Jerzy Pokojski (red.)
Janusz Bonarowski, Jacek Jusis

# Języki programowania

Warszawa 2011

KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI

UNIA EUROPEJSKA
EUROPEJSKI
FUNDUSZ SPOŁECZNY

Politechnika Warszawska
Wydział Samochodów i Maszyn Roboczych
Kierunek studiów "Edukacja techniczno informatyczna"
02-524 Warszawa, ul. Narbutta 84, tel. (22) 849 43 07, (22) 234 83 48
ipbmvr.simr.pw.edu.pl/spin/, e-mail: sto@simr.pw.edu.pl

Opiniodawca: prof. dr hab. inż. Andrzej BAIER

Projekt okładki: Norbert SKUMIAŁ, Stefan TOMASZEK

Projekt układu graficznego tekstu: Grzegorz LINKIEWICZ

Skład tekstu: Janusz BONAROWSKI

Publikacja bezpłatna, przeznaczona dla studentów kierunku studiów
"Edukacja techniczno informatyczna"

Copyright © 2011 Politechnika Warszawska

Utwór w całości ani we fragmentach nie może być powielany
ani rozpowszechniany za pomocą urządzeń elektronicznych, mechanicznych,
kopiujących, nagrywających i innych bez pisemnej zgody posiadacza praw
autorskich.

ISBN 83-89703-68-8

Druk i oprawa: STUDIO MULTIGRAF SP. Z O.O.,
ul. Ołowiana 10, 85-461 Bydgoszcz

# Spis treści

**Wstęp.......................................................................... 5**

**Część I**

**1. Zagadnienia podstawowe ................................... 9**

**2. Funkcje i struktura programu ........................... 53**

**3. Tablice i wskaźniki ............................................ 69**

**4. Struktury i unie ................................................ 85**

**5. Struktury danych, listy .................................... 105**

**6. Struktury danych, stosy, kolejki ..................... 115**

**7. Klasy .............................................................. 117**

**8. Dziedziczenie.................................................. 135**

**9. Polimorfizm .................................................... 153**

**Część II**

**10. Przykłady programów obliczeniowych ........ 161**

10.1. Rekurencja ..................................................................162
10.2. Praca z tekstami - wykorzystanie klasy String.........163

**11. Przykłady programów użytkowych i multimedialnych ........................................ 167**

11.1. Interaktywna grafika wektorowa ..............................168
11.2. Prosty edytor tekstów ................................................177
11.3. Dźwięk i pliki muzyczne ............................................182
11.4. Współpraca z systemem CAD/CAE ...........................189

**12. Podsumowanie ............................................ 193**

**13. Literatura.................................................... 197**

# Wstęp

Niniejsze materiały zostały opracowane w ramach realizacji Programu Rozwojowego Politechniki Warszawskiej współfinansowanego ze środków PROGRAMU OPERACYJNEGO KAPITAŁ LUDZKI. Przeznaczone są dla studentów kierunku „Edukacja Techniczno-Informatyczna”.

Celem opracowania było przedstawienie zasadniczych składników języka programowania C i jego wersji obiektowej C++. Założony poziom opanowania tych narzędzi to uzyskanie umiejętności pisania oprogramowania dedykowanego, współpracującego z komercyjnymi systemami CAD/CAE.

Analiza dostępnych obecnie kompilatorów języka C/C++ skłoniła autorów niniejszego opracowania do następujących wniosków:

1. W procesie nauczania języka C/C++ można wyodrębnić następujące komponenty: (a) podstawowe składniki języka C/C++ w ujęciu klasycznym, (b) posługiwanie się określonym środowiskiem programistycznym do tworzenia aplikacji w języku C/C++, (c) wykorzystanie istniejących narzędzi, bibliotek przeznaczonych do zastosowania w aplikacjach napisanych w C/C++. Obecnie wszystkie powyższe komponenty należy traktować jako jednakowo ważne – w związku z tym w programie przedmiotu powinny wystąpić zarówno podstawowe składniki języka jak i zagadnienia związane z określonym środowiskiem programistycznym i dostępnymi bibliotekami.

2. Z dostępnych obecnie kompilatorów języka C/C++ jedynie kompilator firmy Microsoft posiada szerokie możliwości w zakresie budowy różnych typów aplikacji opartych na istniejących bibliotekach. Jest to jednak narzędzie rozbudowane i dosyć trudne w opanowaniu dla osób początkujących.

Podsumowując, ponieważ zamierzeniem autorów było warsztatowe przygotowanie czytelnika do podjęcia tematyki tworzenia oprogramowania w języku C/C++ przyjęto, że ważne dla osiągnięcia tego celu są zarówno zagadnienia podstawowe jak i umiejętności praktycznego pisania elementarnych aplikacji oraz posługiwania się istniejącymi narzędziami i bibliotekami, a także kompilatorem Microsoft C++.

Książki przeznaczone do nauki języka C/C++ można podzielić na trzy grupy:

1. Książki napisane przez twórców języka C i jego wersji C++, [8, 13]

2. Książki wprowadzające do określonych zagadnień języka lub do określonych klas zastosowań [11].

3. Poradniki, zawierające kompletny opis języków w wersji związanej z określonym kompilatorem.

Niniejsze opracowanie reprezentuje przede wszystkim grupę drugą z uwzględnieniem specyfiki związanej z określonym kompilatorem. Autorzy zakładają, że uczący się za pomocą tej książki będą korzystać z literatury należącej do grupy (1) i (3).

W opracowaniu skoncentrowano się na prezentacji treści, które są kluczowe dla posługiwania się językiem i stanowią jego esencję. Rozdziały 1-6, w części I, omawiają zasadnicze konstrukcje języka C, z elementami C++ i jednoczesną prezentacją możliwości środowiska programistycznego. Dalsze rozdziały (7-9) odnoszą się przede wszystkim do rozwiązań dostępnych w C++.

W pracy zamieszczono wiele przykładów konkretnych programów. Prezentowane programy były testowane za pomocą kompilatora Microsoft Visual C++, Express Edition, wersja 2008.

W II części pracy przedstawiono szerzej możliwości funkcyjne bibliotek oferowanych przez producenta kompilatora. Dodatkowo zamieszczono przykłady programów głębiej osadzone w określonych realiach.

Przy prezentacji treści przyjęto założenie, że czytający posiada elementarną wiedzę w zakresie znajomości jednego z języków programowania algorytmicznego np. języka Visual Basic.

W każdym z rozdziałów części pierwszej opracowania występuje podsumowanie, które syntetycznie naświetla szerszy kontekst prezentowanej tematyki oraz charakteryzuje dostępną literaturę.

Autorami poszczególnych rozdziałów są: Jerzy Pokojski (rozdziały 1-12 oraz redakcja całości materiałów), Janusz Bonarowski (rozdziały 1-4, 11.2), Jacek Jusis (rozdziały 7-11).

# Część I

# Zagadnienia podstawowe

# Język C/C++ geneza

Język C jest językiem programowania, który powstał w latach 70-tych XX wieku. Jego autorami są B. Kernighan i D. Ritchie [8]. C stworzono jako język mający za zadanie zapewnić możliwość budowy systemów operacyjnych. Przy jego budowie zwracano uwagę zarówno na efektywność i szybkość przetwarzania, jak i na możliwość realizacji operacji, które wcześniej były wykonywane przede wszystkim w asemblerze. Z czasem, język C poddano pewnej standaryzacji, jednocześnie został on wzbogacony o szereg konstrukcji typowych dla klasycznych algorytmicznych języków programowania.

Bardzo szybko język C stał się narzędziem używanym do tworzenia profesjonalnych aplikacji przeznaczonych do wielu realnych zastosowań.

Pod koniec lat 70-tych XX wieku pojawiła się obiektowa odmiana języka C oznaczana jako C++ (nazywana niekiedy jako C z klasami). Język ten również w stosunkowo krótkim czasie stał się językiem szeroko używanym do budowy profesjonalnego oprogramowania.

Język C++ stanowi rozwinięcie języka C. Dlatego opanowując C++ należy najpierw poznać możliwości języka C.

W niniejszym opracowaniu przyjęto założenie, że w pierwszych rozdziałach przedstawione zostaną podstawowe elementy języka C, a w kolejnych rozszerzenia typowe dla C++.

# Visual C++ Express Edition instalacja, interface, przykład pracy ze środowiskiem

Poniżej omówiono pierwsze kroki instalowania i uruchamiania pakietu Visual Studio 2008 Express Editon (instalacja pakietu wersji 2010 odbywa się bardzo podobnie).

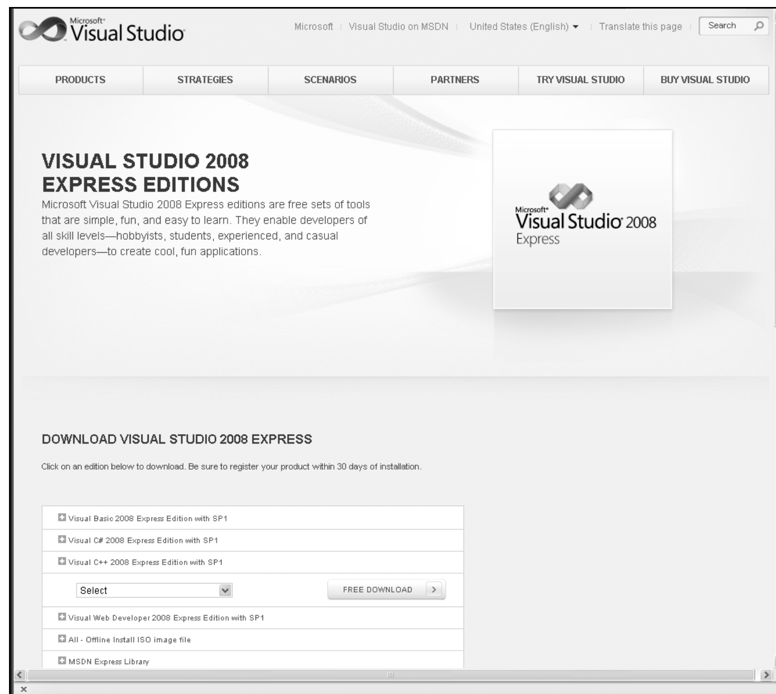
Microsft bezpłatnie udostępnia pakiety programistyczne dla kilku języków programowania, zebrane w nadrzędny pakiet o nazwie Visual Studio Express Edition. Wersja Express, mimo że jest w pełni funkcjonalna – jest całkowicie bezpłatna i to nie tylko do nauki programowania lecz także do zastosowań komercyjnych. Pakiet ten ma obecnie swoje

wersje 2008 i 2010. Wszystkie aplikacje zamieszczone w książce były uruchamiane w wersji 2008.

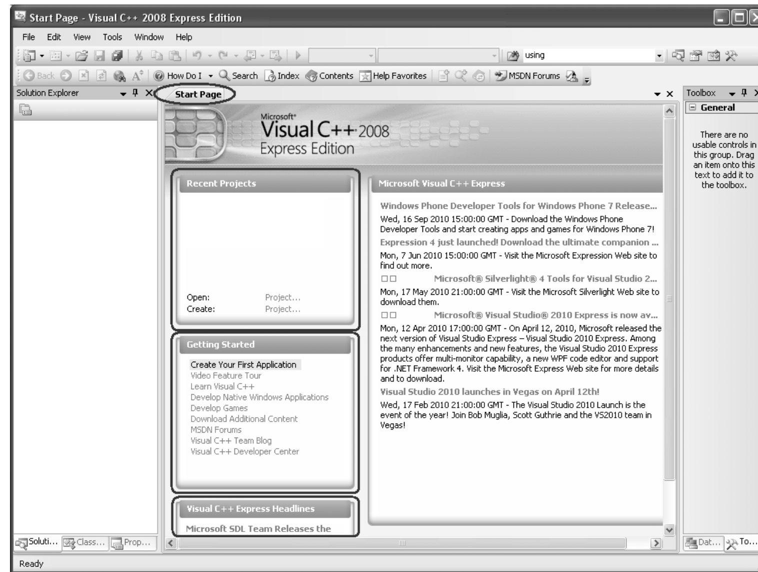
Pakiet możemy samodzielnie pobrać ze stron Microsoftu [9], rysunek 1.1. Można go pobrać w całości i sporządzić płytę instalacyjną lub zainstalować tylko interesującą nas część - jeden z języków. Na rysunku 1 zaznaczono pobierany do instalacji pakiet języka Visual C++ Express Edition.

Po zainstalowaniu Visual C++ należy zarejestrować, wybierając z menu Help, Register Product. Rejestracja nie wymaga żadnych opłat, a daje dostęp do dodatkowych zasobów, co potwierdza tekst umieszczony na stronie, patrz rysunek 1.1.

Po zainstalowaniu pakiet Microsoft Visual C++ Express Edition można od razu uruchomić w celu napisania pierwszego programu. Po wywołaniu zgłasza się środowisko – pokazano to na rysunku 1.2. Jest ono bardzo rozbudowane, służy do tworzenia, uruchamiania i testowania aplikacji pisanych w języku C++/CLI. Środowisko nosi angielską nazwę Integrated Development Environment, w skrócie IDE.




Rysunek 1.1. Pobieranie pakietu do instalacji




Rysunek 1.2. Postać środowiska programistycznego po wywołaniu

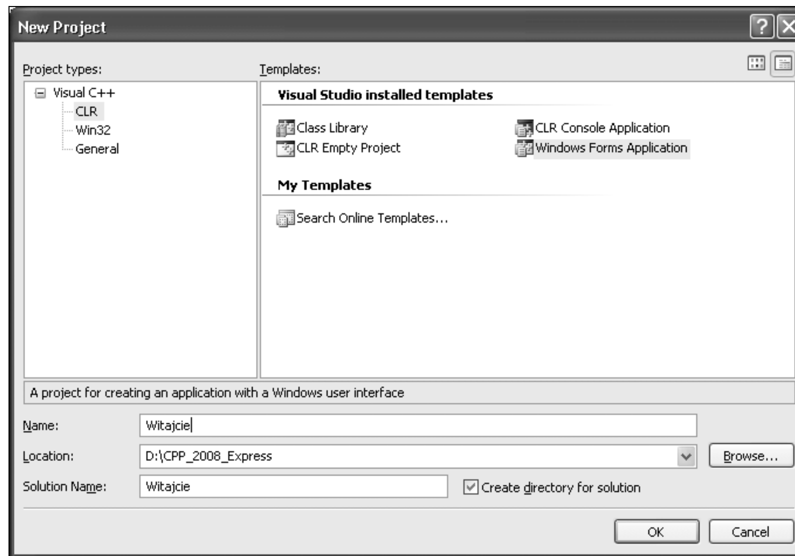
Wszystkie polecenia tego środowiska mają nazwy anglojęzyczne. Firma Microsoft dostarcza pakiety środowiska w 10 językach. Podczas pobierania pakietu instalacyjnego ze strony internetowej firmy można sobie wybrać jeden z tych języków, rysunek 1.3.

Rysunek 1.3. Języki, w których możemy zainstalować środowisko IDE

Po wywołaniu na ogół (choć można skonfigurować środowisko inaczej) zgłasza się strona startowa - Start Page, rysunek 1.2. Jej centralną część stanowi okno, w którym, jeśli komputer na którym pracujemy ma dostęp do Internetu, pojawią się odsyłacze do najnowszych informacji firmy Microsoft dla programistów. Po lewej części okna Start Page znajdują się 3 panele:

- Recent Projects – zawiera listę poprzednio wykonywanych projektów, do których możemy łatwo powrócić,
- Geting Started – zawiera listę odsyłaczy do materiałów edukacyjnych dla programisty C++ stawiającego pierwsze kroki w programowaniu w tym języku,
- Visual C++ Express Headlines – panel z tytułami artykułów proponowanych przez Microsft.

W celu napisania pierwszej aplikacji w C++ wybierzmy z menu File ->New->Project... Otworzymy w ten sposób okno z przygotowanymi szablonami różnych typów projektów, które możemy tworzyć w tym środowisku, rysunek 1.4.




Rysunek 1.4. Okno wyboru typu projektu - szablonu pierwszej aplikacji

Na ogół aplikacją tworzoną jako pierwsza, jest aplikacja typu „Hello World”, czyli drukująca (wyświetlająca) komunikat powitalny. Przyjmijmy, że aplikacja ta korzysta z typowego formularza środowiska systemu operacyjnego Windows, zaznaczymy zatem w oknie Templates (szablony) pozycję Windows Forms Application.

Nazwijmy naszą aplikację „Witajcie”. Taki tekst wpiszemy w pozycji Name, rysunek 1.4. W pozycji Location (położenie) wskażemy, wykorzystując przycisk Browse (przeszukać), katalog przeznaczony na

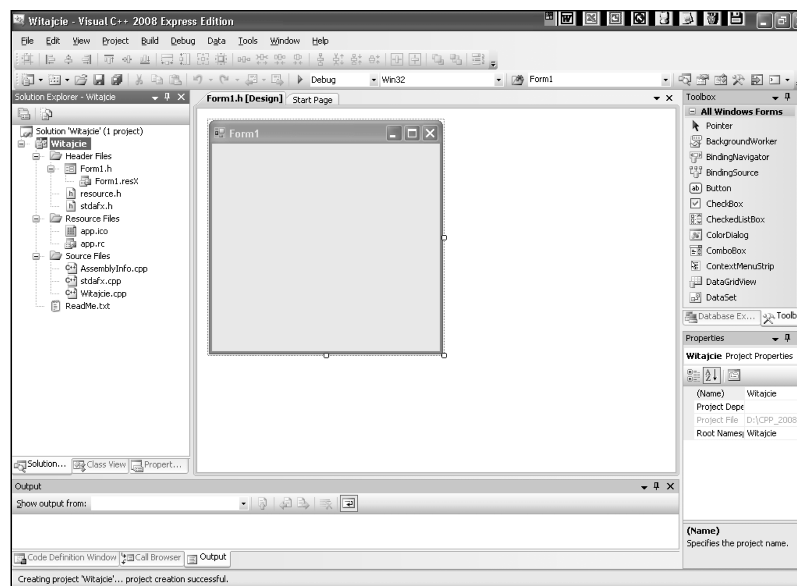
składowanie naszej aplikacji. Pozycja Solution Name (nazwa rozwiązania) wypełni się automatycznie nazwą identyczną jak w pozycji Name.

Pojęcie Solution (rozwiązanie) jest pojęciem nadrzędnym nad pojęciem Projektu (czyli pojedynczej aplikacji). Kilka aplikacji (projektów) odnoszących się do jakiegoś wspólnego problemu można umieszczać w jednym rozwiązaniu (solution).

Dla naszych potrzeb będziemy zawsze tworzyć pojedyncze projekty, z których każdy będzie pojedynczym rozwiązaniem.

Dobrze jest zaznaczać kratkę Create directory for solution, co zapewni umieszczanie całej struktury projektu w odrębnym podkatalogu.

Po kliknięciu przycisku OK, utworzony zostanie nowy projekt, a środowisko przyjmie postać jak na rysunku 1.5.



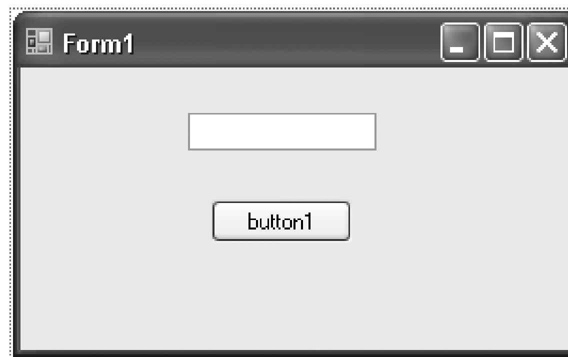

Rysunek 1.5. Postać środowiska po wygenerowaniu nowego projektu

Centralną część zajmuje plik Form1.h umożliwiający graficzne projektowanie formularza aplikacji. Po lewej stronie w oknie Solution Explorer przedstawione są, utworzone w strukturze drzewiastej, katalogi i pliki projektu. Po dwukrotnym kliknięciu pliku pojawia się on w głównym oknie do edycji. Dwie pozostałe zakładki tego okna zostaną omówione gdy pojawi się potrzeba skorzystania z ich funkcji.

Na dole mamy poziome okno z trzema zakładkami. W początkowym okresie nauki programowania interesować nas będzie okno Output, w którym podczas kompilacji i konsolidacji programu, będą wyświetlane komunikaty i ewentualne informacje o błędach.

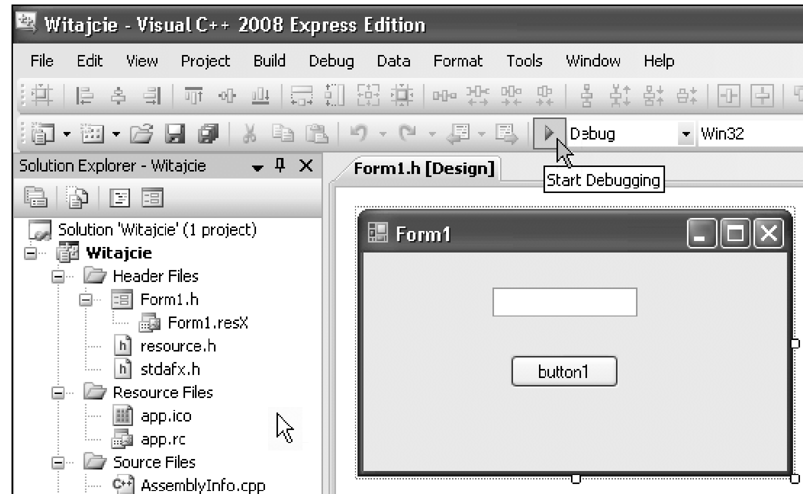
Po prawej stronie u góry widoczne jest okno Toolbox (Przybornik), zawierające prototypy obiektów, które możemy umieszczać na formularzu podczas projektowania przeciągając je myszą, a u dołu znajduje się okno Properties (Właściwości), będące listą właściwości wskazanego (klikniętego) na formularzu obiektu. W oknie tym możemy także zmieniać wartości właściwości.

Przeciągnijmy z okna panelu Toolbox, z zakładki All Windows Forms, na formularz dwa obiekty: TextBox i Button, Formularz może mieć postać jak na rysunku 1.6.



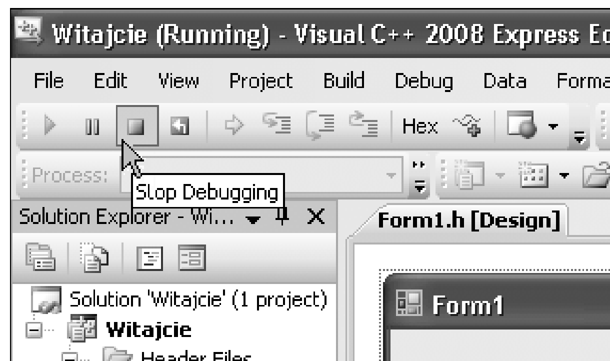

Rysunek 1.6. Rozmieszczenie obiektów na formularzu

Możemy sprawdzić jak działa tak wykonana aplikacja. Aby skompilować, skonsolidować i uruchomić program najwygodniej jest kliknąć myszą przycisk Start Debugging, rysunek 1.7.



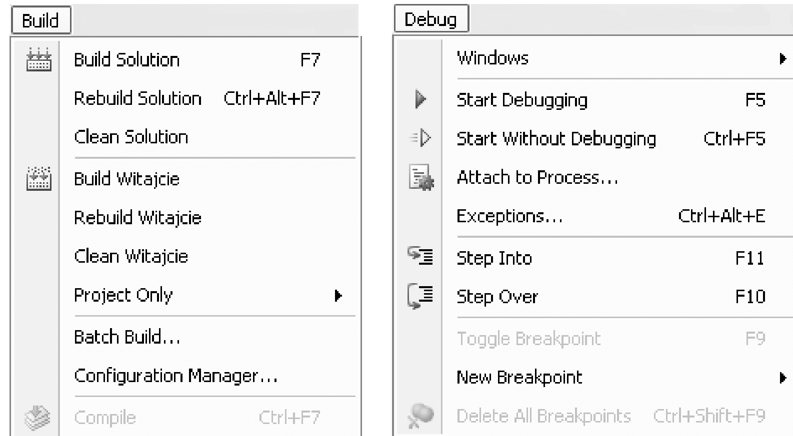

Rysunek 1.7. Uruchamianie testowe aplikacji w trybie debugera

Komentarze z tych procesów można śledzić w oknie Output w dolnej części środowiska. Na koniec działająca aplikacja zostanie wywołana (przejdzie w tzw. stan run) i ponieważ nie wpisano żadnego kodu – obiekty będą zachowywały się w sposób domyślny – przycisk klikany będzie symulował uginanie się, okno tekstowe pozwoli na wpisanie jednego wiersza dowolnego tekstu z klawiatury, formularz pozwoli przesuwać się po ekranie, zmieniać rozmiar i działały będą przyciski pozwalające na maksymalizację formularza, „zrzucenie" formularza na pasek zadań i zamknięcie aplikacji poprzez zamknięcie formularza przyciskiem zamknij. Aplikację można też zamknąć klikając przycisk Stop Debugging, rysunek 1.8.




Rysunek 1.8. Zamknięcie uruchomionej aplikacji

Pełną gamę możliwych poleceń kompilacji i uruchomień uzyskujemy poprzez menu Buid i menu Debug, rysunek 1.9.



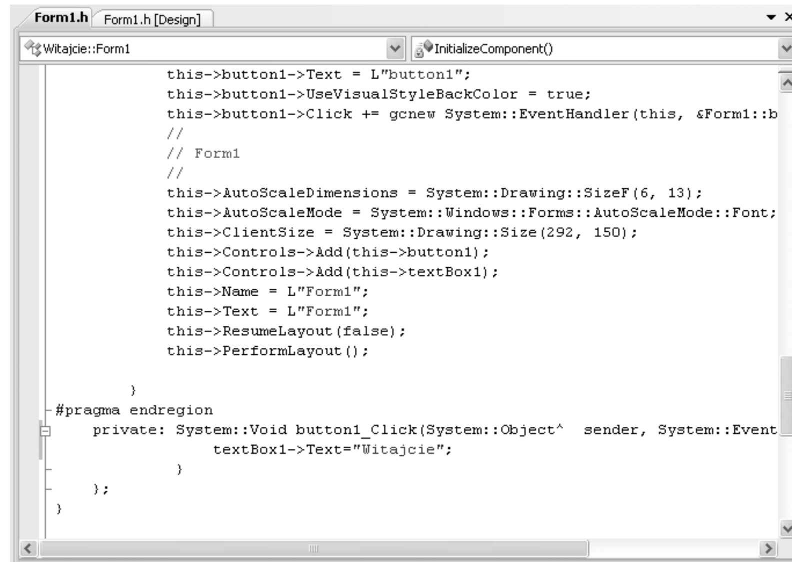

Rysunek 1.9.

Jeśli będąc w stanie projektowania formularza dwukrotnie klikniemy przycisk – otwarte zostanie okno edycji kodu pliku Form1.h i środowisko wygeneruje szkielet procedury obsługi zdarzenia: kliknięcie obiektu o nazwie button1.

Aby aplikacja spełniła nasze zamierzenia, czyli napisała w oknie tekstowym treść powitania, należy dopisać do tej procedury jeden wiersz:

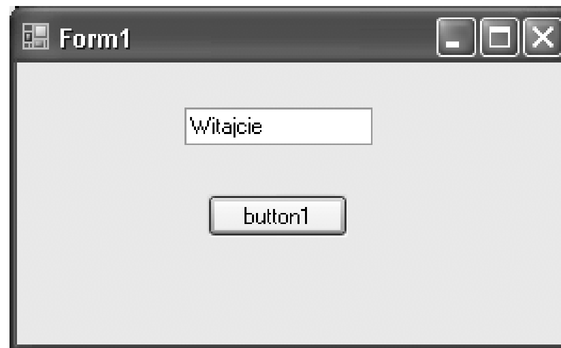
```
textBox1->Text="Witajcie";
```

Okno kodu będzie miało postać jak na rysunku 1.10.




Rysunek 1.10. Okno kodu

Po wpisaniu tego wiersza kodu, ponownym skompilowaniu programu i uruchomieniu go – aplikacja po kliknięciu na przycisk wypisze w oknie tekstowym treść powitania – słowo Witajcie, rysunek 11.



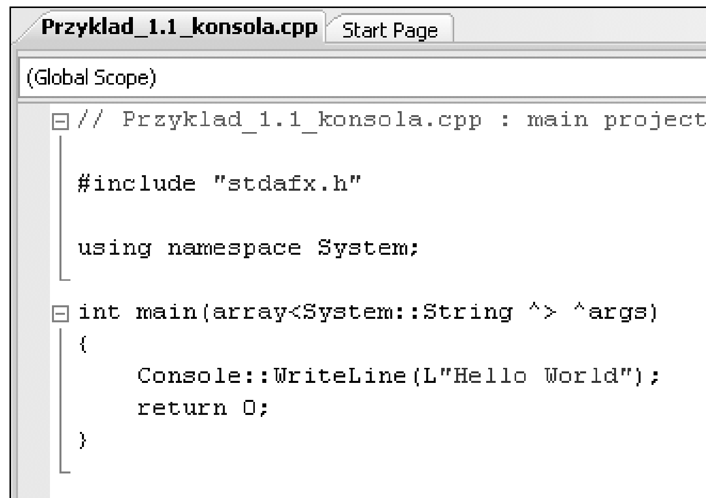

Rysunek 1.11. Widok okna pierwszej aplikacja w trakcie działania

# Przykład prostego programu w języku C

### Przykład 1.1

Poznawanie nowego języka programowania dobrze jest zacząć od stosunkowo prostego programu pokazującego podstawowe cechy i elementy tego języka. Pierwszy w tej książce, prosty, przykładowy program (program 1.1) napisany w języku C przeznaczony jest do przeliczania jednostek – wielkości wyrażone w kilometrach przeliczane są na metry.

Otwórzmy środowisko Visual C++ Express Edition i wybierzmy File > New -> Project. Jako szablon wybierzmy **CLR Console Application**. Nadajmy nazwę projektowi: Przykład_1.1_konsola. Po kliknięciu przycisku OK. środowisko zgłosi się jak na rysunku 1.12.




Rysunek 1.12. Początek pisania programu

Zmieńmy kod jak na rysunku 1.13.

```
#include "stdafx.h"
#include <iostream>

using namespace System;
using namespace std;

void main(){
	int kilometr, metr;
	printf ("\n Liczba ujemna konczy program");
	printf ("\n Wpisujemy liczby calkowite. \n");

	printf ("\n podaj dlugosc [km]:");
	scanf ("%d", &kilometr);
	while (kilometr >= 0)
		{
			metr = 1000 * kilometr;
			printf ( "\nkilometry:  %d  \n", kilometr);
			printf ( "    metry:  %d \n", metr);

			printf ("\n podaj dlugosc [km]:");
			scanf ("%d", &kilometr);
		}
	printf("koniec programu");
}
```

Rysunek 1.13. Kod programu

Cztery początkowe wiersze, które zawsze będziemy umieszczać na początku programów konsolowych dołączają do programu odpowiednie biblioteki.

Sam program napisano w postaci jednej funkcji **void main(). main()** jest nazwą zastrzeżoną funkcji, która musi zawsze występować w programie.

Wykonywanie każdego programu zaczyna się od uruchomienia funkcji **main()**. **void main()** oznacza, że funkcja nie będzie zwracać żadnej wielkości. Linia programu **void main()** stanowi nagłówek funkcji. Nawiasy okrągłe stojące po main mogą zawierać listę parametrów formalnych. Pozostała część nazywana jest ciałem funkcji. Ciało funkcji jest ograniczone nawiasami klamrowymi: **{ … }**. Otwierający nawias klamrowy występuje bezpośrednio po nagłówku, zamykający kończy funkcję.

Linia kodu zaczynająca się od **int** stanowi definicję dwóch zmiennych lokalnych (występujących i używanych wyłącznie w danej funkcji) kilometr i metr. Linia każdej instrukcji w języku C jest zakończona znakiem średnika: **;**. **int** oznacza typ całkowity zmiennej. Typ rzeczywisty pojedynczej precyzji jest określany przez **float**, typ rzeczywisty podwójnej precyzji przez **double**.

W klasycznym C w operacjach wejścia/ wyjścia na ogół używane są funkcje **printf**() i **scanf**(). Funkcja **printf()** służy do wykonania wydruku zgodnego z wzorcem znajdującym się pomiędzy znakami ″ ″. W tym przypadku będzie to wydruk tekstu zawartego pomiędzy ″ ″. Funkcja **scanf**() zapewnia możliwość wczytania wartości zmiennej.

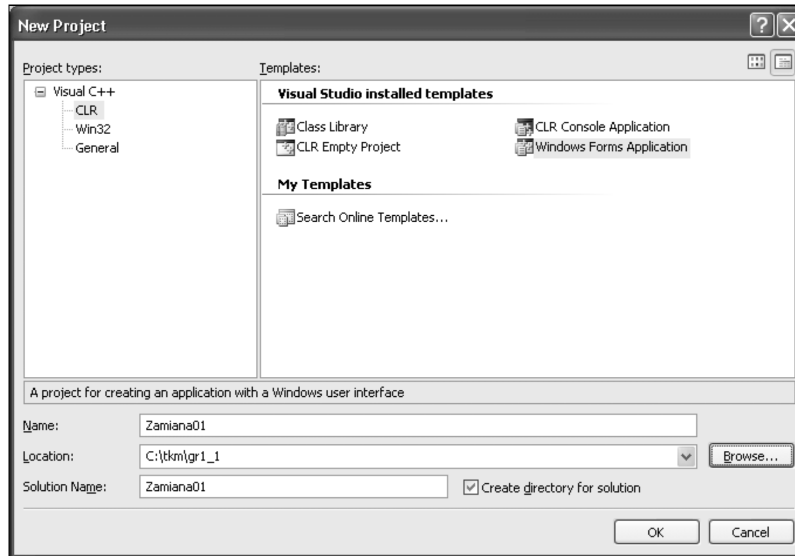
W kolejnej instrukcji programu odbywa się wczytanie wartości zmiennej kilometr zgodnie z wzorcem **%d**.

Kolejna konstrukcja programu przykładowego to instrukcja cyklu **while**() w wersji blokowej. While ( ) powoduje wielokrotne wykonywanie znajdującego się bezpośrednio dalej bloku ograniczonego przez { … }. Każdorazowe wykonanie bloku jest zależne od prawdziwości warunku zawartego bezpośrednio po while w ( ...). Ten warunek to sprawdzenie czy zmienna kilometr jest większa lub równa 0. Program kończy instrukcja wydruku informująca o fakcie zakończenia programu.

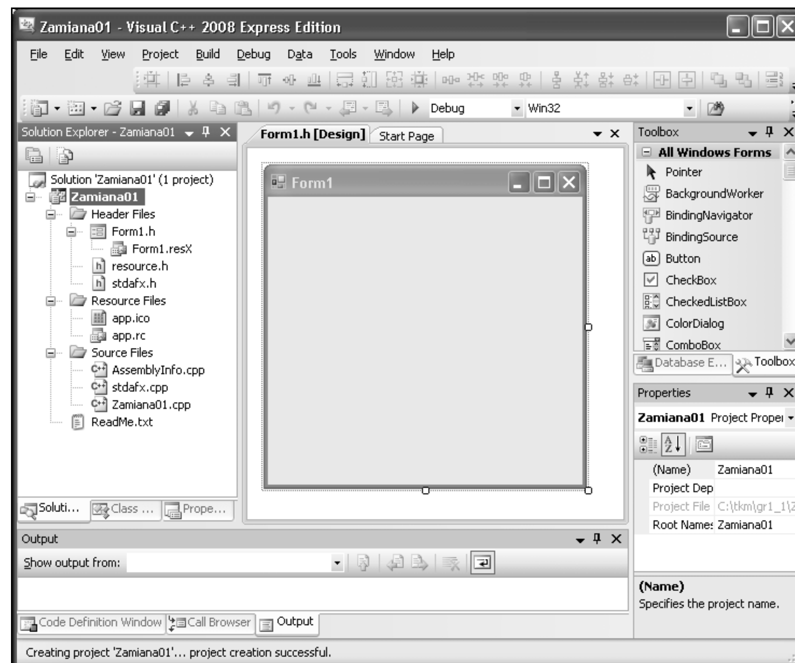
### Przykład 1.1a

Napiszmy program wykonujący podobną zamianę: kilometrów na metry lecz działający w środowisku „okienkowym" Windows. Wybierzmy z menu File -> New -> Project, co otworzy okno nowego projektu, New Project, rysunek 1.14.

W oknie tym w podoknie *Project* types wybierzmy *CLR*, w oknie *Templates* wybierzmy *Windows Forms Application*, w polu *Name* wpiszmy *Zamiana01*, wskażmy przyciskiem [Browse...] położenie projektu w polu *Location* i zaznaczmy kratkę *Create directiory for solution*, aby całość pracy znajdowała się w oddzielnym katalogu. Po tych działaniach i kliknięciu przycisku OK środowisko programistyczne będzie miało postać jak na rysunku 1.15.

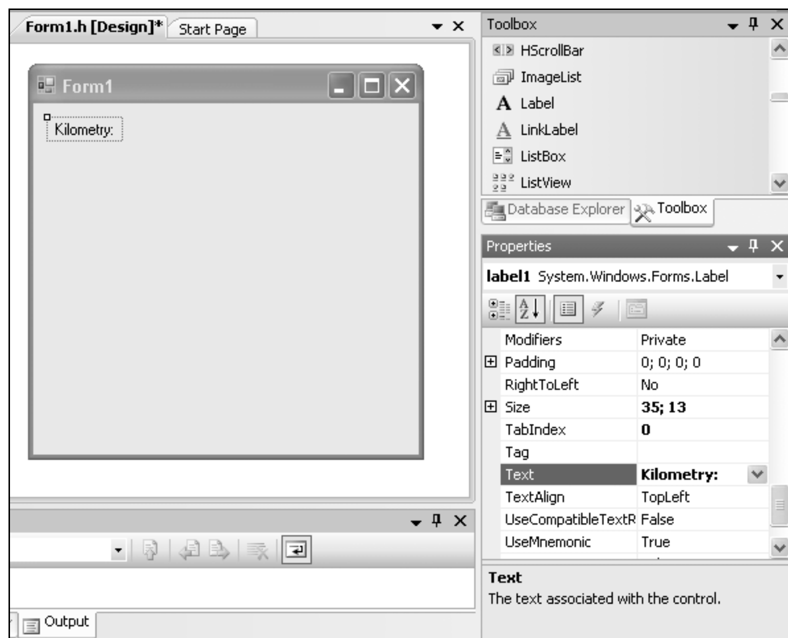



Rysunek 1.14. Okno wyboru szablonu nowego projektu



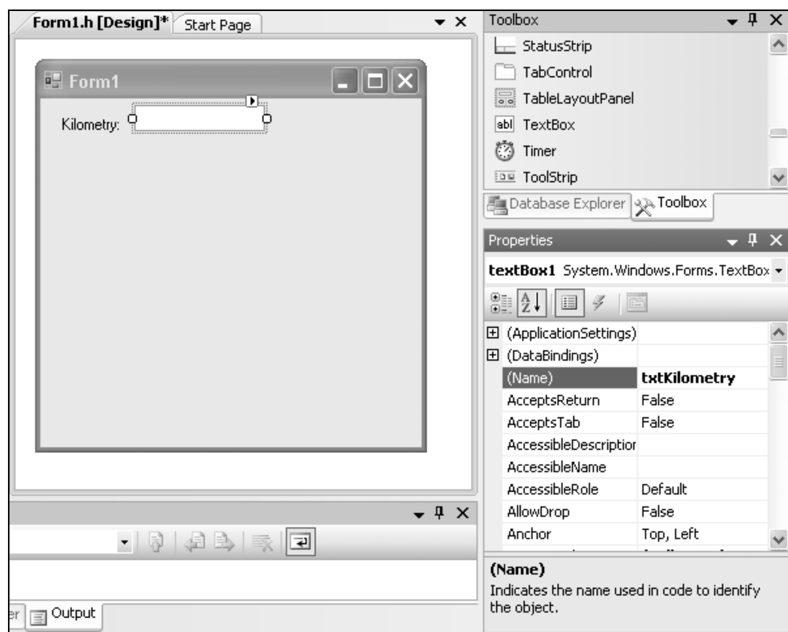

Rysunek 1.15. Środowisko pracy programisty

Przenieśmy z okna Toolbox, na formularz, obiekt Label i w oknie Properties, określmy jego właściwość Text jako: Kilometry:, rysunek 1.16.




Rysunek 1.16. Tworzenie etykiety z napisem Kilometr

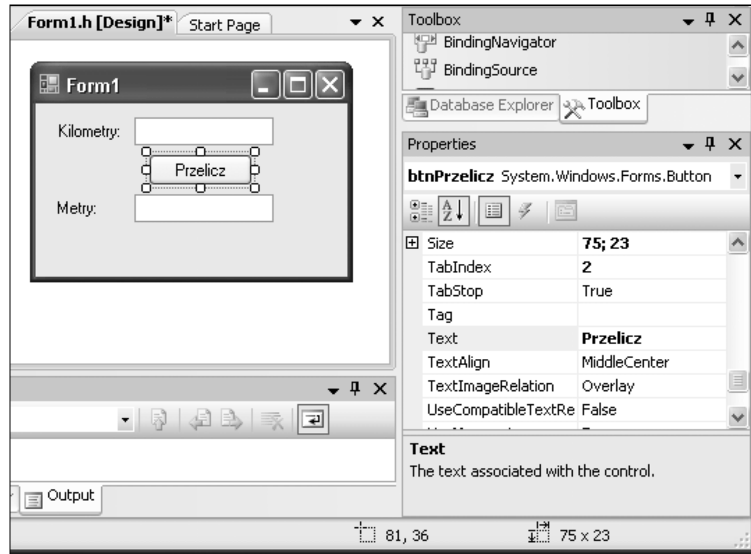
Obok utworzonej etykiety przenieśmy, z Toolbox'u na formularz, obiekt TextBox i nadajmy mu nazwę txtKilometry, rysunek 1.17.




Rysunek 1.17. Dodanie obiektu TextBox i nadanie mu nazwy txtKilometry

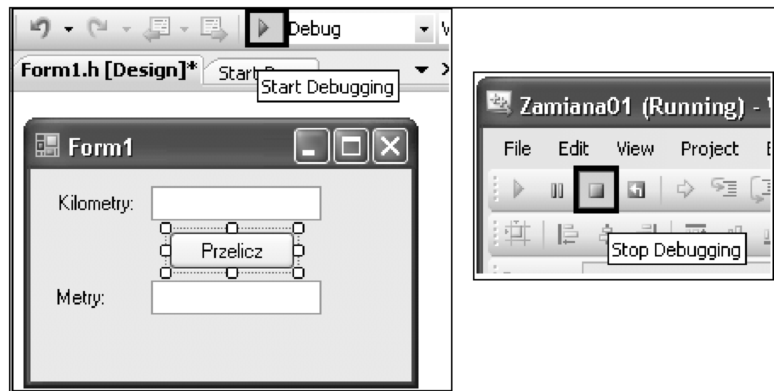
Podobnie dodajmy drugi obiekt Label i drugi textBox odpowiednio określając, w oknie Properties, dla obiektu Label właściwość Text jako „Metry:”, a dla obiektu textBox właściwość Name: txtMetry.

Pomiędzy wierszem Kilometry, a wierszem Metry umieśćmy obiekt Button, dla którego, w oknie Properties wpiszmy dwie właściwości: Name jako btnPrzelicz i właściwość Text jako „Przelicz”. Możemy też zmniejszyć rozmiary formularza, rysunek 1.18.




Rysunek 1.18. Formularz z obiektami

Możemy już uruchomić tak przygotowaną aplikację. W tym celu kliknijmy przycisk *Start Debugging* . Gdy chcemy zatrzymać działającą aplikację klikamy przycisk *Stop Debugging*, rysunek 1.19.




Rysunek 1.19. Przyciski uruchamiania i zatrzymywania aplikacji

Aplikacja uruchomiona zachowuje się domyślnie, tzn. w okna tekstowe można po kliknięciu wpisywać znaki, przycisk Przelicz po kliknięciu symuluje uginanie. Działają też, zgodnie ze swymi funkcjami przyciski w prawym górnym rogu formularza, minimalizuj, maksymalizuj i zamknij.

Z uwagi na to iż nie wpisano kodu – aplikacja nie wykonuje innych czynności.

Należy dodać kod do aplikacji. Aplikacje w środowisku Windows działają w wyniki zaistnienia zdarzeń. Zdarzeniem jest kliknięcie na przycisk Przelicz. Jeśli dwukrotnie klikniemy na przycisku - środowisko automatycznie wygeneruje szkielet procedury obsługującej zdarzenie „kliknięcie na przycisku". Otwarte zostanie też nowego okna, w którym będziemy mogli wpisać kod procedury:

Szkielet procedury bez kodu wygenerowany przez środowisko:

```
private: System::Void
btnPrzelicz_Click(System::Object^  sender,
             System::EventArgs^  e) {
                 }
```

Kod procedury:

```
private: System::Void
btnPrzelicz_Click(System::Object^  sender,
    System::EventArgs^  e) {
     double kilometr, metr;
     kilometr=Convert::ToDouble(txtKilometry->Text);
     metr=kilometr*1000;
     txtMetry->Text=metr.ToString();
 }
```

Omówienie wierszy programu

```
double kilometr, metr;
```

Deklaracja dwóch zmiennych typu double: kilometr i metr

```
kilometr = Convert::ToDouble(txtKilometry->Text);
```

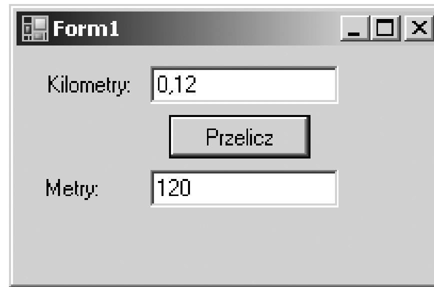
Pobranie łańcucha tekstowego z txtKilometr, przekonwertowanie go na wartość liczbową typu double i podstawienie tej wartości pod zmienną kilometr.

```
metr = kilometr * 1000;
```

Wykonanie mnożenia wartości w zmiennej kilometr przez 1000 i podstawienie wyniku do zmiennej metr.

```
txtMetry->Text = metr.ToString();
```

Konwersja na tekst zawartości zmiennej metr i podstawienie tego tekstu jako właściwość Text do okna tekstowego txtMetr.




Rysunek 1.20. Widok okna aplikacji w trakcie działania

## Operatory relacyjne i logiczne

W warunkach, które mogą być umieszczane w określonych instrukcjach, np. w pokazanej w przykładzie 1.1 instrukcji **while (... )**, można badać prawdziwość różnych relacji. W przykładzie 1.1 sprawdzano czy zmienna kilometr jest większa lub równa 0. Operatory relacyjne i logiczne pozwalają na budowę warunków, których prawdziwość jest badana.

Operatory relacyjne to:

```
==       równość
!=       różne
>        większe
>=       większe, równe
<        mniejsze
<=       mniejsze, równe
```

W wielu przypadkach stosujemy bardziej złożone postacie warunków. Potrzebne są wówczas operatory logiczne:

```
&&     i
||     lub
!      negacja
```

Poniżej przedstawiono przykłady wyrażeń zapisanych w języku C przy wykorzystaniu niektórych operatorów:

zakładamy, że x= 13,  y= 23,

```
x<15   &&   y==23
!(x>10)
```

```
!x>10
!(x<=40 || y==33)
```

Spróbuj określić prawdziwość powyższych wyrażeń, dokonaj również sprawdzenia komputerowego poprzez napisanie programu.

# Operatory arytmetyczne (dwu-argumentowe)

W języku C podobnie jak w każdym innym języku programowania istnieje możliwość budowy wyrażeń arytmetycznych. Ich budowa odbywa się przy wykorzystaniu zmiennych i operatorów arytmetycznych. Podstawowe operatory arytmetyczne dostępne w języku C to:

```
+   operator dodawania
-   operator odejmowania
*   operator mnożenia
/   operator dzielenia
%   operator obliczający resztę z dzielenia
```

Budując złożone wyrażenia arytmetyczne można stosować nawiasy ( ), odpowiednio je otwierając i zamykając. Nawiasy ( ) stosowane są również do wyodrębniania struktur zagnieżdżonych. Kolejność wykonywania działań w przypadku wyrażeń opartych wyłącznie na wymienionych powyżej operatorach odpowiada zwykle kolejności stosowanej przy obliczeniach wykonywanych na papierze. Zatem najpierw wykonywane są operacje mnożenia i dzielenia, zwykle od lewej strony, następnie dodawania i odejmowania, również zwykle od lewej strony. Przetwarzanie jest wykonywane począwszy od najbardziej zagnieżdżonych wyrażeń zawartych w odpowiednich nawiasach ( ).

Poniżej przedstawiamy przykład programu obliczającego pole kwadratu o danym boku.

### Przykład 1.2 wykonany jako Console Application

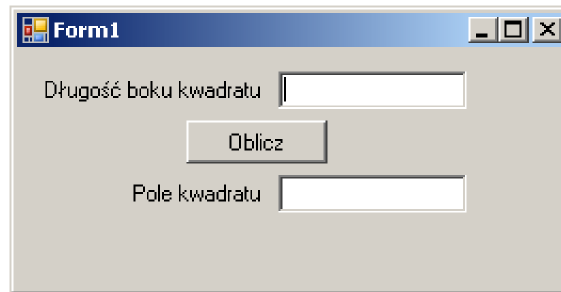
```
// Przyklad_1.2_k.cpp : main project file.

#include "stdafx.h"
#include <iostream>

using namespace System;
using namespace std;

int main()
```

```
 {
 double pole, bok_kwadratu;
 cout<<"\nObliczanie pola kwadratu o zadanym boku ";
 bok_kwadratu = 1.7;
 pole = bok_kwadratu * bok_kwadratu ;
 cout << "\npole kwadratu o zadanym boku " <<
   bok_kwadratu <<" wynosi " << pole;

 bok_kwadratu = 1.7e-2;
 pole = bok_kwadratu * bok_kwadratu;
 cout <<  "\npole kwadratu o zadanym boku " <<
   bok_kwadratu <<" wynosi " << pole;

 bok_kwadratu = 1.7e5;
 pole = bok_kwadratu * bok_kwadratu;
 cout <<  "\npole kwadratu o zadanym boku " <<
   bok_kwadratu <<" wynosi " << pole <<"\n" ;

cin>>bok_kwadratu;
}
```

W przykładzie pokazano różne formy zapisu danych oraz zastosowano inne instrukcje wejścia/wyjścia niźli w przykładzie 1.1. Są to instrukcje wejścia/ wyjścia typowe dla języka C++.

**Przykład 1.2a wykonany jako Windows Forms Application**

Napiszmy teraz aplikację Windows Forms Application wykonującą podobne obliczenia. Utwórzmy formularz i rozmieśćmy na nim obiekty jak na rysunku 1.21



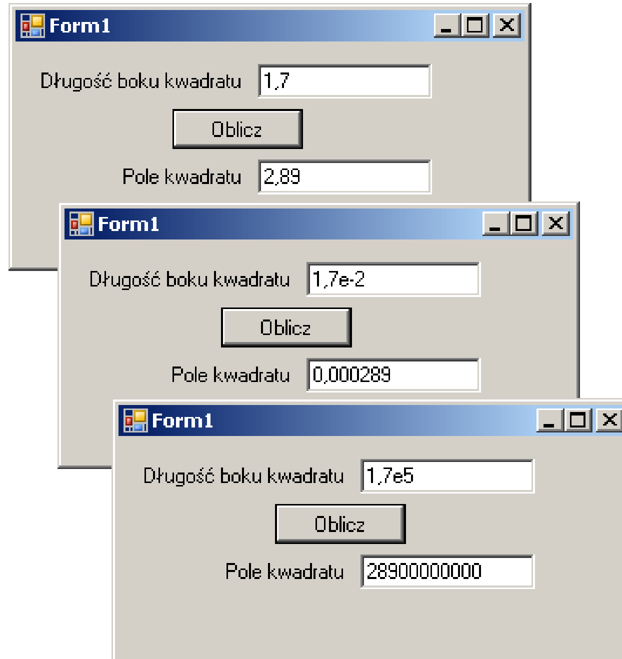

Rysunek 1.21. Postać formularza aplikacji

Niech obiekty na formularzu mają własności wg tabeli poniżej

| Obiekt | Właściwość | Wartość właściwości |
|---|---|---|
| Label1 | Text | Długość boku kwadratu |
| Button | Name | btnOblicz |
| | Text | Oblicz |
| Label2 | Text | Pole kwadratu |
| TextBox | Name | txtPole |

Po dwukrotnym kliknięciu na przycisk btnOblicz środowisko utworzy szkielet procedury btnOblicz_Click, do którego wpiszemy kod procedury:

```
private: System::Void btnOblicz_Click(System::Object^
sender, System::EventArgs^  e) {
      //Oblicznie pola kwadratu
      double bok_kwadratu, pole;
      bok_kwadratu=Convert::ToDouble(txtBok->Text);
      pole=bok_kwadratu*bok_kwadratu;
      txtPole->Text=pole::ToString;
 }
```




Rysunek 1.22. Działanie aplikacji dla danych jak w przykładzie 1.2

# Operatory specjalne

W języku C istnieje możliwość zastosowania tzw. operatorów specjalnych. Umożliwiają one alternatywny sposób zapisywania wyrażeń arytmetycznych. Na przykład wyrażenie:

```
x= x+5
```

można zapisać jako: `x+=5`

Generalnie możliwość ta jest dostępna także dla innych operatorów:

```
x= x operator y
```

można zapisać jako:

```
x operator= y
```

# Operatory ++ --

W języku C bardzo popularnymi operatorami są: ++, --. Mogą one występować zarówno po lewej jak i po prawej stronie określonej zmiennej.

W przypadku gdy operator występują po lewej stronie zmiennej oznacza to wykonanie operacji zwiększenia wartości zmiennej o 1 (przypadek ++) lub zmniejszenia zmiennej o 1 (przypadek --) jeszcze przed obliczeniem wartości całego wyrażenia.

Jeżeli natomiast operator ++ lub -- znajduje się po prawej stronie zmiennej to wówczas zwiększenie lub zmniejszenie wartości zmiennej następuje po obliczeniu wartości wyrażenia. Prześledźmy przykłady:

```
x= 5
y= x++
```

to odpowiednio uzyskane wartości: x=6, y=5

Natomiast

```
x=5
y= ++x
```

to odpowiednio uzyskane wartości: x=6 i y=6

# Inne Operatory

W języku C stosuje się szereg operatorów, które są dosyć nietypowe w porównaniu z operatorami stosowanymi w innych algorytmicznych językach programowania. Zwłaszcza ich łaczne stosowanie przynosi niekiedy zaskakujące efekty.

W języku C nie ma zmiennych logicznych. Jeżeli sprawdzamy warunek:

```
a=5
b=6
if (a>b) ...
```

to wyrażenie **a>b** zwraca **1** (wartość int) w przypadku prawdziwości warunku lub **0** w przypadku nieprawdziwości (również wartość int).

Inną niestandardową konstrukcją jest zapis, który w wersji klasycznej wygląda następująco:

Zadanie

Pod zmienną wynik podstawić większą z dwóch wartości x i y.

Rozwiazanie

```
if(x > y)
   wynik = x;
else
   wynik = y;
```

W wersji w postaci *wyrażenia warunkowego* ma następującą postać:

```
wynik=(x>y) ? x: y;
```

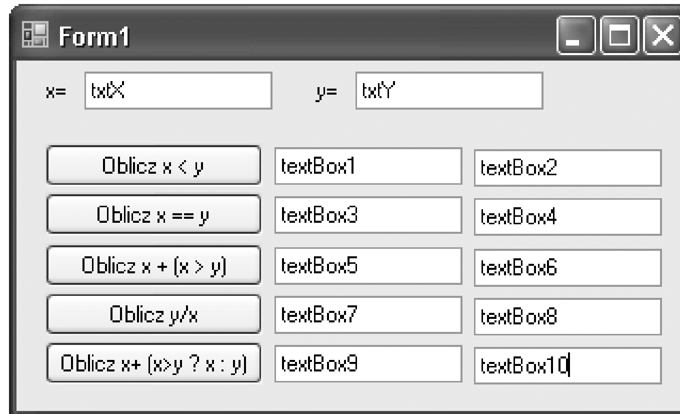
Poniżej przedstawimy kilka przykładów, które ilustrują praktyczne aspekty posługiwania się operatorami w języku C.

Przykładowy program

```
main(){
  int x=3, y=7;
  int wynik_int;
  float wynik_float;

//w dwóch operacjach poniżej najpierw jest ustalana
//wartość logiczna wyrażeń x<y i x==y ,
//w pierwszym przypadku to prawda, czyli 1,
//w drugim fałsz czyli 0.

  wynik_int=x<y;
  printf("\n wynik_int = %d  ", wynik_int);
  wynik_int= x==y;
  printf("\n wynik_int = %d  ", wynik_int);

// poniżej badana jest prawdziwość wyrażenia x>y,
//jest to fałsz czyli 0.

    wynik_int= x + (x>y);
  printf("\n wynik_int = %d  ", wynik_int);

//x i y są typu int, wynik również jest typu int,
//następnie dokonana jest konwersja na typ float.

   wynik_float=y/x; printf("\n wynik_float = %d  ",
wynik_float);

//najpierw określane jest wyrażenie warunkowe.

    wynik_int=x + (x>y ? x :y);
  printf("\n wynik_int = %d  ", wynik_int);
}
```

**Przykład 1.3a wykonany jako Windows Forms Application**

Niech formularz na postać jak na rysunku 10. Nazwy okien tekstowych pokazano na rysunku, a przyciski mają nazwy odpowiednio od button1 do button5.




Rysunek 1.23. Formularz do przykładu 1.3a

Kody procedur

```
private: System::Void button1_Click(System::Object^
         sender, System::EventArgs^  e) {
    // Obliczanie: x < y
     int x, y;
     int wynik_int;
     bool wynik_bool;

     x=Convert::ToInt16(txtX->Text);
     y=Convert::ToInt16(txtY->Text);

     wynik_int=x<y;
     wynik_bool=x<y;
     textBox1->Text=Convert::ToString(wynik_int);
     textBox2->Text=Convert::ToString(wynik_bool);
 }
//-------------------------------------------------
private: System::Void button2_Click(System::Object^
         sender, System::EventArgs^  e) {
     // Obliczanie: x == y
     int x, y;
     int wynik_int;
     bool wynik_bool;
```

```
      x=Convert::ToInt16(txtX->Text);
      y=Convert::ToInt16(txtY->Text);

      wynik_int=x==y;
      wynik_bool=x==y;
      textBox3->Text=Convert::ToString(wynik_int);
      textBox4->Text=Convert::ToString(wynik_bool);
      }
//-----------------------------------------------------
private: System::Void button3_Click(System::Object^
          sender, System::EventArgs^  e) {
      //Obliczanie: x + (x > y)
      int x, y;
      int wynik_int;
      bool wynik_bool;

      x=Convert::ToInt16(txtX->Text);
      y=Convert::ToInt16(txtY->Text);

      wynik_int = x + (x > y);
      wynik_bool= x + (x > y);
      textBox5->Text=Convert::ToString(wynik_int);
      textBox6->Text=Convert::ToString(wynik_bool);
      }
//----------------------------------------------------
private: System::Void button4_Click(System::Object^
          sender, System::EventArgs^  e) {
      //Obliczanie: y / x
      int x, y;
      double wynik_dbl;
      bool wynik_bool;

      x=Convert::ToInt16(txtX->Text);
      y=Convert::ToInt16(txtY->Text);

      wynik_dbl = y / x;
      wynik_bool= y / x;
      textBox7->Text=Convert::ToString(wynik_dbl);
      textBox8->Text=Convert::ToString(wynik_bool);
 }
//----------------------------------------------------
private: System::Void button5_Click(System::Object^
          sender, System::EventArgs^  e) {
      //Obliczanie: x + (x > y ? x : y)
      //tzn. do x dodaj większą ze zmiennych x i y
      int x, y;
      double wynik_dbl;
      bool wynik_bool;
```

```
        x=Convert::ToInt16(txtX->Text);
        y=Convert::ToInt16(txtY->Text);

        wynik_dbl = x + (x > y ? x : y);
        wynik_bool= x + (x > y ? x : y);
        textBox9->Text=Convert::ToString(wynik_dbl);
        textBox10->Text=Convert::ToString(wynik_bool);
}
```

Przykład działania aplikacji dla różnych danych

Rysunek 1.24. Dla danych x=3, y=7

Rysunek 1.25. Dla danych x=7, y=7

Rysunek 1.26. Dla danych x=8, y=7

Poniżej proponujemy samodzielne przeanalizowanie następujących wyrażeń:

```
int x=3, y=7;
int wynik_int;
wynik_int= x>y ? x++ : y++;
printf("\n wynik_int = %d  ", wynik_int);
wynik_int=x==y + (x>y ? x++ : y++);
printf("\n wynik_int = %d  ", wynik_int);
```

### Przykład 1.4 wykonany jako Console Application

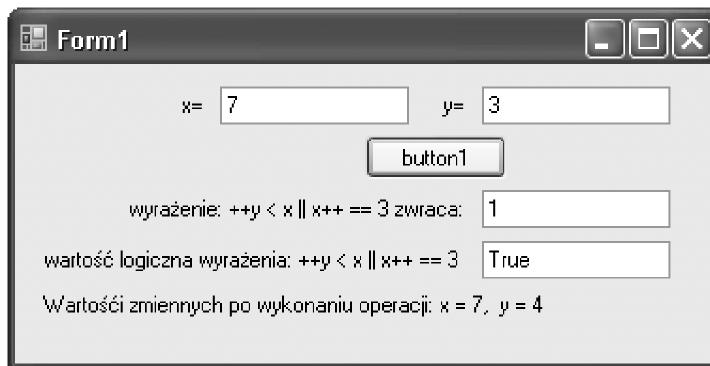
```
main()
{
  int x, y, wynik;

  x=7; y=3;

//najpierw należy określić
//wartość logiczną warunków składowych.

  wynik=++y < x || x++ == 3;
  …
```

### Przykład 1.4a wykonany jako Windows Forms Application

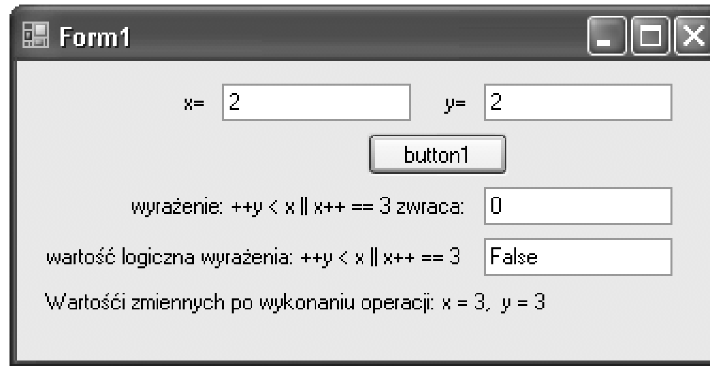
Kod procedury

```
private: System::Void button1_Click(System::Object^
         sender, System::EventArgs^  e) {
   int x, y, wynik;
   bool wynik_bool;
   x=Convert::ToInt16(txtX->Text);
   y=Convert::ToInt16(txtY->Text);

   wynik = ++y < x || x++ == 3;
   wynik_bool = Convert::ToBoolean(wynik);

   txtWynik->Text=Convert::ToString(wynik);
   txtWynik_bool->Text=Convert::ToString(wynik_bool);

   lblZmienne->Text =
     "Wartośći zmiennych po wykonaniu operacji: " +
     "x = " + Convert::ToString(x) +
     ",  y = " + Convert::ToString(y);
 }
```

Widok okna aplikacji, dla różnych danych, rysunki 1.27 i 1.28.




Rysunek 1.27.




Rysunek 1.28.

Przeanalizuj przykład 1.4 dla przypadku innych operatorów ++ i - - oraz dla &&.

## Instrukcja warunkowa

W języku C dostępna jest instrukcja warunkowa, która podobnie jak w innych algorytmicznych językach programowania może przyjmować postać blokową. Składa się ona, w podstawowej postaci, z następujących składników: linii instrukcji **if** z warunkiem umieszczonym w ( ), i bloku określonego klamrami **{ … }**.

```
if (<wyrażenie> )
{…….
…….. . .
…….. . .
…….. . .
}
…….. . .
```

W trakcie wykonywania instrukcji następuje sprawdzenie prawdziwości wyrażenia zawartego w **( )**. Jeżeli jest ono prawdą to nastąpi wykonanie ciągu instrukcji zawartych w bloku określonym za pomocą klamer **{… }**. Jeżeli nie jest prawdą, to blok określony klamrami zostanie pominięty i wykonanie programu będzie się odbywać dalej, zgodnie z pierwszą instrukcją umieszczoną po klamrze zamykającej **}**.

Innym wariantem instrukcji warunkowej jest wariant z dwoma blokami:

```
if (<wyrażenie> )
{…….
…….. .
…….. .
…….. .
}
else
{…….
…….. .
…….. .
…….. .
}
```

W tym przypadku po sprawdzeniu warunku i stwierdzeniu jego prawdziwości nastąpi przejście do bloku (określonego klamrami) znajdującego się bezpośrednio po linii z instrukcją **if ( )**. Po wykonaniu instrukcji zawartych w tym bloku przetwarzanie programu przejdzie do pierwszej instrukcji po całej konstrukcji **if ()** (po obu blokach). Jeżeli nie zostanie stwierdzona prawdziwość warunku to nastąpi przejście do bloku drugiego, czyli znajdującego się po linii **else**. Po wykonaniu instrukcji zawartych w tym bloku będzie wykonywana pierwsza instrukcja po całej konstrukcji **if ()** .

# Instrukcja cyklu while

Konstrukcją bardzo podobną do instrukcji warunkowej z jednym blokiem jest **while ()**. W tym przypadku w zależności od prawdziwości warunku zawartego w wyrażeniu każdorazowo wykonywane są instrukcje zawarte w bloku ograniczonym klamrami **{ … }**.

```
while (<wyrażenie> )
{…….
…….. .
…….. .
…….. .
}
…….. .
```

Istnieje jeszcze inna postać instrukcji z **while** ():

```
do
{…….
……...
……...
……...
} while (<wyrażenie> );
```

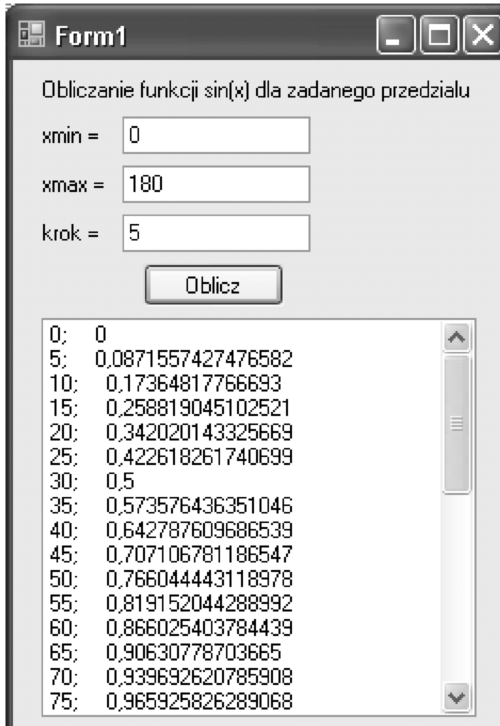
Instrukcja ta jest wykonywana po raz pierwszy bez sprawdzania prawdziwości wyrażenia umieszczonego po while. Warunek ten jest sprawdzany po raz pierwszy dopiero po jednokrotnym wykonaniu instrukcji znajdujących się w bloku { … }. W zależności od jego prawdziwości nastąpi ponowne wykonanie instrukcji zawartych w bloku, lub nie, w przypadku niespełnienia tego warunku. Poniżej przedstawiono przykłady wykorzystania instrukcji while do tablicowania funkcji. W przykładzie 1.5 odbywa się tylko obliczanie wartości funkcji w określonym przedziale z określonym krokiem. W przykładzie 1.6 proces przetwarzania odbywa się podobnie. Ze względu na zastosowanie funkcji pierwiastek przeprowadzane jest każdorazowe sprawdzenie znaku zmiennej pierwiastkowanej. Wykorzystana do tego celu jest instrukcja warunkowa oraz generowany jest odpowiedni komunikat. W przedstawionym przykładzie wykorzystano makrodefinicję #include, która powoduje wstawienie w to miejsce wyspecyfikowanego dalej pliku. Wstawiane są pliki zawierające funkcje związane z operacjami wejścia/wyjścia oraz funkcje matematyczne.

### Przykład 1.5 wykonany jako Console Application

```
#include "stdafx.h"
#include <iostream>
#include <math.h>

using namespace System;
using namespace std;


void main()
{
    double x, y, xmin, xmax, krok;
    cout <<
     "\nObliczanie sin(x) dla zadanego przedzialu " ;

    xmin = -180.0;
    xmax = 180.0;
    krok = 10.0;
```

```
        x=xmin ;
        while (x <= xmax)
        {
              y = sin(x * Math::PI / 180.0);
              cout <<  "\nx= " << x <<" y= " << y ;
              x = x + krok;
        }
        cout <<"\n";
        getchar();
}
```

**Przykład 1.4a wykonany jako Windows Forms Application**




Rysunek 1.29. Widok okna aplikacji w trakcie działania

Kod procedury

```
private: System::Void txtOblicz_Click(System::Object^
         sender, System::EventArgs^  e) {
     double x, y, xmin, xmax, krok;
     xmin = Convert::ToDouble(txtXmin->Text);
     xmax = Convert::ToDouble(txtXmax->Text);
     krok = Convert::ToDouble(txtKrok->Text);

     x = xmin;
     while (x <= xmax)
     {
           y = Math::Sin( x * Math::PI / 180);
           listBox1->Items->Add(x + ";     " + y);
           x = x + krok;
     }
 }
```

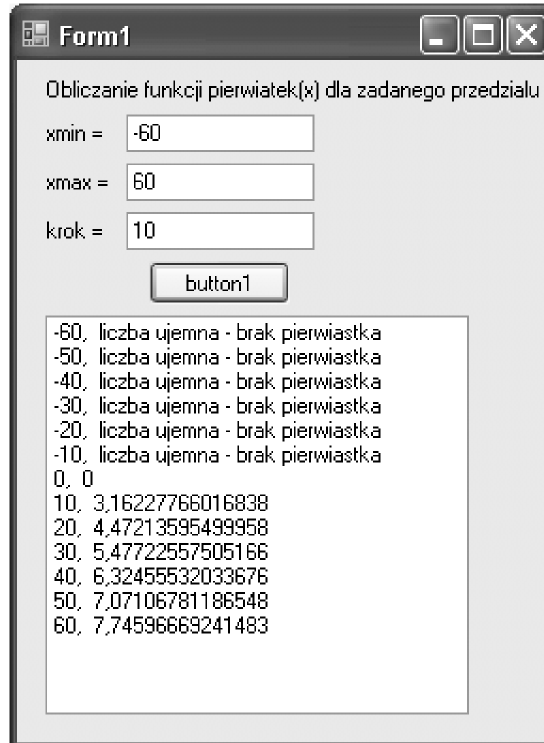
**Przykład 1.6 wykonany jako Console Application**

```
#include "stdafx.h"
#include <iostream>
#include <math.h>

using namespace System;
using namespace std;

void main()
{
    double x, y, xmin, xmax, krok;

    cout <<  "\nObliczanie funkcji pierwiatek(x) " ;
    cout <<  "dla zadanego przedzialu " ;

    xmin = -60;
    xmax = 60;
    krok= 10;

    x=xmin ;
    while (x <= xmax)
    {
      if (x < 0)
        {
         cout<< "\nx = " << x  <<
           "   liczba ujemna-brak pierwiastka ";
        }
      else
        {
```

```
            y = pow(x,0.5);
            cout <<  "\nx = " << x <<" y = " << y ;
            }
        x = x + krok;
      }
      cout <<  "\n "  ;
      getchar();

}
```

**Przykład 1.6a wykonany jako Windows Forms Application**

Postać aplikacji w działaniu przedstawia rysunek 1.30.



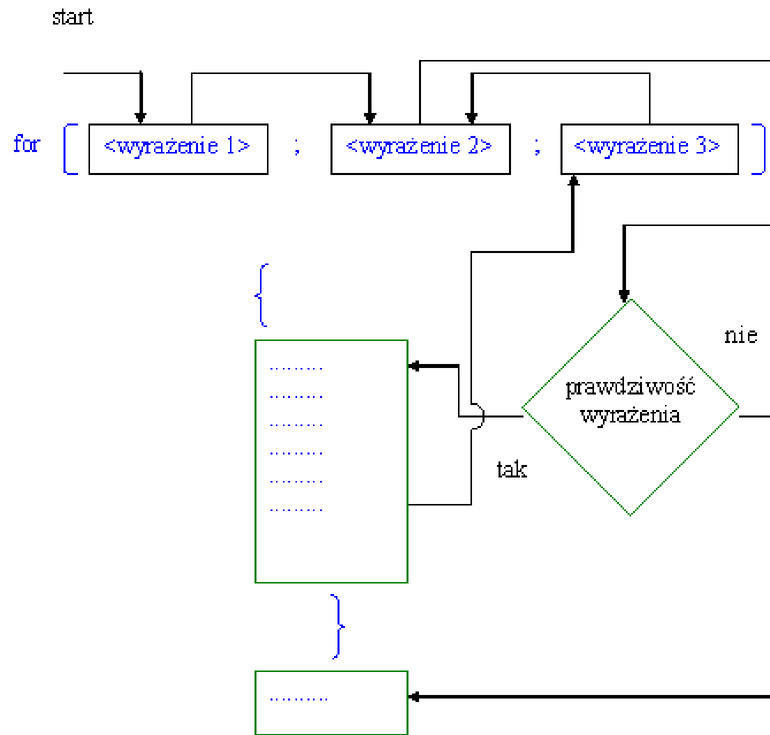

Rysunek 1.30. Widok okna aplikacji w trakcie działania

Kod aplikacji

```
private: System::Void button1_Click(System::Object^
         sender, System::EventArgs^  e) {
     double x, y, xmin, xmax, krok;
     xmin = Convert::ToDouble(txtXmin->Text);
     xmax = Convert::ToDouble(txtXmax->Text);
     krok = Convert::ToDouble(txtKrok->Text);
     x = xmin;
     while(x <= xmax)
      {
          if (x<0)
           {
             listBox1->Items->Add(x + ",  " +
               "liczba ujemna - brak pierwiastka");
           }
          else
           {
             y = Math::Sqrt(x);
             listBox1->Items->Add(x + ",  " + y);
           }
          x = x + krok;
      }
 }
```

## Instrukcja cyklu for

Instrukcja cyklu **for** pozwala na zaplanowanie krotności wykonania instrukcji cyklicznych. W języku C pozwala także na dosyć niestandardowe usytuowanie poszczególnych składników instrukcji **for**. Ogólny schemat zarówno wykonania jak i działania tej konstrukcji przedstawiono na rysunku 1.31.




Rysunek 1.31. Schemat i funkcjonowanie instrukcji for

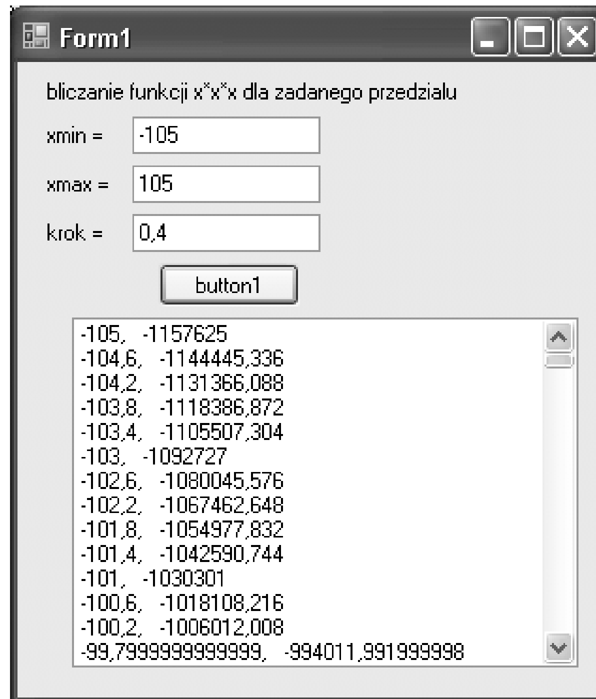
**Przykład 1.7 wykonany jako Console Application**

```
#include "stdafx.h"
#include <iostream>
#include <math.h>

using namespace System;
using namespace std;

void main()
{
    double x, y, xmin, xmax, krok;

    cout <<  "\nObliczanie funkcji x*x*x " ;
    cout <<  " dla zadanego przedzialu " ;

    xmin = -15.0;
    xmax = 15.0;
    krok = 0.4;
```

```
    for (x = xmin ; x <= xmax ; x = x + krok)
          {
                y=x*x*x;
                cout <<  "\nx= " << x <<" y= " << y;
          }
    cout <<  "\n ";
    getchar();
}
```

**Przykład 1.7a wykonany jako Windows Forms Application**

Postać aplikacji w działaniu przedstawia rysunek 1.32.




Rysunek 1.32. Widok okna aplikacji w trakcie działania

Kod aplikacji

```
private: System::Void button1_Click(System::Object^
         sender, System::EventArgs^  e) {

     double x, y, xmin, xmax, krok;

     xmin = Convert::ToDouble(txtXmin->Text);
     xmax = Convert::ToDouble(txtXmax->Text);
     krok = Convert::ToDouble(txtKrok->Text);

     for (x = xmin; x <= xmax; x = x + krok)
           {
             y=x*x*x;
             listBox1->Items->Add(x + ",    " + y);
           }
  }
```

# Konstrukcja switch

Bardzo przydatną konstrukcją w wielu działaniach praktycznych jest konstrukcja **switch**. W zależności od wartości wyrażenia całkowitoliczbowego następuje przejście do określonej pozycji case, tej której odpowiada właściwa stała stala1, itd.:

```
switch (<wyrażenie całkowitoliczbowe>){
case stala 1:
   instrukcja 1
case stala 2:
   instrukcja 2
.
.
.
case stala n:
   instrukcja n
}
```

Innym wariantem instrukcji switch jest konstrukcja oparta na wykorzystaniu składnika default, który jest wykonywany w przypadku braku stałej odpowiadającej wartości badanego wyrażenia całkowitoliczbowego. Rozwiązanie to pozwala na zabezpieczenie procesu wykonania całej instrukcji.

```
switch (<wyrażenie całkowitoliczbowe>){
case stala 1:
   instrukcja 1
case stala 2:
   instrukcja 2
.
.
.
case stala n:
   instrukcja n
default:
   instrukcja

}
```

## Funkcja printf()

Posługując się funkcją **printf** () wykorzystujemy różne wzorce formatów dla różnych typów zmiennych. Pokazane to jest na przykładzie:

```
int x=423;
float z=423.123;

printf("\n  %d   %f", x, z);
printf("\n  %6d   %10f", x, z);
printf("\n  %3d   %3f", x, z);
printf("\n  %10.3f   %10.3e", z, z);
```

## Komentarze

W języku C++ linie komentarza, ignorowane przez kompilator, zaznacza się w następujący sposób:

```
// - komentarz po tych znakach może występować do
//   końca linii,
/*
- komentarz, który może występować pomiędzy tymi
znakami może obejmować kilka linii.
*/
```

# Podsumowanie i uzupełnienia

W rozdziale omówiono podstawowe zagadnienia związane z tworzeniem elementarnych programów pisanych w języku C/C++. W trakcie prezentacji skoncentrowano się przede wszystkim na konstrukcjach podstawowych dla języka C i jednocześnie obecnych w języku C++. Przedstawiono zasadniczą strukturę programu, operatory arytmetyczne, logiczne, relacyjne, specjalne oraz instrukcję warunkową i instrukcje cyklu.

Poniżej dodano pewne uzupełnienia, które nie są istotne dla zasadniczych wątków rozdziału, niemniej jednak czynią bardziej kompletnymi prezentowane treści.

### Nazwy zmiennych

Nazwy zmiennych w języku C/C++ mogą w pełni opisywać sens przechowywanych wartości liczbowych. Mogą one być dosyć długie, rozróżniane są litery duże i małe. W nazwach nie mogą występować spacje, pierwszym znakiem nazwy nie może być cyfra.

### Deklarowanie zmiennych i przypisywanie wartości

Deklarując zmienne można jednocześnie przypisać im wartości początkowe za pomocą operatora przypisania: =.

### Instrukcje złożone

Programy pisane w języku C/C++ składają się z instrukcji, z których każda kończy się znakiem: **;** . Często mówi się o tzw. instrukcjach złożonych, które stanowią ciąg instrukcji zakończonych znakiem: **;**, ale ich całość objęta jest klamrami **{ }**, tworząc jedną instrukcję złożoną (zwaną niekiedy blokiem).

### Wyrażenie logiczne – zmienne logiczne

Dawne wersje języka C/C++ stosowały do reprezentowania prawdziwości bądź fałszywości wyrażeń zmienne całkowite. W standardzie ANSI wprowadzono typ logiczny zmiennych: bool.

### Kolejność działań w wyrażeniach logicznych

Budując złożone wyrażenia logiczne należy pamiętać, że kolejność wykonywania wchodzących w ich skład operacji może niekiedy rozmijać się z intuicją programującego. Dlatego dobrymi rozwiązaniami są: zapo-

znanie się z priorytetami działań wprowadzonymi w używanym kompilatorze, stosowanie nawiasów zagnieżdżonych (pozwalają one określać priorytety programującemu i jednocześnie czynią program czytelniejszym).

#### Deklarowanie i definiowanie funkcji

W rozdziale tym wprowadzono pojęcie funkcji. Funkcje mogą być deklarowane i definiowane. Deklarowanie funkcji oznacza określenie jej nazwy, typu zwracanej wartości oraz listy jej parametrów (niekiedy deklaracja funkcji nazywana jest prototypem funkcji). Definiowanie funkcji dotyczy zapisu zawartości funkcji. Nie można wywoływać funkcji przez inną funkcję jeżeli nie wystąpiła deklaracja tej funkcji.

#### Konstrukcja goto

Poza instrukcjami cyklu przedstawionymi w rozdziale dostępna jest jeszcze instrukcja **goto**. Jest to najstarsza historycznie instrukcja tego typu. Po instrukcji goto występuje etykieta, do której ma nastąpić przeskok w wyniku wykonania instrukcji goto. Obecnie jest ona używana stosunkowo rzadko. Wynika to z faktu słabej czytelności kodu z instrukcjami goto.

#### Instrukcje continue i break

Dodatkowo w instrukcjach cyklu dostępne są jeszcze instrukcje **continue** (dokonuje ona skoku do początku pętli jeszcze przed zakończeniem aktualnej iteracji) i **break** (przerywa wykonywanie pętli i powoduje przeskok do pierwszej instrukcji po instrukcji cyklu).

## Zadania do samodzielnego wykonania

1. Dla rzutu ukośnego zbuduj program obliczający dla ustalonych danych początkowych (dane: prędkość początkowa, kąt nachylenia) położenia ciała w kolejnych chwilach czasowych. Wyniki zaprezentuj w postaci wydruku w formie tabeli.

2. Dla rzutu ukośnego zbuduj program obliczający dla ustalonej prędkości początkowej kąt nachylenia, dla którego zasięg rzutu jest największy. Przeszukaj iteracyjnie dopuszczalny przedział zmienności kąta nachylenia.

3. Zbuduj program obliczający objętość bryły powstałej w wyniku obrotu krzywej $y=x^2$ wokół osi OX. Objętość oblicz w prze-

dziale <0, 20>. Obliczenia wykonaj iteracyjnie przybliżając bryłę zbiorem stożków ściętych. Oblicz objętość każdego stożka i zsumuj objętości wszystkich stożków.

4. Zbuduj program obliczający kolejne położenia, w kolejnych chwilach czasowych, pociągu poruszającego się od stacji A do stacji B odległych o L z prędkością v.

5. Zbuduj program obliczający kolejne położenia, w kolejnych chwilach czasowych, 3 pociągów wyruszających kolejno o godzinie 16:00, 17:00 i 18:00 z miejscowości A do B (odległych o L) z prędkościami odpowiednio v1, v2, v3. Pociągi poruszają się tym samym torem. Zabezpiecz w programie możliwość zbliżenia się pociągów do siebie na mniej niż 1km.

# Funkcje i struktura programu

# Funkcje

Programy pisane w języku C zwykle składają się z większej liczby segmentów, z których każdy to funkcja. Przetwarzanie odbywa się poprzez kolejne odwoływanie się do poszczególnych funkcji. Funkcje mogą być uruchamiane z innych funkcji. Jeżeli w funkcji **A** nastąpi odwołanie do funkcji **B** to w tym momencie przetwarzanie zostanie przekazane do funkcji **B**. Po wykonaniu funkcji **B** nastąpi powrót do funkcji **A** do miejsca gdzie była wywołana funkcja **B**.

Każda funkcja posiada wyraźnie wyodrębniony początek – nagłówek funkcji. Następnie znajdują się instrukcje przetwarzające i całość zamyka zakończenie funkcji.

Zasadniczym celem podziału programu pisanego w C na funkcje jest zapewnienie modularności oprogramowania, która z kolei sprawia, że łatwiej jest pisać i testować programy pisane w C. Łatwiejsze może się także stać ponowne wykorzystanie fragmentów programów napisanych wcześniej.

Funkcja może posiadać opcjonalnie listę parametrów (zwanych argumentami funkcji) za pośrednictwem, których może odbywać się przekazywanie wielkości liczbowych. Funkcja może zwracać wielkość wynikową pod swoją nazwę. Zwykle określamy typ zwracanej wartości określając typ funkcji.

Przedstawimy te zagadnienia na przykładzie.

### Przykład 2.1 wykonany jako Console Application

```
float wartosc_wieksza(liczba1, liczba2)
float liczba1;
float liczba2;
{
  if (liczba1 > liczba2 )
    return(liczba1)
  else
    return(liczba2)
}
```

Funkcja **wartosc_wieksza ()** ma za zadanie wybrać większą z dwóch liczb **liczba1** i **liczba2**, i właśnie ją zwrócić pod nazwę funkcji. Zarówno **liczba1** jak i **liczba2** są typu rzeczywistego **float**. Również typ funkcji w związku z tym, że zwracana liczba jest typu **float** musi być **float** (typ

**float** umieszczony przed nazwą funkcji). Sama funkcja jest stosunkowo prosta – występuje tam jedna instrukcja warunkowa. W przykładzie poszczególne bloki składają się z pojedynczych instrukcji co w konsekwencji pozwala pominąć blokowe klamry instrukcji warunkowej. Instrukcja return powoduje powrót do miejsca wywołania funkcji. Po instrukcji return może wystąpić wyrażenie, którego wartość jest zwracana przez funkcję.

## Deklaracje i działanie funkcji

W języku C trzeba określić typ zwracanej wartości przez funkcję. Oznacza to konieczność określenia w nagłówku funkcji typu zwracanej wartości. Jeżeli funkcja nie zwraca żadnej wartości to w nagłówku piszemy **void**. W starszych kompilatorach języka C pominięcie określenia typu funkcji oznaczało typ całkowity **int**.

Funkcja stanowi jedną całość, nie można definiować innej funkcji wewnątrz danej funkcji. Funkcje są wykonywane jak odrębne programy – kolejno są przetwarzane poszczególne instrukcje aż do ostatniej lub do instrukcji **return**. Instrukcja **return** może występować sama lub wraz z towarzyszącym wyrażeniem. Wartość obliczonego wyrażenia jest odpowiednio zwracana przez wykonaną funkcję.

W trakcie wywołania funkcji musi nastąpić dopasowanie parametrów aktualnych i argumentów funkcji z punktu widzenia ich ilości i typów. Pokażemy to zagadnienie na przykładzie z poprzedniego rozdziału.

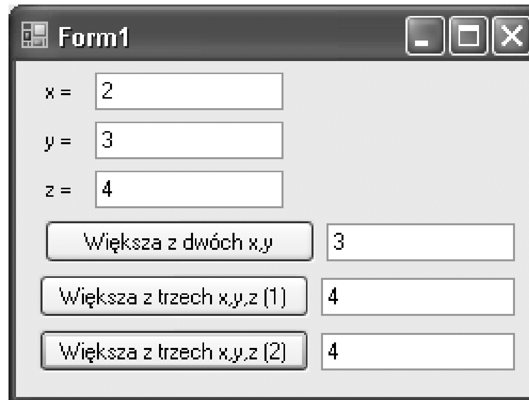
### Przykład 2.2 wykonany jako Console Application

```
void main()
{
float a= 5;
float b= 10;
float wynik;
wynik=wartosc_wieksza(a, b);
printf(„%f „ wynik);
}

float wartość_wieksza(liczba1, liczba2)
float liczba1;
float liczba2;
{
if (liczba1 > liczba2 )
    return(liczba1)
```

```
else
    return(liczba2)
}
```

**Przykład 2.2a wykonany jako Windows Forms Application**

W przykładzie pokazano dodatkowo rekurencję czyli sytuację, w której funkcja może wywoływać sama siebie.




Rysunek 2.1. Postać aplikacji w działaniu

Kod aplikacji

```
//---------------------------------------------------
double wartosc_wieksza(double liczba1,
                       double liczba2){
//Własna funkcja zwracającą większą z dwóch liczb.
     if (liczba1 > liczba2)
     {
           return liczba1;
     }
     else
     {
           return liczba2;
     }
}
//---------------------------------------------------
private: System::Void btnWiększaZdwochXY_Click
          (System::Object^  sender,
          System::EventArgs^  e) {
      double x, y;
      x = Convert::ToDouble(txtX->Text);
      y = Convert::ToDouble(txtY->Text);
      // wywołanie funkcji
```

```
     txtWiekszaZdwochXY->Text =
         Convert::ToString(wartosc_wieksza(x,y));
}
//---------------------------------------------------
private: System::Void btnWiekszaZtrzechXYZ_1_Click
        (System::Object^  sender,
         System::EventArgs^  e) {
 //Wersja pierwsza - funkcja wywoływana pojedynczo
     double x, y, z, wiekszaXY, wiekszaXYZ;
     x = Convert::ToDouble(txtX->Text);
     y = Convert::ToDouble(txtY->Text);
     z = Convert::ToDouble(txtZ->Text);

     // wywołanie funkcji dla zmiennych x i y

     wiekszaXY = wartosc_wieksza(x,y);

     // wywołanie funkcji dla zmiennych wiekszaXY i z

     wiekszaXYZ = wartosc_wieksza(wiekszaXY,z);

     txtWiekszaZtrzechXYZ_1->Text =
         Convert::ToString(wiekszaXYZ);
 }
//---------------------------------------------------
private: System::Void btnWiekszaZtrzechXYZ_2_Click
        (System::Object^  sender,
         System::EventArgs^  e) {
//Wersja druga - funkcja wywołuje sama siebie
//(rekurencja)
     double x, y, z, wiekszaXY, wiekszaXYZ;
     x = Convert::ToDouble(txtX->Text);
     y = Convert::ToDouble(txtY->Text);
     z = Convert::ToDouble(txtZ->Text);

 // wywołanie funkcji w funkcji
     txtWiekszaZtrzechXYZ_2->Text =
       Convert::ToString(
       wartosc_wieksza(wartosc_wieksza(x,y),z));
 }
//---------------------------------------------------
```

## Zmienne, zasięg, typy

Zmienne stosowane w języku C mają swój obszar w programie, w którym są widoczne tzn. mogą być w zakresie tego obszaru używane do

przechowywania wielkości liczbowych. Np. zmienne lokalne, zadeklarowane wewnątrz funkcji są widoczne jedynie (w danej funkcji) w trakcie gdy odbywa się przetwarzanie instrukcji tej funkcji.

Jeżeli program jest zawarty w jednym pliku to można w tym przypadku wyodrębnić zmienne **auto**, **extern** i **static**.

Zmienne auto są widoczne tylko w jednym bloku, w którym zostały zdefiniowane. Przy czym przez blok rozumiemy fragment programu ograniczony przez klamry **{ }**. Pamięć dla tych zmiennych jest przygotowywana w momencie gdy sterowanie wykonaniem programu dochodzi do danego bloku. Dalej zmienne widoczne są w trakcie wykonywania instrukcji zawartych w tym bloku. Po wykonaniu instrukcji należących do bloku następuje zwolnienie obszaru pamięci wykorzystywanego na te zmienne. Można nie pisać słowa **auto**.

Zmienna **extern** jest definiowana poza funkcjami i jest widoczna we wszystkich funkcjach od miejsca zdefiniowania do końca pliku. Słowo **extern** musi występować zawsze.

Zmienne statyczne **static** są definiowane raz na całe życie programu, jeszcze zanim program jako całość będzie wykonywany. Zmienne statyczne mogą być definiowane wewnątrz i na zewnątrz ciała funkcji. Określenie static musi wystąpić zawsze. Pamięć dla zmiennej statycznej jest przygotowywana na czas życia programu. Jeżeli zmienna statyczna jest definiowana na zewnątrz ciała funkcji to będzie widoczna we wszystkich funkcjach do końca pliku; jeżeli będzie definiowana wewnątrz ciała funkcji będzie widoczna w zawierającym ją bloku.

W związku z powyższym zmienna może być widoczna w jednej lub w większej liczbie funkcji.

Oprócz zasięgu zmiennej określa się również jej typ. Poza scharakteryzowanymi do tej pory typem całkowitym i rzeczywistymi **int**, i **float**, **double** istnieją inne. Często używanym jest typ znakowy **char**.

Wszystkie zmienne występujące w programie mają określony swój zasięg działania oraz typ przechowywanych wielkości.

Poza wymienionymi typami zasięgu występuje jeszcze zasięg typu **register**. Zdefiniowanie tego typu zmiennych ma charakter postulatywny, oznacza postulat zapewnienia, w miarę możliwości, możliwie dużej szybkości przetwarzania tych zmiennych.

# Przykłady funkcji

Pokażemy na kilku przykładach koncepcje stosowania funkcji w budowie oprogramowania. Przykład 2.3 ilustruje w jaki sposób budując program poszukujący minimum funkcji w zadanym przedziale podzielić go na dwie funkcje składowe. Z uwagi na to iż może się zdarzyć, że funkcja matematyczna, której minimum jest poszukiwane, w przyszłości, może ulec zmianie wygodnie jest formułę minimalizowanej funkcji zapisać w postaci osobnej funkcji w języku C – w przykładzie funkcja o nazwie **f()**.

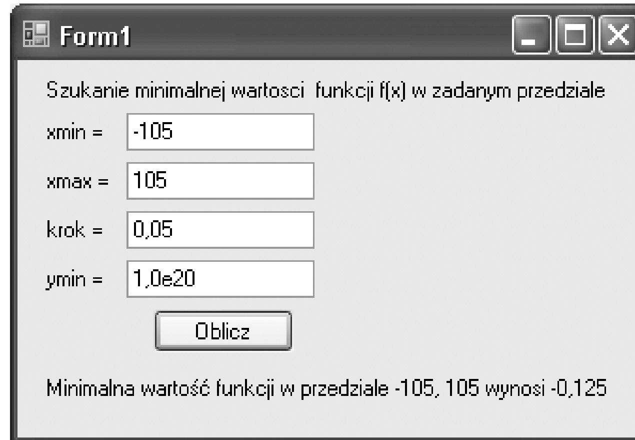
Wyznaczanie minimum odbywa się w tym przypadku w oparciu o przetwarzanie zapisane w funkcji **main()**.

### Przykład 2.3 wykonany jako Console Application

```
#include <iostream.h>
#include <math.h>

//--------------------------------------------------
double f(double x )
{
    return 2*x*x + 3*x  + 1;
}
//--------------------------------------------------
main()
{
    double x, y, xmin,xmax, krok;
    double  ymin;

    cout <<  "\nSzukanie minimalnej wartosci " ;
    cout <<  " funkcji f(x) w zadanym przedziale " ;

    xmin = -105;
    xmax = 105;
    krok = 0.05;
    ymin = 1.0e20;

    for (x = xmin ; x <= xmax ; x = x + krok)
    {
          y=f(x);
          if (y < ymin)
          {
                ymin = y;
          }
    }
```

```
    cout <<  "\n Minimalna wartosc funkcji f(x) w
przedziale  <" <<xmin <<","<< xmax<<
                "> wynosi "<< ymin <<"\n";
    cin>>x;
}
```

**Przykład 2.3a wykonany jako Windows Forms Application**




Rysunek 2.2. Formularz podczas pracy aplikacji

Kod aplikacji

```
double f(double x){
    return 2*x*x + 3*x + 1;
}
//---------------------------------------------------
private: System::Void btnOblicz_Click(System::Object^
         sender, System::EventArgs^  e) {
     double x, y, xmin,xmax, krok;
     double  ymin;
     xmin = Convert::ToDouble(txtXmin->Text);
     xmax = Convert::ToDouble(txtXmax->Text);
     krok = Convert::ToDouble(txtKrok->Text);
     ymin = Convert::ToDouble(txtYmin->Text);

     for (x=xmin ; x<=xmax ; x=x+krok)
     {
           y=f(x);
           if(y<ymin){
                 ymin=y;
           }
```

```
      }
      lblWynik->Text =
           "Minimalna wartość funkcji w przedziale "
            + Convert::ToString(xmin) + ", "
            + Convert::ToString(xmax) + " wynosi "
            + Convert::ToString(ymin);
 }
```

W przykładzie 2.4 pokazano to samo zagadnienie zrealizowane w nieco inny sposób. Przetwarzanie realizowane w funkcji **main()** w przykładzie 2.3 odbywa się teraz w funkcji **minimum()**.

### Przykład 2.4 wykonany jako Console Application

```
#include <math.h>

//----------------------------------------------------
double f(double x )
{
     return 2*x*x + 3*x  + 1;
}
//---------------------------------------------------
double minimum(double xmin,double xmax, double krok)
{
     double x, y;
     double  ymin;

     ymin=1.0e20;
     for (x=xmin ; x<=xmax ; x=x+krok)
     {
           y=f(x);
           if (y<ymin)
           {
                 ymin=y;
           }
     }
     return ymin;
}
```

**Przykład 2.4a wykonany jako Windows Forms Application**

Form1

Szukanie minimalnej wartosci funkcji f(x) w zadanym przedziale

xmin = -105

xmax = 105

krok = 0,05

ymin = 1,0e20

Oblicz

Minimalna wartość funkcji w przedziale -105, 105 wynosi -0,125

Rysunek 2.3. Formularz podczas pracy aplikacji

Kod aplikacji

```
double f(double x )
{
     return 2*x*x + 3*x  + 1;
}
//-------------------------------------------------------
double minimum(double xmin,double xmax, double krok)
{
     double x, y;
     double  ymin;
     ymin = Convert::ToDouble(txtYmin->Text);
     for (x=xmin ; x<=xmax ; x=x+krok)
            {
                   y=f(x);
                   if (y<ymin)
                          {
                                 ymin=y;
                          }
            }
     return ymin;
}
//-------------------------------------------------------
private: System::Void btnOblicz_Click(System::Object^
          sender, System::EventArgs^  e) {
      double xmin, xmax, krok;
      xmin = Convert::ToDouble(txtXmin->Text);
      xmax = Convert::ToDouble(txtXmax->Text);
```

```
     krok = Convert::ToDouble(txtKrok->Text);

     lblWynik->Text =
     "Minimalna wartość funkcji w przedziale "
      + Convert::ToString(xmin) + ", "
      + Convert::ToString(xmax) + " wynosi "
      + Convert::ToString(minimum(xmin, xmax, krok));

}
```

W przykładzie 2.5 zaprezentowano sposób wykorzystania funkcji minimum(), która stała się w tym przypadku bardziej uniwersalnym narzędziem.

**Przykład 2.5 wykonany jako Console Application**

```
//---------------------------------------------------
main()
{
    double  xmin,xmax, krok;
    double  ymin;

    printf("\nSzukanie minimalnej wartosci funkcji ");
    printf("f(x) w zadanym przedziale ") ;

    xmin = -50;
    xmax = 50;
    delta = 0.1;

    ymin = minimum(xmin,xmax,krok);
//         =======

    printf( "\n Minimalna wartosc funkcji f(x) w
przedziale xmin= %f  xmax= %f \n wynosi %f ", xmin,
xmax, ymin );
    scanf("%f", &xmin);

}
//---------------------------------------------------
```

Kolejny przykład to program realizujący funkcje prostego kalkulatora. Sterowanie pracą kalkulatora odbywa się za pomocą zmiennej typu **char** o nazwie **znak**. Przetwarzanie jest zorganizowane za pomocą szeregu funkcji odpowiadających poszczególnym działaniom kalkulatora.

**Przykład 2.6 wykonany jako Console Application**

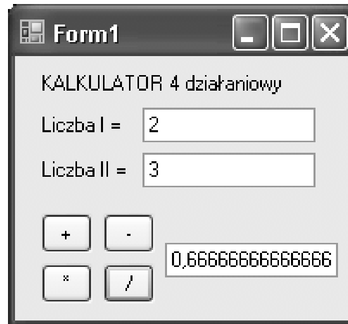
```
main ()
{
    float a, b, c;
    float dodaj(float a, float b),
          odejmij (float a, float b),
          mnoz (float a, float b),
          dziel (float a, float b);
    char znak;

    printf(" KALKULATOR - 4 działaniowy \n");
    printf(" +  dodawanie \n");
    printf(" -  odejmowanie \n");
    printf(" *  mnożenie \n");
    printf(" /  dzielenie \n");
    printf(" wybierz -> ");
    scanf("%c", &znak);
    printf(" \n liczba I -> ");
    scanf("%f", &a);
    printf(" \n liczba II -> ");
    scanf("%f", &b);

    if (znak == '+')
          c= dodaj(a, b);
    if (znak == '-')
          c= odejmij(a, b);
    if (znak == '*')
          c= mnoz(a, b);
    if (znak == '/')
          c= dziel(a, b);

    printf(" wynik %f", c);
    scanf("%f", &c);

}
//---------------------------------------------------
    float dodaj(float a, float b)
    {return a+b;
    }
//---------------------------------------------------
    float odejmij(float a, float b)
    {return a-b;
    }
//---------------------------------------------------
    float mnoz(float a, float b)
    {return a*b;
    }
```

```
//--------------------------------------------------
    float dziel(float a, float b)
    {if (b == 0.0)
          {
          printf("\n dzielisz przez zero \n");
          return 0;
          }
    return a/b;
    }
//--------------------------------------------------
```

Koncepcja podziału na funkcje składowe programu napisanego w języku C wywiera bardzo duży wpływ na proces przetwarzania tego programu, oraz odgrywa dużą rolę w dalszym procesie jego rozwoju.

**Przykład 2.6a wykonany jako Windows Forms Application**

Kalkulator wykonany jako aplikacja formularza Windows jest jeszcze łatwiejsza do napisania i można ją zrealizować nawet bez zastosowania funkcji. Rodzaj działania wymuszany jest procedurą obsługi zdarzenia kliknięcie na klawisz z odpowiednią etykietą: +, -, *, /.




Rysunek 2.4. Postać aplikacji po kliknięciu przycisku dzielenie

Kod aplikacji

```
//--------------------------------------------------
private: System::Void btnDodaj_Click(System::Object^
         sender, System::EventArgs^  e) {
     //Dodawanie
     double liczba1, liczba2, wynik;
     liczba1=Convert::ToDouble(txtLiczbaI->Text);
     liczba2=Convert::ToDouble(txtLiczbaII->Text);
     wynik=liczba1+liczba2;
     txtWynik->Text = Convert::ToString(wynik);
 }
```

```
//--------------------------------------------------
private: System::Void btnOdejmij_Click(
   System::Object^  sender, System::EventArgs^  e) {
      //Odejmowanie
      double liczba1, liczba2, wynik;
      liczba1=Convert::ToDouble(txtLiczbaI->Text);
      liczba2=Convert::ToDouble(txtLiczbaII->Text);
      wynik=liczba1-liczba2;
      txtWynik->Text=Convert::ToString(wynik);
 }
//--------------------------------------------------
private: System::Void btnMnoz_Click(System::Object^
         sender,  System::EventArgs^  e) {
      //Mnożenie
      double liczba1, liczba2, wynik;
      liczba1=Convert::ToDouble(txtLiczbaI->Text);
      liczba2=Convert::ToDouble(txtLiczbaII->Text);
      wynik=liczba1*liczba2;
      txtWynik->Text=Convert::ToString(wynik);
 }
//--------------------------------------------------
private: System::Void btnDziel_Click(System::Object^
         sender,  System::EventArgs^  e) {
      //Dzielenie
      double liczba1, liczba2, wynik;
      liczba1=Convert::ToDouble(txtLiczbaI->Text);
      liczba2=Convert::ToDouble(txtLiczbaII->Text);
      wynik=liczba1/liczba2;
      txtWynik->Text=Convert::ToString(wynik);
 }
//--------------------------------------------------
```

## Podsumowanie i uzupełnienia

W rozdziale omówiono problematykę związaną z tworzeniem funkcji w programie pisanym w języku C/C++. Przedstawiono także zagadnienia związane z typami zmiennych oraz ustalaniem ich zasięgu. Poniżej przedstawimy istotne uzupełnienia.

### Typy zmiennych i zajmowana pamięć

Zmienne, jak już wspomniano mogą być określonego typu. Omówiono zmienne typu **int**, **float**, **char**. Zmienne zadeklarowane jako zmienne określonego typu zajmują określony obszar pamięci. Rozmiar tego obszaru zależy od typu zmiennej oraz niekiedy od użytego kompilatora.

Standardowa funkcja **sizeof()**, której parametrem może być określony typ zmiennej zwraca, w bajtach, rozmiar danego typu zmiennej.

### Typy całkowite zmiennych

Typy całkowite zmiennych mogą mieć wyspecyfikowaną możliwość przyjmowania lub nie wartości ze znakiem za pomocą **signed** oraz **unsigned**. Pozwala to w różnych przypadkach dla typów całkowitych dopuszczać lub nie możliwość przyjmowania wartości ujemnych.

### Typy zmiennych

Poza wymienionymi wcześniej typami zmiennych występują jeszcze inne. Pełną ich listę podajemy poniżej (różnią się one rozmiarem i zakresem przyjmowanych wartości):

```
unsigned short int
short int
unsigned long int
long int
char
bool
float
double
```

### Zmienne globalne

W rozdziale wspomniano o możliwości stosowania zmiennych globalnych. Nie jest to zalecane. Używając innych rozwiązań (będą omówione w dalszej części książki) - przede wszystkim zmiennych wskaźnikowych, można uniknąć tego typu konstrukcji.

## Zadania do samodzielnego wykonania

1. Zbuduj program obliczający pole powierzchni ścian pokoju w kształcie prostopadłościanu, o wymiarach a, b, c, odejmując daną liczbę drzwi o wymiarach 2m * 1m, i daną liczbę okien o wymiarach 1.5m * 1.5 m. Napisz program w formie funkcji.

2. Zbuduj program obliczający powierzchnię wszystkich tynkowanych ścian wewnątrz budynku złożonego z n pomieszczeń w kształcie prostopadłościanu. Wykorzystaj funkcję z zadania 1).

3. Zbuduj dwie funkcje obliczające pola powierzchni fragmentów dachu w kształcie prostokąta i w kształcie trójkąta. Następnie zbuduj program do obliczania powierzchni dachu złożonego z n fragmentów w kształcie prostokąta i k fragmentów w postaci trójkąta.

# Tablice i wskaźniki

# Tablice

Podobnie jak w innych językach programowania również w języku C dostępne są obiekty typu tablice. Tablice stanowią uporządkowane zbiory obiektów tego samego typu.

Odwołanie do elementów tablicy następuje poprzez odwołanie się zarówno do nazwy tablicy jak i do indeksu określonego jej elementu.

Zdefiniowanie tablicy może nastąpić w sposób zbliżony do definicji zwykłej zmiennej, np.:

```
float silnik_moc[20];
```

Zapis ten oznacza, że tablica **silnik_moc** składa się z **20** elementów. Przy czym ich indeksy zmieniają się od **0** do **19**.

Odwołanie do określonego elementu tablicy może nastąpić poprzez wstawienie jako indeksu stałej np. **silnik_moc[7]**, poprzez wstawienie jako indeksu zmiennej **silnik_moc[i]** lub poprzez wstawienie jako indeksu wyrażenia np. **silnik_moc[i +2]**.

Elementy tablicy mogą być określonych typów. Mogą to być typy int, float, double, char.

Bardzo często definiując tablicę nadajemy jej elementom wartości początkowe np.:

```
static float silnik_moment[ ] = {200, 260, 280};
```

Tablice typu **char** mają swoją specyfikę. Jeżeli deklarujemy pojedynczą zmienną typu char to stanowi ona pojedyncze pole, np.

```
char typ_paliwa;
```

Możemy wówczas dokonać podstawienia:

```
typ_paliwa='D';
```

Stosowanym formatem (np. w funkcji **printf**()) jest w tym przypadku **%c**.

Jeżeli natomiast mamy ciąg znaków to możemy do jego składowania wykorzystać tablicę, np.:

```
char typ_nadwozia[6];
typ_nadwozia[6]= ”kombi”
```

Wykorzystanie ”” oznacza, że 5 kolejnych wyrazów to kolejne znaki słowa kombi, natomiast 6 znak to generowany automatycznie znak zakończenia ciągu znaków: **\0**. Formatem stosowanym do wydruku ciągu znaków jest %s. Obiekty tego typu nazywamy łańcuchami znaków lub po angielsku **stringami**.

Tablice wielowymiarowe definiujemy za pomocą większej niż jeden odpowiedniej liczby klamer. Np.

```
autobusy_typy[ ] [ ] [ ];
```

Tablica wielowymiarowa jest tablicą zbudowaną z wielu tablic.

# Wskaźniki

W wielu konstrukcjach programistycznych pisanych w języku C używa się wskaźników. Wskaźniki to nic innego tylko adresy zmiennych. Adresy mogą występować w postaci stałych wskaźnikowych – są to konkretne adresy, lub zmiennych wskaźnikowych, które z kolei są przeznaczone do przechowywania adresów określonych typów zmiennych.

W programach pisanych w języku C możemy definiować zmienne wskaźnikowe np.:

```
char *wskaźnik;
```

oznacza zdefiniowanie zmiennej wskaźnikowej przeznaczonej do przechowywania adresów zmiennych typu char.

Wskaźniki mogą być zwiększane i zmniejszane. Ich wartości mogą być efektem przeprowadzanych obliczeń. Możliwe jest również stosowanie operatorów.

Nazwa tablicy stanowi adres do tej tablicy (jest to stała wskaźnikowa).

Nazwa funkcji to również stała wskaźnikowa.

Zilustrujmy to zagadnienie przykładem elementarnych operacji wykonywanych na wskaźnikach.

### Przykład 3.1 wykonany jako Console Application

```
#include "stdafx.h"
#include <iostream>

using namespace System;

//-----------------------------------------------
void main()
{
char *char_wskaznik;
char char_zmienna1;
char char_zmienna2;

char_zmienna2 = 'S';

char_wskaznik = &char_zmienna2;
char_zmienna1 = *char_wskaznik;

printf("\n char_zmienna  %c", char_zmienna1);
}
//-----------------------------------------------
```

W powyższym przykładzie zadeklarowano trzy zmienne: dwie zmienne typu **char char_zmienna1**, **char_zmienna2** oraz jedną zmienną wskaźnikową przeznaczoną do przechowywania adresów dla zmiennych typu **char char_wskaznik**.

Instrukcja:

```
char_zmienna2 = 'S';
```

oznacza podstawienie **S** do zmiennej **char_zmienna2**.

Instrukcja:

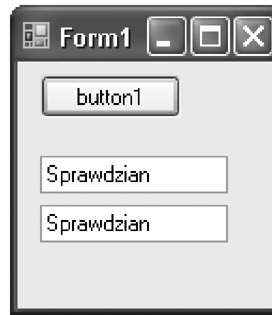
```
char_wskaznik = &char_zmienna2;
```

powoduje znalezienie adresu dla zmiennej **char_zmienna2** (służy do tego operator **&**) i podstawienie go do zmiennej **char_wskaznik**. Następnie za pomocą operacji (*****):

```
*char_wskaznik
```

jest znaleziona **zawartość** przechowywana **pod adresem** składowanym w zmiennej **char_wskaznik**. Znaleziona zawartość to nic innego jak **S**. Jest ona następnie podstawiana pod zmienną char_zmienna1, która z kolei jest w kolejnym kroku drukowana.

### Przykład 3.1a wykonany jako Windows Forms Application




Rysunek 3.1. Widok okna aplikacji w trakcie działania

Kod aplikacji

```
private: System::Void button1_Click(System::Object^
         sender,  System::EventArgs^  e) {

//Zmienna wskaźnikowa
     String^ *str_wskaznik;

//Pierwsza zmienna typu String
     String^ str_zmienna1;

//Druga  zmienna typu String
     String^ str_zmienna2;

//Odczyt tekstu do zmiennej str_zmienna2, typu String
     str_zmienna2 = textBox1->Text;

//Odczytanie adresu zmiennej str_zmienna2 do zmiennej
//str_wskaznik
     str_wskaznik = &str_zmienna2;

//Odczytanie zawartość zmiennej znajdującej się pod
//adresem znajdującym się w zmiennej str_wskaznik
//i podstawienie tej zawartości do zmiennej
//str_zmienna1.
     str_zmienna1 = *str_wskaznik;

//Wyświetlenie zawartości zmiennej str_zmienna1
//w textBox1.
     textBox2->Text = str_zmienna1;
}
```

# Przesyłanie wartości liczbowych do/z funkcji

W rozdziale zaostanie pokazane w jaki sposób przesyłać wartości liczbowe zmiennych do funkcji i w jaki sposób możemy je przesyłać z funkcji na zewnątrz.

Generalnie, wynik przetwarzania dokonanego w funkcji, jeżeli jest on jedną wartością liczbową, można przesłać na zewnątrz wykorzystując do tego celu return. Jest to wówczas wartość zwracana przez funkcję.

Jeżeli chodzi o wielkości liczbowe to do funkcji można je przesłać za pośrednictwem parametrów na liście parametrów funkcji. Pokażemy to poniżej na przykładzie.

### Przykład 3.2 wykonany jako Console Application

```
//-----------------------------------------------
void main()
{
int dana;
dana = 100;

funkcja (dana);
printf("\n a zmienna ma wartosc : %d", dana);
}
//----------------------------------------------
funkcja (x)
int x;
{
x = 20;
}
//----------------------------------------------
```

Efektem działania mechanizmu pokazanego powyżej jest to, że zmienna **dana** przyjmuje początkową wartość **100**. Wartość **100** jest przekazana przez zmienną **dana** zmiennej **x**. Z kolei zmienna **x** przyjmuje wartość **20**. Jednak zmienna **dana** w dalszym ciągu posiada wartość **100** i wartość ta ostatecznie zostanie wydrukowana. Omówiony mechanizm pozwala przekazywać wartości zmiennych do funkcji, nie pozwala natomiast ich przekazywać w kierunku przeciwnym. Nazywany jest *przekazywaniem przez wartość*.

### Przykład 3.2a wykonany jako Windows Forms Application

Rysunek 3.2. Widok okna aplikacji w trakcie działania

Kod aplikacji

```
//---------------------------------------------------
int funkcja (int x)
    {
          x = 20;
          return 0;
    }
//---------------------------------------------------
private: System::Void button1_Click(System::Object^
         sender, System::EventArgs^  e) {
     int dana;
     dana = Convert::ToInt16(txtDana->Text);
     funkcja(dana);
     txtWynik->Text = "Zmienna dana ma wartosc: " +
           Convert::ToString(dana);
 }
//---------------------------------------------------
```

Poniżej zostanie pokazany inny, oparty na wskaźnikach, mechanizm przekazywania wartości do funkcji i z funkcji. Zrobione to zostanie również na przykładzie.

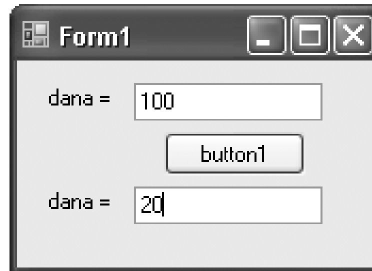
### Przykład 3.3 wykonany jako Console Application

```
//--------------------------------------------------
void main()
{
    int dana;
    int *wskaznik_danej;

    dana = 100;
    wskaznik_danej = &dana;
```

```
    funkcja (wskaznik_danej);

    printf( „ \n a zmienna ma wartosc : %d", dana);
}
//--------------------------------------------------
funkcja (wskaznik)
    int *wskaznik;
{
    *wskaznik = 20;
}
//--------------------------------------------------
```

W powyższym przykładzie pokazano, że zmienna **dana** przyjęła wartość **100**. Następnie zmienna wskaźnikowa **wskaznik_danej** przyjęła jako wartość **adres** zmiennej **dana**. Dalej zmienna wskaźnikowa **wskaznik_danej** przekazała swoją **wartość** (**adres**) zmiennej **wskaznik**. Wartość liczbowa przechowywana pod adresem umieszczonym w zmiennej **wskaznik** stała się dostępna w funkcji **funkcja()**. Można ją także zmienić – w funkcji dokonano podstawienia liczby **20** pod adresem przechowywanym w zmiennej **wskaznik** czyli dokonano zmiany wartości zmiennej **dana**. Wydruk końcowy to liczba **20**. Zaprezentowany mechanizm pozwala na przekazywanie wartości liczbowych zarówno do funkcji jak i z funkcji. Nazywa się on *przekazywaniem przez referencję*.

**Przykład 3.3a wykonany jako Windows Forms Application**




Rysunek 3.3. Widok okna aplikacji w trakcie działania

Kod aplikacji

```
//--------------------------------------------------
int funkcja (int *wskaznik)
{
    *wskaznik = 20;
    return 0;
}
```

```
//------------------------------------------------
private: System::Void button1_Click(System::Object^
         sender, System::EventArgs^  e) {
     int dana;
     int *wskaznik_danej;
     dana=Convert::ToInt16(txtDana->Text);
     wskaznik_danej = &dana;
     funkcja (wskaznik_danej);
     txtDanaNowa->Text=Convert::ToString(dana);
 }
//------------------------------------------------
```

# Tablice i wskaźniki

W języku C tablice pozwalają na wykonywanie bardzo wielu operacji na wskaźnikach. Stała będąca identyfikatorem tablicy występująca bez indeksu oznacza adres początku tablicy czyli jej pierwszego wyrazu. Kolejne wyrazy tablicy mają adresy powiększone o odpowiednie stałe. Zostanie to pokazane na niewielkim przykładzie.

Definiujemy tablicę:

```
int moc_pojazdu[4];
```

Wówczas adresem pierwszego wyrazu tablicy

```
moc_pojazdu[0]
```

jest:

```
moc_pojazdu
```

Kolejne adresy dla wyrazów:

```
moc_pojazdu[1]
moc_pojazdu[2]
moc_pojazdu[3]
```

to odpowiednio:

```
moc_pojazdu+1
moc_pojazdu+2
moc_pojazdu+3
```

Przedstawione związki adresowe pozwalają na bardzo efektywne wykonywanie wielu operacji na elementach tablic. Przede wszystkim nie

zawsze konieczne jest przemieszczanie zawartości elementów tablic. W wielu zastosowaniach wystarczają operacje na adresach.

Osobnego omówienia wymaga zagadnienie przesyłania tablic do funkcji. Tablice przesyłamy **przez adres**, natomiast rozmiar tablicy **przez wartość**. Poniżej pokazano to na przykładzie.

### Przykład 3.4 wykonany jako Console Application

```
//--------------------------------------------------
void main()
{
float suma_wyrazow(float *, int);
float tablica[5] = {3, 4, 5.1, 6.2, 2.1};

printf("suma elementów tablicy : %f \n",
       suma_wyrazow(tablica, 5) );
}
//--------------------------------------------------
float suma_wyrazow (float tab[], int n)
{
int i;
float suma = 0;
for (i=0 ; i<n ; i ++)
suma+=tab[i];
return suma;
}
//--------------------------------------------------
```

W przykładzie w funkcji **main** zdefiniowana jest tablica 5 elementowa o nazwie **tablica**. Jednocześnie nadane są wartości poszczególnym jej wyrazom. Dodatkowo umieszczono na początku informację, że **suma_wyrazow()** to funkcja. Następnie wywołana jest funkcja **suma_wyrazow(tablica, 5), tablica** występuje tu jako stała wskaźnikowa, natomiast **5** przekazuje liczbę wyrazów tablicy do funkcji **suma_wyrazow()**. W funkcji **suma_wyrazow ()** odbywa się proces sumowania wyrazów tablicy.

### Przykład 3.4a wykonany jako Windows Forms Application

Rysunek 3.4. Widok okna aplikacji w trakcie działania

Kod aplikacji

```
//---------------------------------------------------
double suma_wyrazow(double tab[], int n)
    {
          int i;
          double suma = 0;
          for (i=0 ; i<n ; i++)
                suma+=tab[i];
          return suma;
    }
//---------------------------------------------------
private: System::Void button1_Click(System::Object^
         sender, System::EventArgs^  e) {
     double tablica[5], suma;
     tablica[0] =
            Convert::ToDouble(txtTablica0->Text);
     tablica[1] =
            Convert::ToDouble(txtTablica1->Text);
     tablica[2] =
            Convert::ToDouble(txtTablica2->Text);
     tablica[3] =
            Convert::ToDouble(txtTablica3->Text);
     tablica[4] =
            Convert::ToDouble(txtTablica4->Text);
     tablica[5] =
```

```
              Convert::ToDouble(txtTablica5->Text);
       txtSuma->Text=
       Convert::ToString(suma_wyrazow(tablica, 6));
 }
//---------------------------------------------------
```

# Przykłady

W rozdziale zostana pokazane przykłady konstrukcji, które ilustrują w jaki sposób można wykorzystać zmienne wskaźnikowe w operacjach wykonywanych na tablicach.

W przykładzie 3.5 przedstawiono funkcję obliczającą sumę wyrazów tablicy jednowymiarowej. Przekazywanie parametrów odbywa się przez referencję. Sumowanie jest realizowane za pomocą instrukcji for( ), która w tym przypadku nie steruje blokiem a jedynie jedną instrukcją. Na zakończenie odbywa się podstawienie ostatecznego wyniku. Zmienne suma i i mają charakter lokalny. Są używane jedynie do wewnętrznych operacji w funkcji.

### Przykład 3.5 wykonany jako Console Application

```
//---------------------------------------------------
#include <iostream.h>
#include <math.h>

void oblicz_suma(float wektor[], int n, float *wynik)
{
     float suma;
     int i;
     suma=0;
     for (i=0; i<n ; i++)
           suma+=wektor[i];
     *wynik=suma;
}
//---------------------------------------------------
```

Odwołujac się do poszczególnych elementów tablicy można tę operację realizować na kilka sposobów. Pierwszy to wykorzystanie do tego celu indeksów tablicy:

### Przykład 3.6

```
int tablica[4];
int i;

for ( i=0 ; i<4 ; i++ )
tablica[i] = i*i;
```

W powyższym przykładzie generowane są wartości poszczególnych elementów 4-elementowej tablicy. Sterowanie zmianami inedeksu tablicy odbywa się za pomocą zmiennej **i** i instrukcji **for**.

Odwoływanie się do poszczegolnych elementów tablicy w trakcie jej dalszego przetwarzania np. wydruku może się odbywać np. tradycyjnie:

```
for ( i=0;  i<4 ;  i++ )
printf("\n %d", tablica[i]);
```

Inny sposob to odwoływanie się do **tablica** – stałej wskaźnikowej (adresu początku tablicy) i operowanie odpowiednio indeksem **i**. Operator * powoduje odwołanie się do konkretnej zawartości danego elementu tablicy.

```
for ( i=0 ;  i<4 ; i++)
printf("\n %d", *(tablica +i) );
```

Innym sposobem jest budowa instrukcji cyklu w oparciu o zmienną wskaźnikową. Należy wówczas przygotować zmienną adresową **wskaznik** pozwalającą na przechowywanie adresów elementów tablicy. Zmienna **wskaznik** na początku przyjmuje wartość adresu początku tablicy **tablica**, następnie jest powiększana o 1 za pomocą operatora **++**. Tym samym przyjmuje ona wartości adresów kolejnych elementów tablicy. Sprawdzany jest warunek czy adres przechowywany w zmiennej **wskaznik** nie przekracza adresu końcowego elementu tablicy.

```
int *wskaznik;
for ( wskaznik = tablica ;  wskaznik < tablica + 4 ;
wskaznik++ )
printf("\n %d", *wskaznik);
```

Operacje na adresach tablicy mogą być również realizowane zaczynając od adresu końcowego elementu tablicy. Wówczas używanie operatora **--** pozwala na przechodzenie do adresu elementu poprzedzającego dany element.

W wielu przypadkach programując w języku C w chwili tworzenia programu nie znamy rozmiarów pisanego oprogramowania, nie wiemy

np. jak dużą tablicę przygotować do prowadzenia procesu przetwarzania. Wygodnym rozwiązaniem w tym przypadku jest wykorzystanie makrodefinicji **#define**, która pozwala zdefiniować nazwę symboliczną. Nazwa symboliczna może występować w wielu miejscach programu. Ciąg znaków przypisany danej nazwie symbolicznej będzie wstawiany przez kompilator wszędzie tam gdzie wystąpi dana nazwa symboliczna. Z daną nazwą symboliczną można związać nie tylko zmienne określające rozmiary obiektów można także powiązać właściwe procesy przetwarzania. Poniższy przykład ilustruje to zagadnienie.

**Przykład 3.7 wykonany jako Console Application**

```
#include <stdio.h>
#define  ROZMIAR 5

main()
{
int i;
int tablica[ROZMIAR];

printf ("podaj  %d danych liczbowych \n", ROZMIAR);
for ( i =0 ; i<ROZMIAR ; i++ )
scanf ("%d", &tablica[i] );
}
```

W przypadkach gdy tablice nie są zbyt duże i ich zawartość nie ulega w zasadzie zmianie w procesie przetwarzania stosuje się bardzo praktyczne rozwiązanie polegające na nadaniu wartości elementom tablicy w momencie ich deklarowania.

```
main()
{
int i;
float tablica[6] = {1, 2.1, 3.2, 7.1, 3.4, 7.8};
…
}
```

# Podsumowanie i uzupełnienia

W rozdziale omówiono obiekty typu tablice, zmienne wskaźnikowe, związane z nimi operacje oraz możliwość przekazywania wartości liczbowych przez referencję.

### Odwołania do nieistniejących elementów tablicy

Zaznaczono, że tablica jest zbiorem przechowywanych obiektów tego samego typu. Tablica posiada określoną, zadeklarowaną ilość elementów. Oznacza to przygotowanie obszaru pamięci o określonych, ustalonych rozmiarach. Odwołanie się do elementu spoza tego obszaru, w trakcie pracy programu, spowoduje nieprzewidywalne uszkodzenia kodu, które mogą zaowocować zakończeniem działania programu lub jego błędnym funkcjonowaniem. Dlatego ważne jest zabezpieczenie się przed tego typu ewentualnością i bieżące kontrolowanie odwołań do elementów tablicy.

### Deklarowanie elementów tablicy z przypisywaniem wartości

W wielu przypadkach deklarując tablice dokonuje się ich inicjalizacji (pokazano to w rozdziale dla przypadku tablicy jednowymiarowej). Tablice wielowymiarowe mogą być także inicjalizowane w trakcie ich deklarowania. Przypisywanie wartości odbywa się z listy zaczynając od indeksu ostatniego, a kończąc na indeksie pierwszym.

### Dynamiczna rezerwacja pamięci

Wskaźniki, przedstawione w rozdziale, poza zaprezentowanymi zastosowaniami pozwalają budować struktury, przeznaczone do dynamicznego rezerwowania pamięci w trakcie pracy programu. Polega to na tym, że za pomocą specjalnych funkcji można dynamicznie zarezerwować pewien obszar pamięci, dla określonego typu obiektu, uzyskując jego adres. Posługując się tym adresem można wprowadzać w zarezerwowany obszar określone dane, można znając adres również je pobierać. Jeżeli dynamicznie zarezerwowany obszar pamięci nie będzie już potrzebny można go również, za pomocą innej specjalnej funkcji, zwolnić. Mechanizm ten zostanie przedstawiony w dalszej części opracowania w wersji standardowej dla języka C++.

## Zadania do samodzielnego wykonania

1. Zbuduj program zawierający dwie tablice, jedną do przechowywania nazw modeli samochodów, drugą do przechowywania ich cen. Dane tego samego samochodu mają ten sam indeks w każdej z tablic. Zbuduj też trzecią tablicę jako tablicę wskaźników do tablicy cen. Jako pierwszy wyraz tej tablicy niech będzie zapisany adres ceny najtańszego samochodu (w tablicy cen),

jako drugi odpowiednio drugi, itd. Całość wyposaż w funkcje czytania danych, przetwarzania i prezentacji wyników.

2. Zbuduj odpowiednio program zawierający 4 tablice, jedną służącą do przechowywania nazw modeli samochodów, drugą do przechowywania ich cen, trzecią do przechowywania ich mocy, czwartą do przechowywania ich zużycia paliwa. Dane tego samego samochodu mają ten sam indeks w każdej z tablic. Dodatkowo zbuduj trzy tablice wskaźników: do adresów samochodów od najtańszego do najdroższego, do adresów nazw samochodów od najmniejszej do największej mocy, do adresów nazw samochodów od najmniejszego do największego zużycia paliwa. Dodaj funkcje czytania danych, przetwarzania i prezentacji wyników.

# 4 Struktury i unie

# Struktury

W języku C istnieje możliwość tworzenia własnych typów obiektów przez użytkownika. Obiekty te stanowią zagregowane całości zbudowane z innych elementów (zmiennych całkowitych, rzeczywistych, zmiennych tekstowych itp). W przeciwieństwie do tablic struktury pozwalają na agregację różnych typów danych.

Poniżej przedstawiono przykładową definicję struktury **samochod**:

### Przykład 4.1

```
struct  samochod {
     char marka [50];
     int  liczba_drzwi;
    float cena;
    };
```

Powyższy opis jest tylko definicją struktury, nie powoduje on powołania żadnej konkretnej struktury typu **samochod**.

W definicji struktury **samochod** określono, że każda struktura tego typu składa się z trzech składników: tablicy znakowej **marka[50]**, zmiennej typu integer **liczba_drzwi**, zmiennej typu rzeczywistego **cena**. Powyższy zapis jest opisem nowej struktury danych, receptą na jej tworzenie.

Mając zdefiniowaną określoną strukturę można definiować zmienne typu tej struktury. Można np. wykorzystując definicję struktury **samochod** definiować zmienne **samochod1**, **samochod2**, czy też tablicę struktur **samochody[100]**:

```
struct samochod samochod1, samochod2, samochody[100];
```

W powyższym zapisie fragment **struct samochod** to definicja nowego typu, **samochod1**, **samochod2**, **samochody[100]** to określone obiekty zdefiniowanego typu.

W języku C istnieje możliwość zastąpienia fragmentu **struct samochod** jedną nazwą. Wykonywane to jest za pomocą **typedef**. Pokazane zostanie to na przykładzie.

**Przykład 4.2**

```
typedef struct samochod {
            char marka [50];
            int liczba_drzwi;
            float cena ;
    }  SAMOCHOD;
```

Wówczas deklaracja wygląda następująco:

```
SAMOCHOD auto1, auto2, auta[100];
```

Kolejnym ważnym elementem jest odwoływanie się do poszczególnych składników struktury. Operacja ta może być wykonywana na kilka sposobów. Pierwszy z nich wygląda następująco:

```
auto1.cena = 34 000.100 ;
```

Możliwa jest również inicjalizacja dla konkretnej struktury samochod:

```
static struct auto1 = {”ford”, 4, 40 000.500};
```

Często definicje struktur umieszczane są w plikach „include”. Wówczas w pozostałej części programu definicje są dostępne ale nie są uwidaczniane.

Bardzo wygodną operacją możliwą do realizacji na strukturach jest możliwość zbiorowego podstawienia wszystkich składników jednej struktury wszystkim składnikom innej struktury tego samego typu np.:

```
struct van  mercedes1, mercedes2;
mercedes2 = mercedes1;
```

## Wskaźniki i struktury

Podobnie jak dla wszystkich innych typów zmiennych również w przypadku struktur istnieje możliwość deklarowania zmiennych wskaźnikowych np.:

**Przykład 4.3**

```
struct  samochod {
     char marka [50];
     int  liczba_drzwi;
    float cena;
    } auto1, *wskaznik_auto;
```

**auto1** to struktura typu **samochod**, **wskaznik_auto** to zmienna wskaźnikowa przeznaczona do przechowywania adresów struktur typu **samochod**.

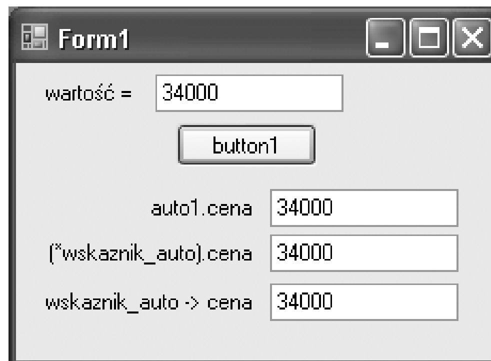
Zmienne wskaźnikowe pozwalają na inne sposoby odwoływania się do składników struktur:

**Przykład 4.4 wykonany jako Console Application**

```
struct  samochod {
     char marka [50];
     int  liczba_drzwi;
    float cena;
    } auto1, *wskaznik_auto;

wskaznik_auto = &auto1 ;

auto1.cena = 34 000.0;
(*wskaznik_auto).cena = 34000.0;
wskaznik_auto -> cena = 34000.0;
```

Operacja **wskaznik_auto = &auto1;** powoduje uzyskanie adresu struktury **auto1** i następnie podstawienie tego adresu do zmiennej wskaźnikowej **wskaznik_auto**. Dalej przedstawiono trzy różne sposoby odwoływania się do składników struktury, pierwszy oparty jest wyłącznie na nazwach, dwa następne wykorzystują zmienne wskaźnikowe zawierające adresy do struktur.

**Przykład 4.4a wykonany jako Windows Forms Application**




Rysunek 4.1. Widok okna aplikacji w trakcie działania

Kod aplikacji

Uwaga: Struktura musi być zdefiniowana **poza klasą Form1, na początku okna kodu**:

```
#pragma once

namespace Przyklad_44a {

    using namespace System;
    using namespace System::ComponentModel;
    using namespace System::Collections;
    using namespace System::Windows::Forms;
    using namespace System::Data;
    using namespace System::Drawing;

// Definiowanie własnej struktury:
//-----------------------------------
    struct  samochod {
           char marka [50];
           int  liczba_drzwi;
           double cena;
           } auto1, *wskaznik_auto;
//-----------------------------------

    ...
    ...
    ...
```

Dalszy kod pisany jest w procedurze obsługi zdarzenia – kliknięcie na przycisku. Kod ten wykorzystuje zdefiniowaną strukturę:

```
private: System::Void button1_Click(System::Object^
          sender, System::EventArgs^  e) {
     double wartosc;
     wartosc=Convert::ToDouble(txtCena->Text);
     wskaznik_auto = &auto1 ;
     auto1.cena = wartosc;
 //Odwołanie do elementu struktury
 //przy wydruku do textBoxów, na kilka sposobów

     textBox1->Text = Convert::ToString(auto1.cena);
     textBox2->Text =
          Convert::ToString((*wskaznik_auto).cena);
     textBox3->Text =
          Convert::ToString((wskaznik_auto -> cena));
     }
```

## Struktura w strukturze

Do budowy struktur poza składnikami takimi jak zmienne typów **int**, **flow**, **char** mogą być wykorzystywane również inne struktury. Warunkiem koniecznym jest to, aby struktura wykorzystywana do budowy innej struktury była zdefiniowana wcześniej. Struktura nie może zawierać składnika, który jest strukturą tego samego typu.

Jedyną możliwością wykorzystania do budowy nowej struktury jej samej jest odwoływanie się we wnętrzu struktury do zmiennych wskaźnikowych do tej struktury. Przykład takiego rozwiązania przedstawiono poniżej.

**Przykład 4.5**

```
struct drzewo_binarne{
  int zawartosc;
  struct drzewo_binarne  *lewa-strona;
  struci drzewo_binarne  *prawa-strona;
```

## Przekazywanie struktur do funkcji

Przekazywanie struktury do funkcji może odbywać się przez wartość lub przez referencję. Pokazuje to poniższy przykład.

**Przykład 4.6**

```
void funkcja1 ();
void funkcja2 ();
struct auto{
    char firma[10];
    char model[10];
    float cena;
    } auto1, *wskaznik_auto;

wskaznik_auto = &auto1;

funkcja1 (auto1);
funkcja2 (wskaznik_auto);
```

## Unie

Unie **union** są w swej istocie strukturami, których składniki zajmują ten sam wspólny obszar pamięci; w każdej chwili jeden składnik unii może dysponować całą pamięcią. Zapis oraz operacje są identyczne jak w przypadku struktur, wystarczy zamienić słowa kluczowe.

Ze względu na rosnące możliwości dzisiejszych komputerów są one coraz rzadziej używane.

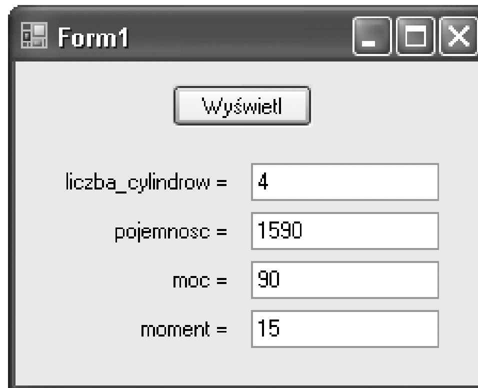
## Przykłady wykorzystania struktur

Struktury odgrywają bardzo ważną rolę w budowie programów pisanych w języku C. Pozwalają na bardzo indywidualne modelowanie oprogramowywanych problemów. Struktury definiowane na potrzeby danego problemu nadają całemu oprogramowaniu bardzo indywidualne, dedykowane cechy. Tym samym pozwalają na dostosowanie oprogramowania do postaci rozwiązywanego problemu. W przykładzie 4.7 pokazano sposób definiowania struktury **silnik**, powołania do życia struktury **s_760** oraz inicjację jej składników. Struktura **s_760** jest strukturą statyczną – istnieje przez cały czas przetwarzania programu.

**Przykład 4.7 wykonany jako Console Application**

```
#include <stdio.h>
void main()
{
static struct silnik {
    int liczba_cylindrow;
    int pojemnosc;
    int moc;
    int moment;} s_760 = {4, 1590, 90, 15};
printf(" \n liczba cylindrow %d",
s_760.liczba_cylindrow);
printf(" \n pojemnosc %d", s_760.pojemnosc);
printf(" \n moc %d", s_760.moc);
printf(" \n moment %d", s_760.moment);
      }
```

**Przykład 4.7a wykonany jako Windows Forms Application**




Rysunek 4.2. Widok okna aplikacji w trakcie działania

Kod aplikacji

Uwaga: Struktura musi być zdefiniowana poza klasą Form1, na początku okna kodu:

```
#pragma once

namespace Przyklad_47a {

    using namespace System;
    using namespace System::ComponentModel;
    using namespace System::Collections;
    using namespace System::Windows::Forms;
    using namespace System::Data;
    using namespace System::Drawing;
//Definicja własnej struktury
//-------------------------------------------
static struct silnik {
    int liczba_cylindrow;
    int pojemnosc;
    int moc;
    int moment;} s_760 = {4, 1590, 90, 15};
//-------------------------------------------
```

Dalszy kod piszemy w procedurze obsługi zdarzenia – kliknięcie na przycisku. Kod ten wykorzystuje zdefiniowaną strukturę:

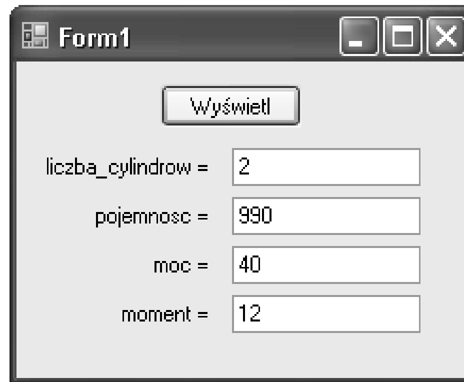
```
private: System::Void
btnWyswietl_Click(System::Object^  sender,
System::EventArgs^  e) {
    textBox1->Text =
           Convert::ToString(s_760.liczba_cylindrow);
    textBox2->Text =
           Convert::ToString(s_760.pojemnosc);
   textBox3->Text = Convert::ToString(s_760.moc);
   textBox4->Text = Convert::ToString(s_760.moment);
 }
```

W przykładzie 4.8 pokazano w jaki sposób można deklarować zmienne wskaźnikowe do struktury określonego typu oraz w jaki sposób te zmienne można wykorzystać.

**Przykład 4.8 wykonany jako Console Application**

```
#include <stdio.h>
void main()
{
struct silnik {
    int liczba_cylindrow;
    int pojemnosc;
    int moc;
    int moment;};
struct silnik silnik1, *wsk_silnik;

wsk_silnik = &silnik1;

wsk_silnik -> liczba_cylindrow = 2;
wsk_silnik -> pojemnosc = 990;
wsk_silnik -> moc = 40;
wsk_silnik -> moment = 12;

printf(" \n liczba cylindrow %d", wsk_silnik ->
liczba_cylindrow);
printf(" \n pojemnosc %d", wsk_silnik -> pojemnosc);
printf(" \n moc %d", wsk_silnik -> moc);
printf(" \n moment %d", wsk_silnik -> moment);
      }
```

**Przykład 4.8a wykonany jako Windows Forms Application**




Rysunek 4.3. Widok okna aplikacji w trakcie działania

Kod aplikacji

Uwaga: Struktura musi być zdefiniowana poza klasą Form1, na początku okna kodu:

```
#pragma once

namespace Przyklad_48a {
    using namespace System;
    using namespace System::ComponentModel;
    using namespace System::Collections;
    using namespace System::Windows::Forms;
    using namespace System::Data;
    using namespace System::Drawing;
//Definicja własnej struktury
//----------------------------------------
struct silnik {
    int liczba_cylindrow;
    int pojemnosc;
    int moc;
    int moment;
};
//----------------------------------------
```

Dalszy kod piszemy w procedurze obsługi zdarzenia – kliknięcie na przycisku. Kod ten wykorzystuje zdefiniowaną strukturę:

```
private: System::Void btnWyswietl_Click(
   System::Object^  sender, System::EventArgs^  e) {

  struct silnik silnik1, *wsk_silnik;

  wsk_silnik = &silnik1;

  wsk_silnik -> liczba_cylindrow = 2;
  wsk_silnik -> pojemnosc = 990;
  wsk_silnik -> moc = 40;
  wsk_silnik -> moment = 12;

  textBox1->Text =
     Convert::ToString(wsk_silnik->liczba_cylindrow);
  textBox2->Text =
     Convert::ToString(wsk_silnik->pojemnosc);
  textBox3->Text =
     Convert::ToString(wsk_silnik->moc);
  textBox4->Text =
     Convert::ToString(wsk_silnik->moment);
 }
```

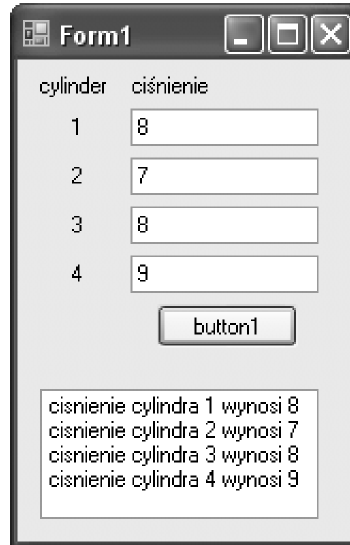
Jeżeli w definicji struktury występują takie elementy jak tablice to wówczas zarówno operacje na strukturach jak i operacje na tablicach wchodzących w ich skład mogą odbywać się przy wykorzystaniu zmiennych wskaźnikowych. Przykład 4.9 ilustruje to zagadnienie.

**Przykład 4.9 wykonany jako Console Application**

```
#include <stdio.h>
void main()
{
struct silnik {
    int liczba_cylindrow;
    int pojemnosc;
    int moc;
    int moment;
    int cisnienie[4];} silnik1;

int licznik;

for (licznik = 0 ; licznik <4 ; licznik++)
   {
   printf ("\n podaj cisnienie cylindra  %d",
                                        licznik+1);
   scanf("%d", &silnik1.cisnienie[licznik]);
   }
```

```
for (licznik = 0 ; licznik <4 ; licznik++)
   {
   printf ("\n  cisnienie cylindra  %d   wynosi %d",
              licznik+1, silnik1.cisnienie[licznik]);
   }

}
```

**Przykład 4.9a wykonany jako Windows Forms Application**




Rysunek 4.4. Widok okna aplikacji w trakcie działania

Kod aplikacji

Uwaga: Struktura musi być zdefiniowana poza klasą Form1, na początku okna kodu:

```
#pragma once

namespace Przyklad_49a {

    using namespace System;
    using namespace System::ComponentModel;
    using namespace System::Collections;
    using namespace System::Windows::Forms;
    using namespace System::Data;
    using namespace System::Drawing;
```

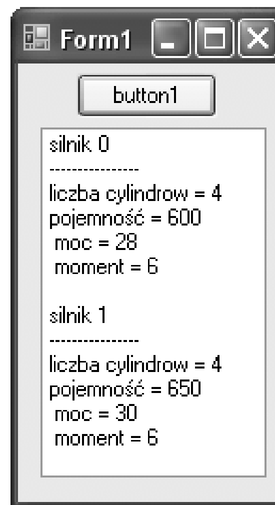
```
//Definicja własnej struktury
//-----------------------------------------
struct silnik {
    int liczba_cylindrow;
    int pojemnosc;
    int moc;
    int moment;
    int cisnienie[3];} silnik1;
    //Elementy tablicy numerowane są od 0,
    //czyli dla silnika 4 cylindrowego: 0, 1, 2, 3.
//-----------------------------------------
. . .
. . .
private: System::Void button1_Click(System::Object^
         sender, System::EventArgs^  e) {

     silnik1.cisnienie[0] =
                  Convert::ToInt16(textBox1->Text);
     silnik1.cisnienie[1] =
                  Convert::ToInt16(textBox2->Text);
     silnik1.cisnienie[2] =
                  Convert::ToInt16(textBox3->Text);
     silnik1.cisnienie[3] =
                  Convert::ToInt16(textBox4->Text);

     for (int licznik = 0 ; licznik <4 ; licznik++)
     {
       listBox1->Items->Add(
          "cisnienie cylindra "
          + Convert::ToString(licznik + 1) +
           " wynosi " + Convert::ToString(
             silnik1.cisnienie[licznik]));
     }
 }
```

Budując struktury danych oparte na elementach takich jak struktury można utworzyć tablice, których poszczególnymi elementami składowymi są struktury wcześniej zdefiniowanego typu. W przykładzie 4.10 przedstawiono ten problem na przykładzie tablicy silników.

### Przykład 4.10 wykonany jako Console Application

```
#include <stdio.h>
void main()
{
int licznik;
 struct silnik {
    int liczba_cylindrow;
    int pojemnosc;
    int moc;
    int moment;} silniki[2];

licznik = 0;
silniki[licznik].liczba_cylindrow = 4;
silniki[licznik].pojemnosc = 600;
silniki[licznik].moc = 30;
silniki[licznik].moment = 6;

printf(" \n liczba cylindrow %d",
silniki[licznik].liczba_cylindrow);
printf(" \n pojemnosc %d",
silniki[licznik].pojemnosc);
printf(" \n moc %d", silniki[licznik].moc);
printf(" \n moment %d", silniki[licznik].moment);

licznik = 1;
silniki[licznik].liczba_cylindrow = 4;
silniki[licznik].pojemnosc = 600;
silniki[licznik].moc = 30;
silniki[licznik].moment = 6;

printf(" \n liczba cylindrow %d",
silniki[licznik].liczba_cylindrow);
printf(" \n pojemnosc %d",
silniki[licznik].pojemnosc);
printf(" \n moc %d", silniki[licznik].moc);
printf(" \n moment %d", silniki[licznik].moment);
}
```

### Przykład 4.10a wykonany jako Windows Forms Application




Rysunek 4.5. Widok okna aplikacji w trakcie działania

Kod aplikacji

Uwaga: Struktura musi być zdefiniowana poza klasą Form1, na początku okna kodu:

```
#pragma once


namespace Przyklad_410a {

    using namespace System;
    using namespace System::ComponentModel;
    using namespace System::Collections;
    using namespace System::Windows::Forms;
    using namespace System::Data;
    using namespace System::Drawing;
//Definicja własnej struktury
//---------------------------------------
struct silnik {
    int liczba_cylindrow;
    int pojemnosc;
    int moc;
    int moment;} silniki[2];
//---------------------------------------
. . .
. . .
```

```
private: System::Void button1_Click(System::Object^
         sender, System::EventArgs^  e) {
     int licznik;
     licznik = 0;
     silniki[licznik].liczba_cylindrow = 4;
     silniki[licznik].pojemnosc = 600;
     silniki[licznik].moc = 28;
     silniki[licznik].moment = 6;

     licznik = 1;
     silniki[licznik].liczba_cylindrow = 4;
     silniki[licznik].pojemnosc = 650;
     silniki[licznik].moc = 30;
     silniki[licznik].moment = 6;

     for (licznik = 0; licznik < 2; licznik++)
     {
       listBox1->Items->Add("silnik " +
               Convert::ToString(licznik));
       listBox1->Items->Add("----------------");
       listBox1->Items->Add("liczba cylindrow = " +
               Convert::ToString(
               silniki[licznik].liczba_cylindrow));
       listBox1->Items->Add("pojemność = " +
               Convert::ToString(
               silniki[licznik].pojemnosc));
       listBox1->Items->Add(" moc = " +
               Convert::ToString(
               silniki[licznik].moc));
       listBox1->Items->Add(" moment = " +
               Convert::ToString(
               silniki[licznik].moment));
       listBox1->Items->Add("");
     }
 }
```

Bardzo często do budowy struktury jednego typu wykorzystywane są struktury wcześniej zdefiniowane. W przykładzie 4.11 struktura **data** (przeznaczona do zapisu takich danych jak poszczególne składniki daty) została wykorzystana w strukturze **silnik**.

**Przykład 4.11 wykonany jako Console Application**

```
#include <stdio.h>
void main()
{
struct data {
    int dzien;
    int miesiac;
    int rok;};
 struct silnik {
    int liczba_cylindrow;
    int pojemnosc;
    int moc;
    int moment;
    struct data data_badania_1;} silnik1;


silnik1.liczba_cylindrow = 4;
silnik1.pojemnosc = 600;
silnik1.moc = 30;
silnik1.moment = 6;

silnik1.data_badania_1.dzien = 1;
silnik1.data_badania_1.miesiac = 2;
silnik1.data_badania_1.rok = 2010;

printf(" \n liczba cylindrow %d",
                          silnik1.liczba_cylindrow);
printf(" \n pojemnosc %d", silnik1.pojemnosc);
printf(" \n moc %d", silnik1.moc);
printf(" \n moment %d", silnik1.moment);


printf(" \n data badania_1");
printf(" \n dzien %d", silnik1.data_badania_1.dzien);
printf(" \n miesiac %d",
silnik1.data_badania_1.miesiac);
printf(" \n rok %d", silnik1.data_badania_1.rok);
 }
```

**Przykład 4.11a wykonany jako Windows Forms Application**

Rysunek 4.6. Widok okna aplikacji w trakcie działania

Kod aplikacji

Uwaga: Struktury muszą być zdefiniowane poza klasą Form1, na początku okna kodu:

```
#pragma once

namespace Przyklad_411 {
    using namespace System;
    using namespace System::ComponentModel;
    using namespace System::Collections;
    using namespace System::Windows::Forms;
    using namespace System::Data;
    using namespace System::Drawing;

// Definicja wlasnych struktur
//-----------------------------------------
struct data {
    int dzien;
    int miesiac;
    int rok;};

struct silnik {
    int liczba_cylindrow;
    int pojemnosc;
    int moc;
    int moment;
    struct data data_badania_1;} silnik1;
//-----------------------------------------
```

```
. . .
. . .
//------------------------------------------------
private: System::Void button1_Click(System::Object^
         sender, System::EventArgs^  e) {

     silnik1.liczba_cylindrow = 4;
     silnik1.pojemnosc = 600;
     silnik1.moc = 30;
     silnik1.moment = 6;

     silnik1.data_badania_1.dzien = 1;
     silnik1.data_badania_1.miesiac = 2;
     silnik1.data_badania_1.rok = 2010;

     listBox1->Items->Add(
           "liczba cylindrów = " +
           Convert::ToString(
                 silnik1.liczba_cylindrow));
     listBox1->Items->Add(
           "pojemność = " +
           Convert::ToString(
                 silnik1.pojemnosc));
     listBox1->Items->Add(
           "moc = " +
           Convert::ToString(silnik1.moc));
     listBox1->Items->Add(
           "moment = " +
           Convert::ToString(silnik1.moment));

     listBox1->Items->Add("Data badania 1");
     listBox1->Items->Add("dzień: " +
           Convert::ToString(
                 silnik1.data_badania_1.dzien));
     listBox1->Items->Add("miesiac: " +
           Convert::ToString(
                 silnik1.data_badania_1.miesiac));
     listBox1->Items->Add("rok: " +
           Convert::ToString(
                 silnik1.data_badania_1.rok));
     }
//------------------------------------------------
```

# Podsumowanie i uzupełnienia

Przedstawione w rozdziale zagadnienia dotyczą struktur jako obiektów definiowanych przez programującego w języku C. Programujący ma możliwość zaprojektowania własnej postaci struktur używanych w programie. Ich postać wynika przede wszystkim z modelowanego problemu.

Zaprojektowane przez programującego struktury mogą być powiązane z innymi zmiennymi różnych typów oraz istniejącymi już strukturami. Możliwe jest również wykorzystanie zmiennych wskaźnikowych przeznaczonych dla danego typu struktur.

Struktury w języku C stanowią formę wstępną dla struktur obiektowych oferowanych przez język C++.

# Zadania do samodzielnego wykonania

1. Zbuduj strukturę zawierającą podstawowe dane samochodu. Wykorzystaj ją w programie, zbuduj funkcje do czytania i prezentacji danych.

2. Zbuduj strukturę zawierającą podstawowe dane samochodu, która wykorzystuje inną strukturę zawierającą podstawowe dane silnika. Wykorzystaj obie struktury w programie, zbuduj funkcje czytania i prezentacji danych.

3. Dla struktur z zadania 2) zbuduj program gdzie będzie wykorzystana tablica struktur zawierających podstawowe dane samochodu. Zbuduj funkcję czytającą dane, prezentującą dane oraz funkcję ustalającą kolejność samochodów od najtańszego do najdroższego – wyniki uporządkowane zapisz w specjalnej tablicy wskaźników do struktur.

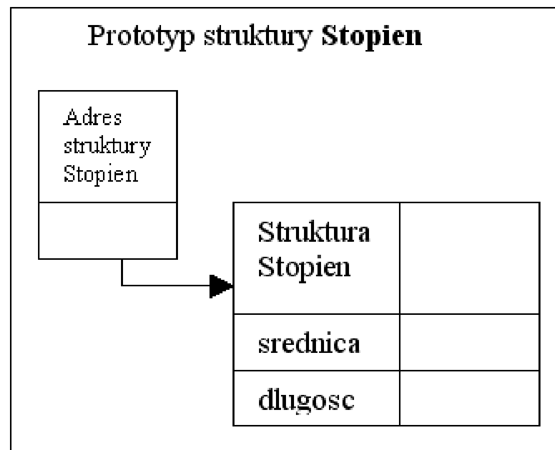
# 5 Struktury danych, listy

W rozdziale przedstawione zaostaną zagadnienia związane ze strukturami danych stosowanymi w programach pisanych w języku C. Przedstawimy zagadnienia ogólne oraz jedno przykładowe rozwiązanie dotyczące struktury danych typu lista.

We wszystkich programach o złożonym procesie przetwarzania konieczne jest opracowanie i zastosowanie odpowiednich struktur danych. Ich zadaniem jest przechowywanie danych występujących w programie oraz zapewnienie efektywnych narzędzi do ich zapisu i odczytu.

Zagadnienie to zostanie omówione na przykładzie. Przykład dotyczy modelu wału maszynowego. Założono, że model wału maszynowego oparty jest na modelu struktury stopnia wału maszynowego.

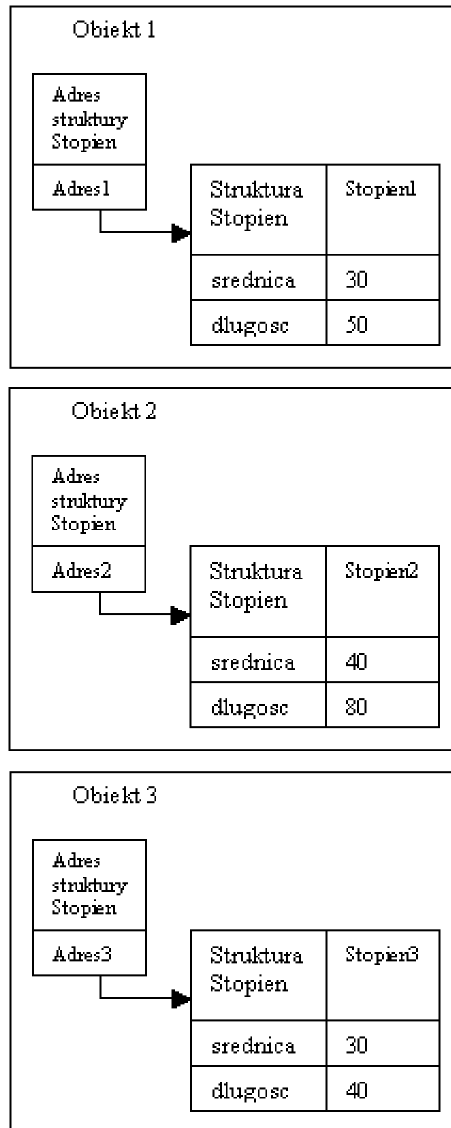
Natomiast sam wał jest modelowany jako lista struktur modelujących poszczególne stopnie wału. Na początek określona zostanie koncepcja opisu stopnia wału, niech będzie to struktura, która zawiera dwa składniki typu **float** pozwalające na zapis średnicy i długości stopnia wału. Zatem określono strukturę **Stopien**, która posiada dwa składniki **srednica** i **dlugosc**. Schematycznie przedstawiono to na rysunku 5.1. Jednocześnie na tym samym rysunku zaznaczono fakt, że dla struktury typu **Stopien** można utworzyć zmienną przeznaczona do przechowywania jej adresu.




Rysunek 5.1. Prototyp struktury **Stopien**

W kolejnym kroku w oparciu o definicję struktury Stopien utworzono kilka obiektów tych struktur: **Stopien1**, **Stopien2**, i **Stopien3**. Przy czym

każdy z tych obiektów ma już wprowadzone dane. Całość przedstawiono na rysunku 5.2.

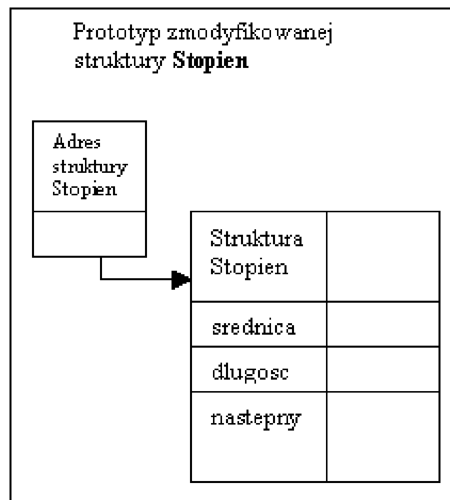



Rysunek 5.2. Utworzone struktury typu **Stopien**

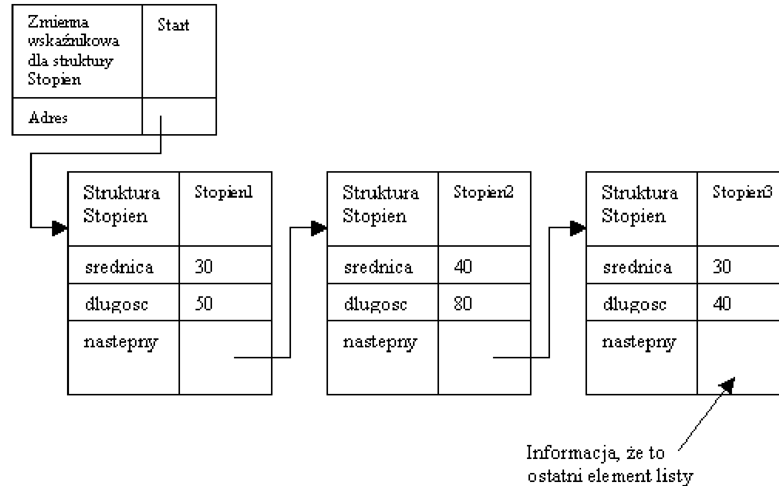
Lista jest strukturą danych, która jest zbudowana w ten sposób, że każdy obiekt należący do listy zna adres następnego po niej obiektu. Przy czym

w strukturze danych opisującej kolejne obiekty składowe listy występuje składnik przechowujący adres następnego obiektu, jest to zmienna wskaźnikowa **nastepny** pozwalająca na zapis adresu do obiektu typu ta sama struktura, która pozwala na opis obiektu. Schematycznie przedstawiono to na rysunku 5.3.

Przykładową strukturę **lista** przedstawiono na następnym rysunku 5.4. Zmienna **Start** jest zmienną wskaźnikową, która ma za zadanie przechowywać adres pierwszego elementu składowego listy. Z kolei pierwszy element listy posiada pole, gdzie przechowywany jest adres następnego elementu składowego listy. Kolejne adresy składowane są w kolejnych składnikach listy. Informacja o tym, że dany składnik jest ostatnim elementem listy najczęściej jest wprowadzana jako odpowiednia stała w polu adresowym ostatniego składnika listy.



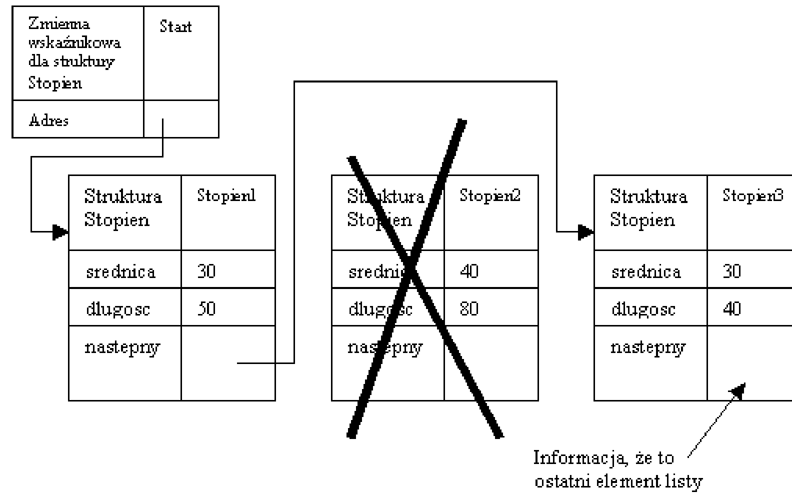

Rysunek 5.3. Zmodyfikowana definicja struktury stopień.



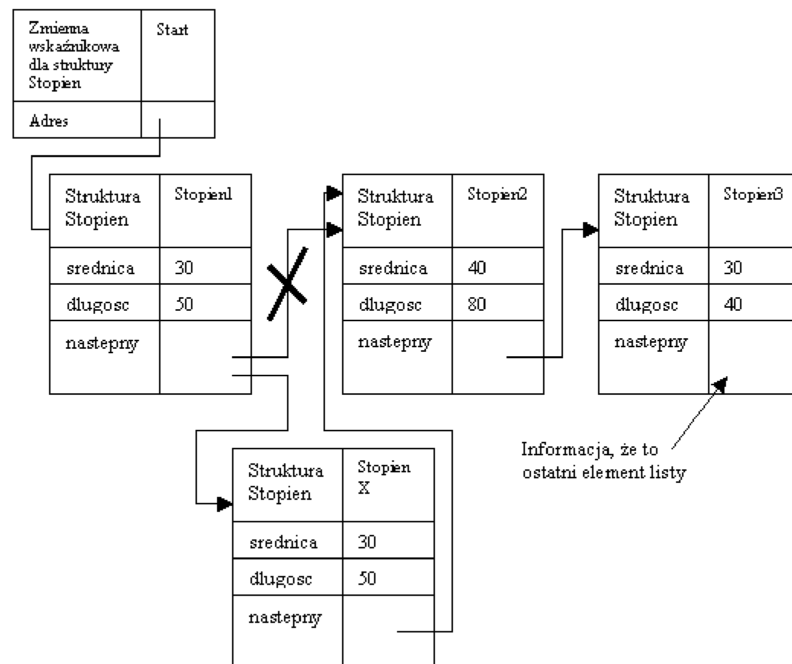

Rysunek 5.4. Struktura listowa złożona ze struktur **Stopien**

Struktury listowe mają to do siebie, że pozwalają na bardzo efektywne wprowadzanie zmian w strukturze listy. Np. usunięcie składnika listy może odbywać się poprzez zmiany w związkach adresowych. Pokazano to na rysunku 5.5. Jak widać w elemencie poprzedzającym element usuwany wstawiamy adres elementu następnego po usuniętym. Odbywa się wtedy praktycznie ominięcie elementu usuwanego w strukturze danych. Na rysunku 5.6 pokazano operację dodania dodatkowego elementu do listy. Również ta operacja polega na wprowadzeniu odpowiednich zmian w związkach adresowych. Po prostu do elementu poprzedzającego element nowo-wprowadzany wstawia się jego adres, a w elemencie nowo-wprowadzanym podaje się adres elementu następującego po nim.

Zaletą operacji wykonywanych na drodze zmieniania związków adresowych w strukturach danych jest unikanie konieczności przemieszczania tych struktur np. poprzez kopiowanie. Miejsca składowania danych pozostają te same, zmianom ulegają jedynie powiązania występujące w strukturach danych.




Rysunek 5.5. Usuwanie pojedynczego elementu listy.




Rysunek 5.6. Dodawanie pojedynczego elementu do listy

Poniżej przedstawiono przykład budowy struktury typu lista gdzie elementem składowym jest stopień wału maszynowego. Struktura nazywa

się **stopien_walka**. Jej składnikami są odpowiednio średnica i długość, chropowatość oraz adres kolejnego stopnia.

Przykład wyposażono w funkcje zapewniające możliwość dodania stopnia, usunięcia stopnia, wydruku danych na temat stopnia oraz wprowadzenia danych stopnia.

### Przykład 5.1 wykonany jako Console Application

```
// Wal_konsola.cpp : main project file.

#include "stdafx.h"

using namespace System;
//-----------------------------------------------------
// projektowanie walka
//-----------------------------------------------------
typedef struct stopien_walka
{
   double srednica;
   double dlugosc;
   double chropowatosc;
   struct stopien_walka *nastepny;
} STOPIEN;
//-----------------------------------------------------
void czytaj(STOPIEN *);
void wydruk(STOPIEN *);
void dodaj(int,STOPIEN *);
void usun(int,STOPIEN *);
//-----------------------------------------------------

int main(array<System::String ^> ^args)
{
   Console::WriteLine(
       L"Wpisywanie danych 5 stopni wału, wydruk, ");
   Console::WriteLine(
       L"a następnie usunięcie stopnia nr 2");
   Console::WriteLine(L"i ponownie wydruk\n.");

   STOPIEN st1;
   STOPIEN *adr;

   adr=&st1;
   czytaj(adr);
   adr->nastepny = 0;
```

```
// dodanie
   dodaj(1,adr);
   dodaj(2,adr);
   dodaj(3,adr);
   dodaj(4,adr);

   wydruk(adr);
   usun(2,adr);
   wydruk(adr);
   Console::ReadLine();
   return 0;
}
//---------------------------------------------------
void czytaj(STOPIEN *wsk)
{
    String^ tmp;
    Console::Write(L"Średnica: ");
    tmp = Console::ReadLine();
    wsk->srednica = Convert::ToDouble(tmp);
    Console::Write(L"dlugosc: ");
    tmp = Console::ReadLine();
    wsk->dlugosc = Convert::ToDouble(tmp);
    Console::Write(L"chropowatosc Ra: ");
    tmp = Console::ReadLine();
    wsk->chropowatosc = Convert::ToDouble(tmp);
}
//---------------------------------------------------
void wydruk(STOPIEN *wsk)
{
    int licznik=1;
    Console::Write("\n\nNr     D       L       Ra");
    while (wsk != 0)
    {
    Console::Write("\n" + licznik++ + "      " +
      wsk->srednica + "      " +
      wsk->dlugosc + "      " + wsk ->chropowatosc);
    wsk = wsk -> nastepny;
    }
    Console::Write("\n");
}
//----------------------------------------------------
```

```
void dodaj(int nr,STOPIEN *wsk)
{
    int licznik=1;
    STOPIEN *adr;

    while (wsk != 0)
    {
    if(licznik == nr)
          {
          adr=new STOPIEN;
          czytaj(adr);
          adr->nastepny = wsk->nastepny;
          wsk->nastepny = adr;
          wsk=adr;
          break;
          }
    licznik++;
     wsk = wsk->nastepny;
    }
}
//---------------------------------------------------
void usun(int nr,STOPIEN *wsk)
{
    int licznik = 1;
    STOPIEN *adres;
    if(nr != 1)
    {
      while (wsk != 0)
      {
          if(licznik == nr-1)
          {
                adres = wsk->nastepny;
                wsk->nastepny = adres->nastepny;
                adres = 0;
                break;
          }
           licznik++;
           wsk= wsk -> nastepny;
      }
    }
}
//---------------------------------------------------
```

## Podsumowanie i uzupełnienia

W rozdziale przedstawiono zasadnicze koncepcje związane z tworzeniem struktur typu lista. Pokazany przykład dotyczy list jednokierunkowych – list gdzie kolejne ich elementy zawierają adresy jedynie swoich sąsiadów – następców. Możliwe jest poszerzenie tej koncepcji o listy dwukierunkowe, gdzie każdy element listy posiada adresy zarówno swojego poprzednika jak i następcy.

Schemat postępowania w przypadku list jest łatwo przenaszalny na inne typowe struktury danych np.: stosy, kolejki, drzewa.

Zaprezentowany przykład nie akcentuje aspektu dynamicznego rezerwowania pamięci w przypadku dodawania nowych elementów do struktury danych typu lista. Jest to popularne, często stosowane rozwiązanie. Podobnie możliwe jest zwalnianie dynamicznie zarezerwowanego obszaru pamięci.

Struktury listowe oparte na strukturach tworzone w języku C mogą być łatwo rozszerzone na struktury listowe zbudowane z obiektów w języku C++.

## Zadania do samodzielnego wykonania

1. Zbuduj strukturę opisującą typ, nazwę elementu osadzonego na wale samochodowej skrzynki przekładniowej. Zbuduj strukturę listową przedstawiającą elementy umieszczone na wale w kolejności wynikającej z ich położenia geometrycznego.
2. Do zadania 1) dodaj tablicę wskaźników do struktur opisujących elementy osadzone na wale w kolejności ich montażu.
3. Do zadania 2) dobuduj możliwość modelowania wałów skrzynek 2 lub 3 wałkowych.

# 6 Struktury danych, stosy, kolejki

W rozdziale krótko zostaną scharakteryzowane inne struktury danych stosowane w języku C. Zdarza się, że w programach budowane są struktury typu stos. W ramach stosu składowanych jest n obiektów tego samego typu. Kolejne obiekty należące do stosu są powiązane ze sobą za pomocą związków adresowych podobnych do tych jakie były stosowane w przypadku struktur listowych. Zasadnicza różnica sprowadza się do tego, że pamiętany jest adres ostatniego wprowadzanego elementu. Jest to adres elementu będącego wierzchołkiem stosu.

Strukturę typu stos najłatwiej sobie wyobrazić w postaci stosu talerzy. Nowy talerz dokładany jest zawsze na górze. „Element pobierany" pobierany jest również zawsze z góry.

Pobieranie elementu należącego do stosu polega na pobraniu elementu ostatnio dokładanego i zwolnieniu jego obszaru pamięci, oraz zapamiętaniu adresu elementu bezpośrednio go poprzedzającego. Przy następnym pobraniu będzie pobierany właśnie ten następny element. Przy dokładaniu nowego elementu do stosu dokładany jest nowy element do elementu ostatnio dołożonego. Wówczas zapamiętywany jest adres ostatnio dołożonego elementu.

Kolejki są strukturami, które również składają się z listy elementów powiązanych adresowo. Przy czym lista ta ma początek i koniec. Nowe elementy dołączane są na końcu. Elementy pobierane są z początku.

Budując struktury danych typu lista, stos, kolejka bardzo trudno jest przewidzieć odpowiednie rozmiary potrzebnej pamięci. Dlatego też w języku C dostępna jest możliwość dynamicznego rezerwowania pamięci (w trakcie wykonywania programu) dla obiektu określonego typu. Dynamicznie zarezerwowany obszar posiada swój adres w pamięci. Znając adres można wykorzystywać ten obszar w używanych strukturach danych. Jeżeli dany obszar nie jest potrzebny można go zwolnić za pomocą odpowiedniej funkcji. Narzędzie to zaprezentowane zostanie w dalszej części opracowania w wersji dla języka C++.

## Zadania do samodzielnego wykonania

1. Wykonaj zadania z rozdziału 5) posługując się stosem.
2. Wykonaj zadania z rozdziału 5) posługując się kolejką.

# Klasy

Prezentacja problematyki programowania obiektowego zwykle rozpoczyna się od opisu pojęcia klasy i formalnego wyjaśnienia tego pojęcia.

**Klasa** jest **wzorcem obiektu**, który agreguje pewne prostsze struktury (mogą to być np. zmienne) w nowe bardziej złożone całości. Nowa złożona struktura może dotyczyć obiektów istniejących fizycznie np. opisywać strukturę określonego produktu czy też jego podzespołu. Może również nawiązywać do opisu bardziej abstrakcyjnych pojęć np. stanu obiektu, procesu, itp.

Posługiwanie się klasami i obiektami ma sens o ile używając tej formy opisu budujemy narzędzia wielokrotnego użytku tzn. ta forma opisu będzie wykorzystywana w opisie szeregu zbliżonych zadań czy też obiektów.

Klasa stanowi abstrakcję, która pozwoliła uchwycić pewną rzeczywistość. Autorzy pomysłu na klasę w jej opisie zawarli swoje, subiektywne widzenie tej rzeczywistości, tego co jest ważne i tego co zostało uznane za mniej ważne i zostało pominięte. Zwykle klasy budowane są z myślą o konkretnych zastosowaniach, pozwalają tworzyć, przechowywać, modyfikować opisy wybranych wycinków rzeczywistości.

Budując oprogramowanie, które jest przeznaczone do realizacji określonych zadań zwykle dostosowujemy koncepcje przyjętych klas do ogólnej koncepcji tego oprogramowania. Bierzemy wówczas pod uwagę szereg aspektów związanych z jego bieżącym wykorzystywaniem jak i z możliwym dalszym rozwojem.

Pojęcie klasy stanowi receptę na tworzenie obiektów tej klasy. **Obiekt** to **egzemplarz** danej klasy.

**Definicja klasy nie tworzy** i nie powinna tworzyć żadnego **obiektu**.

Obiekty są tworzone w oparciu o definicje danej klasy.

Programowanie obiektowe oferuje narzędzia pozwalające na wielokierunkowy, odpowiednio modelowany rozwój oprogramowania. Ma możliwość konstruowania różnych klas, klasy mogą być wykorzystywane do tworzenia nowych klas.

Klasy mogą być modyfikowane poprzez wykorzystanie mechanizmów **dziedziczenia**. Klasa pochodna wywodząca się z klasy bazowej przejmuje jej zasadnicze koncepcje (nie jest potrzebne ponowne ich rozwijanie) i dodaje pewne nowe składniki czy też warianty przetwarzania.

Istnieje cała gama narzędzi, które pozwalają rozwijać te całości w wielu kierunkach na różnym poziomie szczegółowości.

Pojęcie klasy jest w zasadzie pojęciem typu obiektu, gdzie jest on opisany w sposób formalny. Obiekt to jest konkretny egzemplarz struktury typu danej klasy. Poniżej przedstawiono przykład klasy opisującej konkretny obiekt techniczny. Tym obiektem jest połączenie gwintowe.

**Przykład 7.1.**

```
class polaczenie_gwintowe {
float srednica, dlugosc;
public:
  float srednica_rdzenia;
  float funkcja_obliczajaca_naprezenia(float );
};
float
polaczenie_gwintowe::funkcja_obliczajaca_naprezenia
(float sila_osiowa)
    {
      float sigma_r;
      sigma_r = 4* sila_osiowa/(3.14 *
                 srednica_rdzenia*srednica_rdzenia);
      return sigma_r;
    }
```

Opis klasy zawiera zarówno dane:

```
float srednica, dlugosc;
float srednica_rdzenia;
```

jak i funkcje:

```
float funkcja_obliczajaca_naprezenia(float );
```

Struktura odpowiadająca powyższej klasie mogłaby wyglądać następująco:

```
struct polaczenie_gwintowe-bis {
float srednica, dlugosc;
float srednica_rdzenia;
};
```

Utworzenie zmiennej typu tej struktury przyjęłoby z kolei następującą postać:

```
struct polaczenie_gwintowe-bis {
float srednica, dlugosc;
float srednica_rdzenia;
} struktura_gwint1;
```

Odwołanie do składnika struktury **struktura_nakretka1** wyglądałoby w następujący sposób:

```
struktura_gwint1.srednica = 25 ;
```

W przypadku klasy **polaczenie_gwintowe** powołanie do życia obiektu i odwołanie się do jednego z jego składników byłoby zapisane następująco:

```
class polaczenie_gwintowe {
  float srednica, dlugosc;
public:
  float srednica_rdzenia;
  float funkcja_obliczajaca_naprezenia(float );
} obiekt_gwint2;
```

- np.:

```
obiekt_gwint2.srednica_rdzenia = 10;
```

Klasy są generalnie pojęciem szerszym od struktur i pozwalają na wmontowanie w konstrukcję klasy odpowiednich, związanych z tą klasą funkcji. Z tego wynika zarówno forma zapisu tych funkcji jak i możliwość odwoływania się do nich. Poniżej pokazano w jaki sposób możemy uruchamiać funkcje w stosunku do obiektów danej klasy.

```
obiekt_gwint2. funkcja_obliczajaca_naprezenia(60);
```

### Przykład 7.1 wykonany jako Console Application

Poniżej pokazano cały przykład wykorzystania klasy **polaczenie_gwintowe**:

```
include "stdafx.h"
#include <iostream>
#include "polaczenie_gwintowe.h"

using namespace std;

int main(array<System::String ^> ^args)
{
polaczenie_gwintowe obiekt_gwint1;
float sila=600;

obiekt_gwint1.srednica_rdzenia=26;

cout << "średnica rdzenia :" <<
obiekt_gwint1.srednica_rdzenia << endl;
cout << "obciążenie śruby :" << sila << endl;
cout << "napreżenia w śrubie :"
<<obiekt_gwint1.funkcja_obliczajaca_naprezenia(sila)
              << endl;
getchar();
return 0;
}
```

### Przykład 7.1a wykonany jako Windows Forms Application

Utworzona klasa **polaczenie_gwintowe** zawiera dwie właściwości **Srednica_rdzenia** i **Dlugosc** oraz metodę **funkcja_obliczajaca_naprezenia**.

Po naciśnięciu przycisku btnNaprezenia tworzony jest obiekt **gwint1** klasy **polaczenie_gwintowe**

```
polaczenie_gwintowe^ obiekt_gwint1 = gcnew
                           polaczenie_gwintowe();
```

Następnie wartość siły odczytywana jest z okienka tekstowego

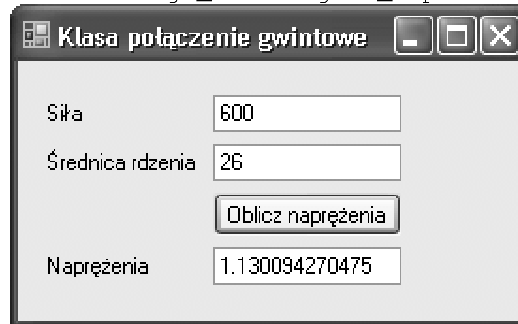
```
sila=Convert::ToDouble(txtSila->Text);
```

Natomiast odczytana wartość Średnicy rdzenia wprowadzana jest jako wartość właściwości **Srednica_rdzenia** obiektu

```
    obiekt_gwint1->Srednica_rdzenia =
       (Convert::ToInt32(txtSrednica_rdzenia->Text));
```

Na koniec wykorzystywana jest metoda funkcja_obliczajaca_naprezenia, której wynik umieszczany jest w okienku tekstowym.

```
txtNaprezenia->Text =
   Convert::ToString(
obiekt_gwint1->funkcja_obliczajaca_naprezenia(sila));
```




Rysunek 7.1. Widok okna aplikacji w trakcie działania

**Kod aplikacji**

```
ref class polaczenie_gwintowe {

private:
double srednica_rdzenia;
double dlugosc;

public:

property double Srednica_rdzenia {
    double get() {
          return srednica_rdzenia;
    }
    void set(double srednica_rdzenia) {
          this->srednica_rdzenia = srednica_rdzenia;
    }
}

property double Dlugosc {
    double get() {
          return dlugosc;
    }
    void set(double dlugosc) {
```

```
            this->dlugosc = dlugosc;
      }
  }

double funkcja_obliczajaca_naprezenia(double
                                sila_osiowa){
    double sigma_r;
    sigma_r=4*sila_osiowa/(Math::PI*srednica_rdzenia*
            srednica_rdzenia);
    return sigma_r;
  }

};

//-----------------------------------------------
#pragma endregion
//-----------------------------------------------
private: System::Void btnNaprezenia_Click(
   System::Object^  sender, System::EventArgs^  e) {

    polaczenie_gwintowe^ obiekt_gwint1 = gcnew
                            polaczenie_gwintowe();
    double sila;
    sila=Convert::ToDouble(txtSila->Text);
    obiekt_gwint1->Srednica_rdzenia =
      (Convert::ToInt32(txtSrednica_rdzenia->Text));
    txtNaprezenia->Text =
      Convert::ToString(
      obiekt_gwint1->funkcja_obliczajaca_naprezenia(
      sila));
  }
};
}
//-----------------------------------------------
```

Definiowanie funkcji przynależnych danej klasie może odbywać się na dwa sposoby:

a. wewnątrz definicji klasy,

b. na zewnątrz definicji klasy.

Poniżej pokazano to zagadnienie na przykładach.

- definiowanie funkcji wewnątrz definicji klasy (strcpy() to funkcja kopiująca string drugi w kolejności na string pierwszy w kolejności):

**Przykład 7.2**

```
class automobil {
   char nazwa_modelu[80];
   int rocznik;
public:

   void zapisz (char * nazwa, int rocznik_dana)
    {
      strcpy (nazwa_modelu, nazwa);
      rocznik = rocznik_dana;
    }

    void wypisz()
    {
     cout << nazwa_modelu  << "rocznik: "
     << rocznik << endl;
    }

};
```

- definiowanie na zewnątrz definicji klasy:

```
class automobil {
   char nazwa_modelu[80];
   int rocznik;
public:

   void zapisz (char * nazwa, int rocznik-dana);

   void wypisz();

};

void automobil::zapisz (char * nazwa,
                              int rocznik_dana)
    {
      strcpy (nazwa_modelu, nazwa);
      rocznik = rocznik_dana;
    }

void automobil::wypisz()
    {
     cout << nazwa_modelu  << "  rocznik: "
                           << rocznik << endl;
    }
```

Cały powyższy przykład w wersji konsolowej wygląda następująco:

```
#include "stdafx.h"
#include "iostream"
#include "resource.h"
using namespace std;
class automobil {
   char nazwa_modelu[80];
   int rocznik;
public:
   void zapisz (char * nazwa, int rocznik_dana);
   void wypisz();
};

void automobil::zapisz (char * nazwa, int
rocznik_dana)
    {
    strcpy (nazwa_modelu, nazwa);
    rocznik = rocznik_dana;
    }

void automobil::wypisz()
    {
  cout << "nazwa modelu : " << nazwa_modelu  << endl;
  cout << "rocznik : " << rocznik << endl;
    }

int main(array<System::String ^> ^args)
{
    automobil auto1, auto2, auto3;
    auto1.zapisz ("Matiz", 2007);
    auto1.wypisz();

    auto2.zapisz ("Lanos", 2003);
    auto2.wypisz();

    auto3.zapisz ("Punto", 2003);
    auto3.wypisz();

    getchar();

    return 0;
}
```

### Przykład 7.2a wykonany jako Windows Forms Application

Utworzona klasa *automobil* zawiera dwie właściwości *Nazwa_modelu* i *Rocznik* oraz metodę *zapisz*. Po naciśnięciu przycisku *btnZapisz* tworzony jest obiekt *auto1* klasy *automobil*

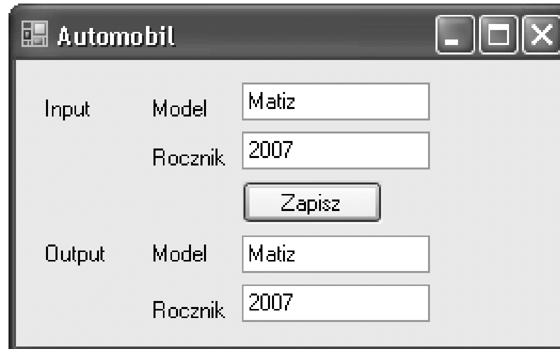
```
    automobil^ auto1 = gcnew automobil();
```

Wywoływana jest metoda *zapisz*, która pobiera dane z okienek tekstowych i ustawia wartości właściwości *Nazwa_modelu* i *Rocznik* obiektu *auto1*.

```
    auto1->zapisz (txtNazwaModelu_in->Text,
          Convert::ToInt32(txtRocznik_in->Text));
```

Następnie wartości właściwości przedstawiane są w dolnych okienkach tekstowych

```
txtNazwaModelu_out->Text = auto1->Nazwa_modelu;
txtRocznik_out->Text = Convert::ToString(
                                   auto1->Rocznik);
```




Rysunek 7.2. Widok okna aplikacji w trakcie działania

### Kod aplikacji

```
    ref class automobil {

private:
    String^ nazwa_modelu;
    int rocznik;
```

```
public:

    property String^ Nazwa_modelu {
          String^ get() {
                return nazwa_modelu;
          }
          void set(String^ nazwa_modelu) {
                this->nazwa_modelu = nazwa_modelu;
          }
    }

    property int Rocznik {
          int get() {
                return rocznik;
          }
          void set(int rocznik) {
                this->rocznik = rocznik;
          }
    }

    void zapisz(String^ nazwa, int rocznik_dana){
          nazwa_modelu = nazwa;
          rocznik = rocznik_dana;
    }

    };
//--------------------------------------------------

#pragma endregion

//--------------------------------------------------
private: System::Void btnZapisz_Click(System::Object^
          sender, System::EventArgs^  e) {

    automobil^ auto1 = gcnew automobil();
    auto1->zapisz (txtNazwaModelu_in->Text,
             Convert::ToInt32(txtRocznik_in->Text));
    txtNazwaModelu_out->Text = auto1->Nazwa_modelu;
    txtRocznik_out->Text =
                  Convert::ToString(auto1->Rocznik);
     }

};
}
//--------------------------------------------------
```

Budując klasy można umieszczać w ich składzie zarówno składniki w postaci danych (zmienne, inne obiekty, itp.) jak i funkcje. Dane

i funkcje są składnikami klasy. Mogą być one publiczne bądź prywatne. Określenie prywatna oznacza, że żadna inna część programu nie ma do nich bezpośredniego dostępu.

To wstępne przedstawienie przykładowej klasy miało na celu pokazanie zasadniczego celu tworzenia klas – budowy oprogramowania, które zapewnia możliwość jego dalszego, kontrolowanego rozwoju.

Inny przykład klasy przedstawiono poniżej:

**Przykład 7.3**

```
class samochod {
  public:
    int pojemnosc;
    int moc;
    char nazwa[80];
    jezdzi();
    hamuje();
};
samochod::jezdzi()
    {
     cout <<  " prędkość maksymalna: " << "200 km/h "
<< endl;
    }
samochod::hamuje()
    {
     cout <<  "droga hamowania:" << "45 m " << endl;
    }
```

Modelując obiektowo rzeczywistość techniczną często stosowane są struktury hierarchiczne. Do budowy jednych klas wykorzystujemy inne klasy. Np. koła zębate mogą być składnikami skrzynki przekładniowej, czujniki i komputer pokładowy składnikami systemu ABS, itd.

Ważnym pojęciem podejścia obiektowego jest enkapsulacja polegająca na tym, że w strukturze klasy umieszczone są zarówno dane jak i funkcje z nią związane. Budując klasę możemy decydować, które jej składniki będą prywatne, a które publiczne.

Składniki wchodzące w skład klasy mogą być typu **private** lub **public**. **Private** oznacza, że są one dostępne jedynie z wnętrza klasy. **Public** pozwala na dostęp z zewnątrz.

Jeżeli nie nastąpiło określenie żadnego z powyższych typów to domyślnie będzie on typu **private**.

W trakcie zapisu opisu klasy i związanych z nią obiektów na komputerze zwykle mamy kilka składników:

- opis klasy,
- opis danych wchodzących w skład obiektów,
- opis funkcji należących do klasy.

Opis klasy i opis funkcji są składowane jednokrotnie, opis danych obiektów jest przechowywany wielokrotnie.

Bardzo ważną konstrukcją w programowaniu obiektowym są **konstruktory**. **Konstruktor** jest metodą, należącą do danej klasy, o nazwie takiej samej jak nazwa klasy. **Konstruktor** właściwy określonej klasie jest uruchamiany przy każdorazowym powstawaniu obiektu. Konstruktor może być bezargumentowy lub z argumentami. W przypadku **konstruktora** z argumentami trzeba każdorazowo deklarując obiekt podawać parametry **konstruktora**. **Konstruktor** z założenia nie zwraca wartości.

Inną funkcją związaną z konstrukcją określonej klasy jest **destruktor**. Nosi on nazwę taką samą jak nazwa klasy, poprzedzoną znakiem **~**. **Destruktor** z założenia jest funkcją bezargumentową i nie zwraca żadnej wartości.

W języku C++ istnieje możiwość tworzenia kilku funkcji o tej samej nazwie różniących się listami parametrów. Np. tworzymy funkcję **rysuj()** w kilku wariantach:

```
int rysuj(int, int)
int rysuj(float)
int rysuj(int, long, float)
```

O tym, która z funkcji będzie uruchomiana w danym przypadku zdecyduje dopasowanie ilości argumentów i ich typów. Mechanizm ten nazywany jest **przeciążaniem funkcji**. Można go stosować również w przypadku konstruktorów.

Poniżej (przykład 7.4) pokazana zostanie aplikacja ilustrującą problem uchwycenia zagadnień geometrycznych za pomocą formalizmu obiektowego. Modelowany obiekt to okrąg.

### Przykład 7.4 wykonany jako Windows Forms Application

Utworzona klasa *Okrag* zawiera trzy właściwości *Xs*, *Ys* (współrzędne środka) i *R* (promień) oraz dwie metody *PoleOkregu* i *OdlegloscOd00*. Po naciśnięciu przycisku *btnOdlegloscOd00* tworzony jest obiekt *mojokrag* klasy *Okrag*.

```
Okrag^ mojokrag = gcnew Okrag();
```

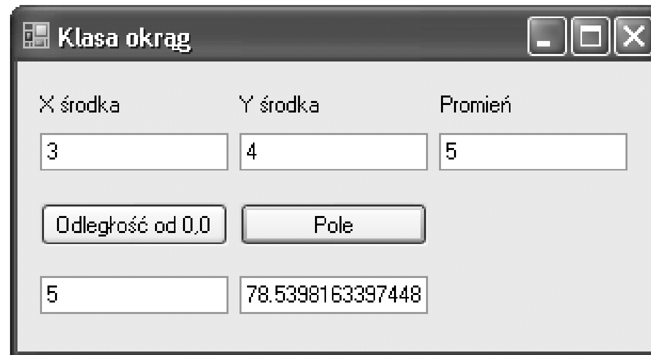
Na podstawie okienek tekstowych ustalane są wartości właściwości *Xs* i *Ys* obiektu *mojokrag*.

```
mojokrag->Xs = (Convert::ToInt32(txtX->Text));
mojokrag->Ys = (Convert::ToInt32(txtY->Text));
```

Wywoływana jest metoda *OdlegloscOd00*, która wylicza wynik umieszczany następnie w okienku tekstowymi t*xtOdlegloscOd00*.

```
txtOdlegloscOd00->Text =
   Convert::ToString(mojokrag->OdlegloscOd00());
```

Przycisk obliczający pole działa podobnie.




Rysunek 7.3. Widok okna aplikacji w trakcie działania

### Kod aplikacji

```
    ref class Okrag {

private:
    int xs;
    int ys;
    int r;
```

```
public:
    property int Xs {
          int get() {
                return xs;
          }
          void set(int xs) {
                this->xs = xs;
          }
    }

    property int Ys {
          int get() {
                return ys;
          }
          void set(int ys) {
                this->ys = ys;
          }
    }

    property int R {
          int get() {
                return r;
          }
          void set(int r) {
                this->r = r;
          }
    }

double PoleOkregu(int r){
    return Math::PI * r * r;
    }

double OdlegloscOd00(){
    return Math::Sqrt(xs*xs + ys*ys);
    }
};
//----------------------------------------------------
#pragma endregion
//----------------------------------------------------
private: System::Void btnOdlegloscOd00_Click(
    System::Object^  sender, System::EventArgs^  e) {
    Okrag^ mojokrag = gcnew Okrag();
    mojokrag->Xs = (Convert::ToInt32(txtX->Text));
    mojokrag->Ys = (Convert::ToInt32(txtY->Text));
    txtOdlegloscOd00->Text =
       Convert::ToString(mojokrag->OdlegloscOd00());
 }
//----------------------------------------------------
```

```
private: System::Void btnPole_Click(System::Object^
                    sender, System::EventArgs^  e) {
    Okrag^ mojokrag = gcnew Okrag();
    mojokrag->R= (Convert::ToInt32(txtR->Text));
    txtPole->Text =
                Convert::ToString(
                mojokrag->PoleOkregu(mojokrag->R));
 }
};
}
//--------------------------------------------------
```

# Podsumowanie i uzupełnienia

W rozdziale przedstawiono koncepcję klasy oraz zasadnicze zagadnienia związane z jej tworzeniem jak i wykorzystywaniem. Klasy używane są w wielu różnych programach stosowanych do rozwiązania realnych codziennych problemów.

### Deklaracja i definicja klasy

Deklarowanie klasy polega na utworzeniu opisu tej klasy. Definiujemy natomiast obiekty określonej klasy.

### Składniki prywatne i publiczne klasy

Składniki klasy mogą być prywatne lub publiczne. Generalnie, jeżeli nie zostanie to określone, to domyślnie wszystkie składniki będą prywatne. Dostęp do składników prywatnych jest możliwy jedynie za pośrednictwem metod należących do danej klasy. Wygodnym rozwiązaniem jest używanie w dużym stopniu prywatnych składników klas. Dostęp do nich można zapewnić budując funkcje publiczne.

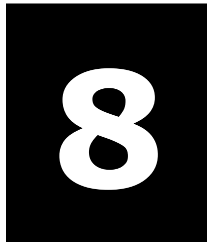
### Przeciążąnie operatorów

Poza przeciążaniem funkcji w języku C++ dostępne jest przeciążanie operatorów. W języku C++ wystepuje cały szereg typów obiektów wbudowanych do kompilatora np. obiekty int, float, itd. Każdy z nich ma przypisane pewne możliwe operacje jakie mogą być wykonane za pomocą określonych operatorów np. operator + używany jest do dodawania liczb i to zarówno całkowitych jak i rzeczywistych, i innych. Każdorazowo oznacza to uruchamianie określonych funkcji. W języku C++ istnieje możliwość zbudowania specjalnych funkcji, które pozwolą przeciążyć standardowy operator nadając mu nowy sens w przypadku gdy działanie

będzie dotyczyć innych niż standardowe obiekty. Np. można zdefiniować nowy sens operatora + tak aby jego stosowanie pozwalało na dodawanie liczb zespolonych czy też macierzy. Nie wszystkie dostępne operatory można przeciążać. Przy przeciążaniu operatorów obowiązuje inny tok postępowania w przypadku operatorów jednoargumentowych, i inny w przypadku operatorów dwuargumentowych.

## Zadania do samodzielnego wykonania

1. Zbuduj klasę opisująca geometrię stopnia wału maszynowego (składniki: średnica i długość stopnia). Utwórz funkcje obliczające objętość i masę wału.
2. Zbuduj na podstawie przykładu 1) tablicę obiektów typu stopień wału tak aby całość stanowiła model geometryczny wału.
3. W przykładzie 2) dodaj możliwość zapisu adresów do obiektów opisujących odpowiednio dane dotyczące geometrii rowka na wpust, rowka pod pierścień osadczy, gwintu.



# Dziedziczenie

Dziedziczenie jest jednym z najważniejszych i najczęściej używanych narzędzi programowania obiektowego. Dziedziczenie pozwala na bazie klasy istniejącej, zwanej **klasą bazową**, zdefiniować nową klasę, zwaną **klasą pochodną**.

Dziedziczenie jest stosowane przede wszystkim w przypadku poszerzenia lub modyfikacji możliwości klas zbudowanych wcześniej. Proces ten wynika najczęściej z konieczności rozwoju istniejącego oprogramowania np. dodania nowych funkcjonalności, modyfikacji funkcjonalności już istniejących. Definiując klasę pochodną definiuje się nowe składowe dane i funkcje. Można również umieścić tam nowe, zmodyfikowane definicje składników klasy bazowej.

Zależność pomiędzy **klasą bazową A** i **klasą pochodną B** wyrażamy następująco:

```
class B: public A
{
…
}
```

Dziedziczenie umożliwia ponowne wykorzystanie powstałego wcześniej oprogramowania. Poniżej przedstawiono prosty przykład dziedziczenia.

**Przykład 8.1**

Klasa bazowa:

```
class pojazd {
public: void start();
    void ruch();
};
void pojazd::start() {cout << "pojazd rusza  \n";}
void pojazd::ruch() {cout << "pojazd porusza się \n";}
```

Klasa pochodna:

```
class rower : public pojazd {
public: void ruch();
};
void rower::ruch() {
cout << "rower porusza się napędzany siłą mięśni \n";}
```

Przykład wykorzystania powyższych klas:

```
void main()
{
    rower gorski;
    gorski.start();
    gorski.ruch();
}
```

W przykładzie przedstawiono możliwości tkwiące w przesłanianiu składników klasy bazowej. W obu klasach występuje funkcja o tej samej nazwie **ruch()**. Jeżeli utworzonym obiektem jest obiekt klasy **rower** to uruchomiona jest funkcja należąca do tej klasy. Jeżeli będzie to obiekt klasy pojazd to będzie to funkcja przynależna klasie **pojazd**.

### Przykład 8.1a jako Windows Forms Application

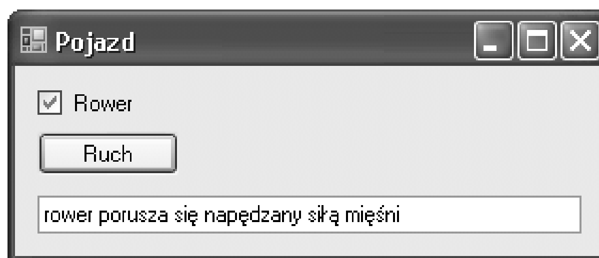
Utworzona klasa bazowa pojazd zawiera właściwość *Sposób_poruszania* oraz metodę *ruch*. Klasa potomna *rower* dziedziczy po klasie *pojazd*.

```
ref class rower: public pojazd
```

oraz posiada własną metodę o tej samej nazwie *ruch*. Po naciśnięciu przycisku *btnRuch* sprawdzane jest pole wyboru *chkRower*. Jeśli pole to zawiera prawdę tworzony jest obiekt oparty na klasie *rower*, w przeciwnym wypadku na klasie *pojazd*. Następnie wywoływana jest metoda *ruch* właściwa dla tego obiektu.

```
if (chkRower->Checked) {
  rower^ mojpojazd = gcnew rower();
  mojpojazd->ruch();
  txtKomentarz->Text = mojpojazd->Sposob_poruszania;
 }
else {
  pojazd^ mojpojazd = gcnew pojazd();
  mojpojazd->ruch();
  txtKomentarz->Text = mojpojazd->Sposob_poruszania;
 }
```

Wynik umieszczany jest w oknie tekstowym txtKomentarz.




Rysunek 8.1. Widok okna aplikacji w trakcie działania

### Kod aplikacji

```
    ref class pojazd {

    public:
          String^ sposob_poruszania;

    public:

    property String^ Sposob_poruszania {
    String^ get() {
          return sposob_poruszania;
    }
    void set(String^ sposob_poruszania) {
      this->sposob_poruszania = sposob_poruszania;
    }
}

    void ruch(){
          sposob_poruszania = "pojazd porusza się" ;
    }

};

//---------------------------------------------------

    ref class rower: public pojazd {

    public:

    void ruch(){
      sposob_poruszania =
        "rower porusza się napędzany siłą mięśni " ;
    }
};
//---------------------------------------------------
```

```
#pragma endregion

//-------------------------------------------------
private: System::Void btnRuch_Click(System::Object^
                   sender, System::EventArgs^  e) {
    if (chkRower->Checked) {
       rower^ mojpojazd = gcnew rower();
       mojpojazd->ruch();
       txtKomentarz->Text =
                      mojpojazd->Sposob_poruszania;
     }
     else {
       pojazd^ mojpojazd = gcnew pojazd();
       mojpojazd->ruch();
       txtKomentarz->Text =
                      mojpojazd->Sposob_poruszania;
     }
 }
};
}
//-------------------------------------------------
```

Dla składników klasy typu dane dostępne są podobne możliwości. Prześledzono to na przykładzie:

**Przykład 8.2**

Klasa bazowa:

```
class rower {
public:
void cena_roweru (int);
void wiek_rower (int);
void predkosc_roweru();
private:
   int cena;
   int wiek;
};
```

Jeżeli r jest obiektem klasy rower to wyrażenie:

```
r. cena_roweru(cena);
```

pozwala na wprowadzanie stosownej danej do pola obiektu. Założono, że nasz pomysł na rower został ukonkretniony jako rower wyposażony w przerzutkę. Daje to nowe cechy roweru, które opisano w klasie pochodnej.

```
class rower_z_przerzutka: public rower {
public:
void wybor_biegu (int);
private:
int numer_biegu;
};
```

Klasa **rower_z_przerzutka** jako klasa pochodna dziedziczy wszystkie składniki z klasy **rower**. Definicja klasy **rower_z_przerzutka** wzbogaca opis klasy o nowy składnik **numer_biegu** i nową metodę **wybor_biegu()**. Numer biegu zapisywany jest jako zmienna typu **int**.

Funkcje **cena_roweru()**, **wiek_roweru()**, **predkosc_roweru()** są typu **public** w obu klasach.

Klasa **rower**:

- cena
- wiek
- cena_roweru ()
- wiek_rower()
- predkosc_roweru()

Klasa **rower_z_przerzutka** zawiera wszystko to co powyżej oraz nowe składniki:

- cena
- wiek
- cena_roweru()
- wiek_rower()
- predkosc_roweru()
- numer_biegu
- wybor_biegu()

Powyższe przykłady ilustrują dziedziczenie dla składników typu **public**. Proces dziedziczenia zapewnia w tym przypadku dużą otwartość tworzonych struktur.

Dziedziczenie w przypadku typu **private** wygląda inaczej i jest bardziej złożone. Składniki typu **private** są widoczne wyłącznie w klasie bazowej. W klasie pochodnej prywatny składnik klasy bazowej nie jest widoczny.

Prywatny składnik klasy bazowej dziedziczy klasa pochodna. Nie jest on jednak w tej klasie widoczny. Prywatny składnik dziedziczony nie jest dostępny dla metod klasy pochodnej. Dostęp do niego jest możliwy za pomocą dziedziczonych metod publicznych z klasy bazowej. Pokazano to na przykładzie:

### Przykład 8.3

```
class punkt_3D {
public:
void wstaw_x (int x1) {x = x1 ;}
void wstaw_y (int y1) {y = y1 ;}
void wstaw_z (int z1) {z = z1 ;}

int podaj_x () {return x;}
int podaj_y () {return y;}
int podaj_z () {return z;}
private:
int x;
int y;
int z;
};

class kolor_punktu_3D: public punkt_3D {
public:
void wstaw_kolor (int i) {kolor = i ; }
int podaj_kolor() {return kolor;}
private:
int kolor;
};
```

Klasa **kolor_punktu_3D** dziedziczy składniki prywatne **x**, **y** i **z** klasy **punkt_3D**. W klasie **kolor_punktu** można się do nich odwoływać za pomocą funkcji publicznych **public wstaw_x()**, **wstaw_y()**, **wstaw_z()**, **podaj_x()**, **podaj_y()**, **podaj_z()**.

### Przykład 8.3a jako Windows Forms Application

Utworzona klasa bazowa *punkt_3D* zawiera trzy właściwości *X*, *Y*, *Z*. Klasa potomna *punkt_3D_kolor* dziedziczy po klasie *punkt_3D*

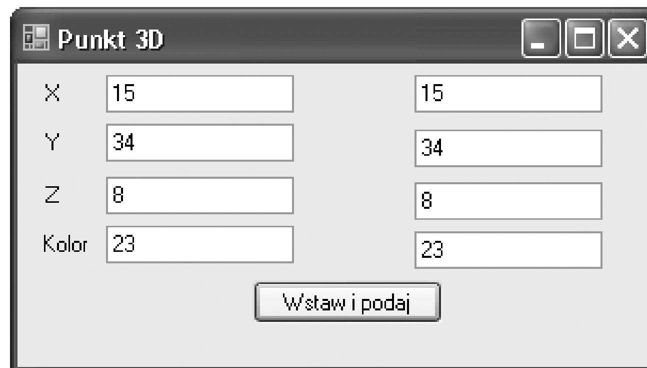
```
ref class punkt_3D_kolor: public punkt_3D
```

oraz posiada własną właściwość *Kolor*. Po naciśnięciu przycisku *btnWstaw* tworzony jest obiekt *mojpunkt* oparty na klasie *punkt_3D_kolor*. Obiekt ten posiada cztery właściwości: *X*, *Y*, *Z* oraz *Kolor*. Procedura ustawia wartości właściwości obiektu na podstawie okienek tekstowych

```
mojpunkt->X = (Convert::ToInt32(txtX_in->Text));
mojpunkt->Y = (Convert::ToInt32(txtY_in->Text));
mojpunkt->Z = (Convert::ToInt32(txtZ_in->Text));
mojpunkt->Kolor =
              (Convert::ToInt32(txtKolor_in->Text));
```

Następnie wartości właściwości pokazywane są w oknach tekstowych po prawej stronie formularza.

```
txtX_out->Text = Convert::ToString(mojpunkt->X);
txtY_out->Text = Convert::ToString(mojpunkt->Y);
txtZ_out->Text = Convert::ToString(mojpunkt->Z);
txtKolor_out->Text =
                 Convert::ToString(mojpunkt->Kolor);
```




Rysunek 8.2. Widok okna aplikacji w trakcie działania

## Kod aplikacji

```
        ref class punkt_3D {

    private:

        int x;
        int y;
        int z;

    public:


        property int X {
              int get() {
                    return x;
              }
              void set(int x) {
                    this->x = x;
              }
        }

        property int Y {
              int get() {
                    return y;
              }
              void set(int y) {
                    this->y = y;
              }
        }

        property int Z {
              int get() {
                    return z;
              }
              void set(int z) {
                    this->z = z;
              }
        }

};
//--------------------------------------------------
```

```
          ref class punkt_3D_kolor: public punkt_3D {

    private:
          int kolor;

    public:
          property int Kolor {
                int get() {
                      return kolor;
                }
                void set(int kolor) {
                      this->kolor = kolor;
                }
          }

};
//-------------------------------------------------

#pragma endregion

//-------------------------------------------------
private: System::Void btnWstaw_Click(System::Object^
                   sender, System::EventArgs^  e) {

    punkt_3D_kolor^ mojpunkt =
                            gcnew punkt_3D_kolor();
    mojpunkt->X = (Convert::ToInt32(txtX_in->Text));
    mojpunkt->Y = (Convert::ToInt32(txtY_in->Text));
    mojpunkt->Z = (Convert::ToInt32(txtZ_in->Text));
    mojpunkt->Kolor =
             (Convert::ToInt32(txtKolor_in->Text));
    txtX_out->Text = Convert::ToString(mojpunkt->X);
    txtY_out->Text = Convert::ToString(mojpunkt->Y);
    txtZ_out->Text = Convert::ToString(mojpunkt->Z);
    txtKolor_out->Text =
                 Convert::ToString(mojpunkt->Kolor);

     }
};
}
//-------------------------------------------------
```

Może się zdarzyć, że składniki o tych samych nazwach występują zarówno w klasie bazowej jak i w klasie pochodnej. Mogą się wówczas pojawić pewne komplikacje. Pokazano to na przykładzie.

```
class punkt_3D{
public:
int x, y, z;
};
class pochodna: public punkt_3D {
public:
int x, y, z;
};

int main () {
pochodna punkt_1;
punkt_1.x = 5
punkt_1. punkt_3D::x = 3
```

W powyższy przykładzie pokazano, że realizowalny jest jednoznaczny dostęp do składników należących do różnych klas. Pierwszy **x** to klasa pochodna, drugi **x** klasa bazowa.

Poza składnikami **private** i **public** istnieje możliwość definiowania składników typu **protected**.

Jeżeli nie następuje proces dziedziczenia to składniki typu **protected** mają identyczne cechy jak składniki typu **private**. Jeżeli natomiast zachodzi dziedziczenie typu **public** to składniki typu **protected**, należące do klasy bazowej, są widoczne w klasie pochodnej.

W przypadku dziedziczenia typu **public** składniki typu **public** klasy bazowej stają się składnikami publicznymi klasy pochodnej. Natomiast dla przypadku dziedziczenia typu **private** składniki publiczne klasy bazowej stają się składnikami **private** klasy pochodnej.

Jeżeli w strukturze klas powiązanych dziedziczeniem występują konstruktory to rodzi to cały szereg złożoności związanych z przesyłaniem parametrów.

Bardziej złożonym problemem pojawiającym się w trakcie procesu dziedziczenia jest dziedziczenie konstruktorów i ewentualne przesyłanie parametrów. Zagadnienie to zilustrowano przykładem.

**Przykład 8.4**

```
class silnik {
public:
silnik() {moc = 100;}
protected:
int moc;
};
class osiagi : public silnik  {
public:
void pisz() {cout < moc << endl;}
};
int main () {
osiagi auto;
auto.pisz()
}
```

W przedstawionym przykładzie konstruktor klasy bazowej generuje daną w chwili powstania obiektu klasy pochodnej.

**Przykład 8.4a jako Windows Forms Application**

Utworzona klasa bazowa silnik zawiera właściwości Moc. Jest w niej zdefiniowany konstruktor:

```
silnik() {
this->moc = 100;
};
```

destruktor

```
~silnik() {
this->moc = 0;
};
```

i finalizator

```
!silnik() {
};
```

Klasa potomna osiagi dziedziczy po klasie silnik

```
          ref class osiagi: public silnik {
};
```

Po naciśnięciu przycisku *btnMoc* tworzony jest obiekt *mojeosiagi* oparty na klasie *osiagi*

```
    osiagi^ mojeosiagi = gcnew osiagi();
```

Wywołana metoda *Moc* w klasie potomnej korzysta z konstruktora klasy bazowej. *Wynik* jest wpisywany do okna tekstowego *txtMoc*.

```
txtMoc->Text = Convert::ToString(mojeosiagi->Moc);
```

Rysunek 8.3. Widok okna aplikacji w trakcie działania

**Kod aplikacji**

```
          ref class silnik {

    protected:

          int moc;

    public:

          silnik() {
          this->moc = 100;
          };
          ~silnik() {
          this->moc = 0;
          };
          !silnik() {
          };

          property int Moc {
                int get() {
                      return moc;
                }
                void set(int moc) {
                      this->moc = moc;
                }
          }

};
//-------------------------------------------------
```

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

          ref class osiagi: public silnik {
};

//------------------------------------------------

#pragma endregion

//------------------------------------------------

private: System::Void btnMoc_Click(
  System::Object^  sender, System::EventArgs^  e) {
    osiagi^ mojeosiagi = gcnew osiagi();
    txtMoc->Text =
                Convert::ToString(mojeosiagi->Moc);
    }
  };
}
//------------------------------------------------
```

Zagadnienie dziedziczenia w oprogramowaniu wspomagającym prace inżynierskie może odnosić się do hierarchizacji używanych obiektów geometrycznych. Najczęściej wyodrębnia się jakiś obiekt nadrzędny, który z kolei jest obiektem bazowym dla szeregu obiektów pochodnych. Np. może to być obiekt generowany prze klasę bazową **prymityw**, która dalej jest dziedziczona przez różne obiekty geometryczne. W przykładzie występuje tylko klasa pochodna **okrag**. Łatwo można powielić ten schemat na inne figury geometryczne. Przykład poniższy to rozwinięcie przykładu z rozdziału 7.

### Przykład 8.5

Przykład dla okręgu z dziedziczeniem wykonany jako Windows Forms Application

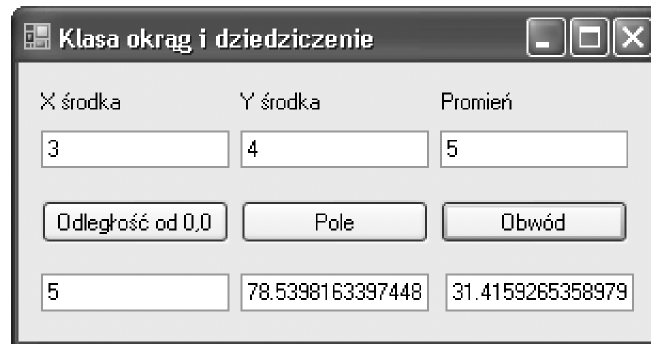
Program jest rozbudowaną o dziedziczenie wersją programu z rozdziału poprzedniego. Utworzona jest klasa potomna *Okrag_obwod* dziedzicząca po klasie bazowej *Okrag* wszystkie właściwości i metody. Dodatkowo posiada własną metodę *Obwod*.

```
ref class Okrag_obwod: public Okrag
```

Po naciśnięciu dowolnego z przycisków tworzony jest obiekt *mojokrag* klasy *Okrag_obwod*

```
    Okrag_obwod^ mojokrag = gcnew Okrag_obwod();
```

Obiekt ten korzysta ze wszystkich właściwości i metod klasy bazowej *Okrag* i potomnej *Okrag_obwod*.




Rysunek 8.4. Widok okna aplikacji w trakcie działania

**Kod aplikacji**

```
      ref class Okrag {

private:

      int xs;
      int ys;
      int r;

public:

      property int Xs {
            int get() {
                  return xs;
            }
            void set(int xs) {
                  this->xs = xs;
            }
      }

      property int Ys {
            int get() {
                  return ys;
            }
            void set(int ys) {
                  this->ys = ys;
            }
      }
```

```
            property int R {
                  int get() {
                        return r;
                  }
                  void set(int r) {
                        this->r = r;
                  }
            }

      double PoleOkregu(int r){
            return Math::PI * r * r;
            }

      double OdlegloscOd00(){
            return Math::Sqrt(xs*xs + ys*ys);
            }
};
//------------------------------------------------

            ref class Okrag_obwod: public Okrag {

      public:

      double Obwod(int r){
            return Math::PI * 2 * r;
            }
};
//------------------------------------------------

#pragma endregion

//------------------------------------------------

private: System::Void btnOdlegloscOd00_Click_1(
  System::Object^  sender, System::EventArgs^  e) {
    Okrag_obwod^ mojokrag = gcnew Okrag_obwod();
    mojokrag->Xs = (Convert::ToInt32(txtX->Text));
    mojokrag->Ys = (Convert::ToInt32(txtY->Text));
    txtOdlegloscOd00->Text =
      Convert::ToString(mojokrag->OdlegloscOd00());
 }

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

```
private: System::Void btnPole_Click_1(System::Object^
                  sender, System::EventArgs^  e) {
    Okrag_obwod^ mojokrag = gcnew Okrag_obwod();
    mojokrag->R= (Convert::ToInt32(txtR->Text));
    txtPole->Text =
Convert::ToString(mojokrag->PoleOkregu(mojokrag->R));
 }
//-------------------------------------------------

private: System::Void btnObwod_Click(System::Object^
                   sender, System::EventArgs^  e) {
    Okrag_obwod^ mojokrag = gcnew Okrag_obwod();
    mojokrag->R = (Convert::ToInt32(txtR->Text));
    txtObwod->Text =
   Convert::ToString(mojokrag->Obwod(mojokrag->R));
     }
};
}
//-------------------------------------------------
```

# Podsumowanie i uzupełnienia

Dziedziczenie jest mechanizmem, który pozwala budować nowe klasy wykorzystując klasy już istniejące. Nowe klasy powstają jako pewne ich modyfikacje lub rozszerzenia.

### Składniki protected

Składniki klas określone jako **protected** w klasie bazowej mają cechy składników typu private. Przy dziedziczeniu funkcje i zmienne określone jako **protected** dostępne są we wszystkich klasach pochodnych. Ich cechy w klasach pochodnych to cechy **private**.

### Nadpisywanie funkcji

Budując klasę pochodną można w niej zbudować funkcję, która posiada tą samą nazwę jak funkcja w klasie bazowej, tę samą wartość zwracaną i tę samą listę parametrów. Jest to nadpisywanie funkcji, w klasie pochodnej uruchamiana będzie nowa funkcja. Nadpisywanie oznacza zastąpienie funkcji z klasy bazowej funkcją z klasy pochodnej.

Jeżeli funkcja z klasy bazowej została nadpisana to nadal istnieje możliwość jej uruchomienia. Konieczne jest pełne jej określenie wraz z podaniem nazwy klasy.

## Zadania do samodzielnego wykonania

1. Do klasy opisującej stopień wału w postaci walca dodaj klasę pochodną modelującą stopień w postaci walca drążonego. Jako funkcję zamodeluj możliwość obliczania objętości wału drążonego.
2. Do zadania 1) dodaj możliwość modelowania wału maszynowego jako listy stopni drążonych.
3. Do zadania 2) dodaj możliwość modelowania dwóch wałów skrzynki przekładniowej.

# Polimorfizm

Polimorfizm jest to mechanizm, który pozwala na bezpośrednie sterowanie procesem przetwarzania programu w trakcie jego wykonywania. Zakładamy, że zbudowane są klasy powiązane mechanizmem dziedziczenia. W każdej z klas jest zdefiniowana funkcja o tej samej nazwie, nazwę każdej z funkcji poprzedza słowo kluczowe **virtual**. Potrzebna jest zmienna wskaźnikowa pozwalająca na składowanie wskaźnika do obiektu klasy bazowej.

W języku C++ istnieje możliwość podstawiania do zmiennej wskaźnikowej klasy bazowej wskaźnika do obiektu klasy pochodnej. Jeżeli uruchamiamy funkcję wirtualną to w zależności od tego co wskazuje wskaźnik zawarty w tej zmiennej wskaźnikowej – obiekt, której z powiązanych dziedziczeniem klas, następuje uruchamianie odpowiedniej funkcji wirtualnej (należącej do odpowiedniej klasy).

Poniżej pokazano przykład zastosowania tej konstrukcji.

**Przykład 9.1 wykonany jako Console Application**

```
class czworokat {
public:
virtual void info () {
  cout << „to jest czworokat”<<endl;}
};

class romb : public czworokat {
public:
virual void info () {
  cout << „to jest romb”<< endl;}
};

class kwadrat : public czworokat {
public:
virtual void info () {
  cout << „to jest kwadrat”<< endl;}
};

int main () {
  czworokat cztery_narozniki;
  romb boki_rownolegle;
  kwadrat boki_katy_rowne;
  czworokat * wskaznik;
  int ktory
  do {
    cout << „1 - czworokat, 2 - romb, 3 - kwadrat”;
    cin>>ktory;
  }
```

```
  while (ktory < 1 || który > 3);

  switch (który){
  case 1 : wskaznik = &cztery_narozniki ; break;
  case 2 : wskaznik = &boki_rownolegle; break;
  case 3 : wskaznik = &boki_katy_rowne; break;
  }

  wskaznik -> info();
  return 0;
}
```

W przedstawionym przykładzie występuje jedna klasa bazowa i dwie klasy pochodne. W każdej z klas zdefiniowano funkcje wirtualne. Dla każdej z klas utworzono obiekty. Jeżeli wskazując obiekty posługujemy się wskaźnikami to w zależności od tego co jest wskazywane następuje uruchamianie właściwej funkcji.

Powyższy mechanizm może również zadziałać w przypadku jeżeli słowo **virtual** wystąpi jedynie w funkcji bazowej.

```
class czworokat {
public:
virtual void info ()
{
  cout << „to jest czworokat”<<endl;}
};

class romb : public czworokat {
public:
 void info ()
{
  cout << „to jest romb”<< endl;}
};
```

Funkcje **wirtualne** mogą być dziedziczone.

Często stosowanym rozwiązaniem jest łączenie funkcji wirtualnych z dynamiczną rezerwacją pamięci.

```
switch (ktory) {
case 1 : wskaznik = new czworokat ; break;
case 2 : wskaznik = new romb; break;
case 3 : wskaznik = new kwadrat; break;
}

wskaznik -> info();
```

Usuwanie obiektu, który utworzono w sposób dynamiczny może być zrealizowane za pomocą **delete**.

```
delete wskaznik;
```

Zmienna wskaźnikowa używana w polimorfizmie może stanowić element tablicy wskaźników. Jest to częsty przypadek, np.

```
czworokat * wskazniki [100];
```

Jeżeli przetwarzamy w programie zagadnienia polegające na generowaniu różnych figur geometrycznych i mamy sytuację, że w tablicy wskaźników **wskaźniki[100]** składowane są w sposób przypadkowy, wynikający z przetwarzania, adresy do różnych obiektów typu **czworokąt** , **romb**, **kwadrat** to do przechowywania możemy wykorzystać dynamiczne rezerwowanie pamięci za pomocą **new**. Natomiast wykorzystanie wirtualnej funkcji info () może być powiązane z usuwaniem obiektów z pamięci za pomocą delete.

```
for (i = 0; i < 100; i++){
  wskazniki[i] -> info();
  delete wskazniki[i];
}
```

### Przykład 9.1a wykonany jako Windows Forms Application

W aplikacji  w zależności od tego, który z przycisków został naciśniety wywoływana jest odpowiednia funkcja wirtualna. W jednoelementowej tablicy wskaźniki przechowywany jest właściwy uchwyt obiektu pochodnego dla klasy *czworokat*.

Rysunek 9.1. Widok okna aplikacji w trakcie działania

### Kod aplikacji

```
    ref class czworokat {

    public:
    String^ opis;

    public:

    property String^ Opis {
          String^ get() {
                return opis;
          }
          void set(String^ opis) {
                this->opis = opis;
          }
    }
      virtual void zapisz(){
          opis = "To jest czworokąt";
    }
};
//---------------------------------------------------

          ref class romb: public czworokat {

    public:

    virtual void zapisz() override {
          opis = "To jest romb";
    }
};
//---------------------------------------------------

    ref class kwadrat: public czworokat {

    public:

    virtual void zapisz() override {
          opis = "To jest kwadrat";
    }
};
//---------------------------------------------------

#pragma endregion

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

```
private: System::Void btnRomb_Click(System::Object^
                  sender, System::EventArgs^  e) {
    cli::array<czworokat^>^ wskazniki =
                   gcnew cli::array<czworokat^>(1);
    wskazniki[0] = gcnew romb();
    for each (czworokat^ b in wskazniki){
          b->zapisz();
    txtOpis->Text = Convert::ToString(b->Opis);
    }
 }
//---------------------------------------------------
private: System::Void btnKwadrat_Click(
  System::Object^  sender, System::EventArgs^  e) {
    cli::array<czworokat^>^ wskazniki =
                  gcnew cli::array<czworokat^>(1);
    wskazniki[0] = gcnew kwadrat();
    for each (czworokat^ b in wskazniki){
          b->zapisz();
    txtOpis->Text = Convert::ToString(b->Opis);
    }
 }
};
}
//---------------------------------------------------
```

Wykorzystanie polimorfizmu zostało opisane w pracy [14]

# Podsumowanie i uzupełnienia

Funkcje wirtualne wykorzystują praktycznie zmienne wskaźnikowe i referencje.

### Klasy abstrakcyjne

Budując cały zbiór klas powiązanych dziedziczeniem może okazać się, że niezbędne są klasy, które nie są potrzebne z punktu widzenia tworzenia nowych obiektów, ale wyłącznie jako klasy – opisy abstrakcji, które spajają pewne inne klasy i są dla nich klasami bazowymi. Przykładem takiej klasy dla różnych figur geometrycznych (takich jak trójkąt, kwadrat, itp.) mogą być klasy będące abstrakcjami np. kształt, prymityw, obiekt. W każdym z tych przypadków trudno jest zrealizować jego konkretyzację polegającą na utworzeniu konkretnego obiektu. Klasy o takich własnościach nazywane są klasami abstrakcyjnymi. W języku C++ abstrakcyjną klasę tworzy się poprzez zapisanie w niej funkcji wirtualnej, która jest inicjowana wartością 0. Taka funkcja

nazywana jest funkcją czysto wirtualną. Wystarczy zadeklarowanie w danej klasie jednej takiej funkcji aby klasa była wirtualna i nie można było budować obiektów tej klasy. Funkcje czysto wirtualne są nadpisywane w klasach pochodnych.

## Zadania do samodzielnego wykonania

1. Zbuduj następujące klasy powiązane dziedziczeniem:
    - element osadzony na wale,
    - łożysko - pochodna klasa od klasy element,
    - koło zębate – pochodna klasa od klasy element
    - synchronizator - pochodna klasa od klasy element.

Zbuduj funkcje wirtualne służące do wprowadzania i wyprowadzania danych na temat elementów osadzonych na wale. Następnie zbuduj tablicę wskaźników do obiektów osadzonych na wale wraz z wprowadzaniem i wyprowadzaniem ich danych.

2. Przykład (1) rozszerz o możliwość modelowania wałów w skrzynce 2 i 3 wałkowej.
3. Przykład (2) rozszerz o klasy odpowiadające elementom takim jak wpusty, pierścienie osadcze.

# 10 Przykłady programów obliczeniowych

# 10.1. Rekurencja

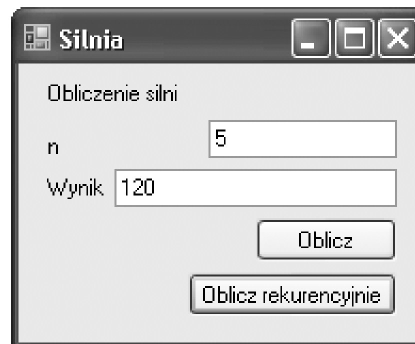
Rekurencja występuje wówczas, gdy funkcja wywołuje samą siebie. Sytuację taką przedstawimy na przykładzie obliczania silni. Program obliczający silnię czyni to na dwa sposoby:

- W Funkcji Silnia wywoływanej przyciskiem **Oblicz**, użyto pętli for

```
for (int i=1; i<=n; i++){
  Silnia_W=Silnia_W*i;
}
```

- W Funkcji SilniaRekurencyjnie wywoływanej przyciskiem **Oblicz rekurencyjnie** użyto rekurencyjnego wywołania tej funkcji

```
if(i>0)
SilniaRekurencyjnie_W=SilniaRekurencyjnie(i-1)*i;
```




Rysunek 10.1. Obliczenie rekurencyjne silni

### Kod aplikacji

```
//---------------------------------------------------
private: System::Void btnOblicz_Click(System::Object^
         sender, System::EventArgs^  e) {
    int n;
    n=Convert::ToInt16(txtN->Text);
    txtWynik->Text=Convert::ToString(Silnia(n));
 }
//---------------------------------------------------
```

```
private: System::Void btnRekurencyjnie_Click(
  System::Object^  sender, System::EventArgs^  e) {
    int n;
    n=Convert::ToInt16(txtN->Text);
    txtWynik->Text=Convert::ToString(
               SilniaRekurencyjnie(n));
 }
//-----------------------------------------------------
    double Silnia(int n){
    double Silnia_W;
    Silnia_W=1;
    for (int i=1; i<=n; i++){
      Silnia_W=Silnia_W*i;
    }
    return Silnia_W;
    }
//-----------------------------------------------------
    double SilniaRekurencyjnie(int i){
    double SilniaRekurencyjnie_W;
    SilniaRekurencyjnie_W=1;
    if(i>0) SilniaRekurencyjnie_W =
                     SilniaRekurencyjnie(i-1)*i;
    return SilniaRekurencyjnie_W;
    }
//-----------------------------------------------------
```

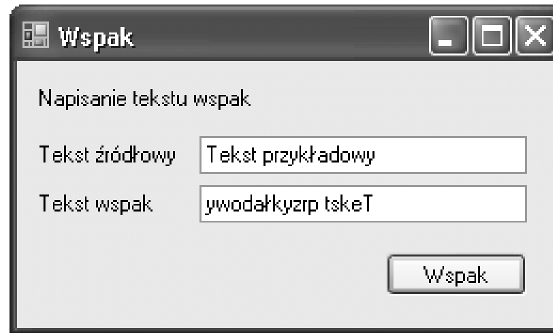
# 10.2. Praca z tekstami - wykorzystanie klasy String

Znak ASCII możemy przechowywać w zmiennej typu **char**. Pamiętany jest kod znaku w postaci liczby całkowitej.

Przy operacjach na tekstach można wykorzystać klasę String.

**Klasa String**

Klasa String znajduje się w przestrzeni nazw System. Obiekt klasy String jest sekwencyjną, kolekcją znaków. Klasa posiada wiele właściwości i metod. W poniższym programie, którego zadaniem jest napisanie tekstu wspak użyto właściwości Length i metody Substring.




Rysunek 10.2. Aplikacja przepisująca tekst wspak

**Kod aplikacji**

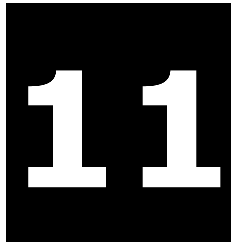
```
//--------------------------------------------------------
private: System::Void btnWspak_Click(System::Object^
                  sender, System::EventArgs^  e) {
    String^ Zrodlo;
    Zrodlo=Convert::ToString(txtZrodlo->Text);
    txtWynik->Text=Convert::ToString(Wspak(Zrodlo));
 }
//--------------------------------------------------------
    String^ Wspak(String^ Tekst_zrodlowy){
    String^ Wspak_W;
    int Dlugosc;
    Dlugosc=Tekst_zrodlowy->Length;
    Wspak_W="";
    for (int i=Dlugosc; i>=1; i--){
      Wspak_W = Wspak_W +
                  Tekst_zrodlowy->Substring(i-1,1);
    }
    return Wspak_W;
 }
//--------------------------------------------------------
```

W programie utworzono funkcję **Wspak**, której parametrem wejściowym jest obiekt klasy **String**, jak również w wyniku działania funkcji otrzymujemy obiekt tej klasy. Szkielet funkcji przedstawiono poniżej.

```
String^ Wspak(String^ Tekst_zrodlowy){
}
```

Znaki przenoszono przy pomocy pętli for. Metoda **Substring** numeruje znaki w ciągu znakowym od zera, tzn. pierwszy znak ma numer 0, drugi 1, a n-ty n-1.

```
for (int i=Dlugosc; i>=1; i--){
  Wspak_W = Wspak_W +
                   Tekst_zrodlowy->Substring(i-1,1);
 }
```



# Przykłady programów użytkowych i multimedialnych

# 11.1. Interaktywna grafika wektorowa

Aby mieć dostęp do funkcji graficznych realizowanych na formularzu aplikacji Windows należy dopisać w bloku określania przestrzeni nazw wiersz:

```
using namespace System::Drawing;
```

Dzięki temu możemy mieć dostęp do klasy *Graphics*. Po utworzeniu obiektów opartych na tej klasie jest możliwe kreślenie podstawowych elementów graficznych jak linie, prostokąty, elipsy, łuki, wielokąty, krzywe itp. Figury zamknięte jak elipsy czy wielokąty mogą być dzięki dodatkowym funkcjom wewnątrz wypełnione.

## Grafika pasywna

Jeśli chcemy, aby za pojawianie się grafiki i jej odświeżanie odpowiadał system operacyjny możemy wykorzystać zdarzenie *Paint* formularza.

Poniższy kod pozwoli na narysowanie prostokąta na ekranie.

Rysunek 11.1. Prostokąt w grafice pasywnej

### Kod aplikacji

```
private: System::Void Form1_Paint(System::Object^
      sender,
      System::Windows::Forms::PaintEventArgs^  e) {
    Graphics^ objRys = e->Graphics;
    objRys->DrawRectangle(Pens::Blue,50,50,100,50);
 }
```

Korzystamy ze zdarzenia Paint formularza.

```
private: System::Void Form1_Paint(System::Object^
sender, System::Windows::Forms::PaintEventArgs^  e) {
```

W linii inicjującej grafikę używamy parametru e procedury zdarzenia

```
    Graphics^ objRys = e->Graphics;
```

Lewy górny narożnik prostokąta ma współrzędne (50,50), długość wynosi 100, a wysokość  50. Linia rysowania jest cienka (domyślnie) i kolor linii jest niebieski.

```
    objRys->DrawRectangle(Pens::Blue,50,50,100,50);
```

System zapewnia odświeżanie rysunku. Jest to grafika pasywna.

## Grafika aktywna

Jeśli chcemy, aby za pojawianie się grafiki i jej odświeżanie odpowiadała nasza aplikacja możemy program zmodyfikować.

Rysunek 11.2. Prostokąt w grafice aktywnej

### Kod aplikacji

```
private: System::Void btnRysujProstokat_Click(
   System::Object^  sender, System::EventArgs^  e) {
     Graphics^ objRys = this->CreateGraphics();
     objRys->DrawRectangle(Pens::Blue,50,50,100,50);
 }
```

Prostokąt będzie narysowany po kliknięciu przycisku btnRysujProstokat. Obsługiwana jest procedura zdarzenia naciśnięcia przycisku.

```
private: System::Void btnRysujProstokat_Click(
   System::Object^  sender, System::EventArgs^  e) {
```

Jeśli okno zostanie np. zminimalizowane to, aby rysunek ponownie się pojawił musimy jeszcze raz nacisnąć przycisk. My decydujemy o odświeżaniu obrazu. W poprzednim przypadku system sam odtworzyłby rysunek. Linia inicjująca grafikę wygląda teraz tak:

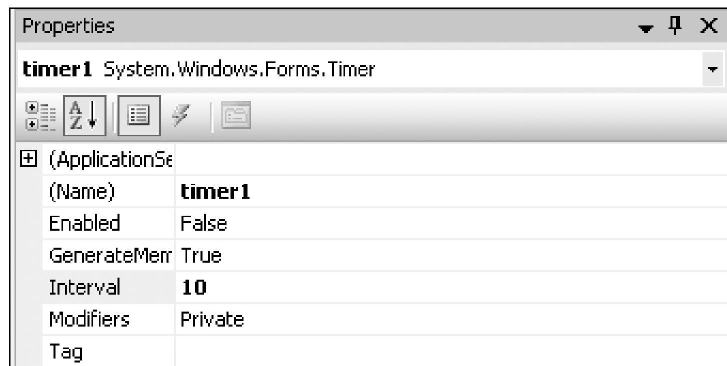
```
     Graphics^ objRys = this->CreateGraphics();
```

Jest to grafika aktywna. Akcja odświeżania rysunku nie musi być związana z naciśnięciem przycisku lub inną akcją użytkownika. Może to być oprogramowana przez nas procedura zdarzenia np. związana z sygnałem od zegara systemowego.

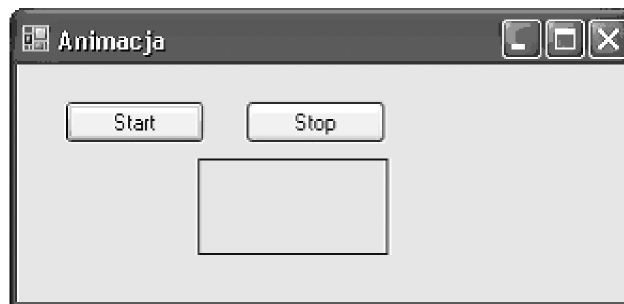
## Animacja wektorowa

Aby uzyskać obraz animowany należy go modyfikować zgodnie z impulsami zegarowymi. Należy zastosować formant Timer przeciągając go z okna Toolbox na formularz.

Aby zegar działał należy go włączyć w oknie właściwości (właściwość Enabled = True) lub zapewnić, że uczyni to nasza aplikacja i ustalić częstotliwość zegara – faktycznie definiowany jest okres określany w ms (właściwość Interval).




Rysunek 11.3. Właściwości formantu Timer




Rysunek 11.4. Prostokąt przesuwający się w prawo

Poniżej przedstawiono kod aplikacji, w której naciśnięcie przycisku Start powoduje przesuwanie się prostokąta w prawo. Naciśnięcie przycisku Stop zatrzymuje animacją.

### Kod aplikacji

```
//-----------------------------------------------------
long x1 = 50;
//-----------------------------------------------------
private: System::Void btnStart_Click(System::Object^
                  sender, System::EventArgs^  e) {
    timer1->Enabled=true;
 }
//-----------------------------------------------------
private: System::Void btnStop_Click(System::Object^
                  sender, System::EventArgs^  e) {
    timer1->Enabled=false;
 }
//-----------------------------------------------------
```

```
private: System::Void timer1_Tick(System::Object^
                     sender, System::EventArgs^  e) {
    Graphics^ objRys = this->CreateGraphics();
    objRys->DrawRectangle(
                SystemPens::Control,x1-1,50,100,50);
    objRys->DrawRectangle(Pens::Blue,x1,50,100,50);
    x1++;
 }
//---------------------------------------------------
```

Procedury zdarzeń naciśnięcia przycisków Start i Stop zmieniają właściwość Enabled Timera odpowiednio na True i False.

W kodzie programu zdefiniowano jedną zmienną globalną, która na początku ma wartość 50.

```
long x1 = 50;
```

Przechowujemy w niej aktualne położenie prostokąta.

W procedurze zdarzenia impulsu zegara inicjowana jest grafika.

```
Graphics^ objRys = this->CreateGraphics();
```

Następnie prostokąt rysowany jest dwa razy, najpierw w kolorze tła dzięki czemu zmazywane jest poprzednie położenie prostokąta, a następnie w nowym położeniu kolorem niebieskim.

```
objRys->DrawRectangle(
           SystemPens::Control,x1-1,50,100,50);
objRys->DrawRectangle(Pens::Blue,x1,50,100,50);
```

Ostatnią instrukcją procedury zdarzenia impulsu zegara jest zwiększenie wartości x1 czyli położenia prostokąta.

```
x1++;
```

W programie zwraca uwagę wielokrotne, przy każdym impulsie zegara, wywoływanie linii inicjowania grafiki.

```
Graphics^ objRys = this->CreateGraphics();
```

Można tę linię przenieść do procedury zdarzenia ładowania formularza, ale trzeba wówczas zadeklarować zmienna objRys jako zmienną publiczną. Kod programu będzie wyglądał wówczas następująco:

```
//--------------------------------------------------
    long x1 = 50;

public:   Graphics^ objRys;
//--------------------------------------------------

private: System::Void Form1_Load(System::Object^
                   sender, System::EventArgs^  e) {
    objRys = this->CreateGraphics();
 }
//--------------------------------------------------
private: System::Void btnStart_Click(System::Object^
                   sender, System::EventArgs^  e) {
    timer1->Enabled=true;
 }
//--------------------------------------------------
private: System::Void btnStop_Click(System::Object^
                   sender, System::EventArgs^  e) {
    timer1->Enabled=false;
 }
//--------------------------------------------------
private: System::Void timer1_Tick(System::Object^
                   sender, System::EventArgs^  e) {
    objRys->DrawRectangle(
                SystemPens::Control,x1-1,50,100,50);
    objRys->DrawRectangle(Pens::Blue,x1,50,100,50);
    x1++;
 }
//--------------------------------------------------
```

Kolejną modyfikacją może być użycie metody **Clear** obiektu rysunkowego w miejsce instrukcji rysującej prostokąt w kolorze tła. Wiersz ten ma postać:

```
    objRys->Clear(SystemColors::Control);
```

## Definiowanie pióra i pędzla

Prezentowany poniżej przykład pokazuje jak wykonać sekundnik zegara.

Rysunek 11.5. Sekundnik

Przedstawiono w nim jak zdefiniować pióro i pędzel do tworzenia figur wypełnionych.

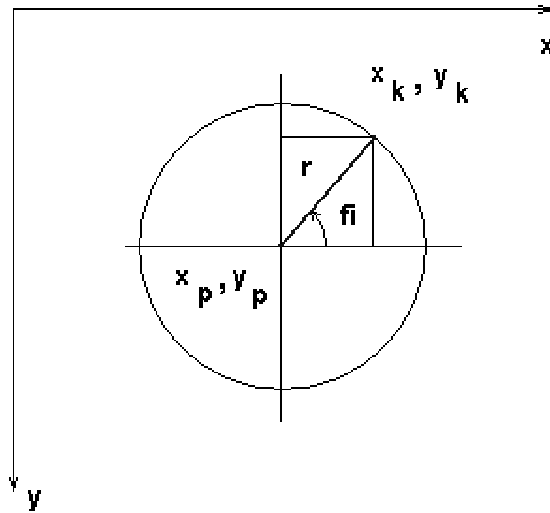
Linia definiująca pióro rysujące wskazówkę sekundnika ma postać:

```
Pen^ mypenWsk = gcnew Pen(Color::Blue,3.0F);
```

Ustalono, że pióro rysować będzie kolorem niebieskim, a grubość pisaka jest równa 3. Linia definiująca pędzel wypełniający kolorem zielono-niebieskim (cyan) tarczę zegara ma postać:

```
SolidBrush^ mybrushTar = gcnew SolidBrush(Color::Cyan);
```

Przycisk na formularzu pełni rolę przełącznika start/stop. Aktualny stan określa zmienna globalna startstop. Współrzędne środka tarczy określone są jako (xp, yp), a współrzędne końca wskazówki poruszającej się po brzegu tarczy to (xk, yk). Promień tarczy oznaczono jako r, a fi to aktualny kąt jaki przemierza wskazówka sekundnika (rysunek 11.6).




Rysunek 11.6. Charakterystyczne punkty sekundnika
w układzie współrzędnych formularza

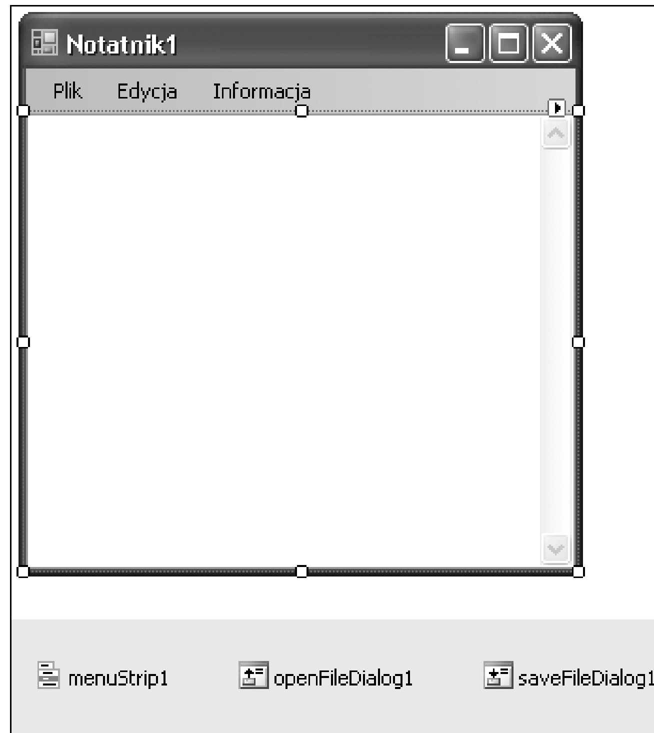
### Kod aplikacji

```
//-------------------------------------------------
    using namespace System::Drawing;
//-------------------------------------------------
    double xp=200, yp=100, xk, yk, fi=-Math::PI/2;
    int startstop=0;
    int r=70;
//-------------------------------------------------
    public:    Graphics^ objRys;
//-------------------------------------------------
private: System::Void Form1_Load(System::Object^
                  sender, System::EventArgs^  e) {
    objRys = this->CreateGraphics();
 }
//-------------------------------------------------
private: System::Void btnstartstop_Click(
  System::Object^  sender, System::EventArgs^  e) {
    if(startstop==0) {
      startstop=1;
      btnstartstop->Text="Stop";
    }
    else {
      startstop=0;
      btnstartstop->Text="Start";
```

```
    }
 }
//-------------------------------------------------
private: System::Void timer1_Tick(System::Object^
                   sender, System::EventArgs^  e) {
    if(startstop==1) {
      objRys->Clear(SystemColors::Control);
      Pen^ mypenWsk = gcnew Pen(Color::Blue,3.0F);
      SolidBrush^ mybrushTar =
                      gcnew SolidBrush(Color::Cyan);
      objRys->FillEllipse(mybrushTar,
              Convert::ToInt32(xp-r),
              Convert::ToInt32(yp-r),
              Convert::ToInt32(2*r),
              Convert::ToInt32(2*r));
      objRys->DrawEllipse(Pens::Magenta,
              Convert::ToInt32(xp-r),
              Convert::ToInt32(yp-r),
              Convert::ToInt32(2*r),
              Convert::ToInt32(2*r));
      xk=xp+r*Math::Cos(fi);
      yk=yp+r*Math::Sin(fi);
      objRys->DrawLine(mypenWsk,
              Convert::ToInt32(xp),
              Convert::ToInt32(yp),
              Convert::ToInt32(xk),
              Convert::ToInt32(yk));
      fi=fi+Math::PI/30;
    }
 }
//-------------------------------------------------
```

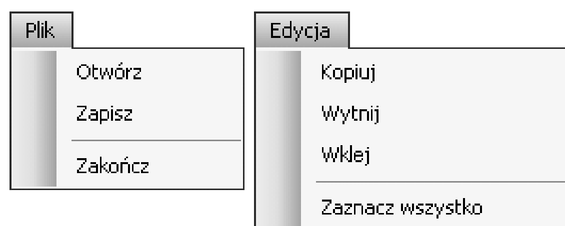
# 11.2. Prosty edytor tekstów

Wykonajmy aplikację podobną do systemowego Notatnika, rysunek 11.7




Rysunek 11.7. Propozycja formularza Notatnika

Aplikacja będzie miała własne menu – dodajmy zatem, z Toolbox'u obiekt realizujący takie zadania. Jest to obiekt *menuStrip*. Obiekt menu została zaprojektowany bardzo przyjaźnie dla programisty. Aby utworzyć pozycje menu należy wpisać w otwierające się na formularzu pola menu odpowiednie ich nazwy. Niech menu posiada trzy pozycje główne: Plik, Edycja i Informacja, a pozycje Plik i Edycja niech posiadają podmenu jak na rysunku 11.8.




Rysunek 11.8. Pozycje menu

Tworzone pozycje menu wyświetlają swoje teksty na formularzu, a ich nazwy widoczne są w oknie Properties, w polu Name. Pozycje menu, zwłaszcza te, dla których napiszemy procedury obsługi zdarzeń Click powinny mieć nazwy proste, krótsze od domyślnych proponowanych przez środowisko. Zmieńmy je zatem na takie jak zaproponowano w tabeli:

Tabela 11.1. Propozycje nazw pozycji menu

| Pozycja | Name |
|---|---|
| Otwórz | mnuOtworz |
| Zapisz | mnuZapisz |
| Zakończ | mnuZakoncz |
| Kopiuj | mnuKopiuj |
| Wytnij | mnuWytnij |
| Wklej | mnuWklej |
| Zaznacz wszystko | mnuZaznaczWszystko |
| Informacja | mnuInformacja |

Dla pozycji Plik i Edycja pozostawiamy nazwy domyślne, ponieważ nie będziemy się do nich odwoływali.

Aby w pozycji menu uzyskać poziomą oddzielającą kreskę należy wpisać w tę pozycję menu pojedynczy znak minus „-”.

Z uwagi na to iż będziemy otwierać do edycji istniejące pliki tekstowe i zapisywać je, może pod inną nazwą – dodajmy do formularza dwa obiekty zaprojektowane do tego celu: openFileDialog i saveFileDialog.

### Kod aplikacji

> **UWAGA**
> Jeśli w sekcji deklaracji, na początku pliku (na górze) dodamy wiersz:
> using namespace System::IO;
> to zamiast pisać, np.:
> System::IO::StreamReader^ srOdczyt =...
> można krócej:
> StreamReader^ srOdczyt =...

```
//--------------------------------------------------
private: System::Void mnuOtworz_Click(System::Object^
         sender, System::EventArgs^  e) {

//Otwarcie okna dialogowego openFileDialog1
     openFileDialog1->InitialDirectory =
             "c:\\tkm\\gr1_1";
     openFileDialog1->Filter =
            "Plik tekstowy (*.txt)|*.txt|" +
            "Wszystkie pliki (*.*)|*.*";
     openFileDialog1->FileName="";
     openFileDialog1->ShowDialog();
 }
//--------------------------------------------------
private: System::Void openFileDialog1_FileOk(
      System::Object^  sender,
      System::ComponentModel::CancelEventArgs^  e) {
 //Wczytanie tekstu z pliku
 //Procedura obsługi zdarzenia:
 //kliknięcie przycisku OK w openFileDialog1

     try
     {
       System::IO::StreamReader^ srOdczyt =
       System::IO::File::OpenText(
            openFileDialog1->FileName);
       txtOkno->Text = srOdczyt->ReadToEnd();
     }
     catch (Exception^ e)
     {
           MessageBox::Show(e->Message);
     }
 }
//--------------------------------------------------
```

```
private: System::Void mnuZapisz_Click(System::Object^
         sender, System::EventArgs^  e) {
 //Procedura wywołuj okno dialogowe saveFileDialog1

     saveFileDialog1->InitialDirectory =
          "c:\\tkm\\gr1_1";
     saveFileDialog1->Filter =
          "Plik tekstowy (*.txt)|*.txt| " +
          " Wszystkie pliki (*.*)|*.*";
     saveFileDialog1->ShowDialog();
 }
//----------------------------------------------------
private: System::Void saveFileDialog1_FileOk(
       System::Object^  sender,
       System::ComponentModel::CancelEventArgs^  e) {
 //Zapis do pliku
 //Procedura obsługuje zdarzenie:
 //kliknięcie przycisku OK w oknie dialogowym
 //saveFileDialog1

     try
     {
       StreamWriter^ sw;
       sw = System::IO::File::CreateText(
            saveFileDialog1->FileName);
       sw->Write(txtOkno->Text);
       sw->Close();
     }
     catch (Exception^ e)
     {
       MessageBox::Show(e->Message);
     }
 }
//----------------------------------------------------
private: System::Void mnuKopiuj_Click(System::Object^
       sender, System::EventArgs^  e) {
 //Kopiowanie do schowka

     if (txtOkno->SelectedText != ""){
       Clipboard::Clear();
       Clipboard::SetText(txtOkno->SelectedText);
     }
     else{
       MessageBox::Show("Proszę zaznaczyć tekst",
                        "Notatnik1");
     }
 }
//---------------------------------------------------
```

```
private: System::Void mnuWklej_Click(System::Object^
       sender, System::EventArgs^  e) {
 //Wkleja tekst ze schowka
     txtOkno->SelectedText=Clipboard::GetText();
 }
//---------------------------------------------------
private: System::Void mnuWytnij_Click(System::Object^
         sender, System::EventArgs^  e) {
 //Wycinanie zaznaczonego tekstu do schowka

     if (txtOkno->SelectedText != ""){
          Clipboard::Clear();
          Clipboard::SetText(txtOkno->SelectedText);
          txtOkno->SelectedText="";
     }
 }
//---------------------------------------------------
private: System::Void mnuZaznaczWszystko_Click(
    System::Object^  sender, System::EventArgs^  e) {
//Zaznacza całą zawartość okna
     txtOkno->SelectAll();
 }
//---------------------------------------------------
private: System::Void mnuInformacja _Click(
    System::Object^  sender, System::EventArgs^  e) {
     MessageBox::Show("Przykładowa aplikacja.",
          "Notatnik - LTK",
          MessageBoxButtons::OK,
          MessageBoxIcon::Information);
 }
//---------------------------------------------------
private: System::Void mnuZakoncz_Click(
    System::Object^  sender, System::EventArgs^  e) {
//Zamykanie aplikacji

     if (MessageBox::Show(
          "Czy rzeczywiście zakończyć aplikację?",
          "Zamknięcie aplikacji",
          MessageBoxButtons::YesNo,
          MessageBoxIcon::Question) ==
          System::Windows::Forms::DialogResult::Yes)
           {
                 Application::Exit();
           }
 }
//---------------------------------------------------
```

# 11.3. Dźwięk i pliki muzyczne

Generowanie dźwięku i praca z plikami muzycznymi została opisana np. w pracy [14]

## Beep

Dla wywołania dźwięku systemowego wystarcza przestrzeń nazw formularza. W aplikacji dla Windows zawsze istnieje wiersz:

```
using namespace System::Windows::Forms;
```

Napisanie linii

```
SystemSounds::Beep->Play();
```

wywołuje dźwięk systemowy.

## Pliki muzyczne

Do odtwarzania dźwięków można wykorzystać obiekt klasy SoundPlayer. Wchodzi ona w skład przestrzeni nazw System::Media . W bloku określania przestrzeni nazw należy dopisać linię:

```
using namespace System::Media;
```

Następnie tworzymy obiekt klasy SoundPlayer np.:

```
SoundPlayer^ dzwiek = gcnew;
```

W kodzie należy zdefiniować ścieżkę dostępu do pliku muzycznego. Metody Play() i PlayLooping() pozwalają na jednorazowe lub wielokrotne odtwarzanie pliku z muzyką.

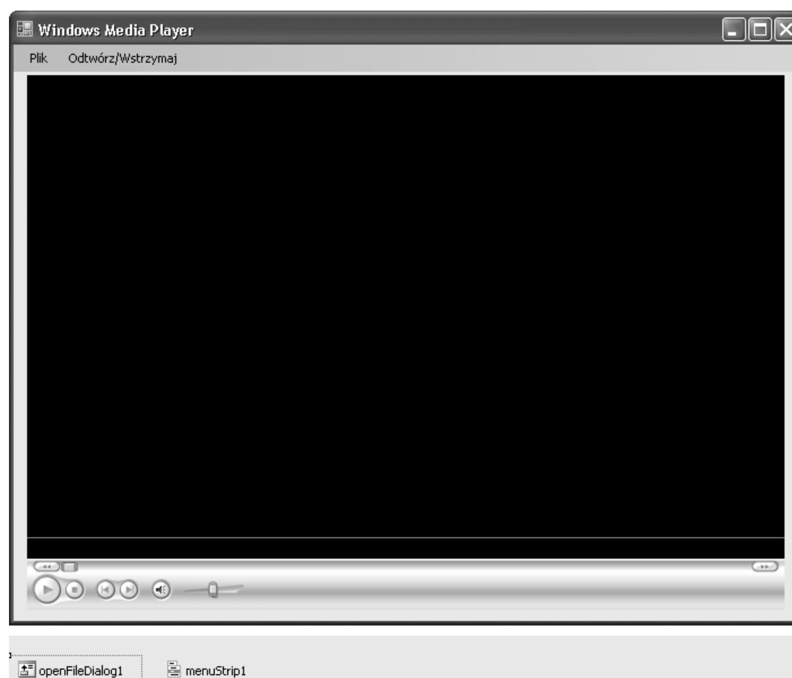
Rysunek 11.9. Aplikacja generująca dźwięki i muzykę

**Kod aplikacji**

```
//-------------------------------------------------
    using namespace System::Media;
//-------------------------------------------------
private: System::Void btn_Beep_Click(System::Object^
                     sender, System::EventArgs^  e) {
    SystemSounds::Beep->Play();
 }
//-------------------------------------------------
private: System::Void btn_wav_Click(System::Object^
                     sender, System::EventArgs^  e) {
    SoundPlayer^ dzwiek = gcnew
          SoundPlayer("C:/tkm/gr1_1/CLIKBSE.wav");
    dzwiek->Play();
 }
//-------------------------------------------------
private: System::Void btn_wav1_Click(System::Object^
                    sender, System::EventArgs^  e) {
    SoundPlayer^ dzwiek = gcnew
          SoundPlayer("C:/tkm/gr1_1/RESTSE.wav");
    dzwiek->PlayLooping();
 }
//-------------------------------------------------
```

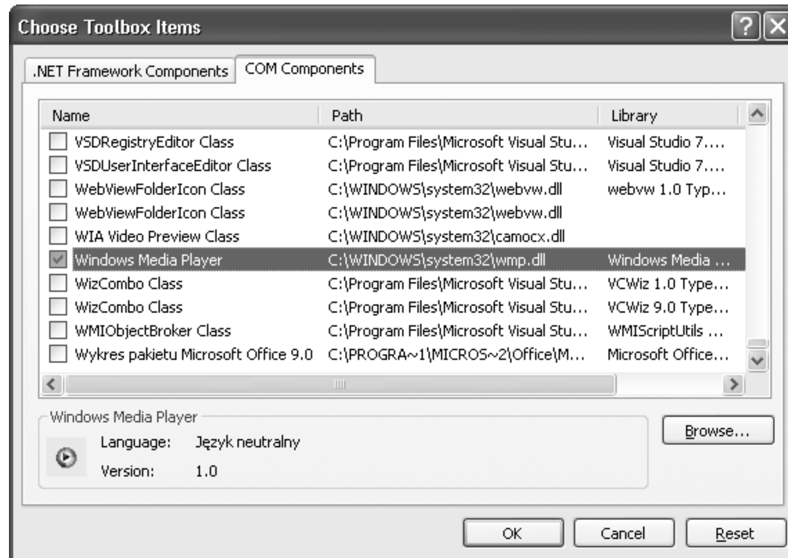
## Windows Media Player

Zbudujmy własny formularz z menu, poprzez które wybierać będziemy pliki do odtwarzania w obiekcie  Windows Media Player, znajdujący się także na formularza, rysunek 11.10.



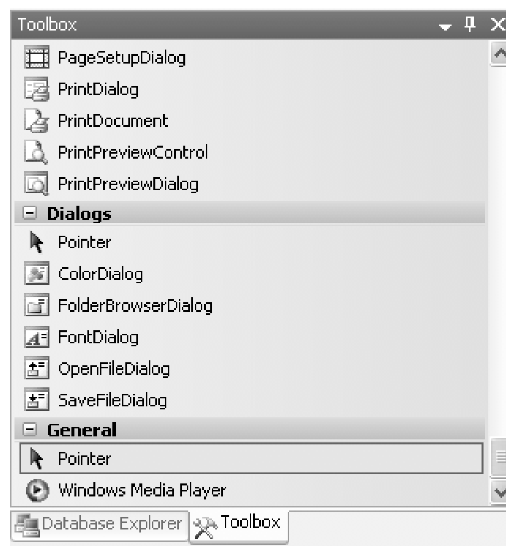

Rysunek 11.10. Aplikacja

Kontrolka potrzebna do utworzenia formantu Windows Media Player nie jest bezpośrednio dostępna w oknie Toolbox. Aby ją zainstalować należy kliknąć prawym klawiszem myszy w dowolne miejsce obszaru Toolbox, a następnie wybrać pozycję **Choose Items...** Pojawi się okno *Choose Toolbox Items*, w którym w zakładce *COM Components* zaznaczamy pozycję *Windows Media Player* i naciskamy przycisk OK.




Rysunek 11.11. Okno Choose Toolbox Items

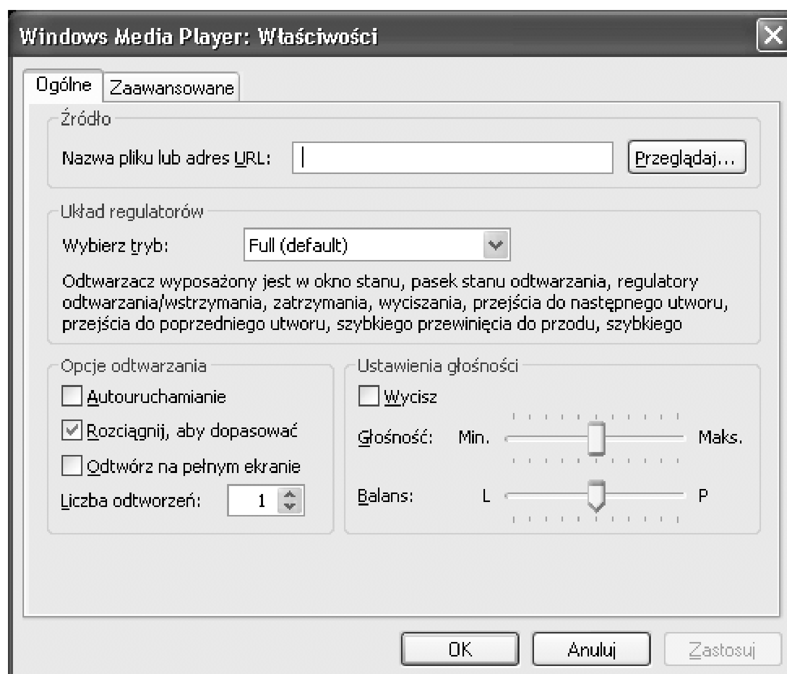
Pozwoli to wstawić do sekcji *General* Toolbox'u kontrolkę *Windows Media Player* (rysunek 11.12).




Rysunek 11.12. Toolbox z dodaną kontrolką Windows Media Player

W aplikacji użyto dodatkowo formantów *menuStrip* dla sterowania odtwarzaniem plików z menu oraz *openFileDialog* dla wygodnego poszukiwania wybranego pliku muzycznego lub filmowego na dysku.

Właściwości formantu *Windows Media Player* ustawiane są w specjalnym oknie właściwości (rysunki 11.13 i 11.14). Mamy do niego dostęp po kliknięciu prawym klawiszem myszy na formancie i wybraniu z menu kontekstowego pozycji *Properties* .




Rysunek 11.13. Okno właściwości obiektu Windows media Player zakładka Ogólne

Rysunek 11.14. Okno właściwości obiektu Windows media Player zakładka Zaawansowane

W menu rozwijalnym oprogramowano cztery akcje:

- Wybór pliku do odtwarzania (*otwórzToolStripMenuItem*),
- Zakończenie pracy (*zakończToolStripMenuItem*),
- Uruchomienie odtwarzania (*odtwórzToolStripMenuItem1*), użyto metody *play()* właściwości *CtlControls*,
- Wstrzymanie odtwarzania (*wstrzymajToolStripMenuItem*), użyto metody *stop()* właściwości *CtlControls*.

### Kod aplikacji

```
//-------------------------------------------------
private: System::Void otwórzToolStripMenuItem_Click(
System::Object^     sender, System::EventArgs^  e) {
     if(openFileDialog1->ShowDialog() ==
                Windows::Forms::DialogResult::OK) {
       axWindowsMediaPlayer1->URL =
                openFileDialog1->FileName->Trim();
     }
 }
//-------------------------------------------------
private: System::Void odtwórzToolStripMenuItem1_Click(
   System::Object^  sender, System::EventArgs^  e) {
    axWindowsMediaPlayer1->Ctlcontrols->play();
 }
//-------------------------------------------------
private: System::Void wstrzymajToolStripMenuItem_Click(
  System::Object^  sender, System::EventArgs^  e) {
    axWindowsMediaPlayer1->Ctlcontrols->stop();
 }
//--------------------------------------------------
private: System::Void zakończToolStripMenuItem_Click(
  System::Object^  sender, System::EventArgs^  e) {
     Application::Exit();
 }
//--------------------------------------------------
```

W aplikacji wybrano właściwość kontrolki Windows Media Player Wybierz tryb – Full.

Powoduje to uwidocznienie na formancie wskaźnika zaawansowania odtwarzania, przycisków start i stop, suwaka do regulacji głośności, przycisku do szybkiego przewijania (rysunek 11.10). Możliwe jest zrezygnowanie z tych narzędzi i pozostawienie w aplikacji jedynie okna odtwarzającego filmy i pliki muzyczne.

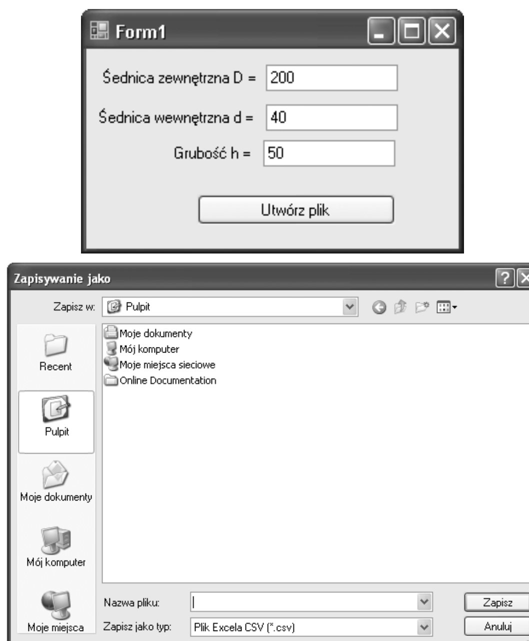
# 11.4. Współpraca z systemem CAD/CAE

W rozdziale przedstawiono przykład organizacji współpracy programu napisanego w języku C++ z komercyjnym systemem CAD/CAE. Zwykle celem połączenia programu komputerowego z systemem CAD/CAE jest zapewnienie możliwości bezpośredniego przesyłania wyników uzyskanych w programie obliczeniowym do systemu CAD/CAE, tak aby przesłane dane decydowały o postaci – wymiarach, strukturze określonego modelu geometrycznego zbudowanego w systemie CAD/CAE. Model geometryczny zbudowany w systemie CAD/CAE jest wówczas modelem sparametryzowanym, którego postać zależy od wprowadzonych wartości liczbowych poszczególnych parametrów. Parametry te mogą być wprowadzane bezpośrednio w systemie CAD/CAE. Mogą również być danymi pobieranymi z innego oprogramowania np. programów obliczeniowych, itp. Ta ostatnia możliwość została wykorzystana w przykładzie.

Przykład pokazuje w jaki sposób można zrealizować współpracę programu napisanego w języku C++ z systemem CATIA. Do realizacji tej współpracy wykorzystano możliwość przesyłania parametrów modeli geometrycznych w systemie CATIA za pośrednictwem aplikacji MS Excel. Przykładowy program w języku C++ zapisuje parametry we wskazanym pliku aplikacji MS Excel. Następnie zapisany plik (plik z rozszerzeniem .csv) jest przez użytkownika konwertowany za pomocą aplikacji MS Excel na plik z rozszerzeniem .xls. Plik z rozszerzeniem .xls jest czytany prze system CATIA i zawarte w nim parametry zmieniają postać modelu geometrycznego zbudowanego w systemie CATIA.

W przykładzie skoncentrowano się na stronie narzędziowej. W związku z tym prezentowany przykład jest stosunkowo prosty. Sparametryzowanym modelem jest model geometryczny 3D tulei o następujących parametrach: D – średnica zewnętrzna tulei, d – średnica wewnętrzna tulei, h – wysokość tulei. Program napisany w języku C++ ma za zadanie wczytać, za pośrednictwem interfejsu okienkowego, odpowiednie parametry i następnie zapisać je we wskazanym pliku z rozszerzeniem .csv. Następnie użytkownik dokonuje przekonwertowania pliku z rozszerzeniem .csv na plik z rozszerzeniem .xls. Plik z rozszerzeniem .xls po wczytaniu do systemu CATIA powoduje nadanie nowo-wczytanych

wartości parametrów modelowi geometrycznemu zbudowanemu w tym systemie.

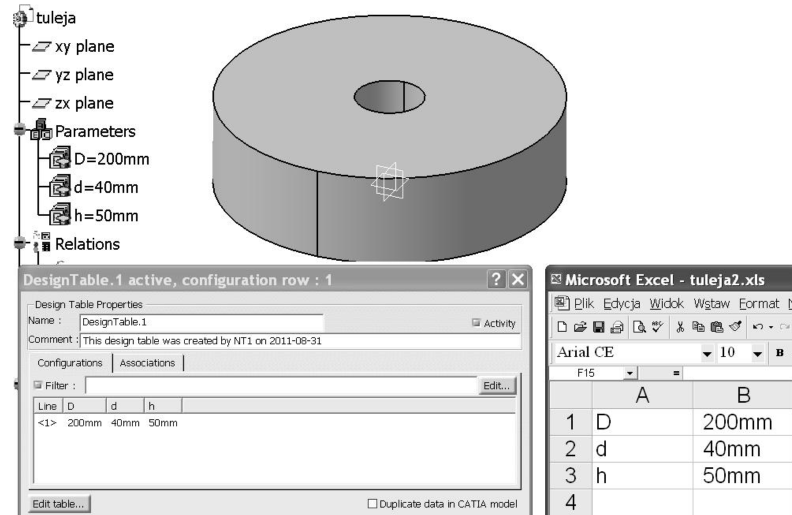



Rysunek 11.15. Interfejs graficzny programu w języku C++ oraz okno zapisu pliku

**Kod aplikacji**

```
#pragma once


namespace Cpp_Excel_csv {

    using namespace System;
    using namespace System::ComponentModel;
    using namespace System::Collections;
    using namespace System::Windows::Forms;
    using namespace System::Data;
    using namespace System::Drawing;
    //
    //Tu dopisujemy:
    using namespace System::IO;

...
```

```
...

#pragma endregion
private: System::Void btnUtworz_Click(System::Object^
            sender, System::EventArgs^  e) {
 //Otwarcie okna dialogowego saveFileDialog1
      saveFileDialog1->InitialDirectory =
            "c:\\przyklad";
      saveFileDialog1->Filter =
            "Plik Excela CSV (*.csv)|*.csv";
      saveFileDialog1->ShowDialog();
      }
//------------------------------------------
private: System::Void
      saveFileDialog1_FileOk(System::Object^  sender,
      System::ComponentModel::CancelEventArgs^  e){
 //Procedura obsługuje zdarzenie:
 //kliknięcie przycisku OK w oknie dialogowym
 //saveFileDialog1
      try
      {
        StreamWriter^ sw;
        sw = System::IO::File::CreateText
                        (saveFileDialog1->FileName);
        sw->Write(     "D; ");
         sw->Write(txtSrednica_zew->Text );
         sw->Write("mm \n");
        sw->Write(     "d; ");
        sw->Write(txtSrednica_wew->Text );
        sw->Write(     "mm \n");
        sw->Write(     "h; ");
        sw->Write(txtGrubosc->Text );
        sw->Write(     "mm \n");
        sw->Close();
      }
      catch (Exception^ e)
      {
       MessageBox::Show(e->Message);
      }
 }
};

}
```




Rysunek 11.16. Model geometryczny tulei wygenerowany w systemie CATIA. Na rysunku widoczny jest także arkusz z parametrami zapisany w pliku z rozszerzeniem .xls oraz parametry wczytane do systemu CATIA jako Design Table1

Zaprezentowany przykład jest jednym z możliwych, bardzo często stosowanych, rozwiązań.

Program napisany w języku C++ może zawierać elementy obliczeń inżynierskich, może również współpracować z innym oprogramowaniem np. bazami danych czy też innymi systemami obliczeniowymi.

# 12 Podsumowanie

W ramach części I niniejszego opracowania przedstawiono zasadnicze składniki języka C i jego wersji obiektowej C++.

Przyjęty układ książki: najpierw prezentacja języka C, potem języka C++, traktowanego jako pewne rozszerzenie, wynikał z wieloletnich doświadczeń w nauczaniu tych zagadnień przez jednego z autorów (Jerzego Pokojskiego). W tym ujęciu klasy, obiektowość to rozszerzenie struktur i opierających się na nich struktur danych. W praktyce układ ten jest przystępniejszy dla studenta i sprawia, że dosyć złożone zagadnienia programowania obiektowego poznaje się stopniowo.

W książce zwracano uwagę na silne wyeksponowanie podstaw, których zadaniem jest zapewnienie niezbędnych umiejętności warsztatowych. W większości przypadków zaawansowane konstrukcje budowane w języku C++ wymagają dobrego przygotowania właśnie w tym zakresie. Prezentacja zaawansowanych zagadnień ze słabą znajomością podstaw na ogół jest niecelowa. Prowadzi do małej elastyczności i przyswajania pewnych „cudownych rozwiązań czy też schematów".

Równolegle do prezentacji zagadnień podstawowych dla C/C++ pokazywano w jaki sposób można je zaimplementować w środowisku MS Windows w formie aplikacji okienkowych lub konsolowych. Zdaniem autorów jest to dosyć istotne aby umieć posługiwać się jednym z najbardziej zaawansowanych i stosunkowo trudnych, środowisk programistycznych.

Część II stanowi uzupełnienie zarówno treści merytorycznych jak i narzędziowych części I.

Siłą rzeczy rozmiary tego opracowania są ograniczone i jest ono właściwie pewnym autorskim, opartym na praktyce, wprowadzeniem do języka C/C++.

Autorzy oczekują, że korzystający z tej książki będą posługiwali się również inną literaturą. Poniżej zamieszczono uwagi na temat literatury wymienionej w spisie znajdującym się na końcu opracowania.

We wstępie zaprezentowano następującą kategoryzację literatury do nauki języków C/C++:

1. książki opracowane przez twórców języka C/C++.
2. książki wprowadzające do określonych zagadnień lub do określonych klas zastosowań.
3. książki typu poradnikowego.

W grupie pierwszej można wymienić pozycje [8] i [13]. Mają one kluczowe znaczenie dla rozumienia zasadniczych idei i koncepcji języka C/C++. Ich dużą zaletą jest syntetyczność ujęcia. Znajomość zawartych w nich treści pozwala na bardzo klarowne widzenie dostępnych struktur programistycznych.

Książki o charakterze wprowadzającym, ujmujące tematykę w sposób bardzo szeroki to pozycje [5] i [6]. Są one dosyć obszerne, opierają się jednak na stosunkowo niedużej liczbie bardzo czytelnych przykładów.

Innym, można powiedzieć bardzo specjalizowanym opracowaniem, wynikającym z zawodowych dokonań autora jest trzytomowa pozycja [4]. Zawiera ona bardzo szerokie spektrum możliwości języka ukazane w przystępnej formie.

Prace [1, 2, 10, 11] są przykładami stosunkowo licznej grupy pozycji wprowadzających o ogólnym i jednocześnie podstawowym charakterze.

Do grupy opracowań specjalizowanych można zaliczyć również pozycje [3, 7] - struktury danych i specyficzne spojrzenie. Pozycja [14] jest skoncentrowana głównie na zaawansowanych typach aplikacji windowsowych.

Pewne cechy poradnika ma opracowanie [12].

Powyższe uwagi można potraktować również jako komentarze do klas opracowań w zakresie języka C/C++ dostępnych na rynku.

# 13 Literatura

1. Barteczko K., Praktyczne wprowadzenie programowania obiektowego w języku C++, Lupus, 1993.
2. Delannoy C.:,Ćwiczenia z języka C, WNT, 1993.
3. Drozdek A., Simow D.L., Struktury danych w języku C, WNT, 1996.
4. Grębosz J., Programowanie w języku C++ orientowane obiektowo, Tom I-III, Oficyna Kallimach, Kraków, 1996.
5. Johnsonbaugh R., Kalin M.: Applications Programming in C. Macmillan Publishing Company, 1989.
6. Johnsonbaugh R., Kalin M.: Applications Programming in C++. Prentice Hall, 1999.
7. Jones R., Stewart I.: Sztuka programowania w języku C. WNT, 1992.
8. Kernighan B.W., Ritchie D.M.: Język C. WNT, 1987.
9. Kompilator MS C++ Express Edition z SP1 . http://www.microsoft.com/visualstudio/en-us/products/2008-editions/express , (data skorzystania: 08. 2011).
10. Liberty J.: Poznaj C++ w 24 godziny. Intersoftland, 1999.
11. Morrisom J.: C++ dla programistów języka Visual Basic. MIKOM 2001.
12. Soulie J.: C++ Language Tutorial. Available online at: http://www.cplusplus.com/doc/tutorial/ (data skorzystania: 05.20011).
13. Stroustrup B., Programowanie. Teoria i praktyka z wykorzystaniem C++, Helion 2010.
14. Wileczek R.: Microsoft Visual C++ 2008 – Tworzenie aplikacji dla Windows. Helion, 2009.