa(int j = 0) { number = j; }
   int __fastcall func1(int i) const {
      Form1->Memo1->Lines->
                Add("Funkcja nie modyfikuje obiektu wywołującego");
      return i;
   }
      __fastcall func1(int i)  {
      Form1->Memo1->Lines->Add("Funkcja modyfikuje dane prywatne");
      return number = i;
      }
   double __fastcall func2(double i) {
      Form1->Memo1->Lines->Add(
                  "Funkcja wywołana dla i= ");
      Form1->Memo1->Lines->Add(i);
      return i;
   }
};
//---------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
}
//---------------------------------------------------------
void __fastcall TForm1::Button1Click(TObject *Sender)
{
   Bazowa *B1 = new Bazowa;
   Bazowa *B2 = new Bazowa;

   // sprawdzenie możliwości odwołania się do
   // poszczególnych funkcji składowych obiektów
   // B1 i B2 klasy Bazowa
   B1->func1(100);
   B1->func2(101.1);

   B2->func1(102);
   B2->func2(103.5);

   // jawne przedefiniowanie specyfikatora const
   int ll = const_cast<int>(B1->func1(100));
   Memo1->Lines->Add(ll);
   //ll = const_cast<int>(B2->func1(102));
   //Memo1->Lines->Add(ll);

   delete B1;
   delete B2;
}
//---------------------------------------------------------
void __fastcall TForm1::FormClose(TObject *Sender,
                                  TCloseAction &Action)
{
  Action = caFree;
}
//---------------------------------------------------------
```

Wynik działania programu bez przedefiniowania specyfikatora `const` operatorem `const_cast` w trakcie wywołania funkcji `func1()` oraz z przedefiniowaniem specyfikatora `const` w funkcji `func1()` pokazany jest odpowiednio na rysunkach 5.2 i 5.3.



Rysunek 5.2. *Rezultat wykonania aplikacji projektu Projekt_R5_03.bpr bez przedefiniowania specyfikatora const w czasie wywołania funkcji func1()*



Rysunek 5.3. *Rezultat wykonania aplikacji projektu Projekt_R5_03.bpr z jawnym przedefiniowaniem specyfikatora const podczas wywołania funkcji func1()*

Łatwo sprawdzić, iż poprzez prosty zabieg polegający na przedefiniowaniu operatorem `const_cast` atrybutu `const` funkcji `func1()` możliwe stało się bezpośrednie uzyskanie wartości powrotnych tej funkcji każdorazowo wywołanych w programie głównym z różnymi parametrami aktualnymi.

Operator reinterpret_cast

Nazwa omawianego operatora rzutowania typów w dosłownym znaczeniu tego słowa odpowiada funkcji, jaką może on pełnić w programie. Operator `reinterpret_cast` pozwala na przypisanie wybranej zmiennej zupełnie innego typu w porównaniu z jej typem aktualnym włącznie

z możliwością rzutowania niezgodnych typów wskaźników. Na listingu 5.5 zamieszczono przykład wykorzystania operatora `reinterpret_cast` w celu zamiany wskaźnika `char *` na typ zmiennej całkowitej `int`.

Listing 5.5. *Kod modułu Unit_R5_04.cpp projektu Projekt_R5_04.bpr jako przykład praktycznego wykorzystania operatora reinterpret_cast*

```
#include <vcl.h>
#pragma hdrstop
#include "Unit_R5_04.h"
#pragma package(smart_init)
#pragma resource "*.dfm"

TForm1 *Form1;

//---------------------------------------------------------
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
}
//---------------------------------------------------------
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  int i;
  // podaje długość zaznaczonego tekstu
  // w komponencie Edit1
  int Size = Edit1->SelLength;

  // dodaje znak NULL
  Size++;

  // tworzy dynamiczny bufor danych
  char *Buffer = new char[Size];

  // kopiuje zaznaczony tekst do bufora danych
  Edit1->GetSelTextBuf(Buffer, Size);

  // umieszcza we własności Text zawartość bufora danych
  Edit1->Text = Buffer;

  // zamiana (rzutowanie) wskaźnika na typ całkowity
  i = reinterpret_cast<int>(Buffer);
  Memo1->Lines->Add(i);

  // usuwa bufor
  delete Buffer;
}
//---------------------------------------------------------
void __fastcall TForm1::FormClose(TObject *Sender,
                                  TCloseAction &Action)
{
  Action = caFree;
}
//---------------------------------------------------------
```

Łańcuch znaków, do którego chcemy uzyskać wskaźnik, wpisywany jest w komponencie edycyjnym `Edit1`. Rozmiar zaznaczonych w jego obszarze znaków ustalany jest za pomocą własności:

```
__property int SelLength = {read=GetSelLength, write=SetSelLength,
                            nodefault};
```

Następnie, zaznaczony tekst kopiowany jest za pomocą funkcji:

```
virtual int __fastcall GetSelTextBuf(char * Buffer, int BufSize);
```

do stworzonego uprzednio dynamicznego bufora danych `Buffer`, którego zawartość zostanie umieszczona we własności `Text` komponentu `Edit1`. W komponencie edycyjnym `Memo1` zostanie wyświetlony efekt rzutowania na typ całkowity wskaźnika do łańcucha znaków umieszczonego w buforze.

Podsumowanie

Obecny rozdział poświecony był omówieniu czterech zdefiniowanych w C++ operatorów rzutowania typów. Używanie niektórych z nich w większości prostych aplikacji może wydawać się czynnością mało przydatną, jednak w programach bardziej zaawansowanych, w których należy wykonywać specyficzne operacje rzutowania typów, umiejętność posługiwania się tymi operatorami jest trudna do przecenienia. Kompletne kody źródłowe aplikacji będących ilustracją poruszanych w tym rozdziale zagadnień można znaleźć na dołączonej do książki płycie CD w katalogu *R05*\.