

Mariusz Owczarek

Microsoft **Visual C++ 2012**

PRAKTYCZNE PRZYKŁADY

Microsoft Visual C++ 2012? To nic trudnego!

- Poznaj składnię języka C++ i nowości wprowadzane przez standard C++11
- Naucz się wykorzystywać typy i konstrukcje programistyczne
- Dowiedz się, jak tworzyć aplikacje oparte na Windows API i .NET Framework



Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiegokolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz Wydawnictwo HELION dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz Wydawnictwo HELION nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Redaktor prowadzący: Michał Mrowiec

Projekt okładki: Studio Gravite / Olsztyn
Obarek, Pokoński, Pazdrijowski, Zaprucki

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: helion@helion.pl
WWW: <http://helion.pl> (księgarnia internetowa, katalog książek)

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

<http://helion.pl/user/opinie?vc21pp>

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Kody źródłowe wybranych przykładów dostępne są pod adresem:

<ftp://ftp.helion.pl/przyklady/vc21pp.zip>

ISBN: 978-83-246-5352-2

Copyright © Helion 2013

Printed in Poland.

- [Kup książkę](#)
- [Poleć książkę](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

Spis treści

Co znajdziesz w tej książce?	9
Rozdział 1. Podstawy środowiska Visual C++ 2012 Professional	11
Opis środowiska	11
Język C++ a .NET Framework	12
Pobieranie i instalacja środowiska	12
Kilka pojęć na początek	14
Zmienne	14
Funkcja	14
Klasy	15
Przestrzenie nazw	16
Z czego składa się aplikacja Windows	16
Główne okno VC++ 2012 RC	17
Zaginiony projekt	18
Tworzenie projektu nowej aplikacji w VC++ 2012	19
Wygląd środowiska w trybie budowy aplikacji	22
Struktura projektu	24
Efektywna praca w środowisku	25
Rozdział 2. Struktura programów C++ i C++/CLI	29
Programy korzystające z konsoli w VC++ 2012	29
Ogólna postać programu pisanego w C++	29
Dyrektywy	31
Dyrektywa #include	31
Dyrektywa #define	33
Dyrektywa #ifdef — kompilacja warunkowa	34
Typy zmiennych	37
Zmienne typu int	37
Zmienne typu float	38
Typ double	38
Typ char	38
Modyfikatory typów	38
Rzutowanie (konwersja) typów	39
Rzutowanie static_cast	39
Rzutowanie const_cast	40
Rzutowanie safe_cast	41
Rzutowanie dynamic_cast	41

Typ wyliczeniowy	41
Silnie typowane wyliczenia	41
Słowo kluczowe auto, czyli dedukcja typu	45
L-wartości i R-wartości	46
Operatory	46
Zapis danych do plików i odczyt z nich za pomocą operatorów << i >>	48
Wskaźniki i referencje	50
Wskaźniki	50
Referencje	50
Referencje do r-wartości	51
Wskaźniki do stałej i rzutowanie const_cast	51
Tablice	52
Operatory new i delete	55
Instrukcje	55
Instrukcje warunkowe	56
Instrukcje iteracji	57
Rozdział 3. Funkcje	59
Tradycyjny zapis funkcji	59
Przeciążanie funkcji	60
Niejednoznaczność	60
Przekazywanie argumentów przez wartość i adres	61
Wskaźniki do funkcji, delegaty	62
Wyrażenia lambda	65
Funkcja main()	67
Przekazywanie parametrów do funkcji main()	68
Szablony funkcji	70
Rozdział 4. Struktury, klasy, obiekty	73
Struktury	73
Klasy	75
Statyczne metody i pola klasy	78
Wskaźnik zwrrotny this	79
Dziedziczenie	80
Funkcje wirtualne	83
Wskaźniki na klasy bazowe i pochodne, rzutowanie	85
Przeciążanie operatorów	88
Szablony klas	89
Wyjątki	92
Przestrzenie nazw	94
Rozdział 5. Konstruowanie i usuwanie obiektów klas	97
Konstruktory i destrukторы	97
Przeciążanie konstruktorów	99
Konstruktor kopiujący	100
Konstruktor przenoszący	102
Konstruktory definiowane w klasach dziedziczonych	104
Konstruktor kopiujący w klasie potomnej	105
Konstruktor definiowany w szablonie klasy	107
Struktury a klasy — porównanie	110
Rozdział 6. Interface win32, główne okno aplikacji	113
Części składowe podstawowego kodu okienkowej aplikacji win32	113
Funkcja główna programu win32	115
Klasa okna głównego	115

Tworzymy nowe okno	118
Procedura okna	120
Pętla komunikatów	122
Zasoby ikon	123
Zasoby menu	128
Okna dialogowe w zasobach	131
Rozdział 7. Obsługa komunikatów	139
Komunikaty w aplikacji Windows	139
WinAPI a standard Unicode	140
Przycisk i okno tekstowe, czyli budujemy warsztat	140
Komunikat WM_COMMAND	142
Odmalowywanie okna — komunikat WM_PAINT	145
Ruch myszy sygnalizuje WM_MOUSEMOVE	146
WM_CREATE kończy tworzenie okna	149
SendMessage() prześle każdy komunikat	150
Rozdział 8. Podstawowe kontrolki w działaniu aplikacji winAPI	153
Wszecznony przycisk Button	153
Obsługa przycisków Button jako pól wyboru	154
Kontrolka ComboBox	155
Rozdział 9. Budowa aplikacji .NET w trybie wizualnym	165
Od WinAPI do .NET Framework	165
Okno w trybie wizualnym	165
Przyciski	171
Etykiety	173
Pola tekstowe	175
Wprowadzanie danych do aplikacji za pomocą pól tekstowych	176
Wprowadzanie danych z konwersją typu	178
Wyświetlanie wartości zmiennych	179
Pole tekstowe z maską formatu danych	180
Pola wyboru, przyciski opcji, kontenery grupujące	183
Rozdział 10. Menu i paski narzędzi	187
Rodzaje menu	187
Komponent MenuStrip	187
Menu podręczne	193
Skróty klawiaturowe w menu	195
Paski narzędzi	197
Rozdział 11. Tablice, uchwyty i dynamiczne tworzenie obiektów	203
Tablice	203
Dostęp do elementów tablicy za pomocą enumeratora	206
Uchwyty	208
Dynamiczne tworzenie obiektów — operator genew	209
Dynamiczna deklaracja tablic	210
Rozdział 12. Komunikacja aplikacji z plikami	213
Pliki jako źródło danych	213
Wyszukiwanie plików	214
Odczyt własności plików i folderów	215
Odczyt danych z plików tekstowych	216
Zapisywanie tekstu do pliku	220
Zapis danych do plików binarnych	222
Odczyt z plików binarnych	223

Rozdział 13. Okna dialogowe	225
Okno typu MessageBox	225
Okno dialogowe otwarcia pliku	227
Okno zapisu pliku	230
Okno przeglądania folderów	231
Okno wyboru koloru	233
Wybór czcionki	234
Rozdział 14. Możliwości edycji tekstu w komponencie TextBox	237
Właściwości pola TextBox	237
Kopiowanie i wklejanie tekstu ze schowka	239
Wyszukiwanie znaków w tekście	240
Wstawianie tekstu między istniejące linie	241
Wprowadzanie danych do aplikacji	242
Prosta konwersja typów — klasa Convert	242
Konwersja ze zmianą formatu danych	243
Konwersja liczby na łańcuch znakowy	246
Rozdział 15. Komponent tabeli DataGridView	249
Podstawowe właściwości komponentu DataGridView	249
Zmiana wyglądu tabeli	253
Dopasowanie wymiarów komórek tabeli do wyświetlanego tekstu	255
Odczytywanie danych z komórek tabeli	257
Zmiana liczby komórek podczas działania aplikacji	261
Tabela DataGridView z komórkami różnych typów	265
Przyciski w komórkach — DataGridViewButtonCell	268
Komórki z polami wyboru — DataGridViewCheckBoxCell	269
Grafika w tabeli — komórka DataGridViewImageCell	270
Komórka z listą rozwijaną — DataGridViewComboBoxCell	272
Odnośniki internetowe w komórkach — DataGridViewLinkCell	274
Rozdział 16. Aplikacja bazy danych	277
Baza danych i aplikacja	277
Instalacja PostgreSQL	277
Wyłączenie usługi bazy	281
Inicjalizacja bazy	281
Organizacja i typy danych w bazach PostgreSQL	283
Język SQL	284
Utworzenie bazy danych	285
Interfejs użytkownika	286
Włączenie sterowników bazy PostgreSQL do projektu	288
Łączenie z bazą i odczyt danych	290
Dodawanie danych do bazy	292
Zmiana danych w bazie	295
Kasowanie danych	297
Obsługa bazy	298
Rozdział 17. Metody związane z czasem — komponent Timer	299
Czas systemowy	299
Komponent Timer	301
Rozdział 18. Grafika w aplikacjach .NET Framework	303
Obiekt Graphics — kartka do rysowania	303
Pióro Pen	308
Pędzle zwykłe i teksturowane	310

Rysowanie pojedynczych punktów — obiekt Bitmap	313
Rysowanie trwale — odświeżanie rysunku	314
Animacje	316
Rozdział 19. Podstawy aplikacji wielowątkowych	319
Wątki	319
Komunikacja z komponentami z innych wątków — przekazywanie parametrów	321
Przekazywanie parametrów do metody wątku	323
Klasa wątku — przekazywanie parametrów z kontrolą typu	324
Kończenie pracy wątku	326
Semafor	328
Sekcje krytyczne — klasa Monitor	331
Komponent BackgroundWorker	334
Rozdział 20. Połączenie aplikacji z siecią Internet	339
Komponent WebBrowser	339
Przetwarzanie stron Web — obiekt HtmlDocument	342
Uruchamianie skryptów JavaScript z poziomu aplikacji	345
Protokół FTP	347
Pobieranie zawartości katalogu z serwera FTP	348
Pobieranie plików przez FTP	350
Wysyłanie pliku na serwer FTP	351
Klasa do obsługi FTP	352
Pobieranie plików w oddzielnym wątku	356
Wysyłanie plików w wątku	357
Rozdział 21. Dynamiczne tworzenie okien i komponentów	359
Wyświetlanie okien — klasa Form	359
Komponenty w oknie tworzonym dynamicznie	361
Przesyłanie danych z okien dialogowych	362
Okno tytułowe aplikacji	363
Obsługa zdarzeń dla komponentów tworzonych dynamicznie	364
Aplikacja zabezpieczona hasłem	365
Rozdział 22. Prosty manager plików	367
Interfejs menedżera	367
Wyświetlanie zawartości folderów	367
Formatowanie prezentacji folderu	369
Przechodzenie w dół i w górę drzewa plików	372
Idziemy w górę	372
Idziemy w dół	373
Kopiowanie plików między panelami	374
Kasowanie plików	375
Skorowidz	377

Rozdział 19.

Podstawy aplikacji wielowątkowych

Wątki

W systemach wielowątkowych wiele programów może być wykonywanych jednocześnie. W systemach jednoprocessorowych wrażenie jednoczesności wykonania powstaje dzięki przydzielaniu czasu procesora dla kolejnych programów na przemian. Każdy z takich wykonywanych równolegle algorytmów nosi nazwę wątku. Znaczenie aplikacji wielowątkowych wzrosło po pojawieniu się procesorów z kilkoma rdzeniami. Dzięki takiej architekturze możliwa jest rzeczywista jednoczesność wykonywania wątków.

Standardowo aplikacja składa się tylko z jednego wątku, związanego z oknem głównym. Taki model nie zawsze jest wystarczający. Ponieważ podczas wykonywania kodu metody nie są przetwarzane zdarzenia dla danego wątku, w przypadku dłuższych metod okno aplikacji nie jest odświeżane i nie jest możliwa obsługa kontroltek. Z tego powodu okno aplikacji wydaje się „zablokowane” i nie jest możliwe wyświetlanie żadnych danych na przykład w kontrolce etykiety Label.

Aby poprawić działanie takiej aplikacji, należy wykonywać część kodu jako oddzielny wątek. Wątek jest reprezentowany w aplikacji przez obiekt klasy Thread. Przy tworzeniu tego obiektu parametrem konstruktora jest metoda, która będzie wykonywana w tym wątku. Następnie do rozpoczęcia wątku służy metoda Start() klasy Thread, którą wykonujemy na utworzonym obiekcie. Prześledzimy zastosowanie oddzielnych wątków na przykładzie programu wymagającego długich obliczeń.

Przykład 19.1

Napisz program odnajdujący liczby pierwsze w przedziale od 2 do 100 000. Liczba pierwsza to taka, która dzieli się tylko przez 1 i przez samą siebie.

Rozwiązanie

Najpierw napiszemy tę aplikację jako klasyczną aplikację z pojedynczym wątkiem.

Utwórz nowy projekt według przykładu 1.4 i wstaw do formatki przycisk `Button` oraz pole tekstowe `TextBox`. Właściwość `Multiline` pola ustaw na `true` i zwiększ wymiary pola tak, aby zmieściło kilka linii. Zdarzenie `Click` przycisku będzie uruchamiała metodę poszukującą liczb pierwszych. W tym przypadku zastosujemy prosty algorytm, który dzieli modulo kolejne liczby i przez liczby od 2 do i, sprawdzając w ten sposób, czy liczba i ma dzielnik inny niż ona sama. W momencie znalezienia dzielnika kończy się pętla `while`. Jeżeli dzielnikiem dla danej liczby jest ona sama, to liczba jest wyświetlana. Nie jest to najbardziej efektywny algorytm, ale nie o to tu chodzi.

```
private: System::Void button1_Click(System::Object^ sender, System::EventArgs^ e) {
    System::Int32 n=2;
    for (System::Int32 i=2;i<100000;i++) {
        n=2;
        while ((i%n))
            n++;
        if (i==n)
            textBox1->AppendText(i.ToString()+System::Environment::NewLine);
    }
}
```

Po uruchomieniu obliczeń przyciskiem program wydaje się działać normalnie, jednak próba przesunięcia okna programu nie powiedzie się, a w przypadku zasłonięcia okna przez inną aplikację i powtórnego odsłonięcia okno będzie puste, dopóki nie skończą się obliczenia.

Przykład 19.2

Napisz program identyczny jak w przykładzie 19.1, ale z uruchamianiem obliczeń w oddzielnym wątku.

Rozwiązanie

Utwórz nowy projekt aplikacji według przykładu 1.4 i wstaw do okna przycisk `Button` i pole tekstowe `TextBox`. Również tu ustaw właściwość `Multiline` pola `TextBox` na `true`.

Na początku kodu w pliku *Form1.h* obok dyrektyw załączania przestrzeni nazw `using namespace` dołącz do programu przestrzeń nazw z metodami wielowątkowości.

```
using namespace System::Threading;
```

Teraz utwórz metodę, która będzie się uruchamiała w wątku jako metoda klasy `Form1`. Metoda będzie poszukiwała liczb pierwszych tak samo jak poprzednio. Zadeklaruj ją po dyrektywie `#pragma endregion`.

```
private: System::Void watek() {
    System::Int32 n=2;
    for (System::Int32 i=2;i<100000;i++) {
        n=2;
```

```
        while ((i%n))
            n++;
    }
}
```

Naciśnięcie przycisku utworzy i uruchomi wątek.

```
private: System::Void button1_Click(System::Object^ sender, System::EventArgs^ e) {
    Thread^ watek_liczenia = gcnew Thread(gcnew ThreadStart(this,&Form1::watek));
    watek_liczenia->Start();
}
```

Argumentem konstruktora obiektu `Thread` jest metoda wątku. Nie podstawiamy jej bezpośrednio, ale używamy delegata `ThreadStart`. O delegatach pisałem w rozdziale o funkcjach. Tak samo jak tam konstrukcję tego obiektu umieściłem bezpośrednio na liście parametrów konstruktora obiektu `Thread`. Parametrami obiektu `ThreadStart` są: obiekt klasy, do której należy metoda wątku (w tym przypadku jest to klasa głównego okna aplikacji pobierana za pomocą wskaźnika `this`), oraz referencja do metody wątku. Referencja musi zawierać nazwę klasy, do której należy metoda wątku, i nazwę samej metody wątku.

Po uruchomieniu programu i naciśnięciu przycisku program rozpoczyna obliczenia, ale nic nie wyświetla. Można sprawdzić, że program liczy, wywołując *Menedżera zadań*. Po naciśnięciu przycisku rośnie stopień wykorzystania procesora przez naszą aplikację. Mimo że aplikacja liczy, można ją swobodnie przesuwac, a po zakryciu i odkryciu okno ma normalny wygląd.

Zapisz aplikację, dokończymy ją w następnym przykładzie.

To, że program z poprzedniego przykładu nic nie wyświetla, nie wynika z zapomnienia, ale wymaga oddzielnego omówienia. W aplikacji wielowątkowej można korzystać z komponentów wizualnych należących do tego samego wątku. Ponieważ pole tekstowe `TextBox` jest w innym wątku niż obliczenia, nie jest możliwe korzystanie z niego bezpośrednio. Dzieje się tak dlatego, że mógłby wtedy wystąpić konflikt między komunikatami przesyłanymi do kontrolki z różnych wątków. Następny przykład pokazuje, jak poradzić sobie z tym ograniczeniem.

Komunikacja z komponentami z innych wątków — przekazywanie parametrów

Ponieważ niemożliwa jest komunikacja z kontrolkami pochodzącymi z innego wątku, każde odwołanie do komponentu musi być realizowane z wątku, do którego należy komponent. Wiele komponentów posiada metodę `Invoke()`, która wywołuje dowolną inną metodę (za pomocą delegata) tak, jakby była ona wywoływana w wątku, do którego należy kontrolka. Komunikację z komponentami z innych wątków można zapisać w punktach:

- a) W klasie okna, do której należy kontrolka okna, definiujemy metodę komunikującą się z tą kontrolką (na przykład piszącą do pola tekstowego).
- b) Również w klasie okna, do którego należy kontrolka, deklarujemy delegat ze słowem kluczowym `delegate` (podobnie do deklaracji zmiennej lub pola klasy).
- c) W metodzie wątku (która też jest metodą klasy tego okna) definiujemy delegat metody, przy czym jako argument konstruktora podajemy metodę zdefiniowaną w punkcie a).
- d) Kiedy potrzebne jest odwołanie do kontrolki w wątku, używamy metody `Invoke()` ze zdefiniowanym w punkcie c) delegatem jako parametrem.

Myślę, że wiele wyjaśni następujący przykład.

Przykład 19.3

Popraw aplikację tak, aby wyświetlała znalezione liczby pierwsze.

Rozwiązanie

Otwórz aplikację z poprzedniego przykładu.

Cała trudność tego przykładu polega na tym, że metoda poszukiwania liczb pierwszych uruchamiana w oddzielnym wątku będzie musiała pisać liczby do okna tekstowego znajdującego się w wątku okna głównego.

Najpierw w klasie `Form1` napisz metodę, która będzie wpisywała podaną jako argument liczbę do pola tekstowego.

```
private: System::Void wyswietl(System::Int32 i) {
    textBox1->AppendText(i.ToString()+System::Environment::NewLine);
}
```

Teraz również w klasie `Form1` utwórz deklarację delegata — tak jak zwyklej metody klasy, ale ze słowem kluczowym `delegate`. Delegat musi mieć listę parametrów taką jak metoda, która będzie posługiwała się komponentem z innego wątku (czyli u nas metoda `wyswietl()`).

```
private: delegate void wyswDelegat(System::Int32 i);
```

W metodzie wątku `watek()` trzeba zdefiniować wystąpienie delegata. Konstruktor delegata ma dwa parametry: pierwszy to klasa, z której pochodzi metoda odwołująca się do kontrolki, a drugi to wskaźnik do samej metody. Po stworzeniu obiektu delegata w celu pisania do pola tekstowego wywołujemy metodę `Invoke()`. Pierwszy parametr tej metody to obiekt delegata, a drugi to tabela obiektów typu obiekt zawierająca wartości parametrów metody. W naszym przypadku metoda `wyswietl()` ma tylko jeden parametr. Oto poprawiony kod metody `watek()`:

```
private: System::Void watek() {
    wyswDelegat^ wyswietlDelegat =
        gcnew wyswDelegat(this, &Form1::wyswietl);
    System::Int32 n=2;
```

```
for (System::Int32 i=2; i<100000; i++) {  
    n=2;  
    while ((i%n))  
        n++;  
    if (i==n)  
        this->Invoke(wyswietlDelegat, gcnew array <System::Object^>(1){i});  
}
```

Można już uruchomić program. Po naciśnięciu przycisku liczby pojawiają się w polu, a okno daje się swobodnie przesunąć i zakrywać.

Jeśli zakończysz działanie aplikacji przyciskiem w prawym górnym rogu, zobaczysz komunikat o błędzie:

Additional information: Cannot access a disposed object.

Pojawia się on dlatego, że zamykamy główne okno aplikacji, a wątek nadal istnieje i próbuje napisać coś w kontrolce TextBox, która już nie istnieje. Jak temu zaradzić, napiszę w dalszej części rozdziału.

Zapisz program, ponieważ będzie przydatny w następnym przykładzie.

Przekazywanie parametrów do metody wątku

Do tej pory metoda wątku była bezparametrowa, czasem jednak konieczne jest przekazanie parametrów. Wtedy przy tworzeniu obiektu klasy Thread posługujemy się delegatem ParameterizedThreadStart zamiast ThreadStart. Wartość parametru przekazujemy w metodzie Start() przy uruchamianiu wątku.

Przykład 19.4

W aplikacji poszukującej liczb pierwszych przekazuj górną granicę poszukiwania liczb do metody wątku za pomocą parametru.

Rozwiązanie

Otwórz aplikację z poprzedniego przykładu.

Dodaj pole tekstowe, w które będziemy wprowadzać górną granicę poszukiwań.

Metodę wątku zmodyfikuj jak niżej. Parametr przekazywany do metody musi być typu Object, dlatego przy podstawianiu do pętli for wykonujemy konwersję.

```
private: System::Void watek(Object^ i_max) {  
    wyswDelegat^ wyswietlDelegat =  
        gcnew wyswDelegat(this, &Form1::wyswietl);
```

```

System::Int32 n=2;
for (System::Int32 i=2;i<Convert::ToInt32(i_max);i++) {
    n=2;
    while ((i%n))
        n++;
    if (i==n)
        this->Invoke(wyswietlDelegat,gcnew array <System::Object^>(1){i});
}
}

```

Metoda obsługująca zdarzenie Click będzie teraz korzystała z delegata ParameterizedThreadStart i metody Start() z argumentem pobranym z drugiego pola tekstowego.

```

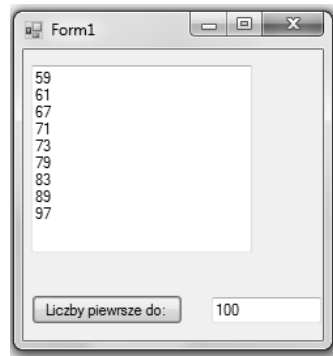
private: System::Void button1_Click(System::Object^ sender, System::EventArgs^ e) {
    Thread^ watek_liczenia = gcnew Thread(gcnew
        ParameterizedThreadStart(this,&Form1::watek));
    watek_liczenia->Start(textBox2->Text);
}

```

Po uruchomieniu programu wpisz liczbę całkowitą w drugie pole tekstowe i naciśnij przycisk. Zostaną wygenerowane tylko liczby mniejsze do zadanej. Wygląd aplikacji przedstawia rysunek 19.1.

Rysunek 19.1.

*Aplikacja wyszukująca
liczby pierwsze*



Niestety, ten sposób przekazywania parametrów ma wady. Po pierwsze, można przekazać tylko jeden parametr (można przekazać tablicę, ale nie zawsze jest to wygodne), a na dodatek musi być on typu Object. Jest to bardzo pierwotny typ i akceptuje większość typów standardowych, co powoduje, że nie ma kontroli nad przekazywanymi danymi i łatwo popełnić błąd, który nie zostanie zauważony.

Klasa wątku — przekazywanie parametrów z kontrolą typu

Aby pozbyć się powyższych problemów z typami danych, można zapisać metodę wątku jako metodę oddzielnej klasy, a parametry jako zmienne tej klasy.

Przykład 19.5

Zapisz metodę wątku jako metodę klasy.

Rozwiązanie

Utwórz nowy projekt aplikacji okienkowej C++/CLI i wstaw do niego dwa pola tekstowe oraz przycisk.

Pole tekstowe `textBox1` będzie służyło do wyświetlania liczb, a `textBox2` do wprowadzania górnej granicy poszukiwań. Właściwość `Multiline` pola `textBox1` ustaw na `true` i powiększ je, aby mieściło kilka linii tekstu.

Nie zapomnij o załączeniu przestrzeni nazw dla wielowątkowości.

```
using namespace System::Threading;
```

Zacznij od napisania klasy wątku, którą nazwiemy `SzukLiczbPierw`. Oprócz metody liczącej wątku klasa będzie zawierać konstruktor inicjalizujący zmienne klasy podanymi wartościami. Wartości te nie są już typu `Object`, ale mają konkretne typy. Z tych zmiennych będzie korzystała metoda wątku. Jedną ze zmiennych klasy jest uchwyt do pola tekstowego, do którego będą wpisywane liczby. Metoda wpisująca liczby do pola również jest częścią klasy wątku. Wywołanie tej metody poprzez metodę `Invoke()` powoduje, że jest ona wywoływana jako metoda wątku, w którym znajduje się pole tekstowe. Zauważ, że teraz metoda `Invoke()` nie jest wywoływana na oknie aplikacji, ale na obiekcie pola tekstowego. Klasę umieść przed klasą `Form1`, zaraz po dyrektywach `using namespace`.

```
public ref class SzukLiczbPierw {
private: System::Int32 i_max;
private: TextBox^ pole;
private: delegate void wyswDelegat1(System::Int32 i);
private: System::Void wyswietl1(System::Int32 i) {
    pole->AppendText(i.ToString()+System::Environment::NewLine);
}
// konstruktor
public: SzukLiczbPierw (System::Int32 gora,TextBox^ ramka)
{
    i_max=gora;
    pole=ramka;
}
// metoda obliczeń
public: System::Void watek1() {
    wyswDelegat1^ wyswietlDelegat =
        gcnew wyswDelegat1(this,&SzukLiczbPierw::wyswietl1);
    System::Int32 n=2;
    for (System::Int32 i=2;i<Convert::ToInt32(i_max);i++) {
        n=2;
        while ((i%n))
            n++;
        if (i==n)
            pole->Invoke(wyswietlDelegat,gcnew array
                <<System::Object^>(1){i});
    }
};
```

Teraz trzeba zaprogramować metodę dla zdarzenia Click przycisku. W tej metodzie będzie tworzony obiekt klasy wątku, a następnie sam wątek. Parametry do metody wątku przekazujemy poprzez konstruktor klasy wątku. Ponieważ metoda licząca jest teraz metodą klasy SzukLiczbPierw, a nie klasy Form1, parametry delegata tej metody to nie wskaźnik this, ale obiekt klasy SzukLiczbPierw. Drugim parametrem jest referencja do metody obliczającej.

```
private: System::Void button1_Click(System::Object^ sender, System::EventArgs^ e) {
    SzukLiczbPierw^ obliczenia =
        gcnew SzukLiczbPierw(Convert::ToInt32(textBox2->Text), textBox1);
    Thread^ watek_liczenia =
        gcnew Thread(gcnew ThreadStart(obliczenia,
            &SzukLiczbPierw::watek1));
    watek_liczenia->Start();
}
```

Program po uruchomieniu będzie działał identycznie jak program z poprzedniego przykładu.



Wskazówka

Tryb graficznego projektowania okna aplikacji wymaga, aby klasa Form1 była pierwszą w pliku *Form1.h*. Jeżeli umieścisz na początku inną klasę, stracisz możliwość wizualnego projektowania aplikacji. Dlatego wszystkie kontrolki należy umieścić przed napisaniem klasy wątku. Także metodę obsługującą zdarzenia trzeba utworzyć przed umieszczeniem klasy wątku.

Kończenie pracy wątku

Przed zakończeniem działania aplikacji powinna ona zamknąć wszystkie wątki, które do niej należą. Inaczej występuje błąd, z którym się już zetknąłeś. Może on prowadzić do tzw. wycieków pamięci i na pewno nie zwiększa bezpieczeństwa aplikacji.

Wątek zakończy pracę w normalnym trybie, jeśli powrócimy z jego funkcji za pomocą instrukcji return. Eleganckie zakończenie wątku może więc wyglądać tak:

1. Deklarujemy zmienną globalną typu bool.
2. W metodzie wątku okresowo sprawdzamy, czy ta zmienna ma wartość np. true. Jeśli tak, to wychodzimy z wątku za pomocą return.
3. W wątku okna głównego wystarczy zmienić w dowolnym momencie wartość tej zmiennej na true i wątek się zakończy.

Zakończenie wątku nie będzie natychmiastowe, ale upłynie pewien czas, zanim wątek sprawdzi wartość zmiennej. Jeśli kończymy wątek przyciskiem, to nie ma problemu, czas ten będzie niezauważalny. Gorzej, jeśli kończymy wątek z powodu tego, że zamykamy okno aplikacji. Wtedy od ustawienia zmiennej kontrolnej może upłynąć za mało czasu i wątek się nie zakończy. Jest kilka sposobów na rozwiązanie tego problemu. Najprościej wstrzymać w jakiś sposób zamykanie aplikacji, aż wątek się zakończy.

Ja zastosuję sposób chyba najprostszy i może mało efektywny, ale skuteczny. Przy zakończeniu aplikacji odwrócimy uwagę użytkownika, wyświetlając okno dialogowe i prosząc o zatwierdzenie przyciskiem OK. Zanim użytkownik zatwierdzi okno, wątek już się zakończy.

Przykład 19.6

Biorąc za punkt wyjścia aplikację z przykładu 19.3, zadaj o zakończenie wątku.

Rozwiązanie

Wykonaj jeszcze raz przykład 19.3, jeśli nie masz tego programu. Dodaj aplikacji drugi przycisk Button. We właściwości Text tego przycisku wpisz *Stop*, będzie on zatrzymywał wątek. Po dyrektywie `#pragma endregion` zadeklaruj zmienną sterującą.

```
private: bool koniec_watku;
```

Do metody `watek` w głównej jej pętli dodamy okresowe sprawdzenie wartości zmiennej. Znajdź w metodzie poniższą linię i dopisz następną:

```
for (System::Int32 i=2;i<100000;i++) { //ta linia istnieje
if (koniec_watku==true) return;
}
```

Teraz kliknij podwójnie przycisk *Stop* i uzupełnij powstałą funkcję jak niżej:

```
private: System::Void button2_Click(System::Object^ sender, System::EventArgs^ e) {
    koniec_watku=true;
}
```

W powyższej funkcji zmieniamy wartość zmiennej sterującej, powodując, że warunek zakończenia wątku będzie spełniony.

Teraz to samo trzeba zrobić, kiedy okno będzie zamykane. Wykorzystamy do tego zdarzenie `FormClosing`. Kliknij okno aplikacji w widoku projektowania. Rozwiń panel *Properties*, przełącz się na widok zdarzeń, znajdź `FormClosing` i kliknij dwukrotnie. Utworzy się funkcja obsługująca to zdarzenie. W tej funkcji umieścimy ustawienie zmiennej `koniec_watku` na `true`, ale to nie daje pewności, że wątek zakończy się przed zakończeniem programu. Wobec tego, aby zyskać na czasie, wyświetlimy okno `MessageBox`, które już znasz. Oto cała funkcja:

```
private: System::Void Form1_FormClosing(System::Object^ sender, System::Windows::
↳Forms::FormClosingEventArgs^ e) {
    koniec_watku=true;
    MessageBox::Show(L"Zatrzymuje wątki liczenia","Zaczekaj",
↳MessageBoxButtons::OK);
}
```

Uruchom aplikację, kliknij przycisk uruchamiający wątek, a następnie, nie czekając na koniec obliczeń, spróbuj zamknąć okno przyciskiem `X` w pasku.

Pojawi się okno dialogowe, a w oknie *Output* na dole środowiska zobaczysz taki komunikat:

The thread '<No Name>' (0xa98) has exited with code 0 (0x0).

Oznacza to, że wątek zakończył się bezpiecznie. Kliknięcie OK zakończy działanie całej aplikacji.

Semafory

Pomimo tytułu dalszy ciąg książki to nie podręcznik kolejnictwa — zostajemy przy programowaniu. Nazwa jest jak najbardziej trafiona. Semafory to obiekty, które przepuszczają określoną liczbę wątków. Stan semafora opisywany jest przez liczbę. Jest to liczba wątków, jaka może być wpuszczona. Wątek poprzez wywołanie określonej metody zapytuje semafor, czy jest dla niego miejsce. Jeśli tak, to wartość jest obniżana o jeden, a wątek może kontynuować działanie. Jeśli wartość wynosi zero, to wątek jest zawieszony. Proces, który został wpuszczony, wychodząc z obszaru działania semafora, wywołuje inną metodę, która podnosi wartość semafora. Wtedy inny czekający wątek może zostać odblokowany, a wartość jest obniżana. W praktyce mamy klasę `Semaphore` i dwie metody: `WaitOne()` i `Release()`. Tworzymy obiekt `Semaphore`, następnie w metodzie wątku wywołujemy na nim `WaitOne()`. Od tego momentu albo wątek jest „wpuszczany” przez semafor i wykonuje się dalej, albo jego wykonanie jest zawieszane, aż semafor będzie akceptował nowe wątki. Jeśli wątek chce zwolnić semafor, wywoła na nim metodę `Release()`. Powoduje to podwyższenie wartości semafora, który może wpuścić następny wątek. Zwolnienie semafora nie musi oznaczać końca wątku, który go opuszcza. Może on nadal działać, ale nie ma już związku z semaforem.

Konstruktorów samego obiektu `Semaphore` jest kilka. Ja użyję formy, która akceptuje dwie liczby. Pierwsza to początkowa wartość semafora, czyli liczba wątków, które może przepuścić zaraz po utworzeniu. Druga liczba to jego maksymalna dopuszczalna wartość.

Przykład 19.7

Napisz program, w którym kolejne wątki uruchamia się za pomocą przycisku. Po rozpoczęciu działania wątki będą pytać o wpuszczenie przez semafor. Niech semafor w aplikacji przepuszcza maksymalnie trzy wątki.

Rozwiązanie

Utwórz nowy projekt aplikacji okienkowej C++/CLI według przykładu 1.4. Do okna wstaw dwa okna `TextBox` i przycisk `Button`. Jedno pole tekstowe będzie pokazywało komunikaty wątków poza semaforem, a drugie wątków przepuszczonych. W bloku dyrektyw `using namespace` dołącz przestrzeń nazw `System::Threading`.

```
using namespace System::Threading;
```

Aby wątki mogły pisać w polach tekstowych, trzeba skorzystać z delegata metody piszącej do pola tekstowego, tak jak w przykładzie 19.3. Tym razem mamy dwa pola i napiszemy dwie metody. Wpisz je po dyrektywie `#pragma endregion`.

```
private: System::Void wyswietl1(System::String^ st)
{
    textBox1->AppendText(st);
}
private: System::Void wyswietl2(System::String^ st)
{
    textBox2->AppendText(st);
}
```

Zaraz niżej zadeklaruj delegata metody o liście parametrów takiej jak `wyswietl1()` i `wyswietl2()`.

```
private: delegate void wyswDelegat(System::String^ st);
```

Wreszcie czas na samą metodę wątku. Tak jak w przykładzie 19.3 mamy deklarację obiektów delegata. Następnie wątek melduje się w pierwszym oknie i zapytuje o wpuszczenie przez semafor. Jeśli zostanie wpuszczony, to symuluje swoją pracę przez 3-sekundowy „sen”, a następnie zwalnia semafor metodą `Release()`. Od razu wyświetla wartość zwróconą z metody — jest to stan semafora *przed* jej wywołaniem. Aktualny stan będzie o jeden większy, bo stan to liczba wątków, które mogą być przepuszczone. Oto cała metoda wątku:

```
private: System::Void watek(System::Object^ num)
{
    wyswDelegat^ wyswietlDelegat1 = gcnew wyswDelegat(this,&Form1::wyswietl1);
    wyswDelegat^ wyswietlDelegat2 = gcnew wyswDelegat(this,&Form1::wyswietl2);
    this->Invoke(wyswietlDelegat1, safe_cast<System::Object^>(L"Wątek "+num->
    ToString()+" czeka pod semaforem."+System::Environment::NewLine));
    //zapytanie o stan semafora
    semafor->WaitOne();

    this->Invoke(wyswietlDelegat2, safe_cast<System::Object^>(L"Wątek "+num->
    ToString()+" przekroczył semafor."+System::Environment::NewLine));
    Thread::Sleep(3000);
    this->Invoke(wyswietlDelegat2, safe_cast<System::Object^>(L"Wątek "+num->
    ToString()+" zwolnił semafor."+System::Environment::NewLine));
    //zwolnienie semafora
    this->Invoke(wyswietlDelegat2, safe_cast<System::Object^>(L"Aktualny stan
    ↪semafora: "+(semafor->Release()+1).ToString()+System::
    ↪Environment::NewLine));
}
```

Trzeba jeszcze zadeklarować uchwyt do obiektu semafora. Możesz to zrobić zaraz po metodach `wyswietl1()` i `wyswietl2()`.

```
private: Semaphore^ semafor;
```

Zaraz pod nim zadeklaruj zmienną, która przyda się przy numeracji wątków.

```
private: System::Int32 i;
```

Na początku wykonywania aplikacji utworzymy obiekt semafora i ustawimy wartość zmiennej pomocniczej. Dobrze do tego celu nadaje się zdarzenie Load okna aplikacji. Kliknij na to okno w widoku projektowania. Ważne, aby kliknąć na samo okno, a nie na którąś z kontroltek. Utworzy się metoda obsługi zdarzenia Load. W tej metodzie wywołamy konstruktor semafora. Parametry oznaczają, że na początku semafor ma wartość 3 i jest to zarazem maksymalna jego wartość. Zmienna *i* ma wartość jeden i jest to numer pierwszego wątku.

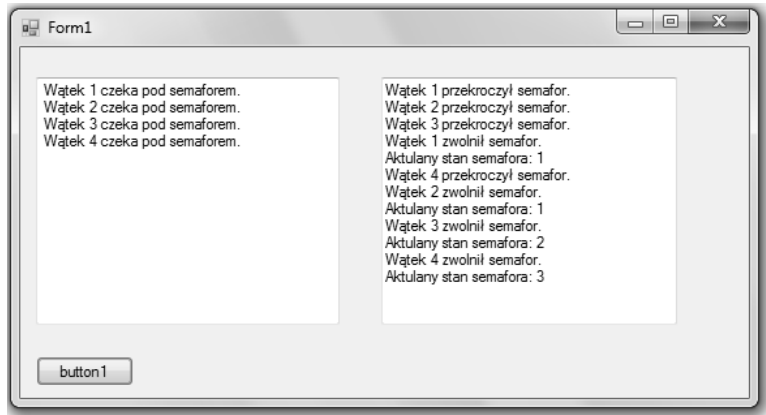
```
private: System::Void Form1_Load(System::Object^ sender, System::EventArgs^ e) {
    semafor = gcnew Semaphore(3, 3);
    i=1;
}
```

Pozostała metoda kliknięcia przycisku. Kliknij go dwukrotnie w widoku projektowania aplikacji. Wnętrze będzie raczej proste: tworzymy nowy wątek, uruchamiamy go, przekazując jako parametr jego numer, i zwiększamy numer o jeden dla następnego wątku.

```
private: System::Void button1_Click(System::Object^ sender, System::EventArgs^ e)
{
    Thread^ t = gcnew Thread(gcnew ParameterizedThreadStart(this,&Form1::watek));
    t->Start(i);
    i++;
}
```

Uruchom aplikację. Kliknij 4 razy w miarę szybko na przycisk. Chodzi o to, żeby uruchomić cztery wątki w ciągu mniej niż trzech sekund. Wynik działania mamy na rysunku 9.2.

Rysunek 9.2.
Działanie aplikacji z semaforem



Wątki 1, 2 i 3 zostały wpuszczone prawie bez czekania, a wątek 4 czeka. Stan semafora wynosi w tej chwili zero. Kiedy wątek 1 zwalnia semafor, jego stan zmienia się na 1 i zostaje wpuszczony wątek 4. Po pracy wszystkie wątki zwiększają stan semafora.

Sekcje krytyczne — klasa Monitor

Niektóre części kodu aplikacji nie powinny być dostępne jednocześnie dla więcej niż jednego wątku. Można to zrealizować za pomocą semafora o początkowej wartości równej jeden. Innym sposobem jest użycie specjalnej klasy .NET Framework o nazwie Monitor. W prostym przypadku ważne będą dla nas dwie metody statyczne tej klasy: Enter() i Exit(). Metoda Enter() oznacza początek sekcji krytycznej, a Exit() koniec. Parametrem tych metod jest obiekt, który chcemy udostępnić tylko danemu wątkowi. Załóżmy, że mamy obiekt o nazwie okno, na którym będzie operował jakiś wątek; może to być dowolna operacja, na przykład zmiana koloru okna. W metodzie wątku piszemy:

```
Monitor::Enter(okno);
```

Od teraz tylko ten wątek ma dostęp do okna. Każdy inny wątek, który spróbuje się powołać na obiekt okno, zostanie zawieszony. Teraz zmieniamy kolor okna:

```
Okno->ForeColor=Blue;
```

i sygnalizujemy, że już nie chcemy mieć wyłączności na dostęp:

```
Monitor::Exit(okno);
```

To cała filozofia sekcji krytycznych z klasą Monitor. Oczywiście, jak to często bywa, problemem są szczegóły. Jeśli wątek zapętlili się w sekcji krytycznej, to może nigdy nie zawołać metody Exit() i zablokować dostęp do okna na stałe. Dlatego korzystamy z obsługi wyjątków. Instrukcje sekcji krytycznej umieszczamy w bloku try, a metodę Exit() umieszczamy w części finally tego bloku. Kod z bloku finally zostanie wykonany zawsze, niezależnie od tego, czy blok try wyrzuci jakiś wyjątek, czy nie. Przykład sekcji krytycznej będzie ostatecznie wyglądał tak:

```
Monitor::Enter(okno);  
try { Okno->ForeColor=Blue;}  
finally { Monitor::Exit(okno);}
```

Jako przykład pokażę różnicę w dostępie do okna bez sekcji krytycznej i z zabezpieczeniem sekcją.

Przykład 19.8

Niech trzy wątki starają się równocześnie pisać do kontrolki TextBox w oknie aplikacji. Dostęp do okna w jednym wariantcie będzie Nielimitowany, a w drugim zabezpieczony sekcją krytyczną.

Rozwiązanie

Utwórz nowy projekt aplikacji okienkowej według przykładu 1.4. Wstaw do okna pole tekstowe TextBox i przycisk Button. Właściwość Multiline kontrolki TextBox ustaw na true i zwiększ jej wymiary tak, aby mogła pomieścić kilka linii. Tak samo jak w poprzednim przykładzie będziemy potrzebować przestrzeni nazw System::Threading. Dołącz ją w bloku using namespace.

```
using namespace System::Drawing; //ta linia istnieje
using namespace System::Threading;
```

Aby metoda wątku mogła pisać do pola tekstowego, potrzebna jest metoda pisząca i delegat, tak jak to robiliśmy już wielokrotnie. Zasada sekcji krytycznej będzie lepiej widoczna, jeśli wszystkie napisy z wątków będą w jednej linii. Wprowadzimy uchwyt do zmiennej `System::String` `tekst`. Wątki będą dopisywać wyniki swojego działania do tej zmiennej i będzie ona wyświetlana w polu tekstowym. Poniższy kod wpisz po `#pragma endregion`:

```
#pragma endregion //ta linia istnieje    private: static System::String^ tekst;

    private: System::Void wyswietl1(System::String^ st)
    {
        tekst=tekst+" "+st;
        textBox1->AppendText(tekst+System::Environment::NewLine);
    }

    private: delegate void wyswDelegat(System::String^ st);
```

Metoda wątku będzie miała dwa warianty, które będziemy uruchamiać kolejno. W każdym wariantcie działanie wątku będzie polegało na wypisaniu liczb od 1 do 5, tyle że raz będzie to realizowane bez żadnych ograniczeń, a raz z zamknięciem dostępu do okna dla innych wątków. Kod wariantu, którego nie chcemy uruchamiać, oznaczymy jako komentarz. Oto cała metoda wątku:

```
private: System::Void watek(Object^ nr)
{
    wyswDelegat^ wyswietlDelegat1 = gcnew wyswDelegat(this,&Form1::wyswietl1);
    //wariant 1 bez kontroli
    for (System::Int32 j=1;j<5;j++)
        this->Invoke(wyswietlDelegat1, safe_cast<System::Object^>
            ↳(nr+"_"+j.ToString()+""));

    //wariant 2 sekcja krytyczna
    /*
        Monitor::Enter(this); //blokada dostępu do całej klasy Form1
    try {
        for (System::Int32 j=1;j<5;j++)
            this->Invoke(wyswietlDelegat1, safe_cast<System::Object^>
                ↳(nr+"_"+j.ToString()+""));
    }
    finally
        {Monitor::Exit(this);}
    */
}
```

Podstawowym działaniem jest wypisanie pięciu liczb do pola tekstowego w pętli `for` zgodnie z zasadami pisania do kontrolerek przez wątki, czyli z użyciem delegatów i metody `Invoke()`. Dodatkowo wypisywany jest też numer wątku. Schemat wpisu w jednym kroku pętli wygląda tak:

Numer Wątku_kolejna liczba

Na przykład liczba dwa wypisana przez wątek pierwszy będzie w postaci 1_2. W podanej formie uruchomi się wariant pierwszy, bo wariant drugi jest w bloku komentarza. Użycie klasy `Monitor` jest takie jak w wyjaśnieniach teoretycznych. Obiektem zamykanym do wyłącznego dostępu w sekcji krytycznej jest główne okno aplikacji, czyli obiekt klasy `Form1`, który jest dostępny we wskaźniku `this`, ponieważ metody wątku są metodami tej klasy. Pozostało zaprogramowanie naciśnięcia przycisku `button1`. Będzie on tworzył trzy wątki i uruchamiał je. Naciśnij dwukrotnie ten przycisk w widoku projektowania i uzupełnij metodę `button1_Click()` jak niżej:

```
private: System::Void button1_Click(System::Object^ sender, System::EventArgs^ e) {
    Thread^ t1 = gcnew Thread(gcnew ParameterizedThreadStart(this,&Form1::watek));
    t1->Start(1);

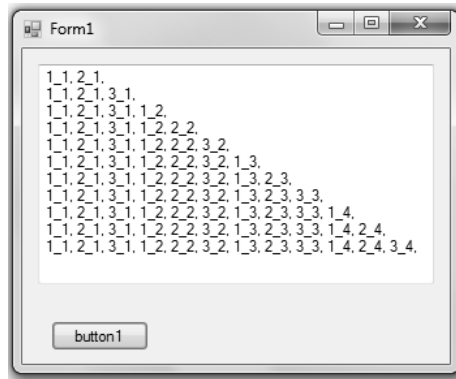
    Thread^ t2 = gcnew Thread(gcnew ParameterizedThreadStart(this,&Form1::watek));
    t2->Start(2);

    Thread^ t3 = gcnew Thread(gcnew ParameterizedThreadStart(this,&Form1::watek));
    t3->Start(3);
}
```

Uruchom aplikację i naciśnij przycisk w oknie. W polu tekstowym zobaczysz napisy jak na rysunku 19.3.

Rysunek 19.3.

*Wynik
działania aplikacji
bez sekcji krytycznej*



W ostatniej linii masz wszystkie napisy wygenerowane przez wątki. Prawdopodobnie (nie jest to w 100% pewne, bo zależy od szybkości wykonywania niezależnych wątków) będziesz miał tam liczbę jeden wpisana przez wątek pierwszy, drugi i trzeci, następnie liczbę dwa wpisana przez kolejne wątki i tak dalej. Na rysunku 19.3 właśnie tak jest. Teraz uruchomimy wariant drugi. Zdejmij znaki komentarzy `/* */` z tego wariantu, a wariant pierwszy zasłoń znakiem komentarza.

```
private: System::Void watek(Object^ nr)
{
    wyswDelegat^ wyswietlDelegat1 = gcnew wyswDelegat(this,&Form1::wyswietl1);
    //wariant 1 bez kontroli
    //for (System::Int32 j=1;j<5;j++)
    //this->Invoke(wyswietlDelegat1, safe_cast<System::Object^>
    //((nr+"_"+j.ToString()+"),"));

    //wariant 2 sekcja krytyczna
    Monitor::Enter(this); //blokada dostępu do całej klasy Form1
}
```

```

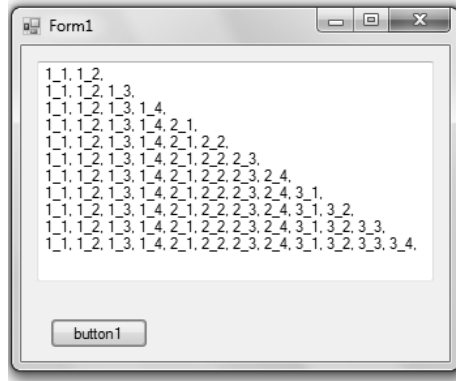
try {
    for (System::Int32 j=1;j<5;j++)
        this->Invoke(wyswietlDelegat1, safe_cast<System::Object^>
            ↳(nr+"_"+j.ToString()+""));
}
finally
{Monitor::Exit(this);}
}

```

Uruchom aplikację. Teraz napisy są jak na rysunku 19.4.

Rysunek 19.4.

*Aplikacja z wątkami
w sekcji krytycznej*



W ostatniej linii widać (teraz mamy pewność, że tak jest), że najpierw swoje liczby wpisał wątek pierwszy, następnie drugi i trzeci. Podczas kiedy jeden wątek pisał, inne były zawieszane. Możliwa jest zmiana kolejności wykonywania wątków. Zależy to od wielu czynników, ale kiedy wątek zostanie dopuszczony do sekcji krytycznej, będzie mógł spokojnie działać, mając wyłączny dostęp do okna.

Komponent BackgroundWorker

Komponent BackgroundWorker jest kolejną możliwością implementacji wielowątkowości. Za jego pomocą można uruchamiać metody w wątkach i kontrolować ich wykonanie. Jego funkcjonowanie opiera się na zdarzeniach. Tabela 19.1 przedstawia trzy ważne zdarzenia związane z tym komponentem.

Tabela 19.1. Zdarzenia komponentu BackgroundWorker

Zdarzenie	Opis
DoWork	Zdarzenie generowane przy rozpoczęciu działania wątku. Metoda obsługi tego zdarzenia uruchamia metodę wątku.
ProgressChanged	Zdarzenie występujące w trakcie działania wątku, po wywołaniu metody ReportProgress(). Może być użyte do pokazywania postępu wykonania wątku lub do innych celów wymagających komunikacji z komponentami okna głównego.
RunWorkerCompleted	Zdarzenie to występuje po zakończeniu pracy przez metodę wątku.

Najczęściej używane metody tego komponentu przedstawia tabela 19.2.

Tabela 19.2. *Niektóre metody komponentu BackgroundWorker*

Metoda	Działanie
RunWorkerAsync() RunWorkerAsync (Object^ parametr)	Generuje zdarzenie DoWork, które uruchamia proces w tle. Wersja z parametrem przekazuje <i>parametr</i> do metody wątku.
ReportProgress(int <i>postęp</i>)	Generuje zdarzenie ProgressChanged. Przekazuje do niego liczbę typu int, która może wyrażać stopień zaawansowania wątku.
CancelAsync()	Metoda do przerywania działania wątku. Nie przerywa ona jego działania natychmiast, a jedynie ustawia właściwość CancellationPending na true. Metoda wątku powinna okresowo sprawdzać tę właściwość i przerwać wykonywanie wątku.

Obiekt BackgroundWorker posiada także właściwości kontrolujące jego zachowanie. Prezentuje je tabela 19.3.

Tabela 19.3. *Właściwości komponentu BackgroundWorker*

Właściwość	Opis
WorkerReportsProgress	Wartość true powoduje, że można korzystać z metody ReportProgress().
WorkerSupportsCancellation	Umożliwia działanie mechanizmu przerywania wątku.
CancellationPending	Wartość true oznacza, że wywołano metodę CancelAsync().

Napiszemy teraz program obliczający w wątku średnią z liczb podanych w tabeli.

Przykład 19.9

Napisz program obliczający średnią z liczb podanych w tabeli. Użyj komponentu BackgroundWorker. Tabela powinna być przekazywana do metody wątku jako parametr.

Rozwiązanie

Utwórz nowy projekt aplikacji według przykładu 1.4 i wstaw do niego przycisk Button, etykietę Label i komponent BackgroundWorker. Ten ostatni znajdziesz w dziale *Components* okna narzędziowego.

Zacznij od oprogramowania zdarzenia Click przycisku Button. W metodzie zdefiniuj tablicę liczb do określenia średniej i wywołaj metodę RunWorkerAsync(), przekazując tę tablicę jako parametr.

```
private: System::Void button1_Click(System::Object^ sender, System::EventArgs^ e) {
    array<System::Single>^ tablica =
        gcnew array<System::Single>(5) {2,2,3,4,5};
    backgroundWorker1->RunWorkerAsync(tablica);
}
```

Metoda `RunWorkerAsync()` wywoła zdarzenie `DoWork` i przekaże parametr do metody obsługującej to zdarzenie. Aby utworzyć metodę obsługi zdarzenia, zaznacz myszką komponent `BackgroundWorker` na pasku niewidocznych komponentów, a następnie przejdź w prawym panelu do widoku zdarzeń i kliknij dwukrotnie zdarzenie `DoWork`. Parametr `sender` metody obsługi zawiera uchwyt do obiektu `BackgroundWorker`, który wywołał zdarzenie `DoWork`. W metodzie obsługi należy rzutować w górę parametr `sender` do obiektu `BackgroundWorker`, aby zapisać ten uchwyt. Następnie wywołujemy metodę wątku. Parametrem metody wątku jest między innymi uchwyt do obiektu `BackgroundWorker` pobrany z parametru `sender`. Wynik działania metody podstawiamy pod pole `Result` drugiego z parametrów metody obsługi zdarzenia `DoWork`.

```
private: System::Void backgroundWorker1_DoWork(System::Object^ sender,
    System::ComponentModel::DoWorkEventArgs^ e) {
    BackgroundWorker^ back_worker = dynamic_cast<BackgroundWorker^>(sender);
    e->Result=watek( safe_cast<array<System::Single^>>(e->Argument),
        back_worker, e );
}
```

Wreszcie sama metoda wątku:

```
private: System::Single watek(array<System::Single^> n, BackgroundWorker^ worker,
    DoWorkEventArgs^ e) {
    System::Single suma;
    for (System::Int16 i=0;i<n->Length;i++)
        suma+=n[i];
    return suma/n->Length;
}
```

Po zakończeniu jej wykonywania wystąpi zdarzenie `RunWorkerCompleted`. Do jego obsługi będziesz potrzebować metody, która wyświetli wartość zwróconą przez metodę wątku. Wartość ta została przekazana przez metodę obsługi zdarzenia `DoWork` do zmiennej `e`, z niej będziemy ją odczytywać. Kliknij pojedynczo na komponent `BackgroundWorker` w pasku niewidocznych komponentów, a następnie przełącz na widok zdarzeń i znajdź zdarzenie `RunWorkerCompleted`. Kliknij po prawej stronie tego zdarzenia, tworząc metodę obsługi, w którą wpisz kod jak niżej:

```
private: System::Void backgroundWorker1_RunWorkerCompleted(System::Object^ sender,
    System::ComponentModel::RunWorkerCompletedEventArgs^ e) {
    label1->Text=e->Result->ToString();
}
```

Po uruchomieniu programu otrzymamy średnią z liczb z tablicy wypisaną w etykiecie. Obliczanie średniej zajmuje mało czasu, dlatego nie widać tu zalet programowania wielowątkowego, ale przy długich procesach jest ono koniecznością.

W celu uzyskania większej kontroli nad procesem wątku niezbędna jest możliwość przerywania tego procesu, a także kontroli postępów jego wykonania. Komponent `BackgroundWorker` posiada mechanizmy, które to umożliwiają.

Przykład 19.10

Za pomocą komponentu `BackgroundWorker` napisz aplikację wyszukującą liczbę pierwszą najbardziej zbliżoną do zadanej. Program ma mieć możliwość przerywania obliczeń w dowolnym momencie, a także powinien informować o zaawansowaniu procesu.

Rozwiązanie

Utwórz nowy projekt aplikacji według przykładu 1.4. Wstaw do okna dwa przyciski `Button`, etykietę `Label`, komponent `BackgroundWorker`, wskaźnik postępu `ProgressBar` (dział *Common Controls* na pasku narzędziowym) oraz pole tekstowe `TextBox`.

We właściwość `Text` przycisku `button1` wpisz *Licz*, a `button2` — *Anuluj*.

Ponieważ będziemy używać mechanizmów raportowania zaawansowania procesu i jego przerywania, za pomocą panelu *Properties* ustaw właściwości `WorkerReportsProgress` i `WorkerSupportsCancellation` obiektu `backgroundWorker1` na `true`.

Sam program będzie wyglądał podobnie jak poprzednio, z tym że znajdują się tu nowe elementy. Zaczynij od metody zdarzenia `Click` pierwszego przycisku `button1`.

```
private: System::Void button1_Click(System::Object^ sender, System::EventArgs^ e) {
    backgroundWorker1->RunWorkerAsync(Convert::ToInt32(textBox1->Text));
}
```

Jako parametr do metody wątku przekazujemy liczbę pobraną z pola tekstowego. Jest to liczba, w okolicy której poszukujemy liczby pierwszej.

Metoda `RunWorkerAsync()` wywołuje zdarzenie `DoWork`, które będziemy obsługiwać za pomocą poniższej metody:

```
private: System::Void backgroundWorker1_DoWork(System::Object^ sender,
↳ System::ComponentModel::DoWorkEventArgs^ e) {
    BackgroundWorker^ back_worker = dynamic_cast<BackgroundWorker^>(sender);
    e->Result = watek( safe_cast<System::Int32>(e->Argument), back_worker, e );
}
```

Metoda różni się od poprzedniego przykładu jedynie typem argumentu (teraz jest to zmienna typu `System::Int32`, a nie tablica typu `System::Single`, jak poprzednio). Teraz napisz samą metodę wątku:

```
private: System::Single watek(System::Int32 i_max, BackgroundWorker^ worker,
↳ DoWorkEventArgs^ e) {
    System::Single liczba;
    System::Int32 n=2;
    System::Int32 procent;
    for (System::Int32 i=2; i<i_max; i++) {
        if (worker->CancellationPending==true) {
            e->Cancel=true; return liczba;
        }
        else {
            n=2;
            while ((i%n))
                n++;
        }
    }
}
```

```

        if (i==n)
            liczba=i; // ostatnia znaleziona
        }
        procent=(int)((float)i/(float)i_max*100);
        worker->ReportProgress(procent);
    }
    return liczba;
}

```

Tu mamy największe zmiany w stosunku do poprzedniego przykładu. W każdym kroku pętli jest sprawdzana właściwość `CancellationPending`. Zmienia ona wartość na `true` w przypadku wywołania metody `CancelAsync()`; jest to znak, że użytkownik chce przerwać działanie wątku. W takim przypadku poprzez parametr `e` informacja ta zostaje przekazana dalej, a instrukcja `return` powoduje opuszczenie metody wątku. Również w każdym kroku jest obliczane zaawansowanie procesu na podstawie aktualnej wartości zmiennej sterującej pętlą. Wartość zaawansowania jest przekazywana poprzez wywołanie metody `ReportProgress()`, która wywołuje zdarzenie `ProgressChanged`. Zauważ, że cała komunikacja między metodami odbywa się poprzez parametry metod.

Utwórz metodę obsługującą zdarzenie `ProgressChanged`, tak jak to robiłeś dla zdarzenia `DoWork`, a następnie doprowadź ją do postaci jak niżej:

```

private: System::Void backgroundWorker1_ProgressChanged(System::Object^ sender,
↳System::ComponentModel::ProgressChangedEventArgs^ e) {
    progressBar1->Value=e->ProgressPercentage;
}

```

Metoda `CancelAsync()` jest wywoływana przez naciśnięcie drugiego przycisku.

```

private: System::Void button2_Click(System::Object^ sender, System::EventArgs^ e) {
    backgroundWorker1->CancelAsync();
}

```

Po zakończeniu działania wątku nastąpi zdarzenie `RunWorkerCompleted`; w metodzie jego obsługi wypiszemy wynik lub informację, że proces został przerwany.

```

private: System::Void backgroundWorker1_RunWorkerCompleted(System::Object^ sender,
↳System::ComponentModel::RunWorkerCompletedEventArgs^ e) {
    if (e->Cancelled==true)
        label1->Text="Anulowano";
    else
        label1->Text=e->Result->ToString();
}

```

Po uruchomieniu programu i wpisaniu liczby w pole tekstowe otrzymamy liczbę pierwszą poprzedzającą wpisaną liczbę. O przebiegu poszukiwań informuje pasek, a przycisk *Anuluj* pozwala anulować obliczenia.

Skorowidz

A

- adapter, 291
- adres serwera bazy, 291
- animacja, 316
- animacja figury, 161
- ANSI, 140
- aplikacja
 - z semaforem, 330
 - zabezpieczona hasłem, 365
- aplikacje
 - .NET Framework, 11
 - Metro, 11
 - Windows, 16, 18
- argument funkcji, 14
- automatyczne
 - dopasowanie komórek, 255
 - rozpoznawanie typu, 45

B

- baza danych, 277
 - postgres, 283
 - serwisu samochodów, 283
 - PostgreSQL, 278
- biblioteka
 - .NET Framework, 165
 - Npgsql.dll, 289
- blok
 - catch, 92
 - try, 246
- błąd odczytu, 110
- błędne dane, 246
- błędny przydział pamięci, 85

C

- CIL, Common Intermediate Language, 12
- czas systemowy, 299
- czasomierz, 301, 363
- czcionka, 234

D

- debugowanie, 24
- definicja
 - destruktora, 97
 - dziedziczenia, 80
 - konstruktora, 97
 - obiektu, 111
 - przeciążania, 88
 - struktury, 73, 75
 - szablonu, 89
 - zasobu okna, 131
- definicje metod klasy, 76
- deklaracja
 - delegata, 64
 - funkcji, 59
 - szablonu, 70
 - tablic, 54, 203, 210
 - uchwyty do okna, 119
 - wskaznika do funkcji, 63
- deklaracje dynamiczne, 209
- delegat
 - FormClosingEventHandler, 366
- delegaty, 64
- destruktor, 97, 108
- dodawanie
 - danych do bazy, 292
 - klas do projektu, 353
 - kolumn, 263

- sterowników PostgreSQL, 288
- wierszy, 263
- dostęp do
 - elementów tablicy, 206
 - składowych, 73
- drzewo plików, 372
- DS, dialog style, 132
- dynamiczna
 - deklaracja obiektów, 209
 - deklaracja tablic, 210
 - tabela łańcuchów, 240
- dynamiczne tworzenie komponentów, 359
- dyrektywa
 - #define, 33, 123
 - #ifdef, 34
 - #include, 30
 - #pragma endregion, 290, 300, 314, 329
 - using, 31
 - using namespace, 289
 - using namespace std, 94
- dziedziczenie, 80

E

- edytor
 - ASCII, 229
 - ikon, 126
 - kolumn, 265
 - menu, 191
 - zasobów, 125
- elementy zarządzane, managed, 12
- enumerator, 206
- etykiety, 173

F

folder Npgsql, 288
 formatowanie folderu, 369
 funkcja, 14, 59
 BeginPaint(), 157
 Button_GetCheck(), 154
 buttonI_Click (), 232, 245
 CreateWindow(), 118, 141
 DialogBox(), 135
 DispatchMessage(), 122
 DodajTekst(), 151
 FromFile(), 201
 getch(), 49
 GetClientRect(), 162
 GetCommandLineArgs(), 69
 GetMessage(), 122
 GetResponse(), 349
 InitInstance(), 26, 119
 InvalidateRect(), 145, 159
 LoadCursor(), 117
 LoadIcon(), 117
 main(), 30, 67
 MyRegisterClass(), 115, 125
 RegisterClassEx(), 118
 rozpoznaj(), 346
 SendMessage(), 143, 150
 SetTimer(), 160
 ShowWindow(), 120
 t_main(), 32
 TranslateAccelerator(), 123
 TranslateMessage(), 122
 tWinMain(), 120
 WndProc(), 135, 154, 163
 WyswietlFold(), 368

funkcje
 GDI, 158
 główne, 115
 operatorowe, 89
 wirtualne, 83
 zwrotne, 114

funkcji
 deklaracja, 59
 przeciążanie, 60
 przekazywanie argumentów,
 61
 szablony, 70
 wskaźnik na adres, 63
 wywołanie, 60
 wywołanie przez wskaźnik, 63

G

GDI, Graphical Device
 Interface, 157

H

hasło, 365
 hermetyzacja danych, 77, 110
 hierarchia
 klas, 85
 wywołań, 28

I

IDE, Integrated Development
 Environment, 29
 identyfikator, 117
 identyfikatory ikon, 126
 ikona, 123, 126
 implementacja FTP, 347
 informacje o procesie, 337
 inicjalizacja
 bazy, 281
 obiektu struktury, 111
 pól, 75
 instalacja PostgreSQL, 277
 instalator Visual Studio, 13
 instancja, instance, 114
 instrukcja
 break, 58
 delete, 98
 do...while, 57
 for, 57
 if...else, 56
 switch, 56
 system(„pause”), 49
 throw, 92
 try, 92
 while, 57
 instrukcje
 iteracji, 57
 warunkowe, 56
 interface
 win32, 113
 managera plików, 367
 użytkownika, 286

J

jawna konwersja, 270
 język
 C++/CLI, 12
 CIL, 12
 SQL, 284

K

kalkulator, 184
 kapsułkowanie, 110

kasowanie

danych, 297
 plików, 375
 klasa, 15, 75
 array, 210
 BinaryWriter, 222
 Bitmap, 201
 Button, 171
 ContextMenuStrip, 187
 Convert, 242
 CultureInfo, 244
 DataGridViewComboBox
 ↳Cell, 273
 DateTime, 299
 Directory, 213
 DirectoryInfo, 214
 enum, 43
 Environment, 69
 File, 214
 FileInfo, 214
 Form, 172, 359
 Form1, 196, 296, 322
 HtmlDocument, 342
 MenuStrip, 187
 MessageBox, 225
 Monitor, 331
 NpgsqlDataAdapter, 291
 okna głównego, 115
 OpenFileDialog, 227
 pobierz_wyslij, 355
 StreamReader, 217
 StreamWriter, 221
 SzukLiczbPierw, 325
 TextBox, 175
 Thread, 323, 325
 WNDCLASSEX, 115

klasy
 bazowe, 81, 85
 dziedziczone, 104
 pochodne, 81, 85
 klawisze skrótów, 128
 klucz główny, 285
 kod zarządzany, 12
 kodowanie, 140
 ASCII, 213
 Unicode, 69, 140
 UTF-7, 224

kolor
 czcionki, 236
 etykiety, 274
 pędzla, 310
 tła, 234
 wierszy, 254

- komenda
 - cmd, 282
 - CREATE TABLE, 285
 - DELETE, 297
 - INSERT, 292, 296
 - psql, 286
 - SELECT, 290–293
 - UPDATE, 295–297
 - komórka
 - DataGridViewButtonCell, 268
 - DataGridViewCheckBoxCell, 269
 - DataGridViewComboBox
 - Cell, 272
 - DataGridViewImageCell, 270
 - DataGridViewLinkCell, 275
 - kompilacja warunkowa, 34
 - komponent
 - BackgroundWorker, 334
 - MenuStrip, 187
 - OpenFileDialog, 358
 - Timer, 301, 363
 - WebBrowser, 339
 - komponenty tworzone dynamicznie, 364
 - komunikat, message, 16, 114, 139
 - BM_GETCHECK, 154
 - EM_GETSEL, 151
 - EM_REPLACESEL, 152
 - EM_SETSEL, 151
 - WM_COMMAND, 122, 142, 151, 162
 - WM_CREATE, 149, 158
 - WM_DESTROY, 122
 - WM_MOUSEMOVE, 146
 - WM_PAINT, 122, 145, 161
 - WM_SETTEXT, 150
 - komunikaty
 - kontrolki ComboBox, 156
 - tekstowe, 151
 - konfigurator Stack Builder, 280
 - konstruktor, 97
 - bezparametrowy, 100
 - domyślny, 97
 - klasy, 48
 - klasy dziedzicznej, 104
 - klasy pochodnej, 104
 - kopiujący, 100, 106
 - przenoszący, 102
 - w szablonie klasy, 107
 - kontekst, 157
 - kontener GroupBox, 184
 - kontrola błędu, 108
 - kontrolka
 - ComboBox, 155
 - DataGridView, 249, 256
 - GroupBox, 183
 - Listbox, 356
 - Panel, 183
 - Picture Control, 137
 - tekstowa, 133
 - WebBrowser, 339, 343
 - konwersja
 - liczby, 147, 246
 - typów, 242
 - ze zmianą formatu, 243
 - kopiowanie
 - obiektu, 100
 - plików, 374
 - wierszy, 264
- L**
- liczba
 - parametrów, 115
 - zaznaczonych komórek, 259
 - liczby makr, 124
 - liczby pierwsze, 319, 337
 - link do strony, 174
 - lista
 - ComboBox, 157, 191
 - rozszerzeń bazy, 280
 - rozwijana, 272
 - l-wartość, 46
- Ł**
- łańcuch
 - formatujący, 247
 - połączenia, connection string, 291
 - znakowy, 140, 176
 - łączenie się z bazą, 279, 290
- M**
- macierz transformacji, 312
 - makra standardowe, 37
 - makro
 - _MSC_VER, 35
 - dla ikony, 124
 - MAKEINTRESOURCE, 117
 - malowanie pędzlem, 311
 - manager plików, 367
 - maska, 180, 182
 - mechanizm
 - garbage collector, 12
 - precompiled headers, 30, 32
 - przeciążania funkcji, 60
 - wyjątków, 245
 - Menedżer zadań, 321
 - menu, 187, 191
 - menu podręczne, 193
 - metoda, 15
 - Add(), 361
 - AppendText(), 175
 - AutoSizeColumns(), 256
 - AutoSizeRows(), 256
 - button1_Click(), 171, 176, 218, 304, 333, 375
 - button2_Click(), 263
 - button3_Click(), 297
 - CancelAsync(), 338
 - Clear(), 293
 - Close(), 172, 298
 - Commit(), 293
 - Copy(), 374
 - CreateGraphics(), 303
 - CreateInstance(), 205
 - CreateSpecificCulture(), 244
 - Current(), 206
 - dataGridView1_
 - CellEndEdit(), 274
 - dataGridView1_Click(), 269
 - dataGridView2_SelectionC
 - hanged(), 295
 - DrawCurve(), 307
 - DrawImage(), 313
 - ExecuteNonQuery(), 292, 294
 - Exit(), 331
 - Fill(), 291
 - FillPie(), 311
 - Form1_Load(), 252, 270, 290, 368
 - Form1_Paint(), 315
 - FormatWpisFolder(), 370
 - FormatWpisPlik(), 371
 - GetAttribute(), 343
 - GetDirectories(), 369, 373
 - GetEncoding(), 218
 - GetEnumerator(), 206
 - GetFiles(), 214–216, 369
 - GetType(), 193
 - IndexOf(), 240
 - InitializeComponent(), 364
 - InsertCopy(), 264
 - Invoke(), 321
 - InvokeScript(), 345
 - KopiujPlik(), 375

- metoda
- LastIndexOf(), 357
 - listBox1_Click(), 373
 - MoveNext(), 206, 207
 - odśwież(), 292
 - opcja1ToolStripMenuItem_Click(), 190, 193
 - Parse(), 180, 218, 243
 - Peek(), 219
 - przycisk_Click(), 365
 - Read(), 219
 - ReadLine(), 218
 - ReadToEnd(), 217
 - Release(), 328
 - Release(), 329
 - Remove(n), 373
 - ReportProgress(), 338
 - Reverse(), 205
 - Rollback(), 293
 - rotate(), 317
 - Rotate(), 312
 - RunWorkerAsync(), 336, 337
 - SetAttribute(), 343
 - SetPixel(), 313
 - SetValue(), 205
 - Show(), 225, 359
 - ShowDialog(), 227, 360
 - Start(), 174, 324
 - Substring(), 357
 - timer1_Tick(), 317
 - toolStripButton1_Click(), 199
 - ToSingle(), 220
 - ToString(), 180, 220, 247
 - Translate(), 312
 - WaitOne(), 328
 - watek(), 322
 - wątku, 332, 336
 - Write(), 220
 - WriteLine(), 31, 220
 - wyslij(), 358
 - wyswietl(), 206
 - WyswietlFold(), 368
- metody
- działające na tablicach, 204
 - klasy System
 - String, 240
 - komponentu
 - BackgroundWorker, 335
 - obiektu FtpWebRequest, 348
 - obiektu WebBrowser, 341
 - odczytu pliku, 217
 - odczytujące, 223
 - operacji na kolumnach, 262
 - operacji na wierszach, 261
 - reakcji, 23
 - rysujące, 305
 - statyczne, 31, 78, 179
 - szablonu, 91
 - transformujące, 312
 - wirtualne, 83
 - zmieniające zawartość pól, 295
- modyfikator const, 51
- modyfikatory typów, 38
- ## N
- nagłówek
- conio.h, 49
 - fstream, 49
 - iostream, 49
- nawiasy
- klamrowe, 53
 - kwadratowe, 67
- nawigacja po folderach, 372
- nazwa
- bazy danych, 291
 - koloru, 117
 - projektu, 288
 - przestrzeni, 288
 - użytkownika, 291
- ## O
- obiekt, 16
- BindingSource, 292
 - Bitmap, 313
 - connection, 290
 - DataSet, 292
 - Graphics, 303, 305
 - HtmlDocument, 342
 - HtmlElement, 344
 - Image, 308
 - inscomm, 292
 - Matrix, 312
 - NpgsqlCommand, 292
 - okno, 331
 - selectcomm, 293
 - Semaphore, 328
 - this, 196
 - Thread, 321
 - TimeSpan, 300
- obiekty graficzne na stronie, 343
- obliczanie średniej, 335
- obsługa
- bazy, 298
 - błędów, 92
 - FTP, 352
 - komunikatów, 139
 - przycisków Button, 154
 - wyjątku, 93
 - zdarzenia Click, 192, 216
- obszar projektowania, 287
- odczyt
- z komórki, 257
 - z pliku, 49, 223
- odnośniki internetowe, 274
- odśmiecacz, garbage collector, 208
- odświeżanie
- okna, 145
 - rysunku, 314
- okno, 114, 166
- Add Class, 353
 - Add Resource, 136
 - DIALOGEX, 134
 - dialogowe, 131, 133, 135
 - EditColumns, 265
 - edycji, 142
 - FontDialog, 235
 - główne, 113, 166, 196
 - hierarchii wywołań, 28
 - IDE, 17
 - klienta FTP, 355
 - komunikatów, 23
 - MessageBox, 225
 - nowego projektu, 20, 21
 - OpenFileDialog, 227, 228
 - przeglądania folderów, 231
 - SaveFileDialog, 230
 - Stack Buildera, 279
 - tytułowe aplikacji, 363
 - VC++, 23
 - wyboru czcionki, 184, 235
 - wyboru folderu, 233
 - wyboru koloru, 233
 - z komunikatami, 144
 - zapisu pliku, 230
- operator
- %, 208
 - &, 84
 - <<, 32
 - =, 101
 - >>, 218
 - delete, 55, 209
 - dostępu ., 73
 - dostępu ->, 73, 98, 173
 - gcnw, 55, 209, 359

new, 55, 98, 209
 przekierowania, 48
 uzyskania wielkości obiektu, 47
 zasięgu ::, 78, 94

operatory
 arytmetyczne, 47
 dostępu, 47
 jednoargumentowe, 47
 logiczne, 47
 porównania, 47
 przypisania, 47

optymalizacja pamięci, 208

P

panel
 Properties, 138
 Toolbox, 137
 z kontrolkami, 137

parametry
 linii komend, 70
 metody Show(), 225

pasek
 niewidocznych komponentów, 287
 ToolStrip, 197

paski narzędzi, 197

pędzle, 310

pętla komunikatów, 114, 122

pióro Pen, 308

plik
 Form1.h, 31, 213
 okienko.rc, 124
 okno.cpp, 114
 pobierz_wyslij.cpp, 354
 pobierz_wyslij.h, 353
 pomoc.html, 340
 README, 357
 resource.h, 124
 skrypt1.htm, 346
 stdafx.h, 31
 windows.h, 115
 WindowsFormApplication1.cpp, 169, 288
 winuser.h, 117

pliki
 .cpp, 25, 30
 .exe, 216
 .h, 30
 .ico, 24, 124
 .rc, 24
 .sdf, 24
 .sln, 24

.suo, 24
 .vcxproj, 25
 .vcxproj.filters, 25
 binarne, 213
 tekstowe, 213

pobieranie
 danych z komponentów, 362
 plików, 350, 356
 z serwera, 349

podgląd znaczników kontekstu, 344

połączenie referencji do biblioteki, 289

podświetlanie, 25

pola
 maskowane, 181
 statyczne, 78

pole, 15
 tekstowe TextBox, 175, 180, 184, 237
 typu SERIAL, 286
 wyboru, 154, 269

predykat, 205
 binarny, 205
 unarny, 205

prekompilowane nagłówki, 31

PRIMARY KEY, 285

procedura
 okna, 120
 WndProc(), 143

program
 createdb, 283
 GIMP, 124
 initdb, 282, 283
 Paint, 168
 pg_ctl, 282
 psql, 286

projekt okienkowy
 C++/CLI, 21, 29
 win32, 21, 29, 114

protokół FTP, 347

przeciążanie
 funkcji, 60
 konstruktorów, 99
 operatorów, 88

przedstawianie danych, 249

przekazywanie argumentów
 przez wartość, 61
 za pomocą adresu, 61

przekazywanie parametrów, 321
 do funkcji main(), 68
 do metody wątku, 323
 z kontrolą typu, 324

przełączanie kolorów, 198

przepuszczanie wątków, 329

przeźreń nazw, 16, 31, 94
 System::Threading, 328
 System::Net, 348
 System::IO, 348

przesyłanie danych, 362

przetwarzanie
 komunikatów, 120
 stron, 342

przycisk, 171
 <-Kasuj, 376
 <-Kopiuj, 374
 Button, 153
 Kopiuj->, 374
 toolStripButton, 200

przyciski
 domyślne, 133
 graficzne, 201
 opcji, 184
 w komórkach, 268

przyporządkowanie metody do zdarzenia, 364

R

RC, Release candidate, 9

referencje, 50

referencje do r-wartości, 51

rejestracja klasy, 114

rodzaje menu, 187

rodzaje piór, 310

rola, role, 283

rola administratora, 283

rozmiar
 komórek, 255
 tabeli, 258

r-wartość, 46

rysowanie, 303
 figur, 160
 linii, 304
 punktów, 313
 tekstu, 306
 trwałe, 314
 wykresów, 307

rysunek na przycisku, 200

rzutowanie
 const_cast, 40, 51
 dynamic_cast, 41
 dynamiczne, 85
 jawne, 44
 niejawne, 40
 safe_cast, 41
 static_cast, 39

rzutowanie

- statyczne, 85
- typów, 39
- w dół, 85
- w górę, 85

S

- sekcje krytyczne, 331, 334
- semafory, 328
- semantyka przenoszenia, 102
- serwer FTP, 348
 - pobranie pliku, 350
 - wysyłanie pliku, 351
- siatka DataGridView, 262, 271
- skrótów klawiaturowe, 195
- skrypt zasobów, 117, 124
- skrypty JavaScript, 345
- słowo
 - delegate, 64
 - DIALOGEX, 131
 - Icon, 126
 - instancja, 114
 - MENUITEM, 128
 - mutable, 67
 - operator, 88
- słowo kluczowe
 - auto, 45
 - const, 41
 - POPUP, 128
 - static, 78
 - struct, 110
 - template, 89
 - typeid, 45
 - virtual, 83
- sortowanie, 260
- specyfikator public:, 110
- sprawdzanie poprawności
 - hasła, 366
- SQL, Structured Query Language, 284
- sterowniki PostgreSQL, 288
- sterta, 55
- struktura, 73
 - DirectoryInfo, 369
 - do przechowywania danych, 281
 - procedury okna, 121
 - projektu, 24
- strumień cout, 32
- styl
 - BS_AUTOCHECKBOX, 154
 - DS_FIXEDSYS, 132

- DS_MODALFRAME, 132
- DS_SETFONT, 132
- WS_OVERLAPPED
 - ↳ WINDOW, 119

Styl Metro, 11

style

- łańcucha znakowego, 244
- okna, 118

sygnalizacja wystąpienia komunikatu, 145

szablon

- klasy, 89-91
- projektu okienkowego, 19
- funkcji, 70

Ś

środowisko

- .NET Framework, 12
- IDE, 22, 29

T

tabela

- DataGridView, 252, 287
- dynamiczna, 266
- odnośników internetowych, 276
- znaków TCHAR, 147

tablica, 52, 203

- dwuwymiarowa, 54
- jednowymiarowa, 53

tekst, 239

- kopiowanie, 239
- wklejanie, 239
- wstawianie, 241
- wycinanie, 239

timer, 160, 317

tryb

- dialogowy, 362
- dziedziczenia, 80
- kompilacji, 24
- Release, 24
- wizualny, 165
- zdarzeń, 23

tworzenie

- bazy danych, 285
- dynamiczne okien, 359
- ikony, 125
- kalkulatora, 184
- menu, 130, 191
- menu głównego, 189
- menu podręcznego, 194
- obiektu, 115

obiektu Graphics, 303

- obiektu klasy potomnej, 105
- okna aplikacji, 168
- okna edycji, 142
- okna dialogowego, 133, 135
- projektu, 19
- przycisku, 133
- struktury folderów, 282
- tabeli, 271
- tabeli dynamicznej, 267
- timer, 160
- tymczasowego obiektu, 104
- wątku, 321, 357
- typ NumberStyles, 244
- typy
 - danych, 178
 - danych w PostgreSQL, 284
 - komórek w tabeli, 265
 - wskaźnikowe, 51
 - wyliczeniowe, 41
 - zmiennych, 37

U

uchwyt, 208

uchwyt do obiektu semafora, 329

uruchamianie bazy, 281

ustawienie Debug, 24

usuwanie

- obiektów, 105
- wiersza, 263

V

Visual Studio, 11

W

wartość

- NULL, 87, 103
- void, 63
- zwracana, 14

wątek, 319

- pobieranie plików, 356
- wysyłanie plików, 357

wczytywanie pliku do pola, 228

wersja kompilatora, 35

wersja RC VC++ 2012, 9

wiązanie późne, 83

widok

- projektowania aplikacji, 256
- RESOURCE VIEW, 125

- WinAPI, 113
 - właściwości, properties, 16
 - klasy FileInfo, 215
 - komórki
 - DataGridViewComboBox
 - ↳ Cell, 272
 - DataGridViewImageCell, 270
 - DataGridViewLinkCell, 275
 - komponentu, 170
 - BackgroundWorker, 335
 - FolderBrowserDialog, 231
 - MaskedTextBox, 181
 - MenuStrip, 188
 - ToolStripMenuItem, 195
 - kontrolki
 - CheckBox, 183
 - DataGridView, 249, 257
 - RadioButton, 183
 - TextBox, 175
 - WebBrowser, 339
 - obiekту
 - DataGridViewCell, 251
 - DataGridViewRow, 250
 - DateTime, 299
 - FtpWebRequest, 347
 - HtmlDocument, 342
 - Pen, 308
 - Timer, 301
 - okna, 137
 - aplikacji, 166
 - ColorDialog, 233
 - FontDialog, 235
 - OpenFileDialog, 228
 - SaveFileDialog, 230
 - paska ToolStrip, 197
 - pędzla TextureBrush, 310
 - pola TextBox, 237
 - właściwość
 - DataGridViewCellStyle, 253
 - ImageScalingSize, 200
 - WM, Windows Message, 122
 - wprowadzanie danych, 176, 242, 294
 - WS, window style, 118
 - wskaźnik, 50
 - myszy, 146
 - postępu, 337
 - this, 79, 304
 - wskaźniki
 - do funkcji, 62
 - do obiektu, 86
 - do stałej, 51
 - na klasy, 85
 - na zmienne, 63
 - współrzędne wskaźnika, 148
 - wstawianie siatki, 266, 271, 307
 - wyjątek, 92, 296
 - wyliczenia, 42
 - wyłączanie usługi bazy, 281
 - wyrażenia lambda, 65
 - wysyłanie
 - komunikatów, 140
 - plików, 351, 357
 - wyszukiwanie, 26
 - liczb pierwszych, 324
 - plików, 214
 - w tablicy, 206
 - znaków, 240
 - wyświetlanie
 - argumentów linii komend, 69
 - czasu, 302
 - figur, 158
 - grafiki w komórkach, 272
 - katalogu serwera FTP, 355
 - liczb pierwszych, 322
 - nazwy przeglądarki, 346
 - obrazka, 308
 - odnośników, 344
 - okien, 359
 - okna dialogowego, 226
 - stron, 340
 - ścieżki, 232
 - tekstu, 175, 306
 - tekstury, 313
 - w etykietce, 190
 - wartości zmiennych, 179
 - zawartości folderu, 367, 372
 - zawartości pliku, 218
 - wywołanie funkcji przez wskaźnik, 63
- Z**
- zakończenie wątku, 326
 - zamiana liter, 238
 - zapis do pliku, 48
 - binarnego, 222
 - tekstowego, 220
 - zapytania z linii komend, 286
 - zarządzanie plikami, 367
 - zasoby
 - ikon, 123
 - menu, 128
 - zaznaczanie
 - komórek, 258
 - komponentu, 138
 - wierszy, 259
 - zdarzenia, 16, 165, 171
 - zdarzenia komponentu
 - BackgroundWorker, 334
 - zdarzenie
 - CellEndEdit, 273
 - Click, 171, 259, 349, 360
 - CurrentCellChanged, 256
 - DoWork, 336
 - FormClosing, 301, 366
 - KeyDown, 341
 - Load, 252, 264, 330, 365
 - Paint, 314
 - RunWorkerCompleted, 336
 - Tick, 316
 - WM_TIMER, 160
 - zmiana
 - danych w bazie, 295
 - liczby komórek, 261
 - zmienianie
 - wielkości okna, 195
 - wyglądu tabeli, 253
 - zmienna, 14
 - CDNumber, 296
 - char, 38
 - double, 38
 - float, 38
 - message, 121
 - TCHAR, 140
 - WCHAR, 140
 - zmiennie sterujące, 327
 - znacznik (.), 372
 - znak
 - *, 208
 - ^, 208
 - &, 66
 - @, 293
 - =, 66
 - komentarza, 333
 - zachęty, 285

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION



1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW
w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

<http://program-partnerski.helion.pl>

Microsoft Visual C++ 2012

PRAKTYCZNE PRZYKŁADY

Dostępne od niedawna środowisko Visual C++ 2012 oferuje użytkownikom szereg zupełnie nowych możliwości. Dzięki wprowadzeniu obsługi standardu C++11 i zwiększeniu przejrzystości oraz uniwersalności kodu źródłowego rozwiązanie ugruntowało swoją renomę nowoczesnego i nieustannie rozwijanego narzędzia programistycznego do wszechstronnych zastosowań. Ulepszone i dostosowane do nowych wymagań z powodzeniem może konkurować z innymi środowiskami obecnymi na rynku. Potwierdzają to również takie posunięcia producenta jak zapewnienie wsparcia dla twórców programów działających w systemie Windows 8 i aplikacji mobilnych uruchamianych na platformie Windows Phone.

Microsoft Visual C++ 2012. Praktyczne przykłady to doskonały przewodnik dla osób chcących poznać język C++ i zacząć pisać programy w środowisku Visual C++. Książka zawiera dokładny opis składni standardowego języka C++, a także praktyczne wskazówki dotyczące tworzenia aplikacji wykorzystujących Windows API oraz .NET Framework. Podręcznik nie tylko prezentuje podstawowe konstrukcje języka i sposoby ich stosowania, lecz także wprowadza czytelnika w bardziej zaawansowane zagadnienia związane z tworzeniem aplikacji działających w systemach operacyjnych Windows. Każdy omawiany tutaj temat został zilustrowany przykładem umożliwiającym praktyczne utrwalenie poznanych wiadomości teoretycznych.

- Instalacja i obsługa środowiska Microsoft Visual C++
- Struktura programów C++ i konstrukcje języka
- Przegląd instrukcji i typów wbudowanych
- Podstawy techniki obiektowej
- Korzystanie z Windows API
- Tworzenie aplikacji wyposażonych w GUI
- Obsługa operacji na plikach
- Programowanie wielowątkowe
- Używanie komponentów .NET Framework

**Chcesz nauczyć się szybko
działać w nowym Visual C++?
Jesteś na dobrej drodze!**

helion.pl
księgarnia
internetowa

Nr katalogowy: 11107



Księgarnia Internetowa:

<http://helion.pl>



Zamówienia telefoniczne:

0 801 339900



0 601 339900



Helion

Sprawdź najnowsze promocje:

• <http://helion.pl/promocje>

Książki najchętniej czytane:

• <http://helion.pl/bestsellery>

Zamów informacje o nowościach:

• <http://helion.pl/nowosci>

Helion SA

ul. Kościuszki 1c, 44-100 Gliwice

tel.: 32 230 98 63

e-mail: helion@helion.pl

<http://helion.pl>

sięgnij po **WIĘCEJ**



KOD KORZYŚCI

ISBN 978-83-246-5352-2



9 788324 653522

Cena: 54,90 zł

Informatyka w najlepszym wydaniu