helion.pl

IDŹ DO

PRZYKŁADOWY ROZDZIAŁ

SPIS TREŚCI

KATALOG KSIĄŻEK

KATALOG ONLINE

ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK

DODAJ DO KOSZYKA

CENNIK I INFORMACJE

ZAMÓW INFORMACJE O NOWOŚCIACH

ZAMÓW CENNIK

CZYTELNIA

FRAGMENTY KSIĄŻEK ONLINE

C++Builder 6. Ćwiczenia zaawansowane



Autor: Andrzej Daniluk
ISBN: 83-7361-089-8
Format: B5, stron: 138

Jeśli opanowałeś już podstawy C++ Buildera i zacząłeś wykorzystywać to środowisko we własnych projektach, przyszedł zapewne czas na dokonanie następnego kroku: poznanie zaawansowanych technik programistycznych.

Książka „C++Builder 6. Ćwiczenia zaawansowane" to ponad 30 kompletnych przykładowych projektów. Jak wszystkie książki z tej serii jest ona skierowana do praktyków: nie znajdziesz więc w niej rozważań teoretycznych, za to w krótkim czasie, krok po kroku, opanujesz C++ Buildera na profesjonalnym poziomie. Także użytkownicy innych środowisk programistycznych wykorzystujących język C++ skorzystają na jej lekturze.

Opisano między innymi:

- Wskazania, adresy i odwołania
- Przeładowywanie operatorów
- Funkcje wirtualne, klasy pochodne, polimorficzne i abstrakcyjne
- Wykorzystanie wątków i procesów
- Operacje na plikach
- Modyfikowanie typów zmiennych w czasie wykonywania programu
- Generowanie liczb pseudolosowych
- Wykorzystanie koprocesora matematycznego
- Tworzenie nowych komponentów C++ Buildera i modyfikowanie istniejących

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

Spis treści

Wstęp 5

Rozdział 1. Wskazania i adresy 7

Organizacja pamięci w komputerze 7
Operatory wskaźnikowe 9
Wskaźniki i tablice 9
Wskaźniki ze słowem kluczowym const 13
Wielokrotne operacje pośrednie 14
Wskaźniki do funkcji 15
Wskaźniki i pamięć alokowana dynamicznie 20
Stos 21
Sterta 22
Dereferencja wskaźnika 27
Operatory (.*) oraz (->*) 29
Podsumowanie 30

Rozdział 2. Odwołania 31

Czym są odwołania? 31
Parametry odwołaniowe 33
Zwracanie odwołań przez funkcje 35
Odwołania do struktur 36
Podsumowanie 38

Rozdział 3. Przeładowywanie operatorów 39

Przeładowywanie jednoargumentowych operatorów ++ oraz -- 40
Przeładowywanie operatorów (!) oraz (!=) 43
Przeładowywanie operatora & 46
Przeładowywanie operatora indeksowania tablic [] 47
Podsumowanie 50

Rozdział 4. Tablice jako urządzenia wejścia-wyjścia 51

Podsumowanie 54

Rozdział 5. Funkcje wirtualne. Klasy pochodne, polimorficzne i abstrakcyjne 55

Odwołania i wskaźniki do klas pochodnych 55
Funkcje wirtualne w C++ 58
Funkcje wirtualne w C++Builderze 61
Klasy abstrakcyjne w stylu biblioteki VCL 64
Podsumowanie 66

Rozdział 6. Typy danych Windows 67

Rozdział 7. Wątki 69

Wątki i procesy 69
Funkcja _beginthread() 70
Funkcja _beginthreadNT() 72
Funkcja BeginThread() 77
Zmienne lokalne wątku 80
Klasa TThread 81
Metody 81
Właściwości 82
Podsumowanie 84

Rozdział 8. Operacje na plikach 85

Moduł SysUtils 85
Windows API 91
Klasa TMemoryStream 98
Podsumowanie 101

Rozdział 9. Zmienne o typie modyfikowalnym w czasie wykonywania programu 103

Struktura TVarData 103
Klasa TCustomVariantType 105
Moduł Variants 107
Tablice wariantowe 109
Podsumowanie 113

Rozdział 10. Liczby pseudolosowe 115

Losowanie z powtórzeniami 116
Losowanie bez powtórzeń 119
Podsumowanie 124

Rozdział 11. Funkcje FPU 125

Podsumowanie 128

Rozdział 12. Komponentowy model C++Buildera 129

Tworzymy nowy komponent 129
Modyfikacja istniejącego komponentu z biblioteki VCL/CLX 135
Podsumowanie 138

Rozdział 3.

Przeładowywanie operatorów

Język C++ udostępnia programistom niezwykle wydajne narzędzie w postaci możliwości przeładowywania (określania nowych działań) wybranych operatorów.



Przeładowywanie (przedefiniowywanie) operatorów umożliwia rozszerzenie obszaru zastosowań wybranego operatora na elementy niezdefiniowanej wcześniej unikalnej klasy.

Projektując algorytm nowego działania wybranego operatora, należy skorzystać ze specjalnej funkcji o zastrzeżonej nazwie `operator`:

```
operator <symbol operatora>( <parametry (opcjonalnie)> )
{
   //instrukcje;
}
```

Zapis ten oznacza, iż, na przykład, najprostsza funkcja opisująca nowy algorytm odejmowania (nowy sposób działania unarnego operatora odejmowania -) będzie mogła przybrać następującą postać:

```
operator-(<parametry>){ //instrukcje ;}
```

Reguły C++ umożliwiają przeładowywanie praktycznie wszystkich operatorów, za wyjątkiem czterech, dla których nie jest możliwe zdefiniowanie nowych działań:

- . operatora kropki umożliwiającego uzyskiwanie bezpośredniego dostępu do pół struktur i klas,
- .* operatora wskazującego wybrany element klasy,
- : : operatora rozróżniania zakresu,
- ? : operatora warunkowego.

Przeładowywanie jednoargumentowych operatorów ++ oraz --

Jako przykład praktycznego wykorzystania przeładowanego operatora postinkrementacji ++ posłuży nam sytuacja zliczania elementów ciągu znaków wprowadzonych z klawiatury. W celu przeładowania jednoargumentowego operatora ++ w pierwszej kolejności musimy zaprojektować odpowiednią funkcję operatorową. Każda funkcja operatorowa powinna mieć możliwość wykonywania odpowiednich operacji na właściwych egzemplarzach klasy (lub obiektu), inaczej mówiąc, powinna w stosunku do odpowiedniej klasy posiadać status funkcji zaprzyjaźnionej lub być normalną metodą w klasie. Zaprojektujemy prostą klasę o nazwie `counter` (licznik):

```
class counter
{
  public:
    int number;
    counter() {this->number = 0;}
    counter operator++() {this->number++; return (*this);}
} add;
```

Ponieważ celem naszym będzie zwiększanie w odpowiedni sposób (postinkrementowanie) wartości pola `number` egzemplarza `add` klasy `counter`, funkcja operatorowa przybierze nieskomplikowaną postać:

```
counter operator++() {this->number++; return (*this);}
```

Zauważmy, iż funkcja ta, będąc normalną metodą w klasie, nie posiada jawnych argumentów i w momencie wywołania otrzymuje *niejawny* wskaźnik `this` do własnego egzemplarza klasy. Dzięki posiadaniu niejawnego wskaźnika `this` funkcja ma możliwość postinkrementowania wartości pola `number` własnego egzemplarza klasy.

Dzięki instrukcji:

```
return *this;
```

funkcja operatorowa jawnie zwraca wskaźnik do zmodyfikowanego egzemplarza `add` klasy `counter`.

Ćwiczenie 3.1.

Każda funkcja składowa klasy otrzymuje niejawnie argument w postaci wskaźnika do obiektu, który ją wywołał, i do którego uzyskuje się dostęp, wykorzystując słowo kluczowe (wskaźnik) `this`. Funkcje składowe przeładowywanych operatorów jednoargumentowych nie potrzebują żadnych jawnie zadeklarowanych parametrów formalnych. Jedynym argumentem, którego należy użyć, jest wskaźnik `this`, będący w rzeczywistości wskaźnikiem do egzemplarza klasy, a nie jego kopią. Konsekwencją tego jest fakt, iż wszystkie modyfikacje wykonane za jego pośrednictwem przez funkcję operatora modyfikują zawartość wywoływanego egzemplarza klasy. Przykład wykorzystania funkcji `operator++()` przeładowanego operatora ++ w celu zliczania znaków wprowadzanych z klawiatury zamieszczono na listingu 3.1. Koniec ciągu wprowadzanych znaków oznaczamy klawiszem *Esc*.

Listing 3.1. *Główny moduł Unit_13.cpp projektu Projekt_13.bpr wykorzystującego normalną funkcję składową przeładowanego operatora jednoargumentowego ++. W przedstawionym algorytmie zrezygnowano z używania niejawnych wskaźników this*

```
#include <iostream.h>
#include <conio.h>
#pragma hdrstop

class counter
{
  public:
    int number;
    counter() {number = 0;}
    counter operator++()
    // postinkrementacja pola number
    {number++; return (*this);}
} add;
//---------------------------------------------------------
int main()
{
   char ch;
   cout << "Podaj imię i nazwisko. "
        << "Aby zakończyć naciśnij [Esc] i [Enter]" << endl;
   while(1)
   {
     cin >> ch;
     if(ch == 0x1B) break;
     // zliczanie znaków metodą preinkrementacji
     ++add;
   }
   cout << "Wprowadzono " << add.number << " znaki (znaków)";
   getch();
   return 0;
}
```

Analizując powyższe zapisy, warto zwrócić uwagę na pewien szczegół. Mianowicie jawny wskaźnik `this` wskazuje własny obiekt funkcji. Jeżeli jednak zażądamy, aby funkcja uzyskiwała dostęp nie do pola własnego egzemplarza klasy, ale do pola obiektu przekazywanego jej jako argument, zawsze możemy nadać jej status `friend`, czyli funkcji zaprzyjaźnionej. Funkcje z nadanym statusem `friend` będą bez problemu uzyskiwać dostęp do pól klasy, nie będąc przy tym traktowane jako zwykłe metody w klasie.

Ćwiczenie 3.2.

Proces przeładowywania operatorów jednoargumentowych może przebiegać z wykorzystaniem *funkcji zaprzyjaźnionych* (ang. *friend functions*). Należy jednak zwrócić uwagę, iż stosując taką technikę przeładowywania operatorów powinniśmy w odpowiedni sposób używać parametrów odwołaniowych po to, aby kompilator przekazywał funkcji operatora adres, a nie kopię egzemplarza klasy, który ją wywołał, umożliwiając zmianę jego zawartości. W przeciwieństwie do normalnych funkcji składowych funkcje zaprzyjaźnione nie mogą otrzymywać wskaźnika `this` (niezależnie od tego, czy traktowany będzie jako wskaźnik jawny czy niejawny), co powoduje, iż nie są w stanie określić wywołującego je egzemplarza klasy, tak jak pokazano to na listingu 3.2.

Listing 3.2. *Zmodyfikowany kod głównego modułu projektu Projekt_13.bpr wykorzystującego zaprzyjaźnioną funkcję operator ++() kategorii friend przeładowanego operatora jednoargumentowego (++)*

```
#include <iostream.h>
#include <conio.h>
#pragma hdrstop

class counter
{
  public:
    int x;
   // parametr odwołaniowy number umożliwia przeładowanie
   // operatora ++
    friend counter operator++(counter &number)
   // postinkrementacji podlega pole x
   // egzemplarza add klasy counter
    {number.x++; return (number);}
}add;

//---------------------------------------------------------
int main()
{
   char ch;
   cout << "Podaj imię i nazwisko. "
        << "Aby zakończyć naciśnij [Esc] i [Enter]" << endl;
   while(1)
   {
     cin >> ch;
     if(ch == 0x1B) break;
     ++add;
   }
   cout << "Wprowadzono " << add.x << " znaki (znaków)";
   getch();
   return 0;
}
```



Postępując zgodnie z regułami języka C++, zalecane jest, aby operatory przeładowywać za pomocą zwykłych funkcji składowych. Możliwość korzystania z funkcji zaprzyjaźnionych została wprowadzona głównie w celu rozwiązywania bardziej skomplikowanych i nietypowych problemów związanych z przeładowywaniem operatorów.

Ćwiczenie 3.3.

Wykorzystując samodzielnie zaprojektowaną normalną funkcję składową, przeładuj jednoargumentowy operator postdekrementacji (`--`).

Ćwiczenie 3.4.

Wykorzystując samodzielnie zaprojektowaną funkcję kategorii `friend` przeładuj jednoargumentowy operator postdekrementacji (`--`).

Przeładowywanie operatorów (!) oraz (!=)

W trakcie pisania programów bardzo często stajemy przez problemem zaprojektowania algorytmów wykonujących określone działania matematyczne. C++ udostępnia szereg operatorów oraz funkcji bibliotecznych, którymi możemy się zawsze posłużyć. Jednak wiele praktycznie użytecznych działań nie doczekało się gotowej postaci funkcyjnej lub operatorowej. Jednym z przykładów ilustrującym to zagadnienie jest problem obliczania *silni* (ang. *factorial*) wybranej liczby:

```
n! = 1*2*3*...*n
0! = 1
```

np.:

```
5! = 1*2*3*4*5 = 120
```

Operator negacji logicznej (`!`) bardzo dobrze nadaje się do tego, aby stać się symbolem nowego działania polegającego na obliczaniu silni nieujemnej liczby całkowitej.

Ćwiczenie 3.5.

W celu określenia nowego rodzaju działania dla operatora (`!`) posłużymy się jednoargumentową funkcją operatorową `operator!()`kategorii `friend`. Funkcja ta, typu `void`, nie powinna zwracać żadnej wartości, gdyż jedynym jej celem będzie obliczenie silni wybranej liczby `number` będącej jej argumentem formalnym. Zgodnie z podstawowymi regułami matematyki silnię możemy wyliczyć jedynie dla nieujemnej liczby całkowitej. Aby zapobiec przykrym niespodziankom mogącym wystąpić w trakcie działania programu i związanym z błędnie wprowadzonym z klawiatury argumentem aktualnym `n` funkcji obliczającej silnię — przed wywołaniem funkcji przeładowanego operatora (`!`) zastosujemy blok instrukcji `try...throw...catch` przechwytującej odpowiedni wyjątek.

Listing 3.3. *Kod głównego modułu projektu Projekt_14.bpr wykorzystującego jednoargumentową zaprzyjaźnioną funkcję kategorii friend przeładowanego operatora jednoargumentowego (!)*

```
#include <iostream.h>
#include <conio.h>
#pragma hdrstop

class factorial //silnia
{
  public:
    unsigned int value;
    factorial (unsigned int n) {value = n;}
    friend void operator!(factorial &);
};
//---------------------------------------------------------
void operator!(factorial &number)
{
  unsigned int result = 1;
  for(unsigned int i=2; i<=number.value; i++)
   result *=i;
```

```
  if(result == 0)
    cout << "przekroczony zakres liczb unsigned int";
    else
     cout << result;
  return;
}
//---------------------------------------------------------
int main()
{
   int n;
   cout << "Podaj liczbę: ";
   try
   {
    cin >> n;
    if (n < 0)
      // nieprawidłowy argument dla
      // funkcji operator!()
      throw(n);
   }
   catch(unsigned int n)
   {};
   factorial factorial_n(n);
   cout << n << "! = ";
   !factorial_n;
   getch();
   return 0;
}
```

Ćwiczenie 3.6.

Testując algorytm z poprzedniego ćwiczenia, ktoś dociekliwy na pewno spróbuje wywołać funkcję operatora (`!`) zgodnie z tradycyjnym matematycznym zapisem:

```
factorial_n!;
```

Wywołanie takie będzie dla kompilatora C++ niezrozumiałe z tego powodu, iż potraktuje je jako „niedokończone" i będzie oczekiwał, że pomiędzy znakiem (`!`) oraz znakiem końca instrukcji (`;`) powinien znajdować się jeszcze jakiś symbol. Należy oczekiwać, iż symbolem tym będzie znak przypisania (=). Jeżeli więc zdecydujemy się używać w programie funkcji przeładowanego operatora zgodnie z intuicyjnym rozumieniem symbolu silni powinniśmy w pewien sposób oszukać kompilator:

```
factorial_n != factorial_n;
```

Otóż funkcja `operator!=()` przeładowanego operatora `!=` powinna być dwuargumentowa (gdyż tradycyjnie traktowany operator relacji `!=` jest w rzeczywistości dwuargumentowy). Do obliczenia silni pojedynczej liczby drugi argument nie jest potrzebny, z tego względu bez większych wyrzutów sumienia dwuargumentową funkcję operatorową przeładowanego operatora `!=` możemy zapisać w ten sposób, aby jej drugi argument `seeming` był *argumentem pozornym* (tzn. argumentem wymaganym przez składnię języka, ale w programie nie odgrywającym żadnej roli).

Listing 3.4. *Kod głównego modułu zmodyfikowanego projektu Projekt_14.bpr wykorzystującego dwuargumentową zaprzyjaźnioną funkcję kategorii friend przeładowanego operatora dwuargumentowego (!=). Pierwszy argument funkcji jest argumentem rzeczywistym, drugi pozornym*

```
#include <iostream.h>
#include <conio.h>
#pragma hdrstop
class factorial
{
  public:
    unsigned int value;
    factorial (unsigned int n, unsigned int s) {value = n;}
    friend void operator!=(factorial &, factorial &);
};

//---------------------------------------------------------
void operator!=(factorial &number, factorial &seeming/*pozorny*/)
{
  unsigned int result = 1;
  for(unsigned int i=2; i<=number.value; i++)
   result *= i;
  if(result == 0)
    cout << "przekroczony zakres liczb unsigned int";
    else
     cout << result;
  return;
}
//---------------------------------------------------------
int main()
{
   int n, s;
  // s jako parametr pozorny musi być zadeklarowany, ale
  // w programie nie otrzymuje żadnej wartości
   cout << "Podaj liczbę: ";
   try
   {
    cin >> n;
    if (n < 0)
      // nieprawidłowy argument dla
      // funkcji operatora != (silnia)
      throw(n);
   }
   catch(unsigned int n)
   {};
   factorial factorial_n(n, s);
   cout << n << "! = ";
   factorial_n != factorial_n;
   getch();
   return 0;
}
```

Ćwiczenie 3.7.

Samodzielnie zaprojektuj dowolną metodę przeładowywania dwuargumentowych operatorów unarnych (+) oraz (-)

Przeładowywanie operatora &

Jak zapewne wiemy, & może być używany jako operator jednoargumentowy — występuje wówczas w programie w roli operatora adresowego (por. ćwiczenie 1.2) lub może być operatorem dwuargumentowym — wtedy odgrywa rolę operatora bitowej koniunkcji. W niniejszym podrozdziale jednoargumentowy operator & przeładujemy tak, aby, używając go w odpowiedni sposób, możliwym było odczytanie wartości wybranego elementu jednowymiarowej tablicy.

Ćwiczenie 3.8.

W celu odpowiedniego przeładowania operatora & skonstruujemy nową klasę `array`. Funkcję operatorową potraktujemy jako funkcję zaprzyjaźnioną po to, aby mogła uzyskiwać dostęp do wszystkich pól egzemplarza swojej klasy, tak jak pokazano to na listingu 3.5. W wyniku działania programu powinniśmy bez trudu odczytać w odwrotnej kolejności wartości elementów tablicy `tab` oraz wykonać na nich wybrane działanie arytmetyczne.

Listing 3.5. *Kod głównego modułu Unit_15.cpp projektu Projekt_15.bpr wykorzystującego jednoargumentową zaprzyjaźnioną funkcję przeładowanego jednoargumentowego operatora &*

```
#include <iostream.h>
#include <conio.h>
#include <system.hpp>
#pragma hdrstop

class array
{
    int tablica;
  public:
    friend int operator&(array &number)
    {return (number.tablica);}
};
//---------------------------------------------------------
int main()
{
   array tab[]={1,2,3,4,5,6,7,8,9,10};
   // wyświetlanie elementów tablicy w odwrotnej kolejności
   for(int i=ARRAYSIZE(tab)-1; i>=0; i--)
     cout << &tab[i] << endl;
   cout << &tab[0] <<"+"<< &tab[9] <<" = "<< &tab[0]+&tab[9];
   getch();
   return 0;
}
```

Testując algorytm funkcji przeładowanego operatora, & możemy samodzielnie stwierdzić, iż powtórne wykorzystanie jednoargumentowego operatora adresowego & w celu pobrania adresów poszczególnych elementów zainicjowanej odpowiednimi wartościami tablicy `tab` z oczywistych względów okaże się czynnością niemożliwą do zrealizowania. Wynika to z faktu, iż tablica `tab` w istocie została zadeklarowana w funkcji `main()` jako pewien obiekt klasy `array`, w której uprzednio zdefiniowano już jednoargumentową funkcję `operator&()`przeładowanego operatora &.

Przeładowywanie operatora indeksowania tablic []

Operator indeksowania tablic [] podczas przedefiniowywania traktowany jest jako operator dwuargumentowy i z powodzeniem może być przeładowany za pomocą funkcji składowej klasy bez potrzeby posługiwania się funkcją zaprzyjaźnioną.

Ćwiczenie 3.9.

Jako praktyczny przykład wykorzystania funkcji operator[][() przeładowanego operatora [] rozpatrzmy prostą klasę array, w której zadeklarowano jednowymiarową tablicę tablica o pięciu elementach typu double.

Konstruktor array() przypisuje każdemu z jej elementów odpowiednią wartość początkową.

```
class array
{
    double tablica[5];
  public:
    // konstruktor
    array(double i, double j, double k, double l, double m)
    {
      tablica[0]=i;
      tablica[1]=j;
      tablica[2]=k;
      tablica[3]=l;
      tablica[4]=m;
    }
```

Wartością powrotną funkcji przeładowanego operatora [] jest wartość elementu tablicy o numerze (indeksie) jednoznacznie określonym poprzez argument funkcji:

```
double operator[](int i)
 {return tablica[i];}
```

Na listingu 3.6 pokazano praktyczny przykład zastosowania w programie omawianych funkcji.

Listing 3.6. *Kod głównego modułu Unit_16.cpp projektu Projekt_16.bpr wykorzystującego jednoargumentową funkcję składową przeładowanego operatora []*

```
#include <iostream.h>
#include <conio.h>
#include <system.hpp>
#pragma hdrstop

class array
{
    double tablica[5];
  public:
    // konstruktor
```

```
    array(double i, double j, double k, double l, double m)
    {
      tablica[0]=i;
      tablica[1]=j;
      tablica[2]=k;
      tablica[3]=l;
      tablica[4]=m;
    }
    double operator[](int i)
    {return tablica[i];}
};
//---------------------------------------------------------
int main()
{
   array tab(1.15,2.25,3.35,4.45,5.55);
   for(int i=0; i<ARRAYSIZE(tab); i++)
     cout << tab[i] << endl;
   getch();
   return 0;
}
```

Ćwiczenie 3.10.

Pokazaną w poprzednim ćwiczeniu funkcję przeładowanego operatora `[]` można zdefiniować również w ten sposób, aby operator `[]` mógł być używany zarówno po lewej, jak i po prawej stronie instrukcji przypisania. W tym celu wystarczy zastosować typ odwoławczy wartości powrotnej funkcji `operator[]()`:

```
double &operator[](int i);
```

Zapis taki spowoduje, iż funkcja zwracać będzie teraz odwołanie do elementu tablicy o indeksie `i`. Skutkiem tego umieszczenie wartości funkcji po lewej stronie instrukcji przypisania spowoduje zmodyfikowanie określonego elementu tablicy, tak jak pokazano to na listingu 3.7. Śledząc poniższy listing, warto zwrócić uwagę na sposób określania błędu przekroczenia dopuszczalnego zakresu tablicy `tablica`. Dzięki zastosowaniu prostego bloku instrukcji `try…throw…catch`, funkcja operatorowa:

```
double &array::operator[](int i)
{
  try
  {
   if(i > 5) throw(&tablica[i]);
  }
  catch(double &tablica)
  {}
  return tablica[i];
}
```

bez trudu wykrywa błąd przekroczenia zakresu tablicy, co skutkuje wygenerowaniem stosownego wyjątku informującego o próbie przypisania nieodpowiedniego miejsca pamięci, czyli błędu naruszenia pamięci.

Listing 3.7. *Kod głównego modułu Unit_17.cpp projektu Projekt_17.bpr wykorzystującego jednoargumentową funkcję składową przeładowanego operatora []. Funkcja operatorowa operator[]() zwraca wartość powrotną poprzez odwołanie*

```
#include <iostream.h>
#include <conio.h>
#include <system.hpp>
#pragma hdrstop

class array
{
    double tablica[5];
  public:
    array(double i, double j, double k, double l, double m)
    {
      tablica[0]=i;
      tablica[1]=j;
      tablica[2]=k;
      tablica[3]=l;
      tablica[4]=m;
    }
    // funkcja operatorowa zwraca wartość powrotną
    // poprzez odwołanie
    double &operator[](int i);
};
// sprawdza zakres tablicy klasy array
double &array::operator[](int i)
{
  try
  {
   if(i > 5) throw(&tablica[i]);
  }
  catch(double &tablica)
  {}
  return tablica[i];
}
//---------------------------------------------------------
int main()
{
   array tab(1.15,2.25,3.35,4.45,5.55);
   for(int i=0; i<ARRAYSIZE(tab);i++)
     cout << tab[i] << endl;

   cout << endl;
   // przeładowany operator []
   // po lewej stronie znaku przypisania
   tab[0]=125.555;
   tab[1]=335.333;
   tab[2]=445.777;
   tab[3]=505.888;
   tab[4]=705.999;
   for(int i=0; i<ARRAYSIZE(tab);i++)
     cout << tab[i] << endl;

   getch();
   return 0;
}
```

Ćwiczenie 3.11.

Zmodyfikuj funkcję `operator[]()` w ten sposób, aby w przypadku wykrycia błędu przekroczenia zakresu tablicy powstałego w wyniku błędnego przypisania typu:

```
tab[900]=705.999;
```

program został bezpiecznie zakończony, nie generując przy tym wyjątku informującego o błędzie naruszenia pamięci.

Podsumowanie

Możliwość swobodnego przedefiniowywania istniejących operatorów jest jedną z wielkich zalet obiektowego języka C++. Rozsądne posługiwanie się samodzielnie przeładowanymi operatorami w wielu przypadkach może znacznie uprościć kod tworzonego programu i spowodować, iż stanie się ona bardziej czytelny dla osób, które będą się nim opiekować w dalszej perspektywie czasu.