C++Builder i Turbo C++. Podstawy

Autor: Jacek Matulewski
ISBN: 83-246-0642-4
Format: B5, stron: 280
Przykłady na ftp: 4122 kB

Wizualne środowiska projektowe od dłuższego czasu cieszą się uznaniem programistów. Możliwość budowania aplikacji z gotowych komponentów, realizujących typowe funkcje, pozwala skoncentrować się na jej funkcjonalności bez potrzeby ponownego wymyślania koła. Najbardziej znanym środowiskiem tego typu jest Delphi, jednak jego producent, firma Borland, wypuścił na rynek kolejne narzędzie: C++Builder. To wizualne środowisko projektowe oparte na języku C++ pozwala tworzyć aplikacje dla platformy Win32 z wykorzystaniem komponentów VCL. W sieci dostępna jest również jego bezpłatna wersja o nazwie Turbo C++ Explorer.

„C++Builder i Turbo C++. Podstawy" to podręcznik programowania w tych środowiskach. Czytając go, nauczysz się tworzyć aplikacje w języku C++ dla systemu Windows z wykorzystaniem C++Buildera lub Turbo C++. Dowiesz się, jak zainstalować i skonfigurować środowisko programistyczne oraz jak utworzyć w nim projekt. Poznasz elementy języka C++, zasady programowania obiektowego i korzystania z komponentów VCL. Stworzysz własne komponenty i aplikacje, zaimplementujesz mechanizm przeciągania i upuszczania, a także zapiszesz dane aplikacji w rejestrze systemu Windows.

- Instalacja środowiska programistycznego
- Pierwszy projekt
- Zmienne i instrukcje w C++
- Programowanie zorientowane obiektowo
- Wyszukiwanie i usuwanie błędów w kodzie
- Komponenty VCL oferowane przez C++Buildera
- Tworzenie interfejsu użytkownika dla aplikacji
- Drukowanie
- Operacje na plikach
- Przechowywanie informacji w rejestrze systemowym
- Tworzenie własnych komponentów VCL

Poznaj nowoczesne narzędzia programistyczne

Wydawnictwo Helion
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl

Spis treści

Rozdział 3.

Typy zmiennych i instrukcje sterujące, czyli o tym, co każdy programista umieć musi

Żeby nie zanudzać Czytelnika suchym wykładem o poszczególnych typach zmiennych predefiniowanych w C++, instrukcjach sterujących i tym podobnych rzeczach, których tak czy inaczej trzeba się nauczyć, od razu proponuję zająć się programowaniem — wiedza o języku pojawiać się będzie jako niezbędny element składowy opisywanych programów. Przy okazji nauczymy się też, jak korzystać z najbardziej podstawowych komponentów: pola edycyjnego `TEdit`, etykiety `TLabel` i przycisku `TButton`. Nie zamierzam bowiem zmuszać nikogo do tworzenia aplikacji konsolowych, na których zwykle uczy się programowania, co w przypadku narzędzi RAD jest mało naturalne i raczej nieatrakcyjne.

Podstawy

Zacznijmy od spraw podstawowych. Na przykład od powtórzenia informacji, że w C++ wielkość liter ma podstawowe znaczenie. Możemy na przykład zadeklarować trzy zmienne: `zmienna`, `Zmienna` i `ZMIENNA`, i każda z nich będzie przez kompilator traktowana jako oddzielna, zupełnie niezależna zmienna.

Nie ma natomiast znaczenia sposób ułożenia kodu. Oznacza to, że pomiędzy słowa kodu można wstawić dowolną ilość spacji i zrobić dowolnie wielkie wcięcia — kompilator nie zwróci na to uwagi.

Wszystkie zmienne w C++ są inicjowane. Jeżeli przy deklarowaniu zmiennej nie wskażemy jej wartości, to zostanie ona zainicjowana wartością domyślną. W przypadku większości typów jest to zero.

W rozdziale pierwszym do zmiany koloru panelu, a więc do przypisania nowej wartości własności `Panel1->Color`, użyliśmy operatora =. W C++ jest to właśnie operator przypisania. To tym operatorem nadajemy nową wartość wszelkiego typu zmiennym. Do porównywania dwóch zmiennych służy natomiast operator ==, który zwraca wartość `true` (prawda), gdy zmienne są równe, i `false` (fałsz) w przeciwnym przypadku.

Równanie kwadratowe

Przygotujmy program rozwiązujący równanie kwadratowe. Jest to przykład na tyle prosty, żeby był łatwo zrozumiały bez większego wysiłku, a jednocześnie informatycznie na tyle złożony, żeby możliwe było przedstawienie wielu aspektów języka programowania. Jest to wręcz idealny przykład na zastosowanie instrukcji wyboru `if` i operacji arytmetycznych.

Najpierw jednak trochę teorii dla tych, którzy zdążyli już zapomnieć, jak oblicza się pierwiastki równania kwadratowego i czym one w ogóle są. Równanie kwadratowe to równanie, w którym wyrażenie typu ax^2+bx+c przyrównuje się do zera, a więc $ax^2+bx+c=0$. Współczynniki równania a, b i c są ustalone i możemy założyć, że je znamy. Zakładamy dodatkowo, że współczynnik a jest różny od zera[1]. Naszym zadaniem jest natomiast wyznaczenie takich wartości liczby x, dla których równanie będzie spełnione, tzn. że po wstawieniu znalezionego x do lewej strony będzie ona równa zero.

Jeżeli pozwolimy, żeby x było liczbą zespoloną, to równanie kwadratowe ma zawsze dwa, choć niekoniecznie różne, rozwiązania. W C++ liczby zespolone nie są jednak jednym z typów wbudowanych, choć obecny jest on w dołączonych do C++Buildera bibliotekach. Proponuję zatem ograniczyć się do liczb rzeczywistych, a wówczas równanie kwadratowe może mieć dwa różne rozwiązania, jedno rozwiązanie „podwójne" lub nie mieć rozwiązań. Wszystko zależy od wartości parametrów, a dokładnie od wartości ich następującej kombinacji: $\Delta = b^2 - 4ac$. Jest to wyróżnik równania kwadratowego nazywany popularnie deltą, bo takiego symbolu używa się zazwyczaj do jego oznaczenia. Jeżeli wartość delty jest dodatnia, to równanie ma dwa różne rozwiązania (pierwiastki). Jeżeli równa jest zero, to pierwiastki stają się sobie równe i mówimy, że równanie ma jedno rozwiązanie będące pierwiastkiem podwójnym. Natomiast jeżeli delta jest ujemna, to równanie nie ma rozwiązań w dziedzinie liczb rzeczywistych. Obliczenie delty to już połowa sukcesu, bo o ile nie jest ujemna, pozwala na bezpośrednie obliczenie wartości pierwiastków, które są równe:

$$x_1 = \frac{-b-\sqrt{\Delta}}{2a} \text{ i } x_2 = \frac{-b+\sqrt{\Delta}}{2a}$$

Widać, że jeżeli wartość delty równa jest zero, a więc znika pierwiastek w liczniku, to x_1 i x_2 mają taką samą wartość i są równe $-b/2a$. To wspomniany pierwiastek podwójny.

[1] Jeżeli a jest równe zero, to równanie kwadratowe degraduje się do równania $bx+c = 0$, którego rozwiązaniem jest $x = -c/b$ (wówczas b musi być różne od zera).

Algorytm obliczania rozwiązań równania kwadratowego jest zatem następujący:

1. Odczytujemy wartości współczynników równania
2. Obliczamy wartość delty Δ.
3. Sprawdzamy, czy wartość delty jest mniejsza od zera: jeżeli tak, kończymy, pokazując komunikat o braku pierwiastków.
4. Jeżeli delta jest nieujemna, obliczamy pierwiastki i prezentujemy je użytkownikowi.

Przygotowanie interfejsu

Aby umożliwić użytkownikowi podanie współczynników równania, zastosujemy jeden z najbardziej podstawowych komponentów biblioteki VCL, a mianowicie `TEdit` — pole edycyjne. A dokładniej trzy tego typu komponenty, po jednym dla każdego współczynnika. Z każdym polem edycyjnym związana będzie etykieta informująca, który współczynnik należy wpisać do pola. Etykieta to komponent `TLabel`. Wynik pokażemy natomiast w okienku dialogowym, a ponadto na dodatkowym komponencie `TEdit`. Obliczenia uruchamiane będą za pomocą przycisku `TButton`. `TEdit`, `TLabel` i `TButton` to trzy chyba najczęściej używane komponenty biblioteki VCL.

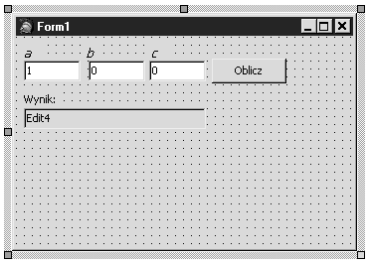
Świadomie pominąłem komponent `TSpinEdit` z zakładki *Samples*, który byłby z pewnością wygodniejszy do kontroli liczb, którymi są współczynniki równania. Chciałem po prostu przedstawić Czytelnikowi komponent `TEdit`.

Stwórzmy zatem nowy projekt aplikacji. Dokładniejszy opis czynności, które należy w tym celu wykonać, znajdzie Czytelnik w pierwszym rozdziale, ale ograniczają się one w zasadzie do wybrania pozycji *VCL Forms Application — C++Builder* z menu *File/New*.

W widoku projektowania na formie należy umieścić trzy komponenty `TEdit` według wzoru na rysunku 3.1. Za pomocą inspektora własności zmieniamy ich własności `Text` odpowiadające zawartości pól na np. `1` w przypadku pierwszego i zera w przypadku pozostałych (zob. rysunek 3.1). Nad każdym z nich warto umieścić komponent `TLabel`. Ich dokładne pozycje można dopasować za pomocą własności `Left` i `Top` widocznych w inspektorze obiektów. Etykiety tych komponentów zmieniamy kolejno na *a*, *b* i *c*. Można też zmienić ich własność `Font` w taki sposób, żeby powiększyć etykiety i użyć kursywy[2]. Dzięki temu będzie jasne, jaką wartość należy wpisać do każdego pola edycyjnego.

Umieszczamy tam także jeszcze jedno pole edycyjne, w którym pokażemy wynik. Jego własność `ReadOnly` (z ang. tylko do odczytu) zmieniamy na `true`. Żeby wyraźnie zaznaczyć, że nie jest to pole, którego wartość będzie dostępna do edycji, proponuję zmienić kolor jego tła na identyczny z kolorem formy (rysunek 3.1). W tym celu z rozwijanej listy w inspektorze obiektów przy własności `Color` wybieramy pozycję `clBtnFace`. Tę samą domyślną wartość ma własność `Color` formy.

[2] O tym, jak wykonać te czynności, dowiedzieliśmy się w pierwszym rozdziale.



Rysunek 3.1. *Interfejs aplikacji znajdującej rozwiązania równania kwadratowego*

Obok tych komponentów kładziemy jeszcze przycisk `TButton`. Za pomocą inspektora własności zmieniamy jego własność `Caption` np. na *Oblicz* (rysunek 3.1). Następnie klikamy go dwukrotnie, aby utworzyć domyślną metodę zdarzeniową. Przeniesieni zostaniemy do edytora, gdzie zobaczymy utworzoną metodę — przez najbliższy czas będzie to nasz cały ogródek, w którym będziemy uczyć się programowania w C++.

Deklarowanie zmiennych

W C++ nie ma wydzielonego miejsca, w którym należy deklarować zmienne. Ogromne znaczenie ma jednak to, czy zadeklarujemy ją wewnątrz, czy na zewnątrz metody `Button1Click`. W drugim przypadku będziemy mieli do czynienia ze zmienną globalną istniejącą przez cały czas działania programu. W pierwszym — ze zmienną lokalną tworzoną tylko na czas wykonywania metody `Button1Click`. Rozwiązanie drugie ogranicza ilość wykorzystywanej pamięci. Jest również znacznie bezpieczniejsze, bo łatwiej kontrolować wartość zmiennej, która nie może być zmieniana nigdzie indziej jak tylko w metodzie `Button1Click`.

Musimy obliczyć wartość delty. **Zadeklarujmy** więc w metodzie `Button1Click` zmienną lokalną `Delta` typu `double`. W tym celu w metodzie wpisujemy typ `double`, a po spacji nazwę zmiennej `Delta` (listing 3.1). Typ `double` potrafi przechowywać liczby rzeczywiste. Na ich przechowywanie posiada 64-bity (8 bajtów), co daje mu możliwość przechowywania liczb o wartości ponad $\pm 10^{300}$.

Listing 3.1. *Deklaracja zmiennej Delta typu Double*

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  double Delta;
}
```

Jeżeli więcej zmiennych ma ten sam typ, to możemy je zadeklarować razem. Dodajmy jeszcze trzy zmienne o nazwach `a`, `b` i `c` typu `double` (listing 3.2). Zmienne te będą przechowywać wartości współczynników równania kwadratowego.

Listing 3.2. *W metodzie zadeklarowane są teraz cztery zmienne lokalne*

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  double a,b,c,Delta;
}
```

Podkreślmy, że są to zmienne lokalne metody `Button1Click`. To znaczy, że powstają w momencie wywołania tej metody, a usuwane są z pamięci w momencie jej zakończenia. Jak wspomniałem, można również tworzyć zmienne globalne, tzn. zmienne, których życie trwa przez cały okres działania programu. Przykładem takiej zmiennej jest `Form1` zadeklarowane w interfejsie modułu *Unit1*. Ogólnie rzecz biorąc, należy jednak ograniczać korzystanie ze zmiennych globalnych, a wszystkie niezbędne dane przesyłać przez argumenty funkcji i metod. Żeby dać Czytelnikowi dobry przykład, w tym rozdziale i w całej książce w ogóle nie będziemy używać zmiennych globalnych.

Inicjacja i przypisanie wartości zmiennej

Zmienne zadeklarowane w listingu 3.1 i 3.2 nie są automatycznie inicjowane! To oznacza, że rezerwowana jest pamięć, w której przechowywana będzie zmienna, ale jej zawartość nie jest czyszczona. W konsekwencji wartość tak zadeklarowanej zmiennej jest przypadkowa. Każda deklarowana zmienna powinna być inicjowana, tj. powinniśmy przypisać jej wartość w momencie utworzenia (takie sformułowanie dotyczy zmiennych lokalnych, do których ograniczamy się w tej książce). Należy to zrobić w następujący sposób:

```
double a=1;
```

Zmienna, która została zainicjowana, może oczywiście zmieniać wartość w trakcie działania programu. W takiej sytuacji mówimy o operacji **przypisania**, np.

```
double a=1;
a=2;
```

W przypadku typów prostych, jak `int`, obie te czynności można utożsamiać. Różnice pojawiają się w przypadku klas, ale to temat wykraczający poza zakres tej książki[3].

Dygresja na temat typów rzeczywistych w C++Builderze

C++Builder nie ma zbyt wielu typów rzeczywistych, ale są one w zupełności wystarczające. Wszystkie (raptem trzy) wymienione zostały w tabeli 3.1.

Tabela 3.1. *Typy rzeczywiste w C++Builder 2006*

Nazwa typu	Zakres (najmniejsza i największa absolutna wartość liczby)	Liczba bajtów (bitów) zajmowana przez zmienną[4]	Postać literału
`Float`	$1{,}5 \cdot 10^{-45}$ .. $3{,}4 \cdot 10^{38}$	4 (32)	`1.0F`
`Double`	$5{,}0 \cdot 10^{-324}$ .. $1{,}7 \cdot 10^{308}$	8 (64)	`1.0`
`long double`	$3{,}4 \cdot 10^{-4932}$ .. $1{,}1 \cdot 10^{4932}$	10 (80)	`1.0L`

[3] Omówienie różnic między inicjacją i przypisaniem w przypadku klas znajdzie Czytelnik w książce Stephena C. Dewhursta *C++. Kanony wiedzy programistycznej*, Helion 2005

[4] Liczbę bajtów zajmowaną przez typ można sprawdzić następującą instrukcją: `ShowMessage(IntToStr(sizeof(`*typ*`)));`.

Konwersja łańcucha na liczbę

Pole edycyjne `TEdit` pozwala użytkownikowi na wpisanie łańcucha. Łańcuch ten może być następnie wykorzystany przez program, który może odczytać go z własności `Text` typu `AnsiString`. `AnsiString` jest najbardziej „typowym typem" łańcuchów w C++Builderze i stosunkowo rzadko zachodzi potrzeba, żeby korzystać z łańcucha typowego dla C i C++, a więc tablicy znaków, tj. tablicy zmiennych typu `char`, który jest znacznie mniej wygodny w użyciu. Łańcuchy `AnsiString` są w C++Builderze identyfikowane za pomocą cudzysłowu, np. `"Helion"`. Nie ma w łańcuchach problemu z polskimi znakami, można ich swobodnie używać.

Zakładamy (zapewne naiwnie), że użytkownik domyśli się, iż w pola edycyjne należy wpisać współczynniki równania kwadratowego, a więc liczby rzeczywiste. Wówczas będziemy mogli **skonwertować** łańcuchy z własności `Text` każdego pola edycyjnego na liczby rzeczywiste i zapisać je do zmiennych `a`, `b` i `c`. Na szczęście nie ma żadnego problemu z konwertowaniem łańcuchów na liczby (rzeczywiste lub naturalne). W przypadku liczb rzeczywistych należy do tego użyć funkcji `StrToFloat`. Listing 3.3 zawiera przykład konwersji łańcucha z pierwszego pola edycyjnego do zmiennej `a`.

Listing 3.3. *Konwersja łańcucha na liczbę*

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  double a,b,c,Delta;
  a=StrToFloat(Edit1->Text);
}
```

Podobnie zróbmy z pozostałymi współczynnikami (listing 3.4). Zrezygnujemy przy tym z deklaracji ich we wspólnej linii na rzecz czytelności kodu:

Listing 3.4. *Konwersja wszystkich danych wejściowych*

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  double a=StrToFloat(Edit1->Text);
  double b=StrToFloat(Edit2->Text);
  double c=StrToFloat(Edit3->Text);
  double Delta;
}
```

Do konwersji łańcucha na liczbę naturalną służy funkcja `StrToInt`.

Z konwersją łańcuchów do liczb rzeczywistych wiążą się problemy. Jednym z podstawowych jest sposób rozdzielania części całkowitej od dziesiętnej, a więc tzw. „przecinek". W Polsce jako przecinka zgodnie ze standardami powinno używać się… przecinka. Wiem, to trochę masło maślane. Ale mniej maślane okazuje się w krajach anglosaskich, w których jako przecinka używa się kropki. Funkcja `StrToFloat` auto-

matycznie wykrywa ustawienia systemowe i odpowiednio do nich interpretuje wstawione znaki, w tym kropkę lub przecinek. Programista może też wymusić, jaki znak ma być podczas konwersji interpretowany jako „przecinek”[5].

Jeżeli konwersja nie powiedzie się, to znaczy gdy użytkownik do pola edycyjnego wpisze tekst, który mimo najszczerszych chęci nie może być zinterpretowany przez funkcję `StrToFloat` jako liczba, to zgłosi ona wyjątek informujący o błędzie. To da nam możliwość zareagowania na błąd bez zawieszania aplikacji, ale tym zajmiemy się później. Na razie załóżmy, że użytkownik naszej aplikacji będzie posłuszny i rzeczywiście w polach edycyjnych umieści liczby.

Obliczenia arytmetyczne i ich kolejność

Jeżeli konwersja powiedzie się, otrzymamy trzy współczynniki równania, które stanowią dane wejściowe naszego algorytmu. Możemy zatem zabrać się za obliczenie delty według wzoru $\Delta = b^2 - 4ac$. Do tego konieczne będzie mnożenie i odejmowanie współczynników od siebie.

Do podstawowych operacji arytmetycznych, czyli do dodawania, odejmowania, mnożenia i dzielenia, służą **operatory**: `+`, `-`, `*` i `/` (zob. tabela 3.2), które pozwalają kolejno na dodawanie, odejmowanie, mnożenie i dzielenie liczb. Typ liczby zwracanej przez te operatory zależy od typów jego argumentów. Zatem dodając dwie liczby `int`, otrzymamy wynik typu `int`. To naturalne także dla operatora odejmowania i mnożenia. Kłopoty sprawia jednak operator dzielenia. Bo przecież iloraz dwóch liczb całkowitych nie musi być liczbą całkowitą. Wręcz przeciwnie — częściej nią nie jest. A jednak operator `/` również przestrzega zasady, według której typ wyniku zależy od typu argumentów. W związku z tym `1/2` ma wartość `0` (zaokrąglenie wyniku następuje zawsze w kierunku zera), ale `1/(double)2` lub `1/2.0` równy jest `0.5`. Należy na to zwracać uwagę, bo to jedno z częstszych źródeł błędów logicznych w programach.

Tabela 3.2. *Operatory arytmetyczne w C++*

Operator	Opis	Przykłady	Priorytet
`*`	mnożenie (jeżeli argumenty są rzeczywiste, to zwracany przez operator typ też jest rzeczywisty)	`2*2` daje `4`, `2.0*2.0` daje `4.0`	4
`/`	dzielenie (jeżeli argumenty są całkowite, to zwracany przez operator typ też jest zaokrąglony do liczby całkowitej)	`2.0/4.0` daje `0.5`, `2/4` daje `0`	4
`%`	reszta z dzielenia całkowitego (nie może być użyta z liczbami rzeczywistymi)	`2 % 4` daje `2`, `4 % 2` daje `0`	4
`+`	dodawanie (zwracany typ zależy od typu argumentów)	`2+4` daje `6`, `2.0+4.0` daje `6.0`	5
`-`	odejmowanie (operator dwuargumentowy) i zmiana znaku (operator jednoargumentowy)	`2-4` daje `-2`, `-2` daje… `-2`	5

[5] Opis tego zagadnienia znajduje się w książce J. Matulewskiego *C++Builder 2006. 222 gotowe rozwiązania*, Helion 2006.

Jeżeli w operatorach / i % drugi argument będzie miał wartość 0, to podczas wykonywania jednej z tych operacji zgłoszony zostanie wyjątek EDivByZero.

Aby obliczyć deltę, wystarczy odjąć od siebie dwa iloczyny, co w C++ należy zapisać jako: b*b-4*a*c. Przyjrzyjmy się temu wyrażeniu. Wiemy dobrze, że oznacza ono różnicę dwóch iloczynów, a więc (b*b)-(4*a*c). Kolejność działań wyznaczona jest przez priorytet każdej operacji arytmetycznej. Wszyscy zostaliśmy nauczeni jeszcze w szkole podstawowej, że mnożenie i dzielenie ma pierwszeństwo przed dodawaniem i odejmowaniem. Ale czy wie o tym kompilator C++Buildera? Na szczęście tak. Priorytety operatorów arytmetycznych w C++ (ostatnia kolumna w tabeli 3.2) całkowicie zgadzają się z tymi, jakie obowiązują w algebrze.

Możemy zatem bez obaw dopisać do metody Button1Click wyrażenie obliczające wartość pola Delta zgodnie ze wzorem z poniższego listingu:

Listing 3.5. *Obliczanie wyróżnika równania kwadratowego*

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  double a=StrToFloat(Edit1->Text);
  double b=StrToFloat(Edit2->Text);
  double c=StrToFloat(Edit3->Text);
  double Delta=b*b-4*a*c;
}
```

Operatory upraszające zapis operacji arytmetycznych wykonywanych na zmiennej

Mówiąc o operatorach arytmetycznych, warto wspomnieć o charakterystycznych dla C++ operatorach przypisania, które zastępują operatory arytmetyczne w szczególnych, ale często spotykanych sytuacjach.

Załóżmy, że w naszym programie jest zmienna n, której wartość chcemy zwiększyć o dwa. Możemy napisać instrukcję

```
n=n+2;
```

Odczytuje ona bieżącą wartość zmiennej n, dodaje do niej 2 i zapisuje nową wartość z powrotem do tej samej zmiennej. Instrukcja z operatorem = wykonywana jest bowiem od prawej do lewej, a więc najpierw obliczana jest wartość wyrażenia stojącego po prawej stronie tego operatora, a dopiero potem następuje przypisanie.

C++ oferuje jednak operator, który upraszcza powyższą instrukcję:

```
n+=2;
```

Ze względu na wynik jest to instrukcja zupełnie równoważna poprzedniej, ale tym razem następuje tylko jedno odwołanie do zmiennej n, a poza tym zamiast dwóch operacji (+ i =) wykonywana jest tylko jedna. Poza wspomnianym wyżej operatorem += mamy do dyspozycji także analogiczne operatory *=, /=, %= i -=. Ich priorytet równy jest 15.

Jeżeli chcielibyśmy zwiększyć wartość zmiennej tylko o jeden — taka sytuacja ma często miejsce w przypadku indeksów pętli — moglibyśmy użyć jeszcze innego operatora, a mianowicie ++. Jest to operator inkrementacji i może być umieszczony zarówno przed, jak i za zmienną. I ma to zasadnicze znaczenie w przypadku, gdy występuje on wspólnie z operatorem przypisania.

```
//przypadek a)
int a1=2;
int a2=++a1;
ShowMessage(a2);

//przypadek b)
int b1=2;
int b2=b1++;
ShowMessage(b2);
```

Jeżeli operator inkrementacji umieszczony zostanie przed zmienną, jak w przypadku a), to wartość zmiennej `a1` zostanie najpierw zwiększona o jeden, a dopiero wtedy jej wartość zostanie użyta do zainicjowania zmiennej `a2`. W przypadku b) kolejność operacji jest odwrotna. Wpierw wartość zmiennej `b1` zostanie przypisana do `b2`, a dopiero wówczas wartość `b1` zostanie zwiększona.

Typ logiczny i operatory logiczne

Poza operatorami arytmetycznymi zdefiniowane są jeszcze operatory logiczne, operatory porównania, operatory związane ze wskaźnikami (omówimy je w następnym rozdziale), operatory działające na bitach liczb (te omówimy w rozdziale 10.), operatory dotyczące zbiorów oraz np. operator pozwalający na łączenie łańcuchów. Operatory porównania to ==, != (różne), <, >, <= i >=. Myślę, że ich działania nie trzeba omawiać, bo działają w sposób jak najbardziej intuicyjny. Natomiast operatory logiczne to `!`, `&&` i `||`. Wszystkie dotyczą zmiennej typu `bool`, która może przyjmować wartość `true` lub `false`. Działanie operatorów logicznych omówione zostało w tabeli 3.3. Zasadniczym zastosowaniem tych operatorów będzie konstruowanie warunków instrukcji warunkowej, którą zaraz poznamy, i warunków przerywania pętli, którymi zajmiemy się trochę później.

Tabela 3.3. *Operatory logiczne (`t = true, f = false`)*

Operator	Opis	Przykłady	Priorytet
!	negacja	`!t = f`, `!f` daje `t`	2
&&	koniukcja	`t && t = t`, `t && f = f`, `f && f = f`	12
||	alternatywa	`t || t = t`, `t || f = t`, `f || f = f`	13

Instrukcja warunkowa if

Wróćmy do naszego równania kwadratowego. Na razie mamy obliczoną wartość delty. Sprawdźmy, czy nie jest ona mniejsza od zera. Jak pamiętamy, równanie nie ma wówczas rozwiązań. Do tego typu zadań służy **instrukcja warunkowa** `if` (ang. jeżeli):

```
if (warunek) polecenie;
```

Polecenie zostanie wykonane jedynie wtedy, gdy *warunek* zostanie spełniony, a więc jeżeli wyrażenie pełniące rolę warunku ma wartość `true`. Jeżeli od warunku chcemy uzależnić wykonanie większej ilości poleceń, to musimy je umieścić w bloku otoczonym nawiasami klamrowymi { i }:

```
if (warunek)
{
  polecenia
}
```

Szkielet instrukcji warunkowej w takiej postaci widoczny jest na listingu 3.6. Dopiszmy go do edytowanej przez nas metody.

Listing 3.6. *Jeżeli delta jest mniejsza od zera, to...*

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  double a=StrToFloat(Edit1->Text);
  double b=StrToFloat(Edit2->Text);
  double c=StrToFloat(Edit3->Text);
  double Delta=b*b-4*a*c;
  if (Delta<0)
  {

  }
}
```

Co zrobimy, jeżeli `Delta` jest mniejsza od zera? Oczywiście poinformujemy użytkownika, że nie ma co liczyć na rozwiązania. Równanie, którego współczynniki podał, nie ma rozwiązań wśród liczb rzeczywistych. Informację o braku rozwiązań najprościej będzie umieścić w przeznaczonym do tego czwartym polu edycyjnym (komponent `Edit4`). Do jego własności `Text` przypiszmy łańcuch z odpowiednim komunikatem (listing 3.7).

Listing 3.7. *Własność Text pól edycyjnych służy nie tylko do odczytania zawartości pola, ale również do jego zmiany*

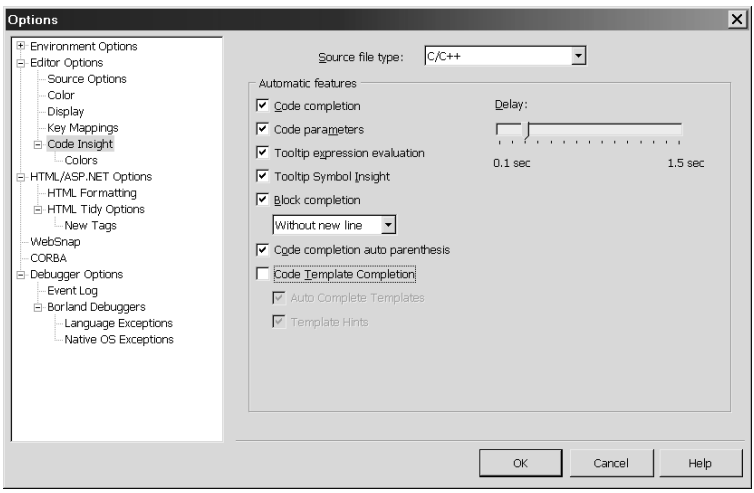
```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  double a=StrToFloat(Edit1->Text);
  double b=StrToFloat(Edit2->Text);
  double c=StrToFloat(Edit3->Text);
  double Delta=b*b-4*a*c;
  if (Delta<0)
  {
        Edit4->Text="Brak rozwiązań (delta mniejsza od zera)";
  }
}
```

W przypadku, gdy w razie spełnienia warunku wykonywane jest tylko jedno polecenie, klamry { i } nie są oczywiście konieczne, ale nawet wówczas warto je umieścić w kodzie, bo podnoszą jego czytelność, nie zmieniając jego sensu i nie wpływając na wielkość skompilowanego pliku *.exe*. Poza tym unikniemy w ten sposób błędu, jeżeli wbrew wcześniejszym intencjom zdecydujemy się jednak dopisać jakieś nowe polecenia.

Proszę zwrócić uwagę, że w C++ operator porównania dwóch zmiennych to ==, a nie =. Zatem w warunku instrukcji `if` może znaleźć się ==, ale raczej nie powinno =.

Jak wyłączyć podpowiadanie szablonów instrukcji w edytorze?

Po napisaniu instrukcji `if` i wstawieniu spacji edytor dopisze za nas część kodu i wyznaczy miejsca, gdzie możemy wstawić warunek i instrukcję. Mnie to doprowadza do szewskiej pasji, bo nie mogę spokojnie pisać dalej kodu, który mam w głowie. Wydaje mi się, że dla Czytelnika, który uczy się C++, takie wyręczenie pamięci na tym etapie też nie jest dobre. Proponuję zatem to cudo wyłączyć. W tym celu z menu *Tools* wybieramy pozycję *Options…*. W oknie *Options* (rysunek 3.2) przechodzimy na zakładkę *Editor Options/Code Insight* i usuwamy zaznaczenie przy opcji *Code Template Completion*. Następnie naciskamy *OK* i wracamy do edytora, w którym możemy już spokojnie wpisywać kod i nic nam w tym nie przeszkadza.



Rysunek 3.2. *Bardzo lubię Code Insight i jego podpowiadanie argumentów metod oraz elementów obiektów, ale szablony instrukcji C++ mnie drażnią*

O błędach w kodzie i części else instrukcji warunkowej

Jeżeli delta nie jest ujemna, to możemy przejść do obliczania rozwiązań. Może to być jednak zrobione tylko i wyłącznie gdy delta jest dodatnia lub równa zero. Nie możemy więc poleceń umieścić za poleceniem `if` (tj. za klamrą zamykającą zbiór poleceń wykonywanych w razie spełnienia warunku instrukcji `if`), bo wówczas wykonane zostałyby bez względu na spełnienie warunku. Umieścimy je wobec tego w sekcji `else`, którą można dodać do instrukcji warunkowej. Zawiera ona polecenia wykonywane w przypadku, gdy warunek określony w instrukcji `if` nie zostanie spełniony. Pełna składnia instrukcji warunkowej jest bowiem następująca:

```
if (warunek) instrukcja_gdy_prawda; else instrukcja_gdy_fałsz;
```

Jeżeli delta nie jest ujemna, to zabierzemy się za obliczanie rozwiązań równania, które zapiszemy do zmiennych `x1` i `x2` także typu `double`. Zastanówmy się nad wyrażeniem obliczającym wartość pierwszego pierwiastka. Czy możemy zapisać je w następujący sposób?

```
x1=-b-Sqrt(Delta)/2*a; //UWAGA!
```

Nie! W tej linii kryją się dwa błędy logiczne. Na słowa „błędy logiczne" Czytelnik powinien poczuć dreszcz obrzydzenia i odrazy. Są to bowiem błędy w wyrażeniach, które są zupełnie poprawne z punktu widzenia składni i kompilator bez protestu je skompiluje, tyle że nie robią tego, czego się po nich spodziewamy. Na przykład powyższe wyrażenie nie oblicza poprawnie pierwiastka równania kwadratowego. Oba błędy w powyższym wyrażeniu wynikają z kolejności działań (por. tabela 3.2). W tej chwili powyższe polecenie jest równoznaczne z:

```
x1=-b-((Sqrt(Delta)/2)*a); //UWAGA!
```

A to oznacza to, że od $-b$ odejmowany jest iloczyn połowy pierwiastka z delty i zmiennej `a`:

$$-b-\frac{\sqrt{\Delta}}{2}a$$

czego oczywiście nie chcemy. Zwróćmy w szczególności uwagę na bardzo często pomyłkę. W wyrażeniu `1/2*a`, zmienna `a` jest mnożona przez `1/2`, a nie `1/(2*a)`. Mnożenie ma ten sam priorytet co dzielenie, dlatego wykonywane są po prostu po kolei.

Nie ma wyboru. Musimy do wyrażenia obliczającego rozwiązanie równania dołożyć nawiasy mówiące kompilatorowi, w jakiej kolejności ma wykonywać działania. Uzupełnijmy polecenie obliczające x_1 w następujący sposób:

```
x1=(-b-Sqrt(Delta))/(2*a);
```

Zamiast drugiego nawiasu możemy również zastosować drugi operator dzielenia, tj.

```
x1=(-b-Sqrt(Delta))/2/a;
```

Zamiast podsumowania tych rozważań dotyczących błędów logicznych, objawię Czytelnikowi mądrość rozpowszechnioną wśród programistów, a sformułowaną, jak wszystkie ważne rzeczy w informatyce, w jednym z praw Murphy'ego: programy nie robią tego, co chcemy, a jedynie to, co my zawarliśmy w ich kodzie.

Druga postać tego samego stwierdzenia jest następująca: jeżeli program nie działa prawidłowo, winny jest programista. Jeżeli do tego dołożymy inną mądrość: bez względu na ilość czasu poświęconą na szukanie błędu, zawsze jakiś w kodzie pozostanie, możemy sformułować wniosek, który ze względu na szacowny charakter tej publikacji wyrażę w sposób łagodny: bez względu na swoje starania, programista jest na przegranej pozycji.

Wróćmy jednak do naszej instrukcji warunkowej i jej części `else`, w której chcemy obliczyć pierwiastki równania. Należy ją uzupełnić o następujące instrukcje:

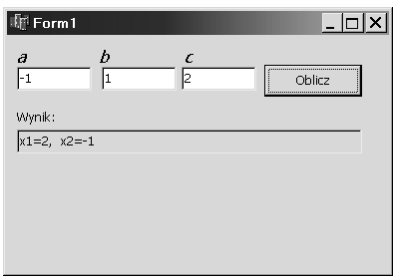
Listing 3.8. *Obliczanie pierwiastków równania kwadratowego*

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  double a=StrToFloat(Edit1->Text);
  double b=StrToFloat(Edit2->Text);
  double c=StrToFloat(Edit3->Text);
  double Delta=b*b-4*a*c;
  if (Delta<0)
  {
        Edit4->Text="Brak rozwiązań (delta mniejsza od zera)";
  }
  else
  {
        double x1=(-b-Sqrt(Delta))/(2*a);
        double x2=(-b+Sqrt(Delta))/(2*a);
        Edit4->Text="x1="+FloatToStr(x1)+",  x2="+FloatToStr(x2);
  }
}
```

Zauważmy, że w instrukcji przypisującej łańcuch do własności `Edit1->Text` użyty został operator `+` do połączenia łańcuchów.

Ponieważ instrukcji wykonywanych w przypadku nieujemnej wartości delty jest więcej, to musimy je umieścić w bloku instrukcji ujętych w nawiasy klamrowe.

No i proszę. Przygotowaliśmy dość złożony program (rysunek 3.3), w którym po raz pierwszy silnik aplikacji jest większy od jednej linii kodu. Brawo!



Rysunek 3.3. *To jest dobry moment, żeby zdecydować, czy Czytelnik polubi programowanie i czy warto inwestować czas i nerwy w jego studiowanie*

Słowo kluczowe return

Rozwiązaniem alternatywnym względem korzystania z części `else` instrukcji warunkowej byłoby wcześniejsze opuszczenie metody. Służy do tego słowo kluczowe C++ `return`. Powoduje ono natychmiastowe opuszczenie metody. Jeżeli umieścimy je w instrukcji warunkowej, to w przypadku jej spełnienia, kod metody znajdujący się za tą instrukcją nie będzie wykonany. Metoda `Button1Click` wyglądałaby wówczas tak jak na listingu 3.9. Ja osobiście jednak nie lubię tego rozwiązania. Gdy je stosowałem, często miałem problemy z rozbudową kodu i w efekcie w końcu i tak zmieniałem je na konstrukcję `if..else`.

Listing 3.9. *Z pozoru kod może wydawać się prostszy, ale moim zdaniem za bardzo przypomina użycie instrukcji goto*

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  double a=StrToFloat(Edit1->Text);
  double b=StrToFloat(Edit2->Text);
  double c=StrToFloat(Edit3->Text);
  double Delta=b*b-4*a*c;
  if (Delta<0)
  {
        Edit4->Text="Brak rozwiązań (delta mniejsza od zera)";
        return;
  }
  double x1=(-b-Sqrt(Delta))/(2*a);
  double x2=(-b+Sqrt(Delta))/(2*a);
  Edit4->Text="x1="+FloatToStr(x1)+",  x2="+FloatToStr(x2);
}
```

W przypadku funkcji i metod zwracających wartość, za słowem kluczowym `return` powinna znaleźć się stała, zmienna lub wyrażenie, którego wartość ma być zwrócona. Oto przykład:

```
int sqr(int argument)
{
  return argument*argument;
}
```

Na tym nie koniec

Jeżeli w powyższym programie do pól edycyjnych wpiszemy liczby $a = 1$, $b = -2$, $c = 1$, to w wyniku uzyskamy dwa identyczne rozwiązania równe 1. Delta $\Delta = (-2)^2 - 4 \cdot 1 \cdot 1 = 4 - 4$ jest bowiem w takim przypadku równa zero i rozwiązanie jest pierwiastkiem podwójnym o wartości $-b/2a$. Przedstawienie wyniku w sposób widoczny na rysunku 3.3 jest wówczas oczywiście także poprawne, użytkownik aplikacji dostaje bowiem prawidłową wartość rozwiązania, choć powtórzoną dwa razy, ale byłoby chyba bardziej elegancko, gdyby program podawał wówczas jedną liczbę i informował o tym, że równanie ma rozwiązanie będące pierwiastkiem podwójnym. Do tego zmierzać będą następne zmiany w kodzie metody `Button1Click`.

Typy całkowite C++

Zadeklarujmy w metodzie `Button1Click` zmienną `IloscPierwiastkow` typu `byte` (listing 3.10):

Listing 3.10. *Deklaracja liczby całkowitej*

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  double a=StrToFloat(Edit1->Text);
```

```
double b=StrToFloat(Edit2->Text);
double c=StrToFloat(Edit3->Text);
double Delta=b*b-4*a*c;
byte IloscPierwiastkow;
...
```

Typ `byte` jest 8-bitową reprezentacją liczby całkowitej bez znaku. Oznacza to, że w tego typu zmiennych można przechowywać liczby o wartościach od 0 do $2^8 - 1 = 255$. To oczywiście o wiele za dużo jak na nasze potrzeby, ale osiem bitów, czyli jeden bajt, jest najmniejszym rozmiarem zmiennej w C++.

Typ `byte` nie jest w zasadzie typem wbudowanym C++, a jedynie „aliasem" do `unsigned char`. Typ `char` jest jednobajtowym typem liczb całkowitych, który używany jest zazwyczaj do kodowania znaków ASCII. Stąd bierze się jego nazwa (char od ang. *character* oznaczającego znak). Modyfikator `unsigned` oznacza, że żaden z bitów tej liczby nie koduje znaku, a więc że wszystkie dopuszczalne wartości są dodatnie. W przypadku typu `signed char` możliwe wartości należałyby do zakresu od –128 do 127. Inne typy całkowite przedstawione zostały w tabeli 3.4.

Tabela 3.4. *Typy całkowite w C++Builder 2006*

Nazwa typu	Obecność znaku	Zakres (najmniejsza i największa wartość liczby)	Liczba bajtów (bitów) zajmowana przez zmienną	Postać literału
`unsigned char` (lub `byte`)	nie	0 .. 255	1 (8)	
`signed char`	tak	–128 .. 127	1 (8)	
`unsigned short`	nie	0 .. 65535	2 (16)	
`short`	tak	–32768 .. 32767	2 (16)	
`unsigned int`, `unsigned long`	nie	0 .. 4294967295	4 (32)	1U,1UL
`int`, `long`	tak	–2147483648 .. 2147483647	4 (32)	1, 1L
`long long`, `__int64`	tak	-2^{63} .. $2^{63}-1$	8 (64)	

Obliczmy ilość pierwiastków równania. Zależy ona tylko od wartości zmiennej `Delta`. Jeżeli jest dodatnia, liczba pierwiastków równa jest dwa, jeżeli `Delta` równa jest 0, to pierwiastek jest jeden, a jeżeli ujemna — zero. Do kodu metody wstawiamy zatem instrukcje `if` widoczne na listingu 3.11.

Listing 3.11. *Sprawdzanie ilości rozwiązań równania*

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  double a=StrToFloat(Edit1->Text);
  double b=StrToFloat(Edit2->Text);
  double c=StrToFloat(Edit3->Text);
  double Delta=b*b-4*a*c;
  byte IloscPierwiastkow=0;
  if (Delta>0) IloscPierwiastkow=2;
  if (Delta==0) IloscPierwiastkow=1;
```

```
  if (Delta<0)
  {
        Edit4->Text="Brak rozwiązań (delta mniejsza od zera)";
  }
  else
  {
        double x1=(-b-Sqrt(Delta))/(2*a);
        double x2=(-b+Sqrt(Delta))/(2*a);
        Edit4->Text="x1="+FloatToStr(x1)+",  x2="+FloatToStr(x2);
  }
}
```

Instrukcja wielokrotnego wyboru switch

Mając liczbę pierwiastków, możemy zreorganizować sposób wyświetlania wyników. Wykorzystajmy do tego instrukcję `switch`. O ile w instrukcji warunkowej `if..else` można określić jedynie dwa rodzaje reakcji: wykonywaną, gdy jej warunek jest spełniony, oraz wykonywany w przeciwnym przypadku, to w **instrukcji wielokrotnego wyboru** `switch` można określić dowolną ilość operacji wykonywanych w zależności od wartości liczby całkowitej.

Najlepiej nauczyć się instrukcji `switch` na przykładzie. Poniższy listing zawiera metodę `Button1Click`, w której zmieniony został sposób przedstawiania wyników tak, że wykorzystywana jest do tego instrukcja wielokrotnego wyboru (listing 3.12).

Listing 3.12. *Przykład użycia instrukcji wielokrotnego wyboru*

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  double a=StrToFloat(Edit1->Text);
  double b=StrToFloat(Edit2->Text);
  double c=StrToFloat(Edit3->Text);
  double Delta=b*b-4*a*c;
  byte IloscPierwiastkow=0;
  if (Delta>0) IloscPierwiastkow=2;
  if (Delta==0) IloscPierwiastkow=1;
  double x1,x2;
  switch (IloscPierwiastkow)
  {
        case 0:
                Edit4->Text="Brak rozwiązań (delta mniejsza od zera)";
                break;
        case 1:
                x1=-b/(2*a);
                x2=x1;
                Edit4->Text="Pierwiastek podwójny:  x="+FloatToStr(x1);
                break;
        case 2:
                x1=(-b-Sqrt(Delta))/(2*a);
                x2=(-b+Sqrt(Delta))/(2*a);
                Edit4->Text="x1="+FloatToStr(x1)+",  x2="+FloatToStr(x2);
                return;
```

```
        default:
                Edit4->Text="Błąd!";
                break;
  }
}
```

Cała konstrukcja rozpoczyna się od linii `switch (IloscPierwiastkow)`. Wskazujemy w ten sposób zmienną (koniecznie typu całkowitego lub wyliczeniowego), która będzie analizowana. W naszym przypadku jest to `IloscPierwiastkow`. Następnie wymienione są poszczególne przypadki, na które chcemy zareagować. Każdy z nich rozpoczyna się słowem kluczowym `case`, a powinien zakończyć słowem kluczowym `break`. To drugie nie jest jednak w C++ obowiązkowe. Gdy go zabraknie, po wykonaniu poleceń z odpowiedniej sekcji `case` wykonywana jest następna. Takie przekazanie sterowania do kolejnych przypadków zazwyczaj nie jest sytuacją zamierzoną — brak `break` jest jednym z częstszych błędów.

Po słowie kluczowym `case` znajdować się powinna wartość zmiennej `IloscPierwiastkow`, która identyfikuje nasz przypadek, a po niej dwukropek. Potem aż do `break` mogą znajdować się dowolne instrukcje.

Za listą przypadków może być umieszczone słowo kluczowe `default`, a po nim instrukcja wykonywana, gdy wartość zmiennej `IloscPierwiastkow` nie jest wymieniona w liście przypadków. Może być ona pominięta, co oznacza, że rezygnujemy z określenia czynności wykonywanych w takiej sytuacji.

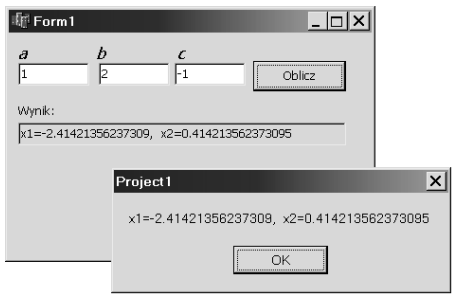
Ten sam efekt można oczywiście uzyskać, stosując instrukcje `if`, czy to w serii, czy zagnieżdżone, ale każdy chyba przyzna, że to rozwiązanie wygląda bardziej przejrzyście. Elegancję rozwiązania w naszym przypadku zmniejsza to, że i tak musimy obliczyć wartość zmiennej `IloscPierwiastkow`, korzystając z instrukcji `if`.

Funkcja ShowMessage

Wynik przedstawiony został w polu edycyjnym. A może by tak rzucić nim jeszcze w oczy użytkownika? Możemy go na przykład wyświetlić w osobnym oknie. Możemy i zróbmy to. Pomoże nam w tym poznana w pierwszym rozdziale funkcja `ShowMessage`. Jej jedynym argumentem jest łańcuch, który ma być pokazany w oknie. My wyświetlimy po prostu zawartość pola edycyjnego, do którego zapisaliśmy wynik (listing 3.13, rysunek 3.4).

Listing 3.13. *ShowMessage wygrałaby pewnie ranking na najczęściej używaną funkcję VCL*

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  double a=StrToFloat(Edit1->Text);
  double b=StrToFloat(Edit2->Text);
  double c=StrToFloat(Edit3->Text);
  double Delta=b*b-4*a*c;
  byte IloscPierwiastkow=0;
  if (Delta>0) IloscPierwiastkow=2;
```



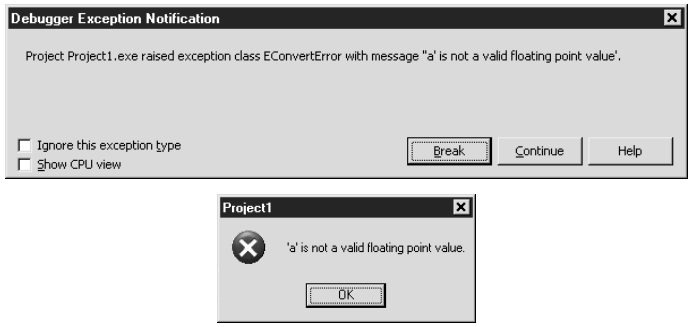
Rysunek 3.4. *Działanie funkcji ShowMessage*

```
if (Delta==0) IloscPierwiastkow=1;
  double x1,x2;
  switch (IloscPierwiastkow)
  {
        case 0:
              Edit4->Text="Brak rozwiązań (delta mniejsza od zera)";
              break;
        case 1:
              x1=-b/(2*a);
              x2=x1;
              Edit4->Text="Pierwiastek podwójny:  x="+FloatToStr(x1);
              break;
        case 2:
              x1=(-b-Sqrt(Delta))/(2*a);
              x2=(-b+Sqrt(Delta))/(2*a);
              Edit4->Text="x1="+FloatToStr(x1)+",  x2="+FloatToStr(x2);
              return;
        default:
              Edit4->Text="Błąd!";
              break;
  }
  ShowMessage(Edit4->Text);
}
```

Obsługa wyjątków

Najsłabszym punktem naszej aplikacji jest sposób wprowadzania do niej danych. To jest zresztą typowe miejsce, gdzie mogą się pojawić błędy, bowiem nikt nic nie jest tak nieprzewidywalne w programie, jak pomysły jego użytkownika. Wystarczy, że użytkownik pomyli się i zamiast cyfry wprowadzi jakąś literę[6]. Wówczas funkcja konwertująca łańcuch na liczbę `StrToFloat` **zgłosi wyjątek**. Jeżeli program uruchamiany jest w środowisku debugera Borland Developer Studio, to pojawi się wówczas komunikat o wystąpieniu wyjątku, który widzimy na rysunku 3.5 (górny). W przypadku aplikacji uruchamianej poza BDS lub bez użycia debugera (co można uzyskać, naciskając *Ctrl+Shift+F9*), okno komunikatu jest nieco prostsze (rysunek 3.5, dolny).

[6] Poza *E* w odpowiednim miejscu, która może służyć do zapisu liczby zmiennoprzecinkowej.



Rysunek 3.5. *Informacje o wyjątkach zgłoszonych przez aplikację*

Czym są i do czego służą wyjątki?

Wyjątki są obiektami, które są tworzone w przypadku wystąpienia błędów. Obiekty te służą do przesyłania informacji o wystąpieniu błędu i o jego charakterze. Informuje o tym zarówno sam typ wyjątku, który może mniej lub bardziej jednoznacznie identyfikować błąd, np. `EDivByZero` (dzielenie przez zero), `EDirectoryError` (problem dotyczący katalogu na dysku) lub `EConvertError` (błąd przy konwersji zmiennych), jak i komunikat umieszczony we własności `Message` każdego wyjątku. Klasą bazową wszystkich wyjątków jest `Exception`[7]. Jeżeli nie chcemy tworzyć własnego typu wyjątku, to należy użyć tej właśnie klasy.

Przechwytywanie wyjątków

Wyjątek zgłoszony przez funkcję `StrToFloat` może być **przechwycony** i obsłużony. Dzięki możliwości przechwycenia zgłoszenie wyjątków nie musi prowadzić do katastrofy, a program ma możliwość, żeby skorygować dane lub w inny sposób zareagować na błąd. Jednak jeżeli aplikacja uruchamiana jest w środowisku BDS przy włączonym trybie debugowania, to komunikat widoczny na rysunku 3.5 (górny) pojawi się mimo wszystko. To ułatwia naprawianie programu na etapie jego projektowania.

Otoczmy krytyczną część naszej metody konstrukcją przechwytywania wyjątków. W tym celu musimy użyć konstrukcji `try..catch` widocznej na poniższym listingu:

Listing 3.14. *Dodajemy do naszej metody obsługę wyjątków*

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  try
  {
        double a=StrToFloat(Edit1->Text);
        double b=StrToFloat(Edit2->Text);
        double c=StrToFloat(Edit3->Text);
        double Delta=b*b-4*a*c;
        byte IloscPierwiastkow=0;
        if (Delta>0) IloscPierwiastkow=2;
```

[7] Klasy wyjątków rozpoczynają się nie od litery *T*, jak zwykłe klasy w VCL, ale od *E*.

```
        if (Delta==0) IloscPierwiastkow=1;
        double x1,x2;
        switch (IloscPierwiastkow)
        {
              case 0:
                    Edit4->Text="Brak rozwiązań (delta mniejsza od zera)";
                    break;
              case 1:
                    x1=-b/(2*a);
                    x2=x1;
                    Edit4->Text="Pierwiastek podwójny:  x="+FloatToStr(x1);
                    break;
              case 2:
                    x1=(-b-Sqrt(Delta))/(2*a);
                    x2=(-b+Sqrt(Delta))/(2*a);
                    Edit4->Text="x1="+FloatToStr(x1)+",  x2="+FloatToStr(x2);
                    break;
              default:
                    Edit4->Text="Błąd!";
                    break;
        }
        ShowMessage(Edit4->Text);
    }
    catch(EConvertError& exc)
    {
          Edit4->Clear();
          ShowMessage("Błąd konwersji współczynników równania!\nKomunikat wyjątku:
          "+exc.Message);
          return;
    }
    catch(...)
    {
          Edit4->Clear();
          ShowMessage("Wystąpił nierozpoznany typ błędu!");
    }
}
```

W argumencie funkcji `ShowMessage` wyświetlającej komunikat o błędzie użyte zostało wyrażenie `\n`. W C++ oznacza to znak o kodzie ASCII numer 13. Pod tym kodem kryje się znak końca linii. W efekcie komunikat wyświetlany przez `ShowMessage` będzie wyświetlany w dwóch liniach.

Zwróćmy uwagę, że obiekt przekazujący informacje o wyjątku „odbierany" jest w sekcji `catch` przez referencję[8]. Jest to najwłaściwszy sposób odbierania obiektu wyjątku.

W części za słowem kluczowym `try` powinny być wszystkie polecenia, co do których mamy obawy, że mogą doprowadzić do zgłoszenia wyjątku, oraz wszystkie te polecenia, które są od nich w bezpośredni sposób zależne. Należy pamiętać, że w razie wystąpienia błędu w sekcji `try` (np. przy konwersji zawartości pola edycyjnego `Edit1`)

[8] Referencje zostaną omówione w następnym rozdziale.

wątek aplikacji przenosi się natychmiast do sekcji `catch` i już z niej nie powraca. Po błędzie w `Edit1` nie będzie wobec tego możliwości, aby spróbować konwersji łańcuchów z kolejnych pól edycyjnych. Po obsłudze wyjątku wykonywane są polecenia znajdujące się bezpośrednio po klamrze zamykającej ostatnią z sekcji `catch`. Czasem warto zatem rozważyć utworzenie oddzielnej konstrukcji `try..catch` dla każdej niebezpiecznej operacji.

Jak widzimy na listingu 3.14, sekcji `catch` może być więcej — przypomina to nieco instrukcję wielokrotnego wyboru. Każda z nich może służyć do przechwytywania innych klas wyjątków, a więc do obsługi innych typów błędów. Należy jednak zwrócić uwagę, aby klasy wyjątków wymieniane były od najbardziej szczegółowej („najbardziej potomnej") do najbardziej ogólnej („najbardziej bazowej")[9]. W ostatniej (względnie jedynej) sekcji `catch` zamiast klasy wyjątku można użyć wielokropka. Wówczas sekcja ta przechwytuje wszystkie możliwe wyjątki, także nieprzekazywane za pomocą obiektów dziedziczących z `Exception` (można przesłać na przykład dowolną zmienną lub stałą). W praktyce oznacza to jednak, że w tej sekcji nieznany jest typ błędu ani związany z nim komunikat.

Zgłaszanie wyjątków

O samodzielnym zgłaszaniu wyjątków chciałbym tylko wspomnieć. Zresztą, o ile nie tworzymy własnych klas wyjątków, to nie ma zbyt wiele do powiedzenia na ten temat. Do zgłaszania wyjątków służy słowo kluczowe `throw`. Za nim powinien znaleźć się obiekt wyjątku. Przykładowa instrukcja zgłaszająca wyjątek widoczna jest w listingu 3.15.

Listing 3.15. *Zgłaszanie wyjątków w przypadku wykrycia błędu w naszym algorytmie*

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  try
  {
        double a=StrToFloat(Edit1->Text);
        double b=StrToFloat(Edit2->Text);
        double c=StrToFloat(Edit3->Text);
        double Delta=b*b-4*a*c;
        byte IloscPierwiastkow=0;
        if (Delta>0) IloscPierwiastkow=2;
        if (Delta==0) IloscPierwiastkow=1;
        double x1,x2;
        switch (IloscPierwiastkow)
        {
case 0:
                    Edit4->Text="Brak rozwiązań (delta mniejsza od zera)";
                    break;
              case 1:
                    x1=-b/(2*a);
                    x2=x1;
                    Edit4->Text="Pierwiastek podwójny:  x="+FloatToStr(x1);
                    break;
```

[9] Jak wspomniałem, klasą bazową wyjątków w bibliotece VCL jest klasa `Exception`.

```
                case 2:
                        x1=(-b-Sqrt(Delta))/(2*a);
                        x2=(-b+Sqrt(Delta))/(2*a);
                        Edit4->Text="x1="+FloatToStr(x1)+",  x2="+FloatToStr(x2);
                        break;
                default:
                        //Edit4->Text="Błąd!";
                        throw Exception("Niemożliwa do określenia ilość rozwiązań
                        równania");
                        break;
        }
        ShowMessage(Edit4->Text);
    }
    catch(EConvertError& exc)
    {
        Edit4->Clear();
        ShowMessage("Błąd konwersji współczynników równania!\nKomunikat wyjątku:
        "+exc.Message);
        return;
    }
    catch(...)
    {
        Edit4->Clear();
        ShowMessage("Wystąpił nierozpoznany typ błędu!");
    }
}
```

Pętle

Do tej pory omawialiśmy instrukcje, które pozwalały na wybór, już w trakcie działania programu, jednej ze ścieżek kodu, która może być wykonywana w zależności od stanu programu (aktualnej wartości zmiennych). Teraz zajmiemy się drugim zasadniczym typem instrukcji obecnym, tak jak powyższe, we wszystkich językach programowania, a mianowicie pętlami. **Pętle** służą do wielokrotnego wykonywania sekwencji instrukcji. Nie musi to wcale oznaczać prostego powtarzania. Każda **iteracja** może być inna, ponieważ różną wartość może mieć indeks pętli.

Pętla for

Zacznijmy od najczęściej używanego typu pętli, a mianowicie od pętli `for`. Stosuje się go wtedy, gdy z góry znana jest ilość iteracji, jaka ma być wykonana. Jej konstrukcja jest następująca

```
for(inicjacja_indeksu;warunek_kontynuowania;zmiana_indeksu) instrukcja;
```

Zazwyczaj pętla ma postać podobną do poniższej:

```
for(int i=0;i<ilość_iteracji;i++) instrukcja;
```

Zmienna `i` pełni tu rolę indeksu pętli. Jeżeli jest zadeklarowana jak w powyższym przykładzie, to jej zakres obejmuje jedynie samą pętlę, a więc może być użyta w warunku kontynuowania pętli, w poleceniu zmieniającym wartość indeksu i w instrukcji lub grupie instrukcji wykonywanych w pętli. Polecenie zwiększające indeks wykorzystuje poznany wcześniej operator ++. W tym przypadku nie ma praktycznego znaczenia, czy jest on umieszczony przed, czy za zmienną indeksu.

Oto przykład konkretnej pętli umieszczonej w domyślnej metodzie zdarzeniowej przycisku w nowym projekcie (listing 3.16):

Listing 3.16. *Klimaty ZX Spectrum (wyczuwalne tylko dla osób po trzydziestce)*

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  for(int i=1;i<=10;i++) Beep(100*i,100);
}
```

Dziesięć razy wywołana zostanie funkcja `Beep`, przez co dziesięć razy wyemitowany zostanie dźwięk o długości 100 milisekund (drugi argument) i częstotliwości od 100 do 1000 herców (pierwszy argument).

Indeks pętli `for` może być nie tylko zwiększany, ale również zmniejszany. Służy do tego operator `--`. Oto prosty przykład:

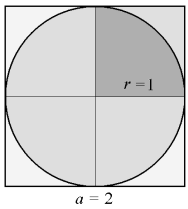
Listing 3.17. *Opadanie w Jet Set Willy*

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  for(int i=10;i>0;i--) Beep(100*i,100);
}
```

Pętla for w praktyce, czyli tajemnica pitagorejczyków

Nieco ambitniejszym przykładem zastosowaniem pętli `for` może być wykonywanie jakichś obliczeń. Załóżmy na przykład, że chcemy symulować rzucanie ziaren piasku na kwadratowy stół. Jeżeli na tym stole wykreślimy ćwierć okręgu o promieniu równym krawędzi stołu, to możemy zastanawiać się, jak wiele ziaren spadnie wewnątrz tej ćwiartki okręgu, a jak wiele poza nią (rysunek 3.6). Dla uproszczenia przyjmijmy, że promień okręgu, a więc krawędź stołu, równy jest jednemu metrowi. Parametr, który będzie nas interesował w szczególności, to pomnożony przez cztery stosunek ilości ziaren z okręgu do wszystkich zrzuconych ziaren, pod warunkiem, że zrzucanie ziaren odbywało się w sposób zupełnie przypadkowy.

Rysunek 3.6. *Część ciemniejsza to stół. Pozostałe trzy ćwiartki mają tylko pomóc wyobraźni*



Można oczywiście podobne doświadczenie wykonać w domu. Nie trzeba nawet liczyć ziaren, wystarczy je zważyć. Większym problemem będzie jednak zapewnienie całkowitej przypadkowości miejsca, na które upuszczone zostanie każde ziarenko, i wyczyszczenie dywanu po takim eksperymencie. Szczególnie to ostatnie może do tej inicjatywy skutecznie zniechęcić. I to jest właśnie typowe zastosowanie dla komputerów. Dlaczego by nie napisać programu, który takie doświadczenie zasymuluje. Można, i to właśnie zrobimy. A wykorzystamy do tego pętlę `for`. W tej pętli będziemy losowali dwie liczby z przedziału od 0 do 1. Będą one pełniły rolę współrzędnych upuszczonego na stół ziarenka mierzonych od lewego dolnego rogu stołu (środka okręgu). Do losowania współrzędnych wykorzystamy funkcję `Random`, która zwraca liczbę z tego przedziału. Przedtem generator liczb pseudolosowych należy zainicjować funkcją `Randomize`, bez tego po każdym uruchomieniu programu będziemy otrzymywali takie same liczby.

Poza `Random` mamy również do dyspozycji funkcję `RandG`, która generuje liczby losowe zgodnie z rozkładem standardowym (Gaussa).

Stwórzmy osobny projekt *VCL Forms Application — C++Builder*. Po pojawieniu się podglądu formy umieśćmy na niej przycisk `TButton` z palety *Standard* i dwukrotnie go klikając, stwórzmy domyślną metodę zdarzeniową. Do tej metody wpisujemy instrukcje z poniższego listingu:

Listing 3.18. *Taki typ obliczeń, w których losuje się dane wejściowe, nazywa się symulacjami Monte Carlo*

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  const unsigned int N = 10000000;
  Randomize();
  unsigned int trafienia=0;
  for(int i=0;i<N;i++)
  {
        double x=Random();
        double y=Random();
        if(x*x+y*y<1) trafienia++;
  }
  double wynik=4*trafienia/N;
  ShowMessage("Wynik: "+FloatToStr(wynik));
}
```

Ilość zer, jaką wpiszemy w definicji stałej `N` określającej ilość iteracji pętli `for`, zależy jedynie od szybkości procesora, jakim dysponuje komputer, oraz cierpliwości, jaką dysponuje Czytelnik. Nie może to być jednak liczba większa od 4294967295, czyli od zakresu zmiennej typu `unsigned int` użytej jako indeks pętli.

Przyjrzyjmy się powyższej metodzie. Pętla `for` wykonywana jest dziesięć milionów razy. Za każdym razem losowane są dwie liczby, które odpowiadają współrzędnym ziarenka rzuconego na stół. Następnie sprawdzamy, czy ziarenko upadło na ćwiartkę okręgu, czy poza nią. W tym celu sprawdzamy jego odległość od lewego dolnego rogu, a więc od naszego środka układu współrzędnych. Aby nie trudzić procesora niepotrzebnym, a pracochłonnym pierwiastkowaniem, obliczmy nie samą odległość, a jej kwadrat x^2+y^2. Jeżeli jej wartość jest mniejsza od kwadratu promienia okręgu, a więc

mniejsza od jedności, to zaliczamy trafienie i zwiększamy zmienną zliczającą trafienie o jeden. Do tego wykorzystujemy operator ++, który wcześniej wykorzystywaliśmy w pętli `for`. Po wykonaniu pętli obliczamy stosunek ilości trafień do ilości wszystkich ziarenek, mnożymy go przez cztery i pokazujemy wynik użytkownikowi aplikacji.

Dzielenie liczb naturalnych

No to sprawdźmy, co nam wyjdzie. Uruchamiamy program, klikamy przycisk i… otrzymujemy wynik równy 3! Łatwo się domyślić, że mało prawdopodobne jest, aby opisana przeze mnie procedura eksperymentu dawała tak prosty wynik. Jest on zatem prawdopodobnie skutkiem błędu. Niestety, sytuacje, kiedy program działa bezbłędnie zaraz po napisaniu, należą do rzadkości. Jednak w ogromnej większości przypadków błędy są na tyle proste, że z pomocą debugera można je bez trudu znaleźć w ciągu kilku chwil. Nigdy jednak nie ma pewności, że błędów nie jest więcej, tzn. że znaleźliśmy ostatni. Ale to już jest przekleństwo programistów. Z jakim błędem mamy do czynienia tym razem? Bardzo prostym, ale bardzo łatwym do przeoczenia. Przyjrzymy się poleceniu obliczającemu wartość zmiennej `wynik`. Zauważmy, że obliczany jest z wartości całkowitej `4` oraz zmiennych całkowitych `trafienia` i `N`. Ponieważ operatory arytmetyczne wykonywane są od lewej do prawej, to najpierw wykonywane jest mnożenie, w wyniku tego otrzymujemy wartość całkowitą, którą następnie dzielimy przez liczbę prób `N`. I to jest właśnie źródło błędu. Dzielimy liczbę całkowitą przez całkowitą, co oznacza, że w wyniku otrzymujemy liczbę całkowitą zaokrąglaną w dół, a nie liczbę rzeczywistą, jaką powinniśmy otrzymać. Dlatego wynik równy jest dokładnie 3.

Jak temu zaradzić? Musimy doprowadzić do tego, że lewy lub prawy argument operatora dzielenia `/` będzie typu `double`. Możemy to osiągnąć, rzutując jedną ze zmiennych na typ `double` lub zmieniając `4` na `4.0`, a więc:

```
double wynik=4.0*trafienia/N;
```

lub

```
double wynik=4*trafienia/(double)N;
```

Uzyskany wynik był dwadzieścia parę wieków temu równie wielką sensacją, jak dziś afera wokół kodu Leonarda Da Vinci. Dziś nie budzi już niestety takich emocji. Uzyskamy bowiem stałą Archimedesa, nazywaną także ludolfiną lub liczbą π. Cała sensacja bierze się z faktu, że jest to liczba niewymierna (podobnie jak pierwiastek z dwóch), a to oznacza, że nie można jej wyrazić ani liczbą całkowitą, ani ułamkiem zbudowanym z takich liczb, ani liczbą rzeczywistą ze skończoną liczbą cyfr. I choć przybliżenia w stylu 22/7 (oszacowanie Archimedesa) lub 355/113 (Metius, XVI wiek) są niezłym oszacowaniem jej wartości, to nadal są to tylko przybliżenia, w których poprawne jest tylko kilku pierwszych liczb znaczących. Doświadczenie, które symulowaliśmy, zaproponowane zostało w drugiej połowie dziewiętnastego wieku przez szwajcarskiego astronoma Rudolfa Wolfa i w fachowej literaturze znane jest jako metoda Monte Carlo. Niestety, jest jednym z najmniej wydajnych sposobów obliczania liczby π. Natomiast posłużyło nam doskonale do ilustracji pętli `for`.

Prawdziwą wartość liczby π można uzyskać dzięki stałej `M_PI` z pliku nagłówkowego *Math.h*. Zwraca ona wartość zaokrągloną do 3.14159265358979. Dzięki temu możemy sprawdzić dokładność powyższych obliczeń, zmieniając ostatnią linię metody na:

```
ShowMessage("Wynik: "+FloatToStr(wynik)+"\nDokładność: "+FloatToStr(fabs(M_PI-wynik)));
```

Pętla do..while

Równie często wykorzystywany jest inny rodzaj pętli, a mianowicie `do..while`. Oto jej składnia:

```
do
{
  instrukcje
} while (warunek);
```

Jest ona wykorzystywana najczęściej wówczas, gdy ilość iteracji nie jest z góry określona, ale umiemy sformułować warunek, który ma przerwać działanie pętli (np. czytaj plik linię po linii aż do jego końca). Oto przykład, w którym dzielimy liczbę 1 przez 2 tak długo, aż wynik będzie mniejszy od jednej milionowej (listing 3.19):

Listing 3.19. *Prosty przykład pętli do..while*

```
void __fastcall TForm1::Button2Click(TObject *Sender)
{
  double d=1;
  do d/=2; while (!(d<1E-6));
  ShowMessage("d="+FloatToStr(d));
}
```

Aby policzyć, ile iteracji zostało wykonanych, możemy wprowadzić coś na kształt indeksu pętli. Wystarczy zadeklarować zmienną typu naturalnego i zwiększać jej wartość przy każdej iteracji. Pokazuje to kolejny listing:

Listing 3.20. *Wprowadzanie indeksu do pętli do..while*

```
void __fastcall TForm1::Button2Click(TObject *Sender)
{
  double d=1;
  int i=0;
  do
  {
        d/=2;
        i++;
  }
  while (!(d<1E-6));
  ShowMessage("d="+FloatToStr(d)+", i="+IntToStr(i));
}
```

Podobnie jak w przypadku instrukcji `if`, należy bardzo ostrożnie formułować warunki zawierające operatory logiczne (zob. ostrzeżenie wyżej). Dla przykładu, dwa poniższe fragmenty kodu:

```
do instrukcja while (!i<20);
```

i

```
do instrukcja while (!(i<20));
```

mają zupełnie inne znaczenie, choć wydają się podobne. Warto przekonać się o tym samemu.

Pętla while

Jest jeszcze jeden rodzaj pętli, który jest podobny do `do..while`, ale różni się jednym, za to istotnym szczegółem. W pętli `do..while` wpierw wykonywana jest pierwsza iteracja, a dopiero potem sprawdzany jest warunek rozstrzygający, czy wykonana będzie następna. Natomiast w pętli `while` warunek sprawdzany jest jeszcze przed wykonaniem pierwszej iteracji. Czy to ważne? Czasami tak. Wyobraźmy sobie czytanie z pliku. Jeżeli plik jest pusty, to nie powinniśmy próbować czytać linii ani razu. W takiej sytuacji pętla `do..while` w stylu „czytaj aż dojdziesz do końca pliku" nie jest najlepszym rozwiązaniem i powinna być zastąpiona pętlą `while` w stylu „dopóki nie doszedłeś do końca pliku, czytaj kolejną linię". Styl tego zdania nie jest najlepszy, ale dokładnie oddaje sens pętli `while`. Przykład takiej pętli znajdzie Czytelnik w rozdziale 11. dotyczącym plików.

Nie należy jednak różnicy między tymi dwoma rodzajami pętli źle zrozumieć. W normalnych przypadkach — a przez normalne rozumiemy takie, w których pętla ma przynajmniej kilka iteracji — ilość iteracji w pętli `do..while` i `while` jest taka sama. Dla przykładu, pętla `while` z listingu 3.21 wykonywana jest tyle samo razy i daje ten sam wynik co pętla `do..while` z listingu 3.20.

Listing 3.21. *Pętla while odpowiadająca pętli z listingu 3.20*

```
void __fastcall TForm1::Button3Click(TObject *Sender)
{
  double d=1;
  int i=0;
  while (!(d<1E-6))
  {
         d/=2;
         i++;
  }
  ShowMessage("d="+FloatToStr(d)+", i="+IntToStr(i));
}
```

Różnica pojawi się dopiero wtedy, gdy początkową wartość zmiennej `d` zmienimy tak, żeby warunek pętli był od razu spełniony. Nadajmy jej na przykład wartość `1E-7`. W tej sytuacji kod w pętli `while` nie zostanie wykonany ani razu, ale w przypadku pętli `do..while` zostanie on wykonany i wartość zmiennej `d` zostanie zmniejszona o połowę.

Zwróćmy jeszcze uwagę, że w pętli `while` warunek opisuje sytuację, w której pętla może być kontynuowana, podczas gdy w pętli `do..while` wskazuje on na warunek przerwania pętli. Kiedy więc kod jest tłumaczony z jednej pętli do drugiej, jak w listingach 3.20 i 3.21, dostawiona musi być negacja lub w inny sposób warunek zmieniony na przeciwny.

Instrukcje break i continue

Załóżmy, że w zbiorze liczb od –10 do 10 szukamy takich, przez które można podzielić liczbę 100 bez reszty. Zadanie bardzo wydumane, ale w końcu nie chodzi o jego praktyczność, a o naukę C++. Do rozwiązania przygotujemy pętlę `for`, w której indeks będzie przebiegał liczby od –10 do 10, i za pomocą operatora `%` zwracającego resztę z dzielenia sprawdzać będziemy, przez które z nich można 100 podzielić bez reszty (listing 3.22).

Listing 3.22. *Pętla, z której będziemy musieli „wyjąć" jedną iteracje*

```
void __fastcall TForm1::Button4Click(TObject *Sender)
{
  for(int i=-10;i<=10;i++)
  {
         if (100%i==0) ShowMessage("100/("+IntToStr(i)+")="+IntToStr(100/i)
          +", bez reszty");
  }
}
```

Umieśćmy tę pętlę w metodzie zdarzeniowej przycisku i uruchommy. Pojawi się seria komunikatów informujących o znalezieniu pierwszych liczb, przez które można podzielić 100 bez reszty. Są to –10, –5, –4, –2, –1 i…. I wtedy pojawia się wyjątek `EDivByZero`. W kolejnej iteracji podejmowana jest bowiem próba dzielenia 100 przez zero. A to nie może skończyć się dobrze. Możemy oczywiście podzielić pętle na dwie od –10 do –1 i od 1 do 10. Ale to oznaczałoby powtarzanie kodu. O wiele łatwiej jest ominąć tę jedną iterację, korzystając z instrukcji `continue`. Powoduje ona przerwanie bieżącej iteracji i rozpoczęcie następnej. Wstawmy zatem do pętli instrukcję `continue` z instrukcją warunkową sprawdzającą, czy indeks pętli równy jest zero (listing 3.23).

Listing 3.23. *Instrukcja powodująca opuszczenie jednej z iteracji pętli*

```
void __fastcall TForm1::Button4Click(TObject *Sender)
{
  for(int i=-10;i<=10;i++)
  {
         if (i==0) continue;
         if (100%i==0) ShowMessage("100/("+IntToStr(i)+")="+IntToStr(100/i)+",
         bez reszty");
  }
}
```

Działanie drugiej z wymienionych w tytule instrukcji, instrukcji `break`, jest silniejsze. Powoduje całkowite opuszczenie pętli. Jeszcze raz załóżmy, że przeszukujemy zbiór liczb od –10 do 10 i szukamy liczb, przez które można podzielić 100 bez reszty. Ale

tym razem potrzebujemy tylko trzech takich liczb. Po ich znalezieniu dalsze poszukiwanie, tj. dalsze wykonywanie pętli, jest tylko stratą czasu. Wobec tego liczmy znalezione wyniki i opuśćmy pętle po znalezieniu trzeciego. Odpowiednią konstrukcję pokazuje listing 3.24.

Listing 3.24. *Przykład wykorzystania instrukcji break*

```
void __fastcall TForm1::Button4Click(TObject *Sender)
{
  int n=0;
  for(int i=-10;i<=10;i++)
  {
        if (i==0) continue;
        if (100%i==0)
        {
              n++;
              ShowMessage("100/("+IntToStr(i)+")="+IntToStr(100/i)+",
              bez reszty");
              if (n==3) break;
        }
  }
  ShowMessage("Znaleziono 3 liczby");
}
```

Podsumowanie

Poznaliśmy już podstawowe instrukcje języka C++: deklarację zmiennej rzeczywistej i całkowitej, zmianę jej wartości, operacje arytmetyczne, instrukcję warunkową `if..else`, instrukcję wielokrotnego wyboru `switch` oraz trzy typy pętli: `for`, `do..while` i `while`. W zasadzie nie zostało już wiele więcej, jeżeli chodzi o instrukcje samego języka programowania. Większość pozostałych funkcji, które wypadałoby poznać, jest raczej związana z biblioteką VCL lub są funkcjami bibliotek systemu Windows, a nie elementami języka C++. Oczywiście do omówienia pozostały jeszcze typy złożone, którymi zajmiemy się za chwilę, definiowanie funkcji i całe zagadnienie projektowania klas. Ale wiedza, którą Czytelnik posiadł już do tej pory, pozwala na pisanie dość złożonych programów, i nawet przy definiowaniu klas, ciała ich metod składać się będą w głównej mierze właśnie z poznanych tu instrukcji.

Typy złożone

A przed nami wstęp do bardziej zaawansowanych zagadnień związanych z programowaniem w C++Builderze. Na pierwszy ogień idą struktury danych, a więc wszelkiego typu tablice jedno- i wielowymiarowe, rekordy, zbiory i typy wyliczeniowe.

Tablice statyczne

Tablica to według najprostszej definicji uporządkowany zbiór elementów tego samego rodzaju. Uporządkowany, to znaczy, że każdy element ma swój numer porządkowy i wynikającą z tego pozycję. Listing 3.25 zawiera przykład deklaracji dziesięcioelementowej tablicy zdefiniowanej w domyślnej metodzie zdarzeniowej przycisku. Tablica ta zdefiniowana jest zgodnie z następującym szablonem:

```
typ nazwa[iloscElementow];
```

Rozpoczyna się od typu elementów, które są w niej przechowywane, a po nim następuje nazwa tablicy. Od deklaracji zwykłej zmiennej odróżnia ją ilość elementów podana w nawiasach kwadratowych.

Listing 3.25. *Dziesięcioelementowa tablica liczb całkowitych*

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  int tablica10Int[10];
  for(int i=0;i<10;i++) tablica10Int[i]=0;
}
```

Analogicznie jak zwykłe zmienne, także elementy tablic nie są inicjowane. Rezerwowana jest jedynie odpowiednia ilość komórek pamięci, w której elementy tablicy mają być umieszczone. Natomiast ich wartości są przypadkowe. Dlatego w listingu 3.25 wykonywana jest pętla, która przypisuje każdemu elementowi wartość 0. Przy okazji możemy zobaczyć, w jaki sposób możliwy jest dostęp do elementów tablicy — po nazwie tablicy należy podać numer indeksu w nawiasach kwadratowych.

Zakres indeksów tablicy w C++ rozpoczyna się od 0, a więc ostatni element równy jest ilości elementów minus 1. Zatem elementy tablicy z listingu 3.25 mają indeksy od 0 do 9.

Zastąpmy inicjowanie elementów tablicy zerami inicjowaniem ich za pomocą liczb losowych z przedziału od zera do dziesięciu. Następnie tak wylosowane liczby umieszczone w tablicy posortujmy. Ponieważ elementów do sortowania nie jest wiele, możemy zastosować najprostszą metodę, która polega na porównywaniu elementów w podwójnej pętli (listing 3.26). Jest to metoda dość wolna, ale przy tak niewielkim zbiorze to nie ma najmniejszego znaczenia. Przed i po posortowaniu zawartość tablicy pokazywana jest użytkownikowi.

Listing 3.26. *Najprostsza metoda sortowania zrealizowana za pomocą podwójnej pętli for*

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  int tablica10Int[10];

  //inicjacja
  Randomize();
  for(int i=0;i<10;i++) tablica10Int[i]=Random(10);

  AnsiString s="Przed sortowaniem: ";
```

```
  for(int i=0;i<10;i++) s=s+" "+IntToStr(tablica10Int[i]);
  ShowMessage(s);

  //sortowanie
  for(int i=0;i<9;i++)
         for(int j=i+1;j<10;j++)
                if(tablica10Int[i]>tablica10Int[j])
                {
                       int t=tablica10Int[i];
                       tablica10Int[i]=tablica10Int[j];
                       tablica10Int[j]=t;
                }

  s="Po sortowaniu: ";
  for(int i=0;i<10;i++) s=s+" "+IntToStr(tablica10Int[i]);
  ShowMessage(s);
}
```

Jak działa ten sposób sortowania? Wyobraźmy sobie talię kart ułożonych na stole. Przyłóżmy palec lewej ręki do pierwszej z nich — to będzie indeks `i` pierwszej pętli (zewnętrznej). Następnie drugi palec przykładamy do drugiej karty. Będzie to indeks drugiej pętli (wewnętrznej). Zwróćmy uwagę, że indeks wewnętrznej pętli zaczyna się od `i+1`, a więc nasz prawy palec zawsze będzie po prawej stronie lewego (ręce nigdy się nie skrzyżują). Teraz porównujemy karty, które wskazujemy palcem. Jeżeli karta wskazywana przez lewy palec jest „wyższa" od tej pod wskazywanej przez prawy, to zamieniamy je miejscami (tu musimy poprosić kogoś o pomoc, żeby nie odrywać palców od miejsc, gdzie są karty). Następnie przesuwamy prawy palec o jedną kartę w prawo. I znowu porównujemy karty. I tak dalej, aż dojdziemy prawym palcem do końca wyłożonych kart, a więc do końca pętli wewnętrznej. Po tej pętli mamy pewność, że na pierwszej pozycji, wskazywanej lewym palcem, leży „najniższa" ze wszystkich kart. Następnie zwiększamy indeks pętli zewnętrznej `i` do 1, co oznacza, że przesuwamy lewy palec o jedną kartę w prawo na pozycję drugą. I ponownie uruchamiamy pętlę wewnętrzną, szukając najniższej pośród kart od lewego palca na prawo. I tak dalej, aż lewym palcem dojdziemy do przedostatniej karty. Zaletą tego sortowania jest to, że przy każdej iteracji zewnętrznej pętli z lewej strony lewego palca mamy już ostatecznie ułożone karty — ich pozycja już się nie zmieni. Wadą jest oczywiście długi czas, jaki musi upłynąć, zanim uzyskamy wynik. Przy dziesięciu elementach nie zdążymy jednak nawet mrugnąć okiem.

Tablice dwuwymiarowe

Tablice w C++ mogą być wielowymiarowe. Bez problemu możemy na przykład zdefiniować macierz, czyli tablicę dwuwymiarową. Analogicznie wygląda także inicjowanie elementów tablicy. Pokazuje to listing 3.27. Musimy wówczas wykorzystać dwie zagnieżdżone pętle.

Listing 3.27. *Tablica dwuwymiarowa? Żaden problem*

```
void __fastcall TForm1::Button2Click(TObject *Sender)
{
  int tablica10x20Int[10][20];
```

```
  for(int i=0;i<10;i++)
        for(int j=0;j<20;j++)
              tablica10x20Int[i][j]=0;
}
```

Definiowanie aliasów do typów

Język C++ pozwala na definiowanie typów. Mam tu na myśli nie definiowanie klas, ale definiowanie aliasów dla istniejących typów lub dla możliwych do skonstruowania z nich typów złożonych. Służy do tego instrukcja `typedef`:

```
typedef definicja_typu nazwa_aliasu;
```

np.

```
typedef int ntyp;
```

Jakie mogą być korzyści z tak zdefiniowanego aliasu? Po pierwsze, możemy oprzeć kod na typie `ntyp` zamiast `int`, co ułatwi ewentualną zmianę typu na `__int64` lub `short`, jeżeli uznamy, że tak będzie lepiej. Tworzenie aliasów może być też wykorzystane do uproszczenia kodu lub uczynienia go bardziej czytelnym. Na przykład, aby uniknąć wielokrotnego używania długich definicji typów złożonych. Załóżmy, że w programie korzystamy wiele razy z tablic o rozmiarach 3×3, a więc z dwuwymiarowej tablicy o dziewięciu elementach. Możemy podnieść przejrzystość programu, definiując alias dla tego typu i nazywając go `Macierz3x3`. Taki nowy typ może być zdefiniowany albo lokalnie w obrębie jednej metody, albo w całym module. W pierwszym przypadku instrukcję `typedef` umieszczamy w metodzie, w której definiujemy nowy typ. Pokazuje to listing 3.28.

Listing 3.28. *Definiowanie lokalnego typu*

```
void __fastcall TForm1::Button3Click(TObject *Sender)
{
  typedef double Macierz3x3[3][3];
  Macierz3x3 a,b,c;

  for(int i=0;i<3;i++)
        for(int j=0;j<3;j++)
        {
              a[i][j]=0;
              b[i][j]=0;
              c[i][j]=0;
        }
}
```

Taki alias możemy też zdefiniować w całym module, w tym celu linię definiującą typ wstawiamy na przykład w pliku nagłówkowym modułu. Nowym typem możemy posługiwać się identycznie jak typami wbudowanymi. A więc przede wszystkim możemy tworzyć zmienne tego typu.

Tablice dynamiczne

Podstawowym ograniczeniem tablic, jakie poznaliśmy do tej pory, jest ich z góry określona wielkość. I dlatego do C++ wprowadzono możliwość deklarowania tzw. tablic dynamicznych. Zastępują one przejęte z języka C mniej wygodne rezerwowanie pamięci za pomocą funkcji `malloc` i `calloc`. Do określenia wielkości tablicy tworzonej dynamicznie nie musimy używać stałej, jak w przypadku zwykłych tablic, a możemy użyć zwykłej zmiennej, której wartość może być określona przez użytkownika programu lub wynikać z bieżącego stanu aplikacji. Listing 3.29 zawiera przykład, w którym tworzona jest tablica dynamiczna, a jej rozmiar ustalany jest przez stałą `rozmiar`, której wartość odczytywana jest z pola edycyjnego. Należy pamiętać, że podobnie jak w przypadku zwykłych zmiennych i tablic statycznych, również elementy tablic dynamicznych nie są inicjowane zerami. Ich wartości są przypadkowe.

Listing 3.29. *Tablice tworzone dynamicznie są automatycznie inicjowane zerami*

```
void __fastcall TForm1::Button4Click(TObject *Sender)
{
  int rozmiar=StrToInt(Edit1->Text);
  int* tablicaDyn=new int[rozmiar];

  //operacje na tablicy

  delete[] tablicaDyn;
}
```

Po utworzeniu tablicy powinniśmy przechowywać wskaźnik do niej, będący w istocie, podobnie jak w przypadku zwykłych tablic statycznych, wskaźnikiem do pierwszego elementu tej tablicy. Wskaźnik ten pozwoli na zwolnienie pamięci zajmowanej przez tablicę dynamiczną. W przeciwieństwie do tablic statycznych, pamięć zarezerwowana przez tablice dynamiczne nie zostanie automatycznie zwolniona po wyjściu z zakresu, w którym ta tablica została utworzona. Do jej zwolnienia korzystamy z operatora `delete[]` i następującego po nim wskaźnika do tablicy. Widoczne jest to na listingu 3.29.

Typy wyliczeniowe

A teraz stwórzmy nowy projekt aplikacji i umieśćmy na formie komponent `TLabel`. Następnie kliknijmy go dwukrotnie, aby utworzyć jego domyślną metodę zdarzeniową, która związana jest z jego zdarzeniem `OnClick` (a więc wywoływana będzie w trakcie działania programu po kliknięciu tego komponentu). W tej metodzie pobawimy się typami wyliczeniowymi i zbiorami.

Typy wyliczeniowe pozwalają tak naprawdę definiować zbiór stałych. Przyjrzyjmy się definicji zbioru `TFontStyle` z biblioteki VCL, która określa możliwe style czcionki:

```
enum TFontStyle {fsBold, fsItalic, fsUnderline, fsStrikeOut};
```

W efekcie uzyskujemy cztery stałe: `fsBold` o wartości 0 i kolejne trzy o wartościach 1, 2 i 3. Oczywiście taki sposób mówienia to znaczne uproszczenie, bo przecież zdefiniowany został nowy typ zmiennej `TFontStyle`, którą możemy zadeklarować i przypisać

jej wartość (listing 3.30). Jednak w przypadku zmiennych typów wyliczeniowych przypisana wartość może być tylko jedną ze zdefiniowanych w tym typie stałych. One określają jednoznacznie zbiór możliwych wartości tej zmiennej. Na poniższym listingu definiujemy własny typ określający styl czcionki, który jest jednak zupełnie równoważny typowi `TFontStyle`:

Listing 3.30. *Do definiowania typów wyliczeniowych używa się nawiasów okrągłych*

```
void __fastcall TForm1::Label1Click(TObject *Sender)
{
  enum TStylCzcionki {scPogrubiony, scKursywa, scPodkreslenie, scPrzekreslenie};
  TStylCzcionki sc=scPogrubiony;
  TFontStyle fs=fsBold;
}
```

Domyślnie stałym typu wyliczeniowego przypisywane są wartości od 0 i zwiększane co jeden. Można też jawnie określić, jakie mają być wartości poszczególnych stałych, np.:

```
enum TDzienTygodnia {stPoniedzialek=1,stWtorek,stSroda,stCzwartek,stPiatek,
stSobota=10,stNiedziela=11};
```

W tym wypadku rozpoczynamy numerację od 1. Kolejne elementy typu mają wartości zwiększane o 1, a ostatnim dwóm stałym przypisujemy wartości 10 i 11.

Zbiory

Zbiory zostały wprowadzone do biblioteki dołączanej do C++Buildera, aby imitować typ zbioru z Delphi, który często wykorzystywany jest przez komponenty VCL. A warto wiedzieć, że biblioteka VCL, także ta z C++Buildera, napisana jest w Object Pascalu. Zbiory w C++Builderze zaimplementowane zostały jako klasa-szablon `Set`, która przyjmuje trzy parametry: typ elementów oraz wartość minimalną i maksymalną elementów. Wiem, że zagadnienie klas nie zostało jeszcze omówione, ale to nie ma wielkiego znaczenia, bo w tej książce w ogóle przemilczę temat szablonów, które byłyby i tak niezbędne do zrozumienia tego, jak w C++Builderze funkcjonują zbiory. Nauczmy się zatem, jak ich używać, pomijając sposób ich działania.

Wiemy już, że w perspektywie biblioteki komponentów VCL zbiory okazują się dość istotnym elementem rozszerzającym język C++ — wiele podstawowych własności komponentów jest zbiorami. Najlepszym przykładem jest `TFont::Style`, czyli własność, która określa styl czcionki. Jest to zbiór, do którego mogą należeć cztery elementy poznanego wcześniej typu wyliczeniowego `TFontStyle`: `fsBold`, `fsItalic`, `fsUnderline`, `fsStrikeOut` odpowiadające pogrubieniu czcionki, kursywie, podkreśleniu i rzadko używanemu przekreśleniu. Zbiory definiuje się za pomocą nazwy klasy-szablonu `Set` i następujących po niej trzech parametrów określających typ elementów zbioru, wartość minimalną i maksymalną. W przypadku stylu czcionki typem elementów jest typ wyliczeniowy `TFontStyle`, a graniczne wartości stałe to `fsBold` i `fsStrikeOut`:

```
typedef Set<TFontStyle,fsBold,fsStrikeOut> TStyleCzcionki;
```

Typ `TFontStyles` jest więc zbiorem, do którego mogą należeć stałe z typu `TFontStyle` od `fsBold` do `fsStrikeOut` (czyli wszystkie cztery). I tak na przykład, jeżeli chcemy, żeby etykieta `TLabel` była pogrubiona, musimy do jej własności `Font->Style` będącej zbiorem typu `TFontStyles` dodać `fsBold`. Jeżeli chcielibyśmy sami zdefiniować odpowiednie typy, to należałoby zrobić to w następujący sposób:

```
enum TStylCzcionki {scPogrubiony, scKursywa, scPodkreslenie, scPrzekreslenie};
typedef Set<TStylCzcionki,scPogrubiony,scPrzekreslenie> TStyleCzcionki;
```

Druga z instrukcji zgłosi błąd jeżeli będzie umieszczona w metodzie, w której znajduje się też pierwsza. Zatem najlepiej wstawić te linie do pliku nagłówkowego.

A teraz poćwiczmy operacje na zbiorach. Z metody, którą przygotowaliśmy w poprzednim paragrafie (listing 3.30), usuńmy wszystkie dodane polecenia i wstawmy tam nową definicję zbioru o nazwie `zbior` typu `TFontStyles`, tj. identycznego jak typ własności `Label1->Font->Style`. Rozpocznijmy od wyczyszczenia zawartości zbioru. Służy do tego metoda `Clear`. Następnie przypiszmy go do własności określającej styl czcionki (listing 3.31). Spowoduje to skopiowanie zawartości zbioru.

Listing 3.31. *Zbiór w C++Builderze jest obiektem*

```
void __fastcall TForm1::Label1Click(TObject *Sender)
{
  TFontStyles zbior;
  zbior.Clear();
  Label1->Font->Style=zbior;
}
```

Jeżeli skompilujemy i uruchomimy aplikację (*F9*) i klikniemy komponent `Label1`, to… nic się nie stanie. Przynajmniej nic, co moglibyśmy zobaczyć. Domyślnym stylem etykiety jest bowiem właśnie zbiór pusty, nie jest ona ani podkreślona, ani pogrubiona, ani nie jest użyte żadne inne formatowanie. Zmieńmy wobec tego sposób inicjacji naszego zbioru tak, żeby zawierał on elementy określające podkreślenie i przekreślenie (listing 3.32). Jedynym sposobem na zmianę zawartości zbioru jest użycie operatora << dodającego element do zbioru. Nie ma w C++ konstrukcji pozwalającej na zbudowanie zbioru ze stałych.

Listing 3.32. *Zmiana stylu czcionki wymaga elementarnej wiedzy o zbiorach*

```
void __fastcall TForm1::Label1Click(TObject *Sender)
{
  TFontStyles zbior;
  zbior.Clear();
  zbior << fsUnderline << fsStrikeOut;
  Label1->Font->Style=zbior;
}
```

Teraz po naciśnięciu klawisza *F9* i kliknięciu etykiety powinniśmy zobaczyć zmianę stylu napisu. Idźmy jednak dalej. Do zbioru można dokładać kolejne elementy. Dołóżmy do naszego zbioru element `fsBold`. Pokazuje to kolejny listing:

Listing 3.33. *Dodawanie elementów do zbioru*

```
void __fastcall TForm1::Label1Click(TObject *Sender)
{
  TFontStyles zbior;
  zbior.Clear();
  zbior << fsUnderline << fsStrikeOut;
  zbior << fsBold;
  Label1->Font->Style=zbior;
}
```

Każdy element może być w zbiorze tylko raz, dlatego ponowne włożenie `fsBold` (powtórzenie wyróżnionego w listingu polecenia) nie spowoduje, że zbiór będzie zawierał dwa takie elementy.

A teraz wyjmijmy jakiś element ze zbioru. Proponuję pozbyć się przekreślenia. Służy do tego operator >>, którego składnia jest identyczna jak << (listing 3.34).

Listing 3.34. *Usuwanie elementów ze zbioru*

```
void __fastcall TForm1::Label1Click(TObject *Sender)
{
  TFontStyles zbior;
  zbior.Clear();
  zbior << fsUnderline << fsStrikeOut;
  zbior << fsBold;
  zbior >> fsStrikeOut;
  Label1->Font->Style=zbior;
}
```

Aby sprawdzić, czy jakiś element znajduje się w zbiorze, możemy użyć instrukcji `if`, w której warunek zawiera wywołanie metody `Contains` zbioru:

```
if(zbior.Contains(element)) instrukcja else instrukcja;
```

Przykład widoczny jest na listingu 3.35.

Listing 3.35. *Sprawdzanie, czy element jest w zbiorze*

```
void __fastcall TForm1::Label1Click(TObject *Sender)
{
  TFontStyles zbior;
  zbior.Clear();
  zbior << fsUnderline << fsStrikeOut;
  zbior << fsBold;
  zbior >> fsStrikeOut;
  if(zbior.Contains(fsUnderline))
        Label1->Caption="Z podkreśleniem";
  else
        Label1->Caption="Bez podkreślenia";
  Label1->Font->Style=zbior;
}
```

Ostateczny efekt manipulacji elementami zbioru określającej styl czcionki w komponencie `Label1` widoczny jest na rysunku 3.7.

Rysunek 3.7. *Efekt manipulacji stylem czcionki*

W ogólności zbiory mogą być złożone z elementów, które przyjmują najwyżej 256 różnych wartości. Doskonale nadają się więc do kolekcjonowania typów wyliczeniowych i tak są najczęściej używane. Ale mogą być również zdefiniowane na liczbach typu `byte` (czyli `unsigned char`) lub znakach `char`. Oto definicje dwóch zmiennych (tym razem nie definiujemy osobnego typu, a po prostu określamy typ zbioru przy deklaracji zmiennej):

```
Set<byte,0,255> ZbiorLiczb;
Set<char,'a','z'> ZbiorZnakow;
```

Do tak zdefiniowanych zbiorów za pomocą operatora << można dorzucać odpowiednio wartości liczb od 0 do 255 lub znaki, można je także oczywiście usuwać ze zbioru za pomocą operatora >> i sprawdzać ich obecność za pomocą metody `Contains`. Dokładnie tak samo jak we wcześniejszym przykładzie.

Pamiętajmy, że elementy zbioru nie mogą się powtarzać, więc jeżeli do zbioru `ZbiorLiczb` włożymy za pomocą operatora << wartość 2, to ponowne jej włożenie nie spowoduje, że w zbiorze będą dwa takie elementy. Nadal będzie tam tylko jeden i użycie operatora >> spowoduje całkowite jego usunięcie. W ten sposób w zbiorze `ZbiorLiczb` może być maksymalnie 256 elementów.

Struktury

Załóżmy, że piszemy program, który wymaga zgromadzenia dużej ilości danych o osobach, np. pracownikach jakiejś firmy. Zazwyczaj w takiej sytuacji korzysta się z baz danych, które są najlepszym rozwiązaniem, ponieważ zwalniają programistę z zadań związanych z przechowywaniem tych danych w plikach, a poza tym są bardzo proste w użyciu dzięki komponentom bazodanowym VCL. Załóżmy jednak, że z jakiegoś powodu, np. przenośności programu, nie chcemy korzystać z baz danych. Wówczas rozwiązaniem są struktury. Struktury to zbiór zmiennych różnego typu (nazywanych w tym kontekście polami), które związane są w całość, np.

```
struct TPracownik
{
  AnsiString Imie,Nazwisko;
  unsigned char KodStanowiska;
  Currency Placa,FunduszPracowniczy;
  unsigned char PremiaProcent;
};
```

Zdefiniowaliśmy nowy typ o nazwie `TPracownik`, który zawiera sześć pól wymienionych w nawiasach klamrowych. Jak widzimy, każde pole może być innego typu, w zależności od danych, jakie ma przechowywać.

Stwórzmy nowy projekt aplikacji. Do pliku nagłówkowego modułu *Unit1* dopiszmy powyższy rekord. Następnie zdefiniujmy listę zawierającą informacje o pracownikach. Lista, a więc taki typ danych, w którym mamy dostęp do wszystkich elementów, ale jej rozmiar może być swobodnie zmieniany, w bibliotece VCL implementowana jest przez klasę `TList`. Po utworzeniu możemy dodawać do niej elementy metodą `Add`. W pliku nagłówkowym deklarujemy pole klasy `TForm1` o nazwie `pracownicy` będące wskaźnikiem do typu `TList` (listing 3.36). Do konstruktora klasy `TForm1` dodajemy natomiast polecenie tworzące obiekt typu `TList` i zapisujące jego wskaźnik do wskaźnika `pracownicy` (listing 3.37). Klasa `TList`, jak wszystkie klasy biblioteki VCL, uniemożliwia tworzenie obiektów na stosie (bez użycia operatora `new`).

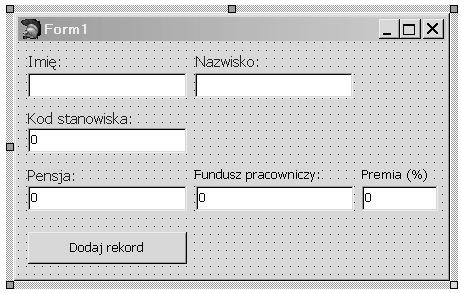
Listing 3.36. *Tak może rozpoczynać się tworzenie aplikacji kadrowej*

```
//---------------------------------------------------------------------------

#ifndef Unit1H
#define Unit1H
//---------------------------------------------------------------------------
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
//---------------------------------------------------------------------------
struct TPracownik
{
  AnsiString Imie,Nazwisko;
  unsigned char KodStanowiska;
  Currency Pensja,FunduszPracowniczy;
  unsigned char PremiaProcent;
};

class TForm1 : public TForm
{
__published:    // IDE-managed Components
private:        // User declarations
  TList* pracownicy;
public:         // User declarations
  __fastcall TForm1(TComponent* Owner);
};
//---------------------------------------------------------------------------
extern PACKAGE TForm1 *Form1;
//---------------------------------------------------------------------------
#endif
```

Listing 3.37. *Konstruktor formy z poleceniem tworzącym listę pracowników*

```
__fastcall TForm1::TForm1(TComponent* Owner)
  : TForm(Owner)
{
  pracownicy=new TList;
}
```

Następnie na formie umieśćmy sześć pól edycyjnych TEdit z etykietami TLabel według wzoru z rysunku 3.8. Obok połóżmy przycisk, który będzie pozwalał na dodanie do tablicy nowego rekordu wypełnionego danymi wpisanymi przez użytkownika do pól edycyjnych. Kliknijmy dwa razy przycisk, żeby stworzyć jego domyślną metodę zdarzeniową. W tej metodzie musimy sprawdzić, czy wypełnione zostały pola, od których tego wymagamy (oznaczone czerwonymi etykietami). Jeżeli tak, to tworzymy strukturę, której pola inicjujemy danymi z pól edycyjnych, i dodajemy ją do listy pracownicy (listing 3.38).



Rysunek 3.8. *Typowy sposób rozmieszczenia komponentów na formularzu*

Listing 3.38. *Dopisywanie rekordów do listy pracowników*

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
  if (!Edit1->Text.IsEmpty() && !Edit2->Text.IsEmpty() && !Edit3->Text.IsEmpty()
      && !Edit4->Text.IsEmpty())
  {
        unsigned int l=pracownicy->Count;
        TPracownik* pracownik=new TPracownik;
        pracownik->Imie=Edit1->Text;
        pracownik->Nazwisko=Edit2->Text;
        pracownik->KodStanowiska=StrToInt(Edit3->Text);
        pracownik->Pensja=StrToInt(Edit4->Text);
        pracownik->FunduszPracowniczy=StrToInt(Edit5->Text);
        pracownik->PremiaProcent=StrToInt(Edit6->Text);
        pracownicy->Add(pracownik);
        ShowMessage("Do listy pracowników dodałem rekord nr "+IntToStr(l));
        Edit1->Text=""; Edit2->Text="";
        Edit3->Text="0";
        Edit4->Text="0"; Edit5->Text="0"; Edit6->Text="0";
  }
  else ShowMessage("Należy wypełnić wszystkie oznaczone pola");
}
```

Zauważmy, że dostęp do pól struktury możliwy jest dzięki identycznemu operatorowi jak w przypadku obiektów, a więc za pomocą kropki.

Jak sprawdzić zawartość tablicy rekordów?

Aby móc sprawdzić aktualną zawartość tabeli pracownicy, dodajmy do formy kolejny przycisk (Button2), stwórzmy jego domyślną metodę zdarzeniową, a w niej umieśćmy polecenie wyświetlające listę nazwisk pracowników wraz z ich kodami stanowisk. Szczegóły widoczne są na listingu 3.39.

Listing 3.39. *Metoda pozwalająca na kontrolę zmian dokonywanych przez poprzednią metodę*

```
void __fastcall TForm1::Button2Click(TObject *Sender)
{
  AnsiString s="Lista pracowników:\n";
  for(int i=0;i<pracownicy->Count;i++)
  {
        TPracownik* pracownik=(TPracownik*)pracownicy->Items[i];
        s+=pracownik->Imie+" "+pracownik->Nazwisko+" ("+IntToStr(pracownik
        ->KodStanowiska)+")\n";
  }
  s+="\nLiczba rekordów: "+IntToStr(pracownicy->Count);
  ShowMessage(s);
}
```

Kilka słów o konwersji i rzutowaniu typów

Mowa była już o konwersji liczby `double` na łańcuch i odwrotnie możliwej dzięki funkcjom `FloatToStr` i `StrToFloat`. C++ umożliwia jednak inny sposób dokonania takiej konwersji, a mianowicie rzutowanie typów. Oto przykłady:

```
double d=1.0;

AnsiString s0=d;
ShowMessage(s0);

AnsiString s1=(AnsiString)d;
ShowMessage(s1);

AnsiString s2=static_cast<AnsiString>(d);
ShowMessage(s2);
```

W przypadku zmiennej `s0` mamy do czynienia z niejawną konwersją liczby typu `double` na łańcuch `AnsiString`. Na szczęście w tym przypadku konwersja ta daje dobre rezultaty, ale generalnie nie polecałbym jej stosowania, biorąc pod uwagę, że niejawne konwersje są źródłem znacznej części błędów w programach. Inicjując zmienną `s1`, wykorzystaliśmy operator rzutowania w „klasycznym" stylu znanym z języka C, tzn. nowy typ umieszczony jest przed zmienną w okrągłych nawiasach: `s1=(AnsiString)d`. Możliwa jest alternatywna składnia, w której nowy typ używany jest podobnie jak nazwa funkcji, a konwertowana zmienna wstawiana jest do nawiasów okrągłych: `s1=AnsiString(d);`. Wreszcie zmienna `s2` inicjowana jest wartością rzutowaną za pomocą operatora C++ `static_cast`. Ta składnia jest nieco bardziej złożona, bo wiąże się z podaniem jako parametru szablonu nowego typu. Konwertowana zmienna musi być umieszczona w nawiasach okrągłych, tak jak argument funkcji. Rzutowanie za pomocą `static_cast` nadaje się do sytuacji, w których użytkownik „wie, co robi", ponieważ podobnie jak w rzutowaniu w stylu C, także przy użyciu `static_cast` konwersja zostaje wykonana bez żadnej kontroli typów — ich wynik powinien być zresztą identyczny.

Poza `static_const` w C++ zdefiniowane są jeszcze trzy operatory rzutowania: `const_cast` pozwalający na usunięcie modyfikatorów `const` i `volatile`, `reinterpret_cast` dokonujący konwersji zmiennych w taki sposób, jakby były tylko zbiorem bitów, i `dynamic_cast`, który pozwala na konwersję wskaźników z uwzględnieniem kontroli typów.

Rzutowanie typów to oczywiście znacznie obszerniejszy i poważniejszy problem. Należy od razu zastrzec, że jest to jedno z najczęstszych źródeł błędów. Każde rzutowanie jest bowiem gwałtem na mechanizmie kontroli typów. Podczas rzutowania może przecież dojść do utraty precyzji, obcięcia bitów i związanej z tym zmiany wartości liczby, czy reinterpretacji znaczenia bitów. Generalnie radzi się, aby unikać konwersji niejawnej — bardzo utrudnia ona konserwację kodu. Zazwyczaj zaleca się, aby tam, gdzie rzutowanie jest jednak konieczne, stosować konwersję jawną z użyciem nowych operatorów C++, zamiast rzutowania w starym stylu znanym z C. Mówiąc szczerze, ja z tej rady korzystam rzadko i zwykle używam rzutowania C, które wydaje mi się bardziej naturalne i mniej mnie „kłuje w oczy”[10].

Na koniec jeszcze jedna uwaga dotycząca konwersji liczb. Za w pełni bezpieczne należy uznać wszystkie konwersje z typów mniej dokładnych i o mniejszym zakresie na typu o większej dokładności i większym zakresie. Można więc bez obaw rzutować `int` na `long` czy `float` na `double`. Rzutowania odwrotne mogą prowadzić do zmiany wartości zmiennych jeżeli oryginalna wartość przekracza zakres nowego typu. W miarę bezpieczne jest także rzutowanie typów całkowitych na rzeczywiste oraz konwersje na łańcuchy.

Łańcuchy

Podsumujmy dotychczasową wiedzę o łańcuchach. Do ich przechowywania używaliśmy typu `AnsiString`. Wiemy, jak łatwo je łączyć operatorem `+`, wiemy, że można w nich używać znaków niedostępnych na klawiaturze, np. `\n` (znak końca linii). Wiemy też, jak za pomocą funkcji `StrToInt`, `StrToFloat` oraz `IntToStr` i `FloatToStr` konwertować łańcuchy na liczby całkowite i rzeczywiste i z powrotem. Warto jednak zwrócić uwagę na fakt, że AnsiString nie jest typem prostym, a klasą. Dzięki temu wyposażony jest w metody, które pozwalają na manipulowanie łańcuchami. Przyjrzyjmy się im w działaniu — to najlepiej pokaże nam, jak można ich używać. Listing 3.40 zawiera szereg poleceń zmieniających początkowy łańcuch typu `AnsiString`. Wszystkie poza ostatnim korzystają z metod klasy `AnsiString`. Zauważmy, że dostęp do nich uzyskujemy za pomocą operatora `.` (kropka), a nie `->`. Jest tak, ponieważ w tym przypadku zmienna `s` nie jest wskaźnikiem do `AnsiString`, a reprezentuje obiekt utworzony na stosie.

[10] Elegancja rzutowania w starym stylu jest zresztą najpoważniejszym zarzutem wobec tego sposobu rzutowania. Według Dewhursta każde umieszczone w kodzie rzutowanie powinno kłuć w oczy, bo jest niebezpieczne (zob. Stephen C. Dewhurst *C++. Kruczki i fortele w programowaniu*, Helion 2004). Ja nie jestem aż tak zasadniczy, szczególnie że rzutowanie może być czasem koniecznością, np. przy dzieleniu liczb całkowitych.

Listing 3.40. *Zabawa łańcuchami to zajęcie nie tylko dla duchów w średniowiecznych zamkach*

```
void __fastcall TForm1::Button2Click(TObject *Sender)
{
  AnsiString s="  Helion  ";
  ShowMessage("$"+s+"$");
  s=s.Trim();                    //usuwanie spacji
  ShowMessage("$"+s+"$");
  ShowMessage(s.UpperCase());    //wielkie litery
  ShowMessage(s.LowerCase());    //małe litry
  s=s.SubString(0,3)+"lo!";      //podłańcuch i łączenie łańcuchów
  ShowMessage(s);
  s.Insert(", hello",6);         //wstawianie
  ShowMessage(s);
  ShowMessage("Pierwszy znak \"o\" znajduje się na pozycji "+IntToStr(s.Pos("o")));
  ShowMessage("Łańcuch \""+s+ "\" ma długość "+IntToStr(s.Length())+" znaków");
  s.Delete(2,7);                 //usuwanie fragmentu
  ShowMessage(s);
  TReplaceFlags rf; rf << rfReplaceAll << rfIgnoreCase;
  s=StringReplace(s,"lo!","ion",rf);  //zastępowanie fragmentu
  ShowMessage(s);
}
```

Zwróćmy uwagę, że w dwóch liniach pojawiają się cudzysłowy poprzedzone znakiem *backslash*: `\"`. Podobnie jak w przypadku znaku `\n`, mamy tu do czynienia ze znakiem, który w inny sposób nie mógłby być zapisany w kodzie źródłowym. Użycie samego cudzysłowu oznaczałoby przecież zakończenie łańcucha, podczas gdy nam chodzi jedynie o ujęcie fragmentu tekstu w cudzysłów.

Ze względu na użycia znaku *backslash* do kodowania znaków, sam znak *backslash* musi być również kodowany w ten sam sposób. Aby umieścić ten znak w łańcuchu, musimy użyć podwójnego znaku *backslash*. Ma to szczególne znaczenie, gdy łańcuch ma zawierać ścieżkę do pliku. Dla przykładu, instrukcje z listingu 3.41 doprowadzą do prezentacji w oknie dialogowym poprawnie zbudowanej ścieżki do katalogu.

Listing 3.41. *Kwestia znaków backslash*

```
void __fastcall TForm1::Button3Click(TObject *Sender)
{
  AnsiString s="c:\\Program Files\\Borland\\BDS\\4.0\\bin";
  ShowMessage(s);
}
```

W rozdziale 11. znajdą się informacje o funkcjach pozwalających w wygodny sposób wyodrębniać z łańcuchów zawierających ścieżki do plików ich nazwy, katalog czy symbol dysku.

Dyrektywy preprocesora

Cykl właściwej kompilacji kodu C++ poprzedzony jest tak zwanym cyklem preprocesora. W nim wyszukiwane są między innymi umieszczone w kodzie dyrektywy preprocesora. Jest ich sporo, dlatego tu wspomnę tylko o tych, które będą ważne w dalszych przykładach lub pojawiają się w kodzie tworzonym automatycznie przez C++Builder.

Dyrektywa #include

O dyrektywie `#include` wspomniałem już w drugim rozdziale. Zwróciliśmy wówczas uwagę na to, że nazwy plików umieszczone za nią mogą być otoczone nawiasami < > lub cudzysłowem " ". Wpływa to na ścieżkę przeszukiwania katalogów, w których poszukiwany jest włączany do kodu plik. W przypadku nawiasów < > plik szukany jest w katalogach, które w opcjach środowiska wskazane są jako katalogi zawierające pliki nagłówkowe (menu *Project\Options...*, gałąź *C++ Compiler (bcc32)\Paths and Defines*, pozycja *Include search path*). Jeżeli zamiast w nawiasach, nazwę pliku umieścimy w cudzysłowach, to będzie on szukany przede wszystkim w bieżącym katalogu, a dopiero jeżeli nie zostanie tam odnaleziony, przeszukane zostaną katalogi z plikami nagłówkowymi.

Ta różnica powoduje, że pliki nagłówkowe bibliotek standardowych dostarczanych razem z kompilatorem dołącza się, korzystając z nawiasów:

```
#include <Math.h>
```

natomiast nagłówki modułów należących do bieżącego projektu, korzystając z cudzysłowu:

```
#include "Unit1.h"
```

Dyrektywy kompilacji warunkowej

Bardzo ważne są dyrektywy `#if`, `#else` i `#endif`, które pozwalają na sterowaniem przebiegiem kompilacji. Listing 3.42 zawiera przykład, w którym sprawdzana jest wersja kompilatora i ewentualnie wyświetlana informacja o wykryciu C++Builder 2006. Można sobie jednak wyobrazić, że od wersji kompilatora zależy coś poważniejszego, np. dołączenie odpowiednich plików nagłówkowych.

Listing 3.42. *Treść komunikatu zależy od wersji kompilatora*

```
#if __BORLANDC__ >= 0x580
ShowMessage("Wykryty kompilator: C++Builder 2006 lub nowszy");
#else
ShowMessage("Starsza wersja kompilatora");
#endif
```

Należy zwrócić uwagę, że znaczenie dyrektyw z listingu 3.42, pomimo podobnego efektu, jest zupełnie inne niż konstrukcji C++ `if..else`. W przypadku dyrektywy `#if` wybór instrukcji, która ma być wykonana, dokonuje się już w momencie kompilacji, a zatem do skompilowanego pliku trafia tylko jedna opcja.

Stałe preprocesora

Stała __BORLANDC__ zdefiniowana została w bibliotekach kompilatora bcc32 za pomocą dyrektywy #define, zapewne w następujący sposób:

```
#define __BORLANDC__ 0x580
```

Oczywiście ta stała pojawia się tylko w kompilatorze C++ firmy Borland. Jeżeli przygotowujemy kod, który może być kompilowany w innych środowiskach, to musimy uwzględnić fakt, że ta stała w ogóle nie jest zdefiniowana[11]. Służą do tego dyrektywy #ifdef i #ifndef. Pierwsza reaguje na obecność stałej, druga na jej brak. Oto przykład:

Listing 3.43. *A co, gdy stała w ogóle nie jest zdefiniowana?*

```
#ifdef __BORLANDC__
  #if __BORLANDC__ >= 0x580
  ShowMessage("Wykryty kompilator: C++Builder 2006 lub nowszy");
  #else
  ShowMessage("Starsza wersja kompilatora");
  #endif
#else
ShowMessage("Nie wykryto kompilatora firmy Borland");
#endif
```

Ze względu na możliwości optymalizacji kodu przez kompilator i możliwość wystąpienia trudnych do wytropienia błędów, nie należy dyrektywy #define używać do definiowania typów, np. #define pdouble double* (zamiast tego należy użyć instrukcji C++ typedef np. typedef double* pdouble;), oraz do definiowania stałych (do tego należy używać zmiennych z modyfikatorem const).

Bardzo ważnym zastosowaniem dyrektyw #define i #ifndef jest ochrona plików nagłówkowych przed wielokrotnym włączaniem za pomocą dyrektywy #include. Zagadnienie to zostanie omówione szczegółowo w rozdziale 5.

Makra

Dyrektywa #define może być użyta z argumentem, co pozwala na definiowanie tzw. makr, np.:

```
#define _kwadrat(arg) (arg*arg)
```

Makra są jednak bardzo silnym narzędziem i przez to zbyt niebezpiecznym. W momencie kompilacji „ciało" makra jest wstawiane w każde miejsce wystąpienia jego nazwy — to może czasem prowadzić do zupełnie zaskakujących rezultatów.

[11] Wszystkie kompilatory pozwalają na identyfikację za pomocą stałych preprocesora np. _MSC_VER (Visual C++), __GNUC__ (g++), _CRAYC (Cray CC), __SUNPRO_CC (Sun CC) itd.

Makra i stałe preprocesora (także generalnie nazywane makrami) mogą być również usunięte i w konsekwencji niewidoczne dla dalszej części kodu. Służy do tego dyrektywa `#undef`.

Zadania

Usilnie namawiam, aby Czytelnik, zanim przejdzie do kolejnych rozdziałów, poświęcił trochę czasu na powtórzenie przedstawionych w tym rozdziale podstaw C++. Doskonałą okazją do tego są poniższe zadania.

Zdegenerowane równanie kwadratowe

Jeżeli współczynnik *a* w równaniu kwadratowym równy jest 0, to równanie to staje się zwykłym równaniem liniowym $bx + c = 0$. Proponuję, aby Czytelnik uwzględnił taką sytuację w metodzie rozwiązującej równanie kwadratowe.

Z kolei jeżeli jednocześnie *a* i *b* są równe zero, a *c* jest różne od zera, to należy zgłosić wyjątek informujący, że równanie jest sprzeczne.

Silnia

Przygotuj metodę, która, korzystając z pętli `for`, będzie obliczała silnię podanej liczby (użytkownik powinien móc wskazać tę liczbę za pomocą pola edycyjnego). Co to jest silnia liczby *n*? To funkcja, która jest iloczynem liczb (naturalnych) od 1 do *n*, a więc $1 \cdot 2 \cdot 3 \cdot 4 \cdot \ldots \cdot 5 \cdot \ldots \cdot n$. Zwykle silnie zapisuje się, stosując wykrzyknik *n*!. Dla przykładu: 1! = 1, 3! = 6 itd.

Gdy już sobie poradzimy z silnią, to przygotujmy drugą funkcję, która będzie obliczała funkcję n!!. Jest to odmiana silni, w której mnoży się co drugą liczbę, a więc $6!! = 6 \cdot 4 \cdot 2$, $5!! = 5 \cdot 3 \cdot 1$, itd.

Pętle

Skonstruuj takie pętle `while` i `do...while`, które będą odpowiadały dokładnie pętli `for(int i=0;i<10;i++)...`.

Korzystając z wybranej przez siebie pętli, znajdź największą liczbę, przez którą można bez reszty podzielić dwie wskazane w polach edycyjnych liczby naturalne typu `byte`.

Ikony formy

Zbiór ikon widocznych z prawej strony paska tytułu formy określany jest za pomocą zbioru `Form1->BorderIcons`. Należy usunąć z niego ikony minimalizacji i maksymalizacji. Wykorzystaj do tego metodę zdarzeniową związaną ze zdarzeniem `OnFormCreate` formy.

Typ wyliczeniowy i zbiór

Zdefiniuj własny typ wyliczeniowy określający dni tygodnia. Zadeklaruj stałą będącą zbiorem dni tygodnia i umieść w nim dni wolne.

Struktury

Do aplikacji z listą pracowników z paragrafu *Struktury* dodaj przyciski z etykietami *Poprzedni* i *Następny*, które pozwolą na przeglądanie w polach edycyjnych zawartości struktur z listy `pracownicy`.