Programowanie obiektowe

dr inż. Sławomir Koczubiej

Politechnika Świętokrzyska
Wydział Zarządzania i Modelowania Komputerowego
Katedra Technologii Informatycznych

(5 października 2025)

Kontakt

Budynek C, pokój 3.26
sk@tu.kielce.pl

Materiały do pobrania, aktualności, terminy zaliczeń

http://staff.tu.kielce.pl/sk

Organizacja wykładów

- Wykłady są nieobowiązkowe (zgodnie z postanowieniami regulaminu), ale...
- Na wykłady warto zaglądać.
- Czasem sprawdzam listę obecności.

Warunki zaliczenia wykładu

Egzamin zaliczeniowy po zakończeniu wykładów.

Organizacja laboratoriów

- Zajęcia laboratoryjne są obowiązkowe.
- Dopuszcza się jedną nieobecność.
- Większa liczba nieobecności powoduje zmniejszenie oceny do niedostatecznej włącznie (3 lub więcej nieobecności).
- W przypadku usprawiedliwionej nieobecności zajęcia można odrobić z inną grupą (jeśli istnieje taka możliwość).

Warunki zaliczenia laboratoriów

Wykonanie ćwiczeń i zaliczenie sprawdzianów kontrolnych.

Tematyka wykładów

- Wprowadzenie do programowania w języku C++. Obiekty i klasy.
- Ochrona i kapsułkowanie. Interfejsy.
- Dziedziczenie, dziedziczenie wielobazowe. Polimorfizm.
- Obiekty i zarządzanie pamięcią. Tworzenie i niszczenie obiektów.
- Operatory przeciążone. Funkcje operatorowe.
- Strumienie i obsługa plików. Struktury i rekordy.
- Wyjątki i ich obsługa.
- Szablony.
- Biblioteka STL.

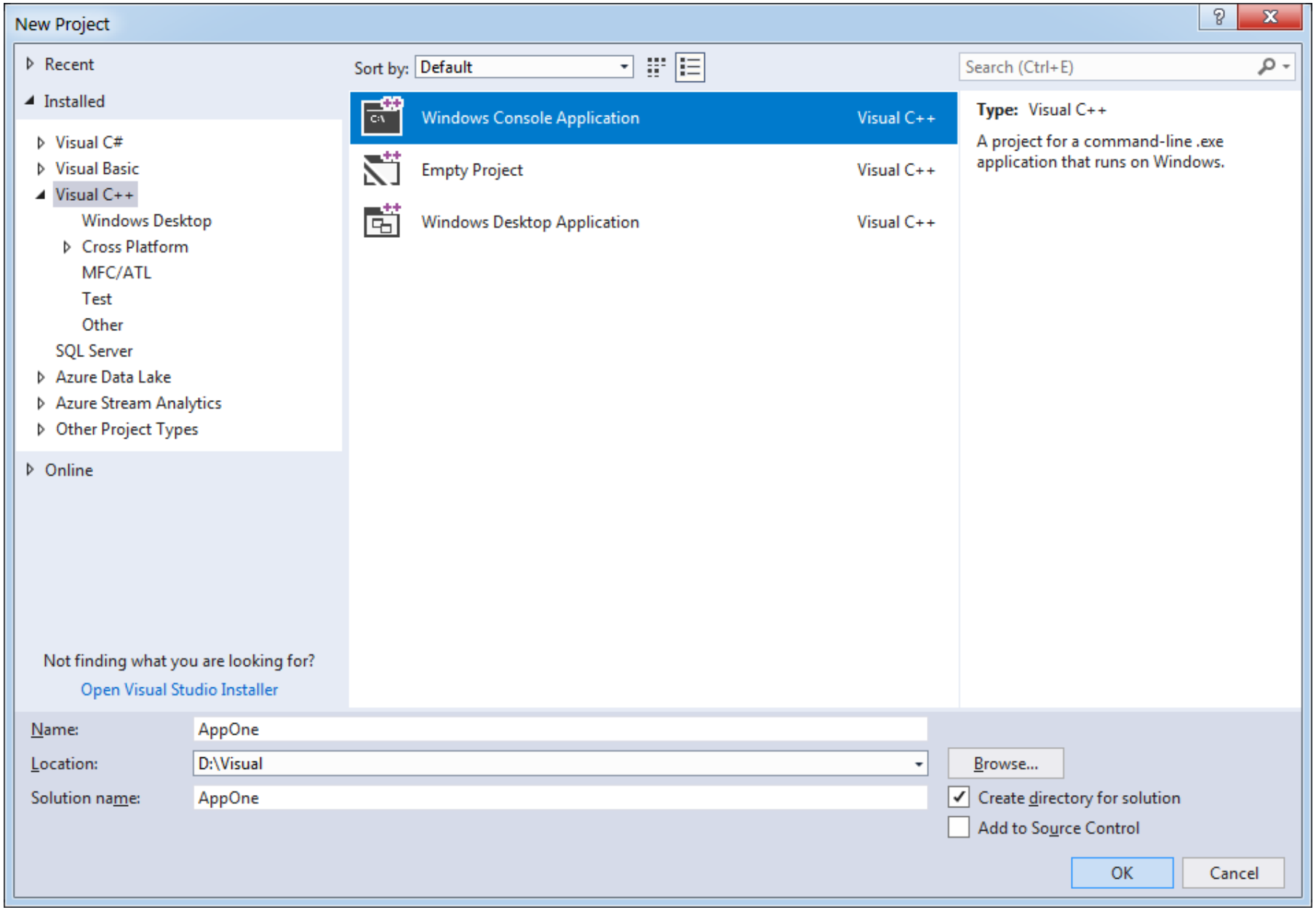
Tematyka laboratoriów

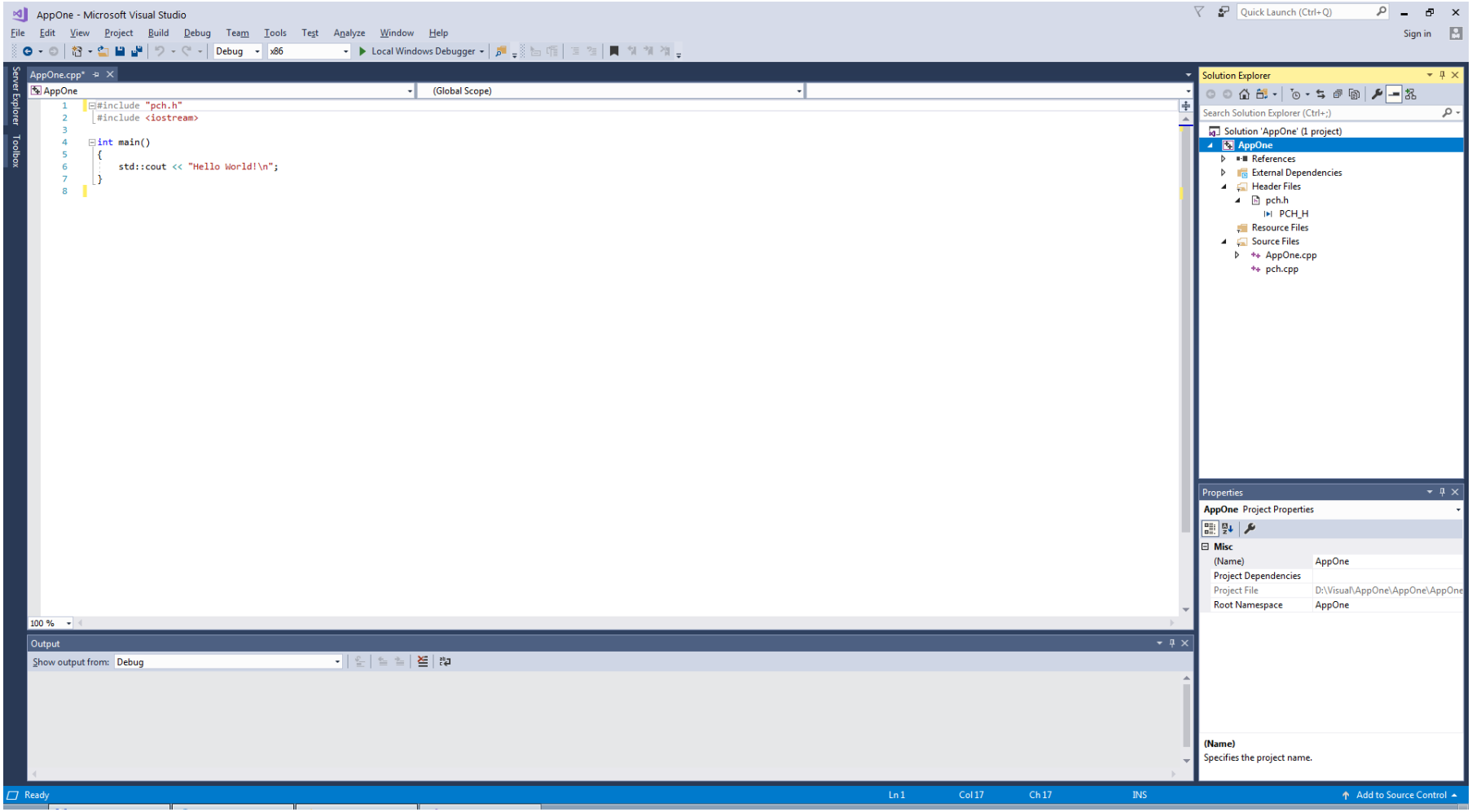
- Struktura programu w języku C++.
- Definiowane klas. Składowe klasy, obiekty.
- Dziedziczenie i dziedziczenie wielobazowe.
- Polimorfizm i tablice wskaźników.
- Tworzenie i niszczenie obiektów. Konstruktor, destruktor i zarządzanie pamięcią.
- Redefiniowanie operatorów.
- Obsługa błędów.
- Obsługa plików.
- Kontenery i iteratory.

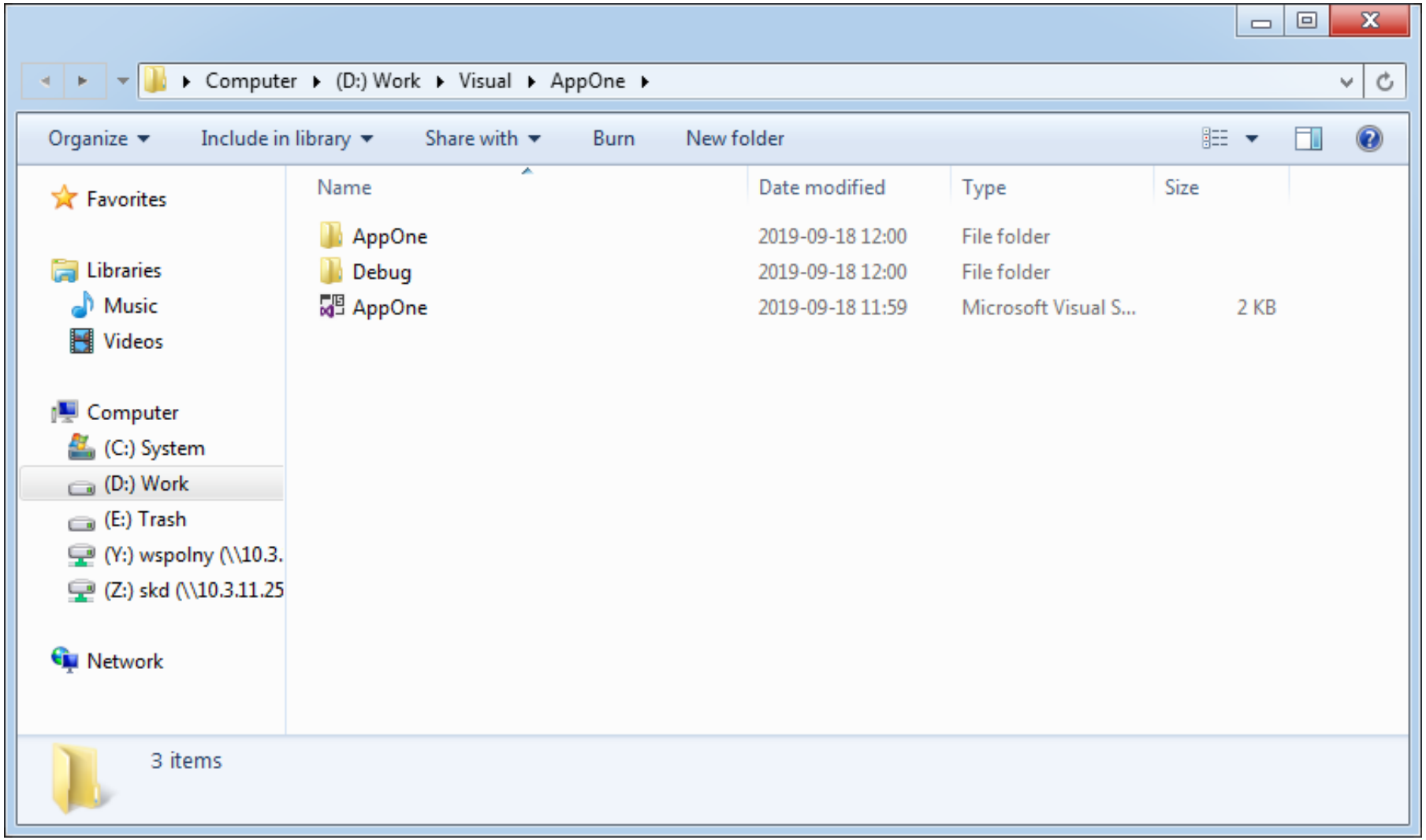
Literatura

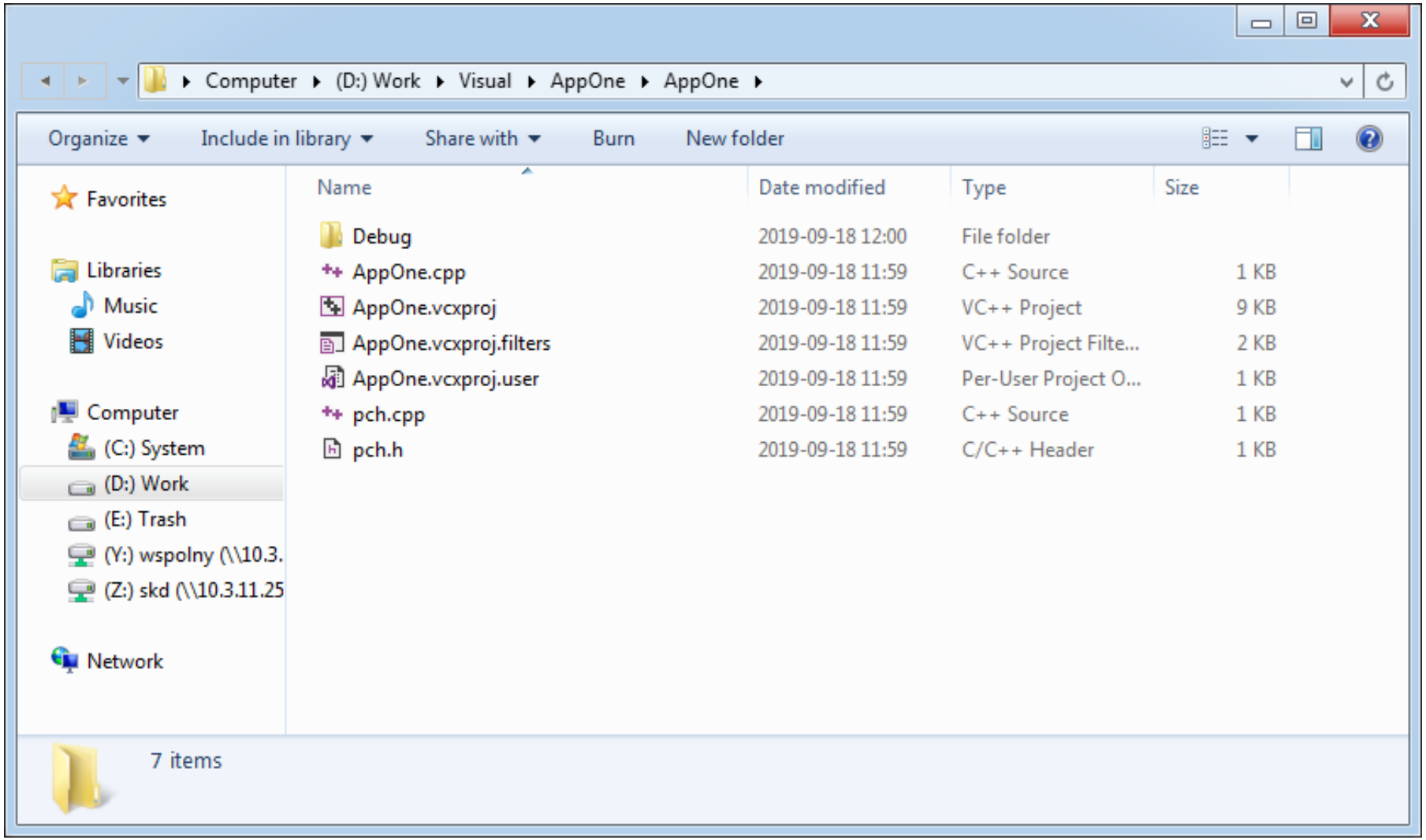
- Drozdek A. *C++. Algorytmy i struktury danych.* Wydawnictwo Helion, Gliwice 2004.
- Eckel B. *Thinking in C++.* Wydawnictwo Helion, Gliwice 2002.
- Gręboszj. *Symfonia C++ Standard.* Wydawnictwo Edition 2000, Kraków 2009.
- Prata S. *Szkoła programowania. Język C++.* Wydawnictwo Helion, Gliwice 2006.

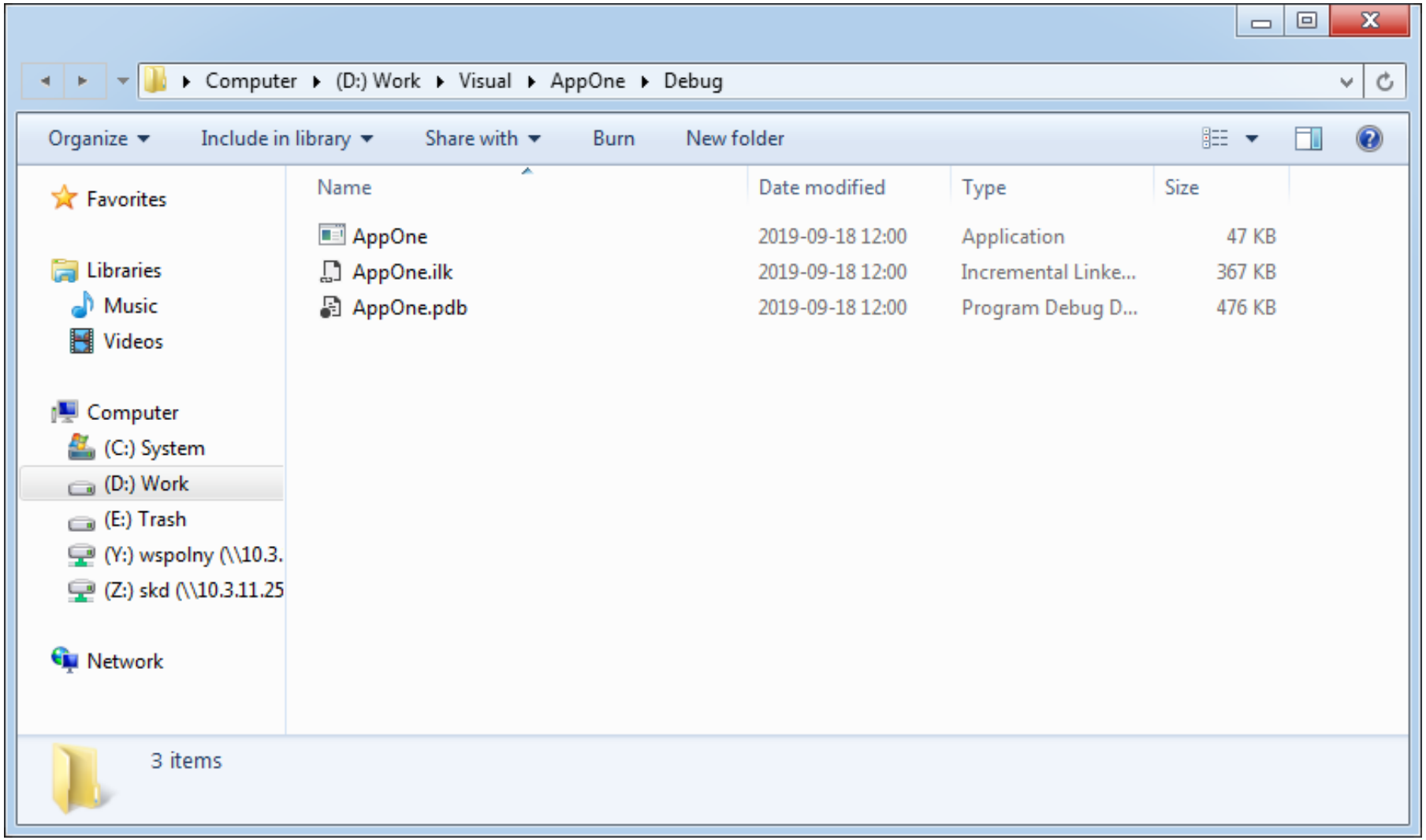
Środowisko programowania Visual Studio - Windows




AppOne - Microsoft Visual Studio
File Edit View Project Build Debug Team Tools Test Analyze Window Help
Debug x86 Local Windows Debugger
Quick Launch (Ctrl+Q)
Sign in
AppOne.cpp*
AppOne
(Global Scope)
#include "pch.h"
#include <iostream>
int main()
{
std::cout << "Hello World!\n";
}
100 %
Output
Show output from: Debug
Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'AppOne' (1 project)
AppOne
References
External Dependencies
Header Files
pch.h
PCH_H
Resource Files
Source Files
AppOne.cpp
pch.cpp
Properties
AppOne Project Properties
Misc
(Name) AppOne
Project Dependencies
Project File D:\Visual\AppOne\AppOne\AppOne
Root Namespace AppOne
(Name)
Specifies the project name.
Ready
Ln 1 Col 17 Ch 17 INS
Add to Source Control

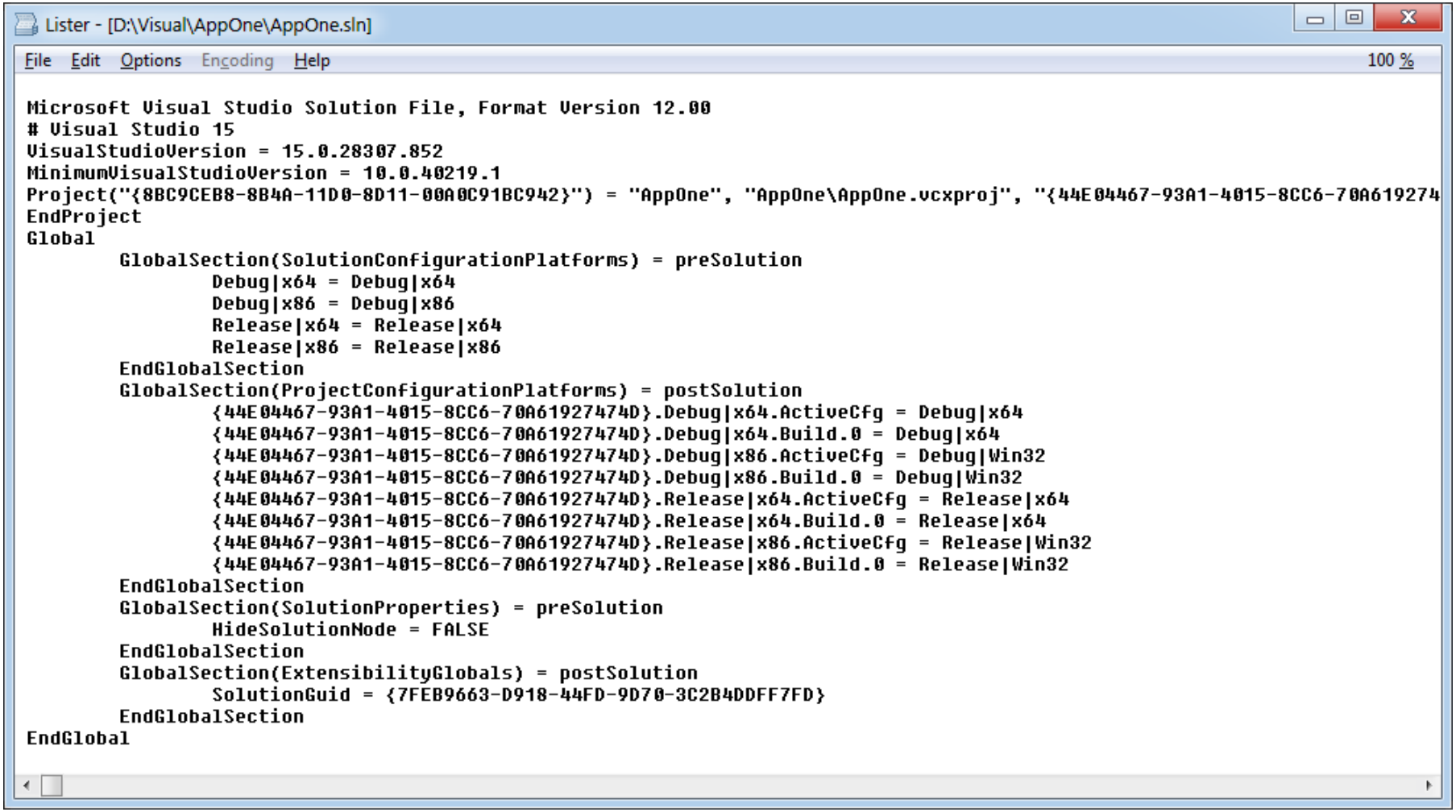

Computer ▸ (D:) Work ▸ Visual ▸ AppOne ▸
Organize
Include in library
Share with
Burn
New folder
Favorites
Libraries
Music
Videos
Computer
(C:) System
(D:) Work
(E:) Trash
(Y:) wspolny (\\10.3.
(Z:) skd (\\10.3.11.25
Network
Name
Date modified
Type
Size
AppOne 2019-09-18 12:00 File folder
Debug 2019-09-18 12:00 File folder
AppOne 2019-09-18 11:59 Microsoft Visual S... 2 KB
3 items


Computer ▸ (D:) Work ▸ Visual ▸ AppOne ▸ AppOne ▸
Organize
Include in library
Share with
Burn
New folder
Favorites
Libraries
Music
Videos
Computer
(C:) System
(D:) Work
(E:) Trash
(Y:) wspolny (\\10.3.
(Z:) skd (\\10.3.11.25
Network
Name
Date modified
Type
Size
Debug 2019-09-18 12:00 File folder
AppOne.cpp 2019-09-18 11:59 C++ Source 1 KB
AppOne.vcxproj 2019-09-18 11:59 VC++ Project 9 KB
AppOne.vcxproj.filters 2019-09-18 11:59 VC++ Project Filte... 2 KB
AppOne.vcxproj.user 2019-09-18 11:59 Per-User Project O... 1 KB
pch.cpp 2019-09-18 11:59 C++ Source 1 KB
pch.h 2019-09-18 11:59 C/C++ Header 1 KB
7 items


Computer ▸ (D:) Work ▸ Visual ▸ AppOne ▸ Debug
Organize
Include in library
Share with
Burn
New folder
Favorites
Libraries
Music
Videos
Computer
(C:) System
(D:) Work
(E:) Trash
(Y:) wspolny (\\10.3.
(Z:) skd (\\10.3.11.25
Network
Name
Date modified
Type
Size
AppOne
2019-09-18 12:00
Application
47 KB
AppOne.ilk
2019-09-18 12:00
Incremental Linke...
367 KB
AppOne.pdb
2019-09-18 12:00
Program Debug D...
476 KB
3 items

Struktura domyślnego projektu

Plik `*.sln` – plik *rozwiązania*, przechowuje globalne informacje o rozwiązaniu (rodzaj kontenera projektów). Rozwiązanie może zawierać projekty wykorzystujące rożne technologie i języki programowania.



Plik `*.vcxproj` – plik *projektu*, przechowuje informacje specyficzne dla każdego projektu.

Lister - [D:\Visual\AppOne\AppOne\AppOne.vcxproj]

File Edit Options Encoding Help 18 %

```
<?xml version="1.0" encoding="utf-8"?>
<Project DefaultTargets="Build" ToolsVersion="15.0"
xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <ItemGroup Label="ProjectConfigurations">
    <ProjectConfiguration Include="Debug|Win32">
      <Configuration>Debug</Configuration>
      <Platform>Win32</Platform>
    </ProjectConfiguration>
    <ProjectConfiguration Include="Release|Win32">
      <Configuration>Release</Configuration>
      <Platform>Win32</Platform>
    </ProjectConfiguration>
    <ProjectConfiguration Include="Debug|x64">
      <Configuration>Debug</Configuration>
      <Platform>x64</Platform>
    </ProjectConfiguration>
    <ProjectConfiguration Include="Release|x64">
      <Configuration>Release</Configuration>
      <Platform>x64</Platform>
    </ProjectConfiguration>
  </ItemGroup>
  <PropertyGroup Label="Globals">
    <VCProjectVersion>15.0</VCProjectVersion>
    <ProjectGuid>{44E04467-93A1-4015-8CC6-70A61927474D}</ProjectGuid>
    <Keyword>Win32Proj</Keyword>
    <RootNamespace>AppOne</RootNamespace>
    <WindowsTargetPlatformVersion>10.0.17763.0</WindowsTargetPlatformVersion>
  </PropertyGroup>
  <Import Project="$(VCTargetsPath)\Microsoft.Cpp.Default.props" />
  <PropertyGroup Condition="'$(Configuration)|$(Platform)'=='Debug|Win32'"
Label="Configuration">
    <ConfigurationType>Application</ConfigurationType>
    <UseDebugLibraries>true</UseDebugLibraries>
```

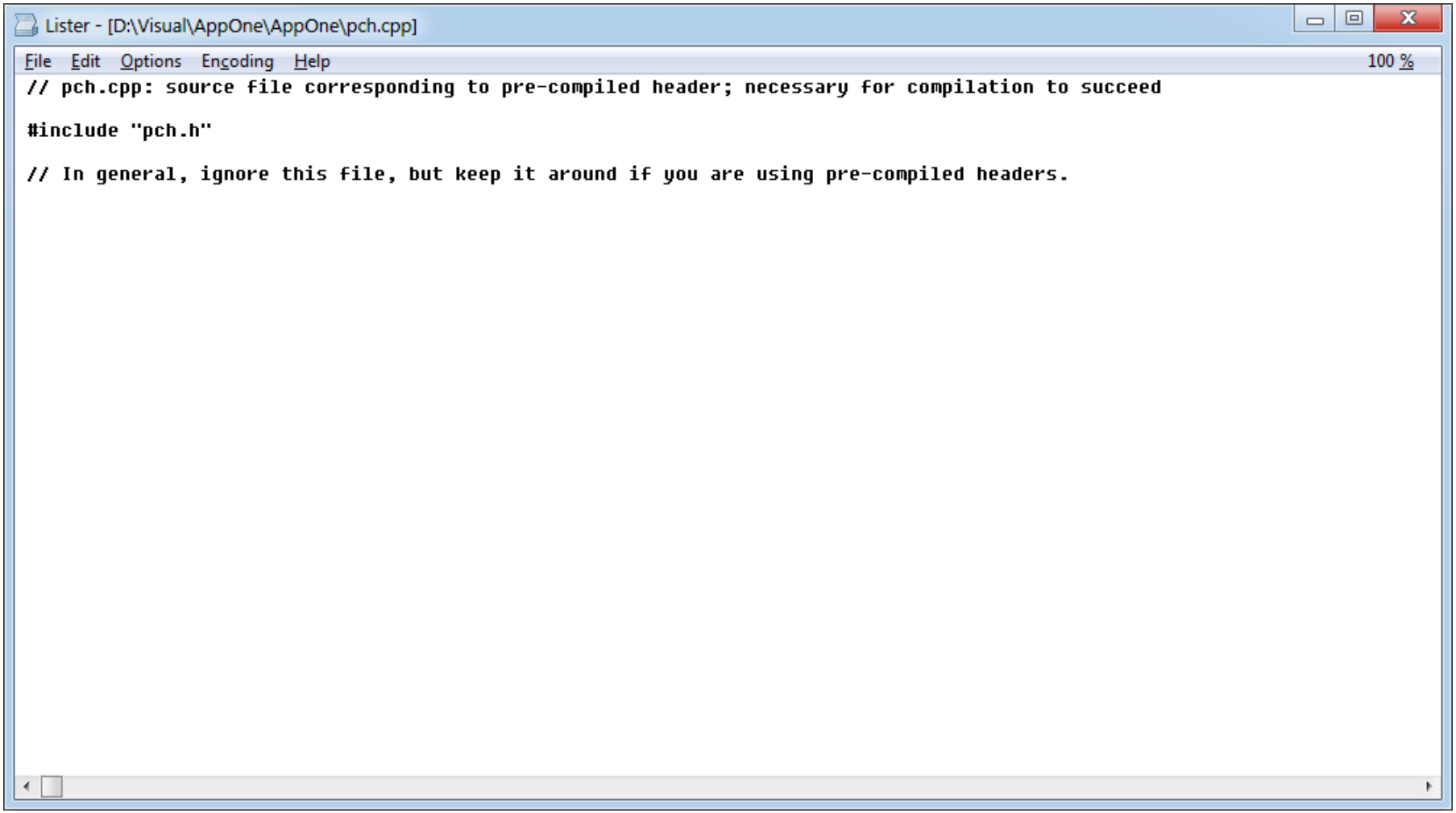
Plik `*.vcxproj.user` – plik *filtrów*, określa miejsce umieszczenia pliku, który jest dodawany do rozwiązania.

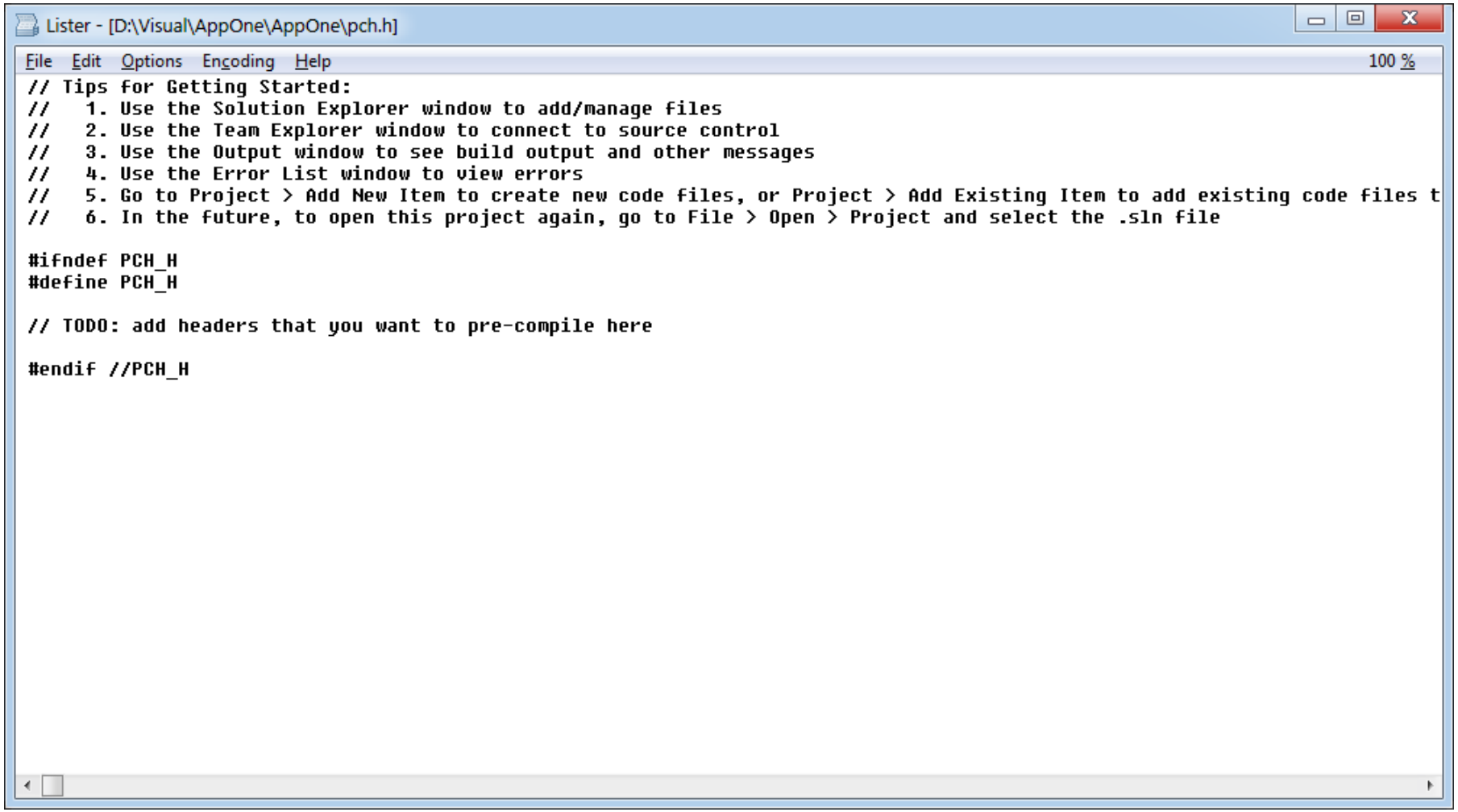
Lister - [D:\Visual\AppOne\AppOne\AppOne.vcxproj.filters]

File Edit Options Encoding Help 100 %

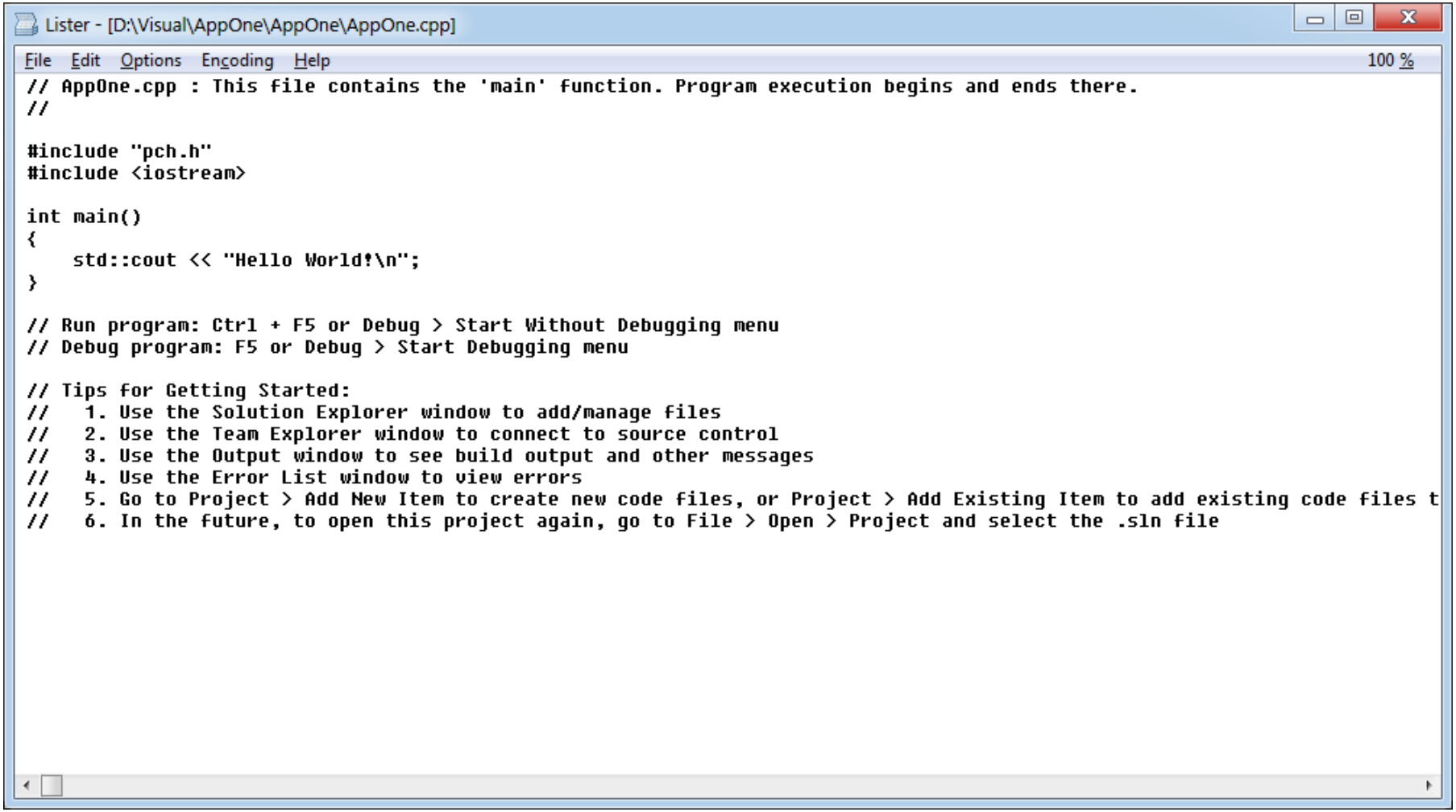
```
<?xml version="1.0" encoding="utf-8"?>
<Project ToolsVersion="4.0" xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <ItemGroup>
    <Filter Include="Source Files">
      <UniqueIdentifier>{4FC737F1-C7A5-4376-A066-2A32D752A2FF}</UniqueIdentifier>
      <Extensions>cpp;c;cc;cxx;def;odl;idl;hpj;bat;asm;asmx</Extensions>
    </Filter>
    <Filter Include="Header Files">
      <UniqueIdentifier>{93995380-89BD-4b04-88EB-625FBE52EBFB}</UniqueIdentifier>
      <Extensions>h;hh;hpp;hxx;hm;inl;inc;ipp;xsd</Extensions>
    </Filter>
    <Filter Include="Resource Files">
      <UniqueIdentifier>{67DA6AB6-F800-4c08-8B7A-83BB121AAD01}</UniqueIdentifier>
      <Extensions>rc;ico;cur;bmp;dlg;rc2;rct;bin;rgs;gif;jpg;jpeg;jpe;resx;tiff;tif;png;wav;mfcribbon-ms</Extensions>
    </Filter>
  </ItemGroup>
  <ItemGroup>
    <ClInclude Include="pch.h">
      <Filter>Header Files</Filter>
    </ClInclude>
  </ItemGroup>
  <ItemGroup>
    <ClCompile Include="pch.cpp">
      <Filter>Source Files</Filter>
    </ClCompile>
    <ClCompile Include="AppOne.cpp">
      <Filter>Source Files</Filter>
    </ClCompile>
  </ItemGroup>
</Project>
```

Pliki `pch.cpp` i `pch.h` – pliki *prekompilowanych nagłówków*, ich celem jest przyspieszenie procesu kompilacji; wszystkie stabilne pliki nagłówkowe, powinny być zawarte w tym miejscu.



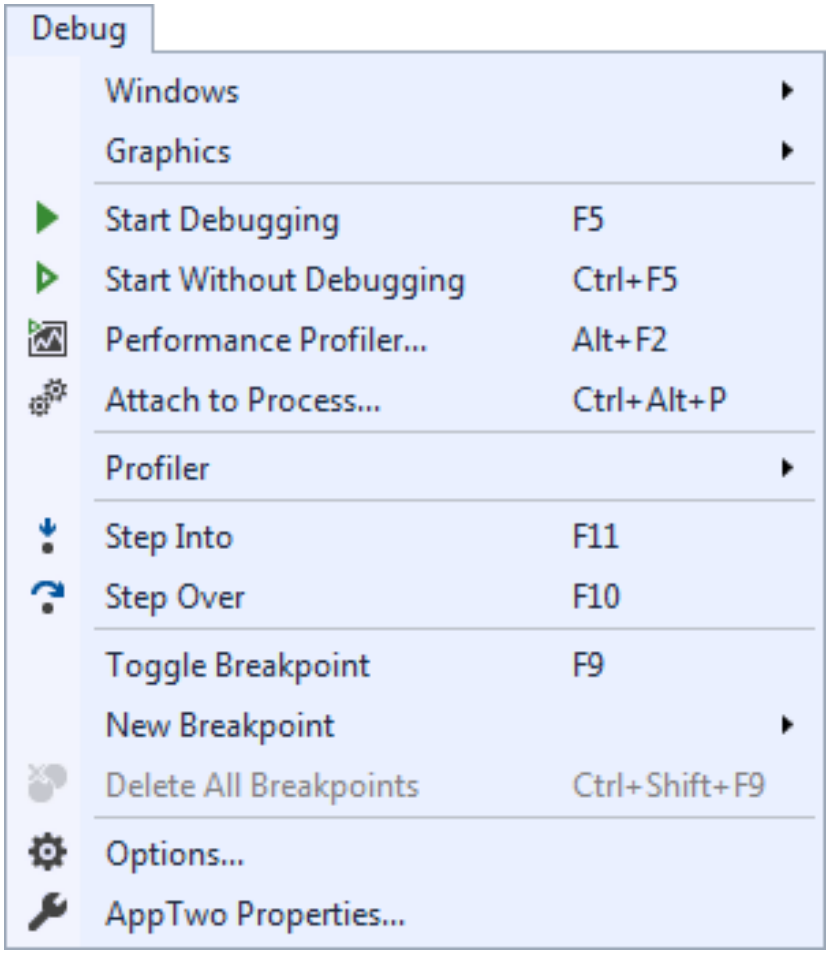

Lister - [D:\Visual\AppOne\AppOne\pch.h]
File Edit Options Encoding Help
100 %
// Tips for Getting Started:
//   1. Use the Solution Explorer window to add/manage files
//   2. Use the Team Explorer window to connect to source control
//   3. Use the Output window to see build output and other messages
//   4. Use the Error List window to view errors
//   5. Go to Project > Add New Item to create new code files, or Project > Add Existing Item to add existing code files t
//   6. In the future, to open this project again, go to File > Open > Project and select the .sln file
#ifndef PCH_H
#define PCH_H
// TODO: add headers that you want to pre-compile here
#endif //PCH_H

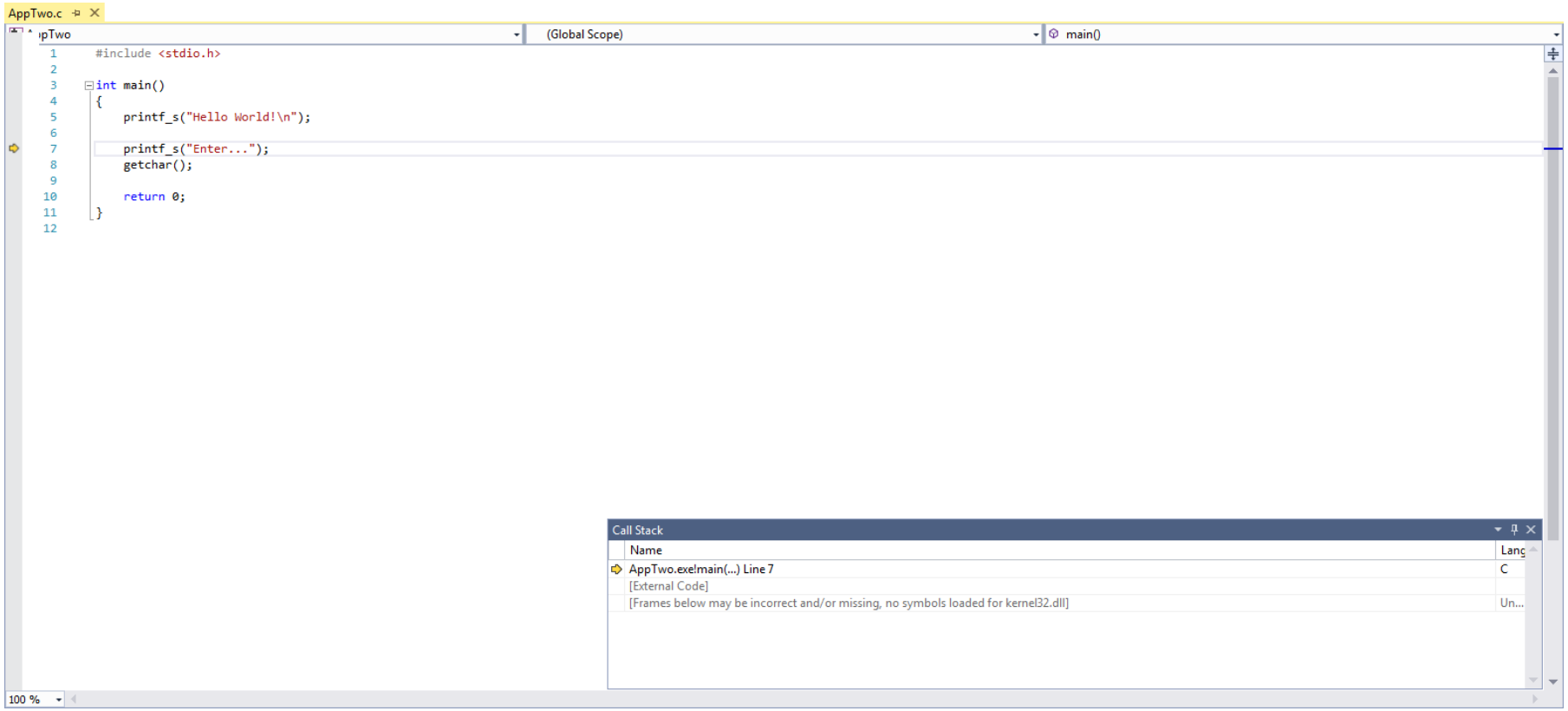
Plik `*.cpp` – plik *źródłowy*, zawiera kod źródłowy programu.

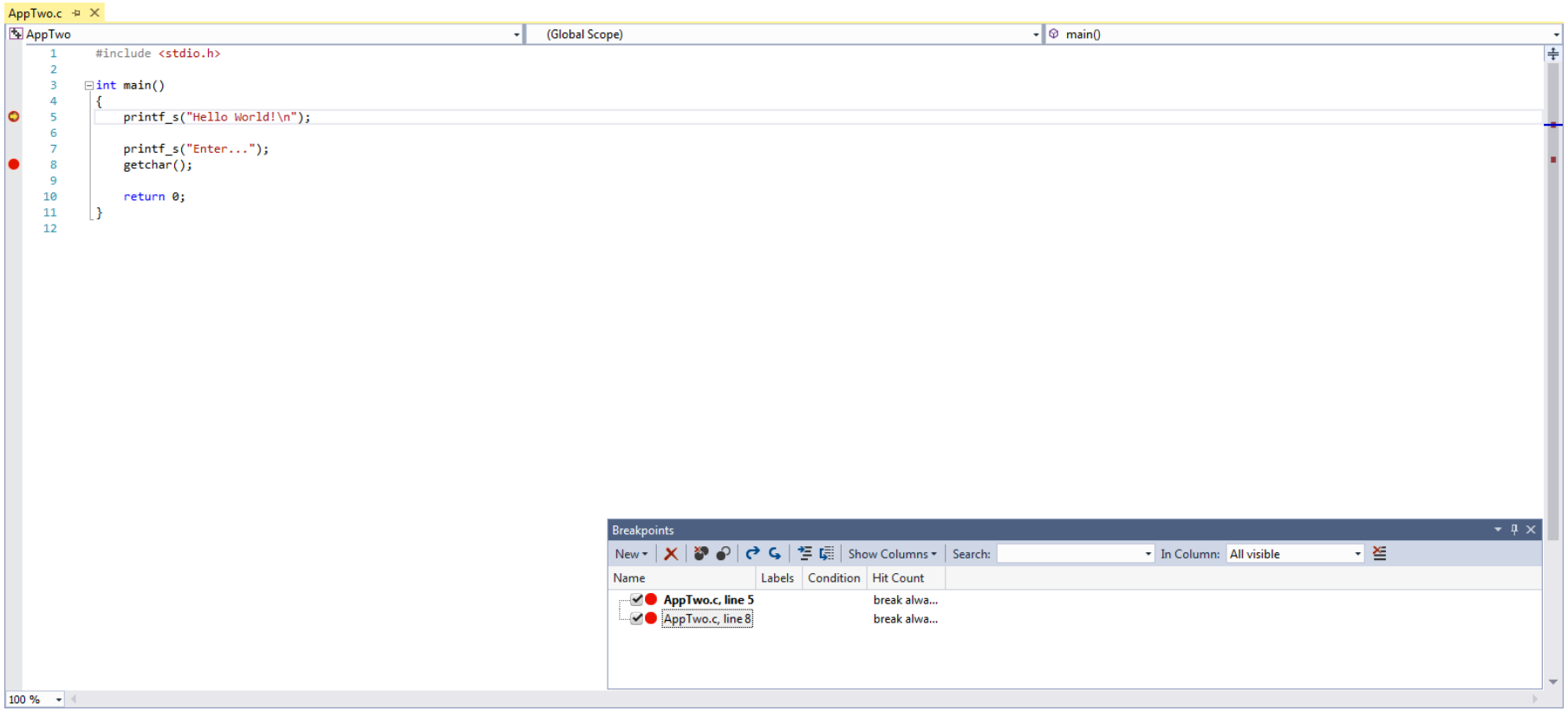


Kompilowanie, debugowanie, śledzenie

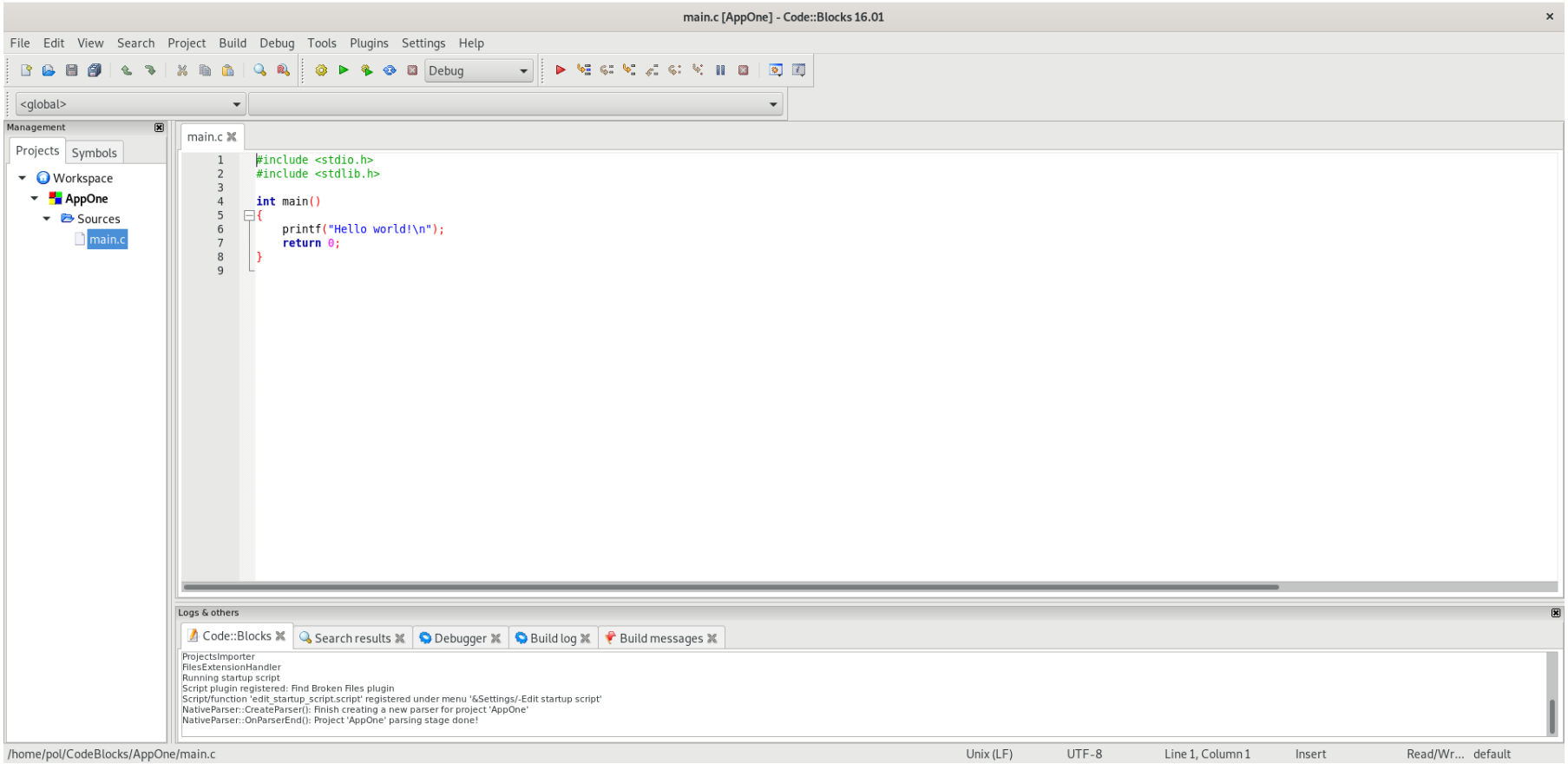
skróty klawiaturowe	
[F5]	start z nadzorowaniem (debugger) i kontynuacja wykonywania programu
[CTRL F5]	start bez debuggera
[F11]	krokowe wykonywanie programu
[F10]	krokowe wykonywanie programu, procedury i funkcje w jednym kroku
[F9]	wstawianie i usuwanie punktów przerwania (*Break Point*)



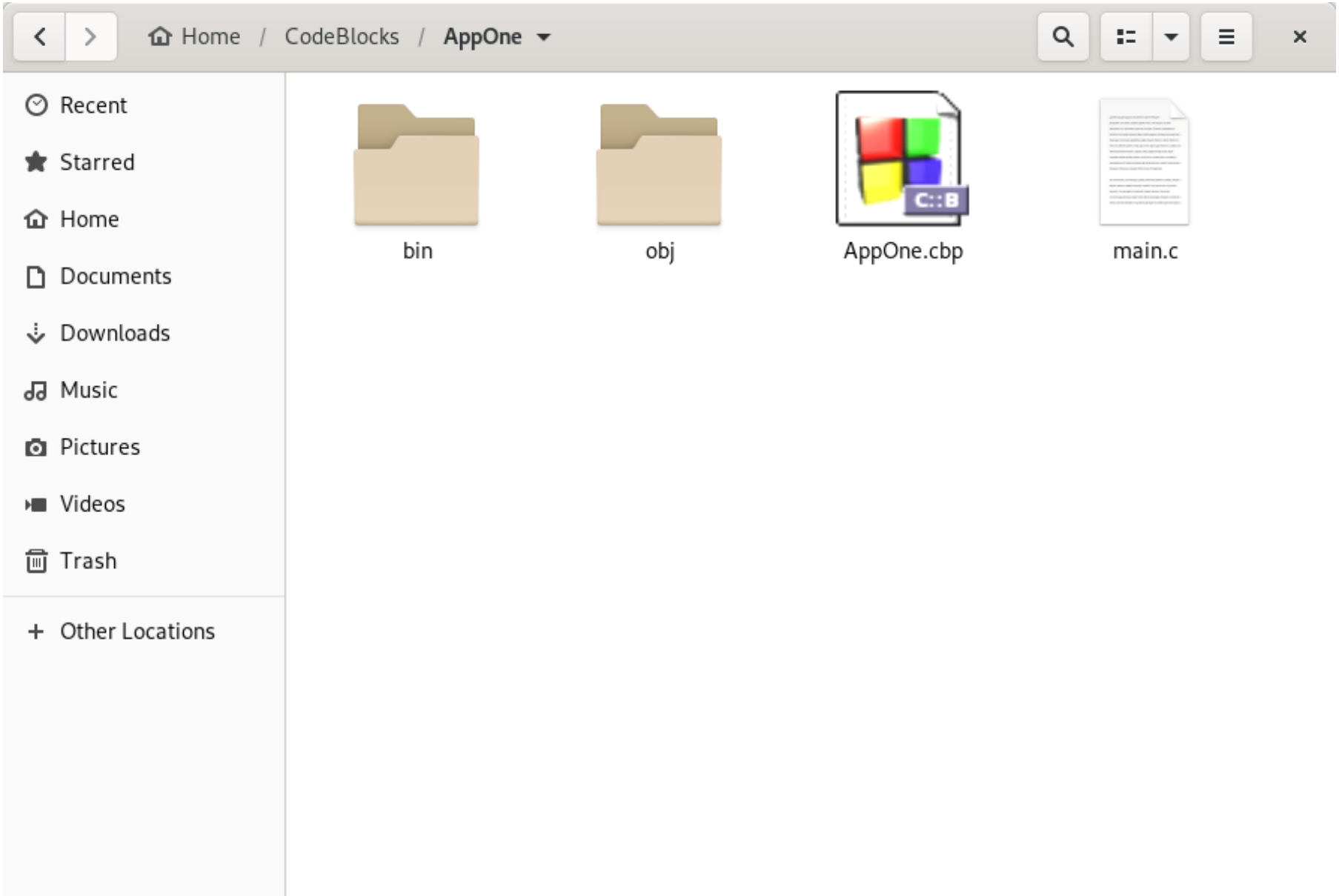

AppTwo.c
pTwo
(Global Scope)
main()
#include <stdio.h>
int main()
{
printf_s("Hello World!\n");
printf_s("Enter...");
getchar();
return 0;
}
Call Stack
Name
Lang
AppTwo.exe!main(...) Line 7
C
[External Code]
[Frames below may be incorrect and/or missing, no symbols loaded for kernel32.dll]
Un...
100 %

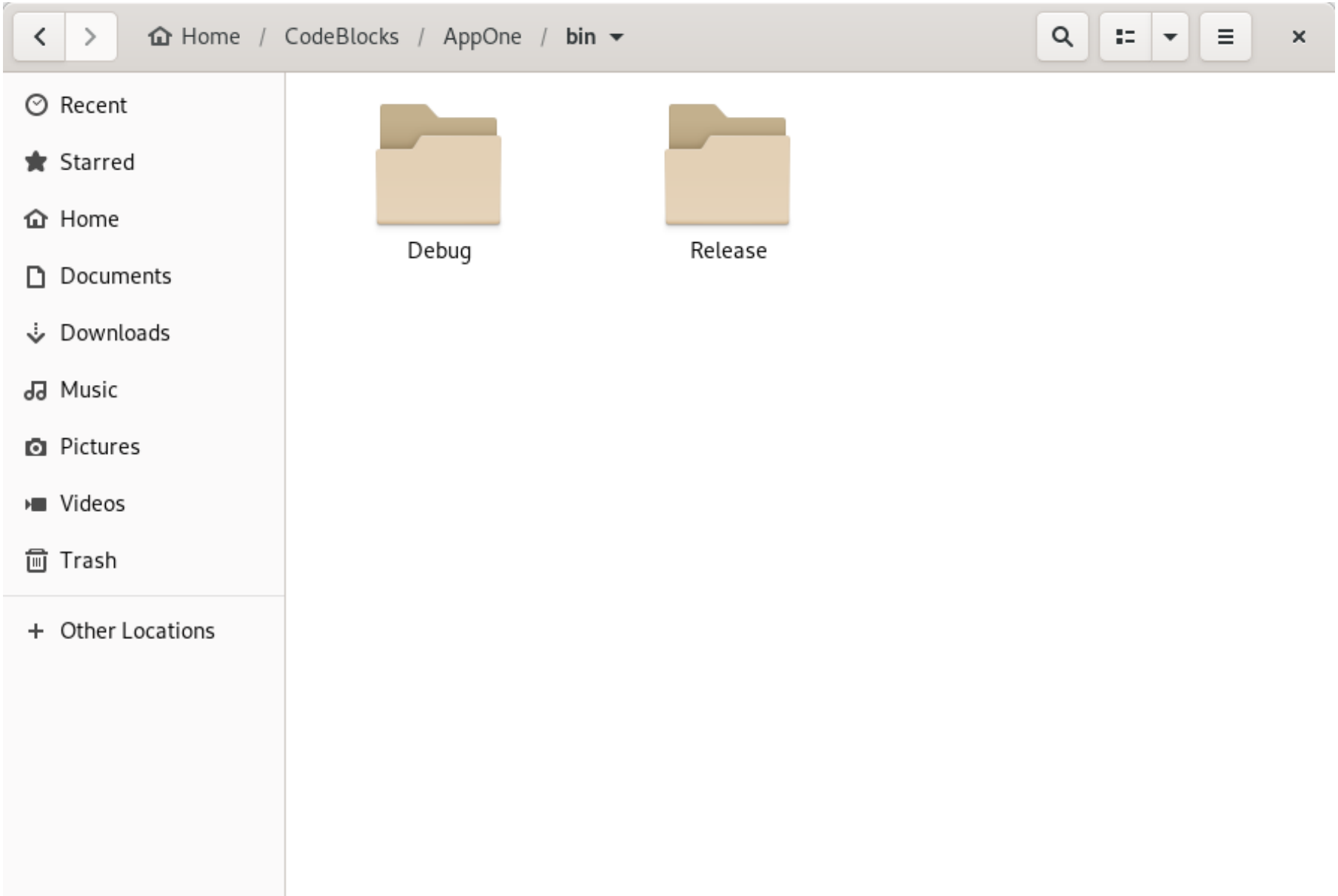

AppTwo.c
AppTwo
(Global Scope)
main()
#include <stdio.h>
int main()
{
printf_s("Hello World!\n");
printf_s("Enter...");
getchar();
return 0;
}
Breakpoints
New
Show Columns
Search:
In Column:
All visible
Name
Labels
Condition
Hit Count
AppTwo.c, line 5
break alwa...
AppTwo.c, line 8
break alwa...
100 %

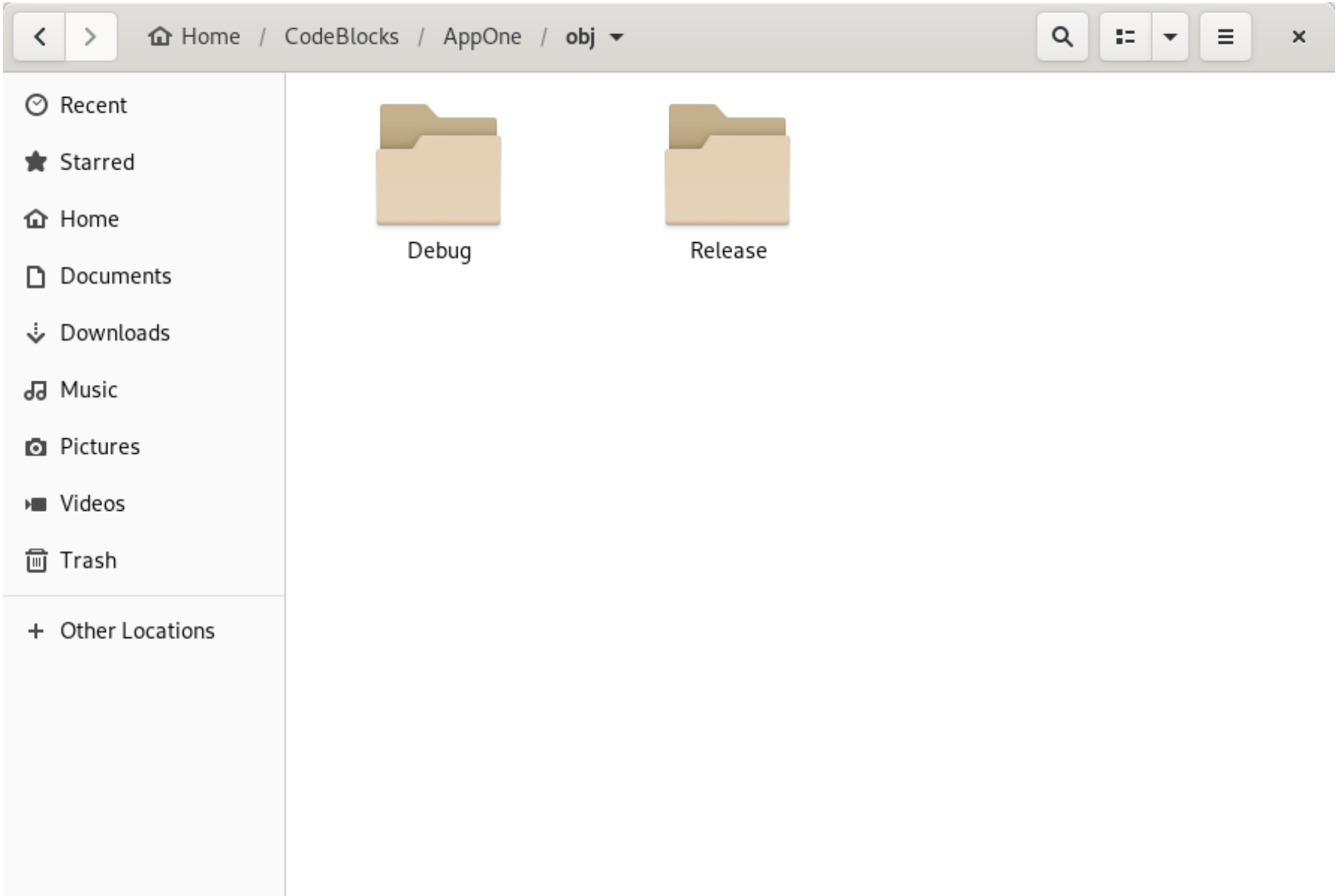
Środowisko programowania Code::Blocks - Linux

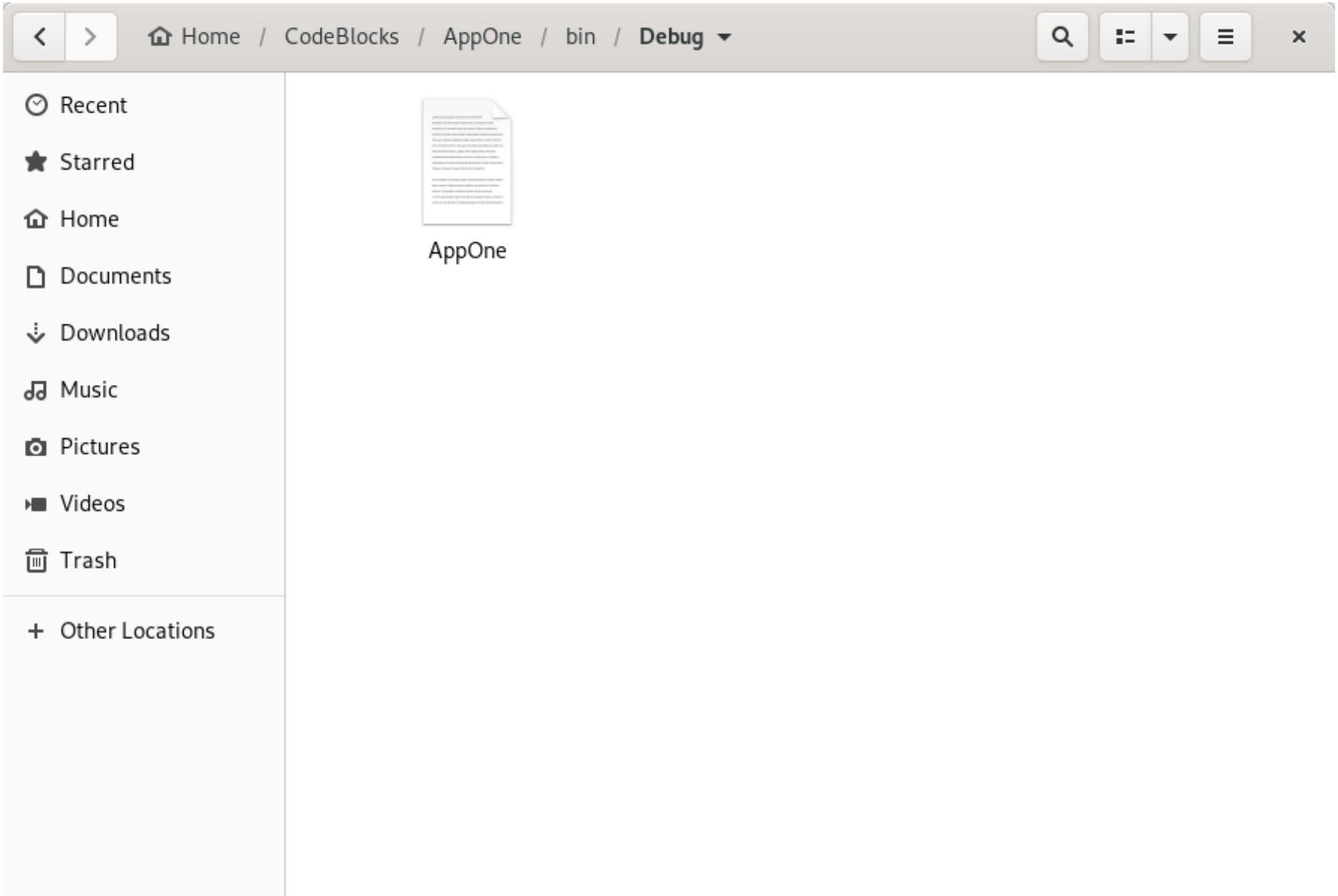


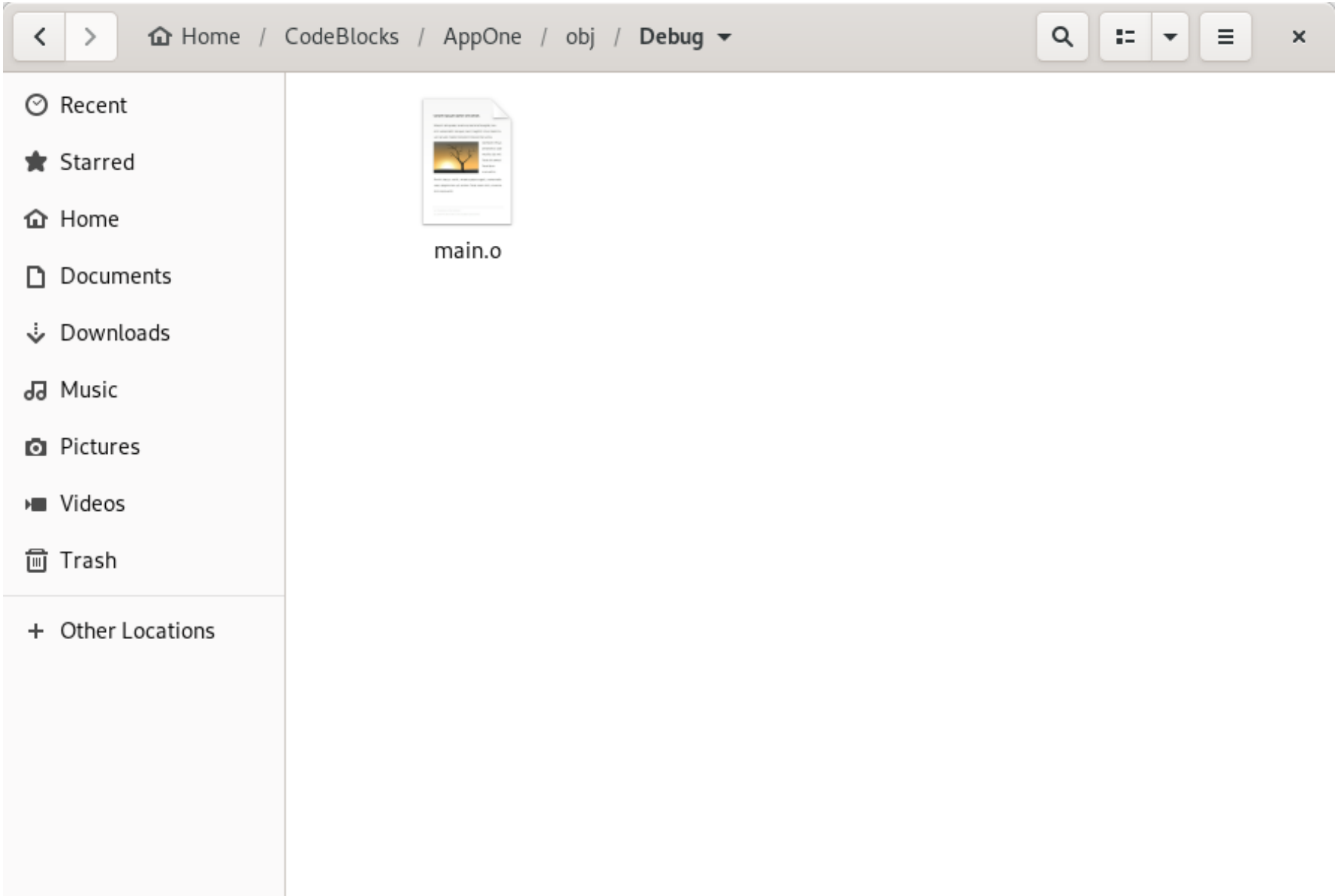
Pliki projektu




Home / CodeBlocks / AppOne / bin
Recent
Starred
Home
Documents
Downloads
Music
Pictures
Videos
Trash
Other Locations
Debug
Release

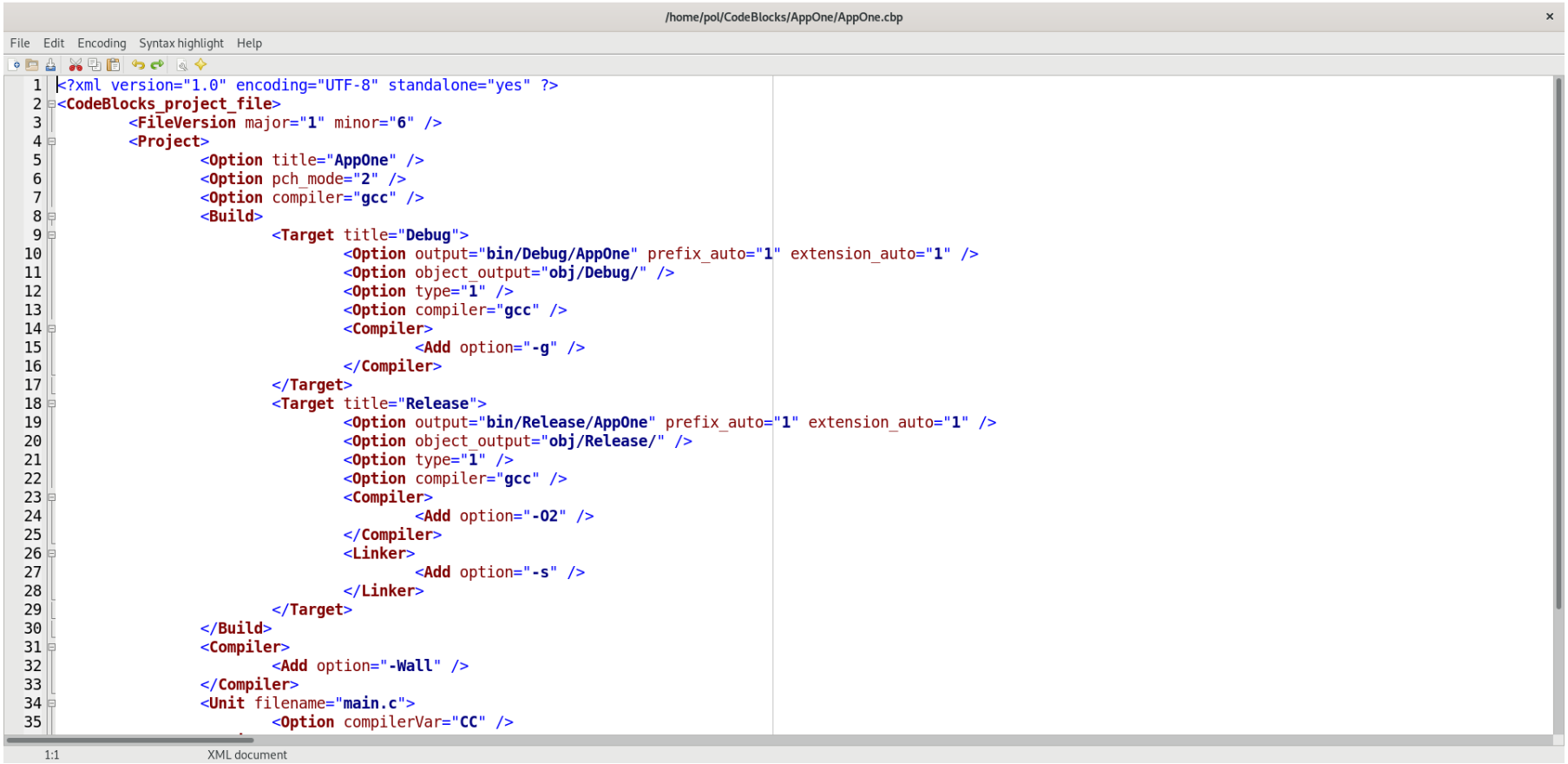

Home / CodeBlocks / AppOne / obj
Recent
Starred
Home
Documents
Downloads
Music
Pictures
Videos
Trash
Other Locations
Debug
Release


Home / CodeBlocks / AppOne / bin / Debug
Recent
Starred
Home
Documents
Downloads
Music
Pictures
Videos
Trash
Other Locations
AppOne


Home / CodeBlocks / AppOne / obj / Debug
Recent
Starred
Home
Documents
Downloads
Music
Pictures
Videos
Trash
Other Locations
main.o

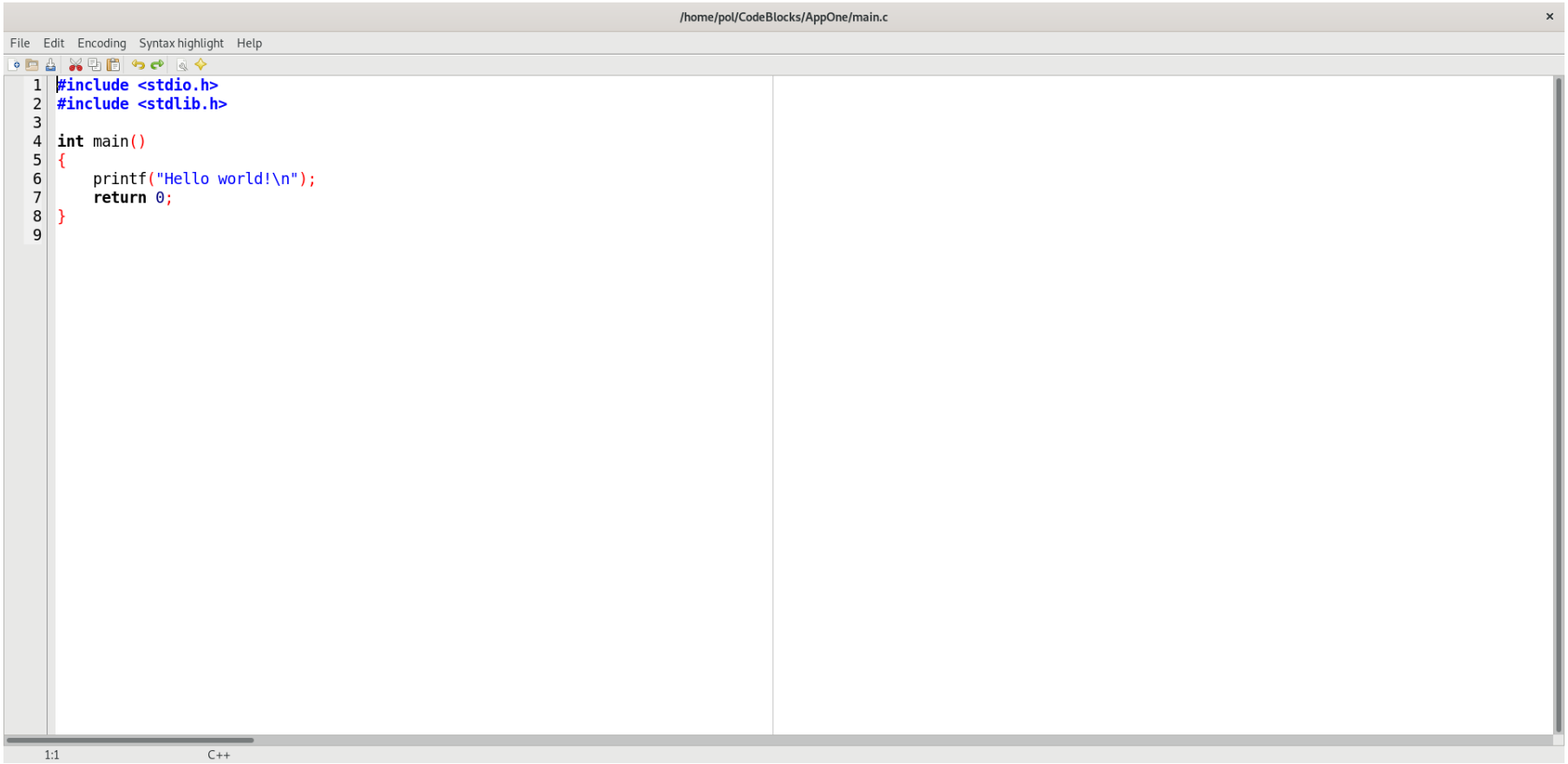
Struktura domyślnego projektu

Plik `*.cbp` – plik *projektu*, przechowuje informacje specyficzne dla każdego projektu.



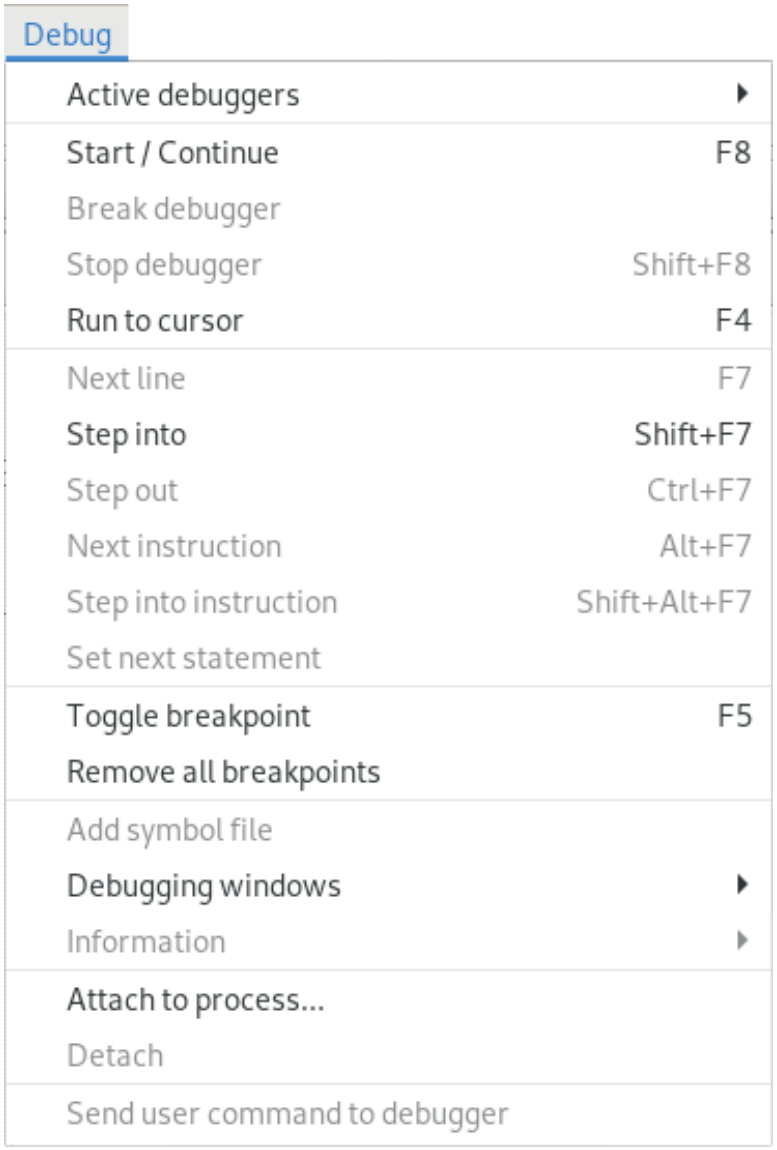

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<CodeBlocks_project_file>
	<FileVersion major="1" minor="6" />
	<Project>
		<Option title="AppOne" />
		<Option pch_mode="2" />
		<Option compiler="gcc" />
		<Build>
			<Target title="Debug">
				<Option output="bin/Debug/AppOne" prefix_auto="1" extension_auto="1" />
				<Option object_output="obj/Debug/" />
				<Option type="1" />
				<Option compiler="gcc" />
				<Compiler>
					<Add option="-g" />
				</Compiler>
			</Target>
			<Target title="Release">
				<Option output="bin/Release/AppOne" prefix_auto="1" extension_auto="1" />
				<Option object_output="obj/Release/" />
				<Option type="1" />
				<Option compiler="gcc" />
				<Compiler>
					<Add option="-O2" />
				</Compiler>
				<Linker>
					<Add option="-s" />
				</Linker>
			</Target>
		</Build>
		<Compiler>
			<Add option="-Wall" />
		</Compiler>
		<Unit filename="main.c">
			<Option compilerVar="CC" />
```

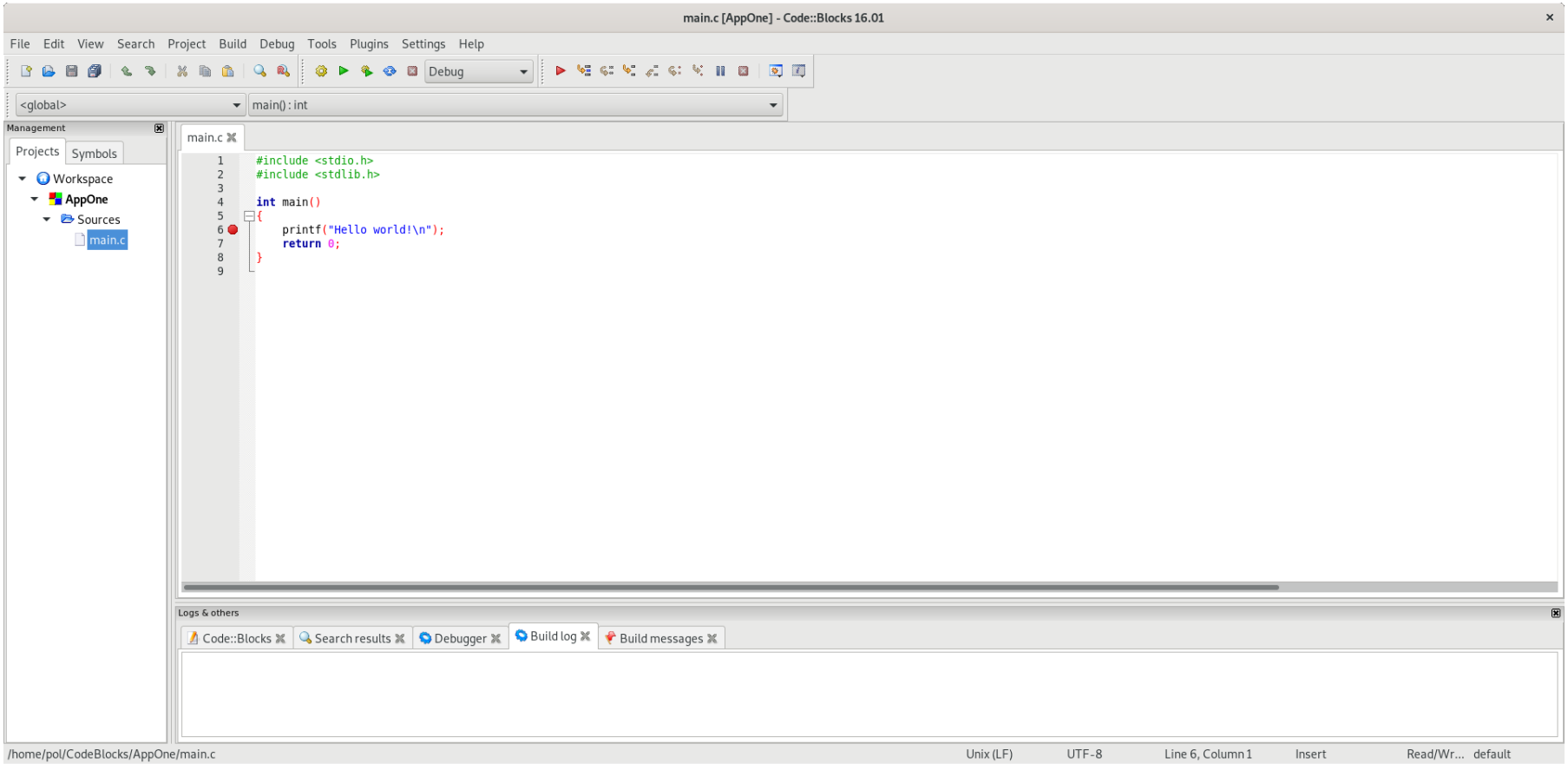
Plik `*.cpp` – plik *źródłowy*, zawiera kod źródłowy programu.

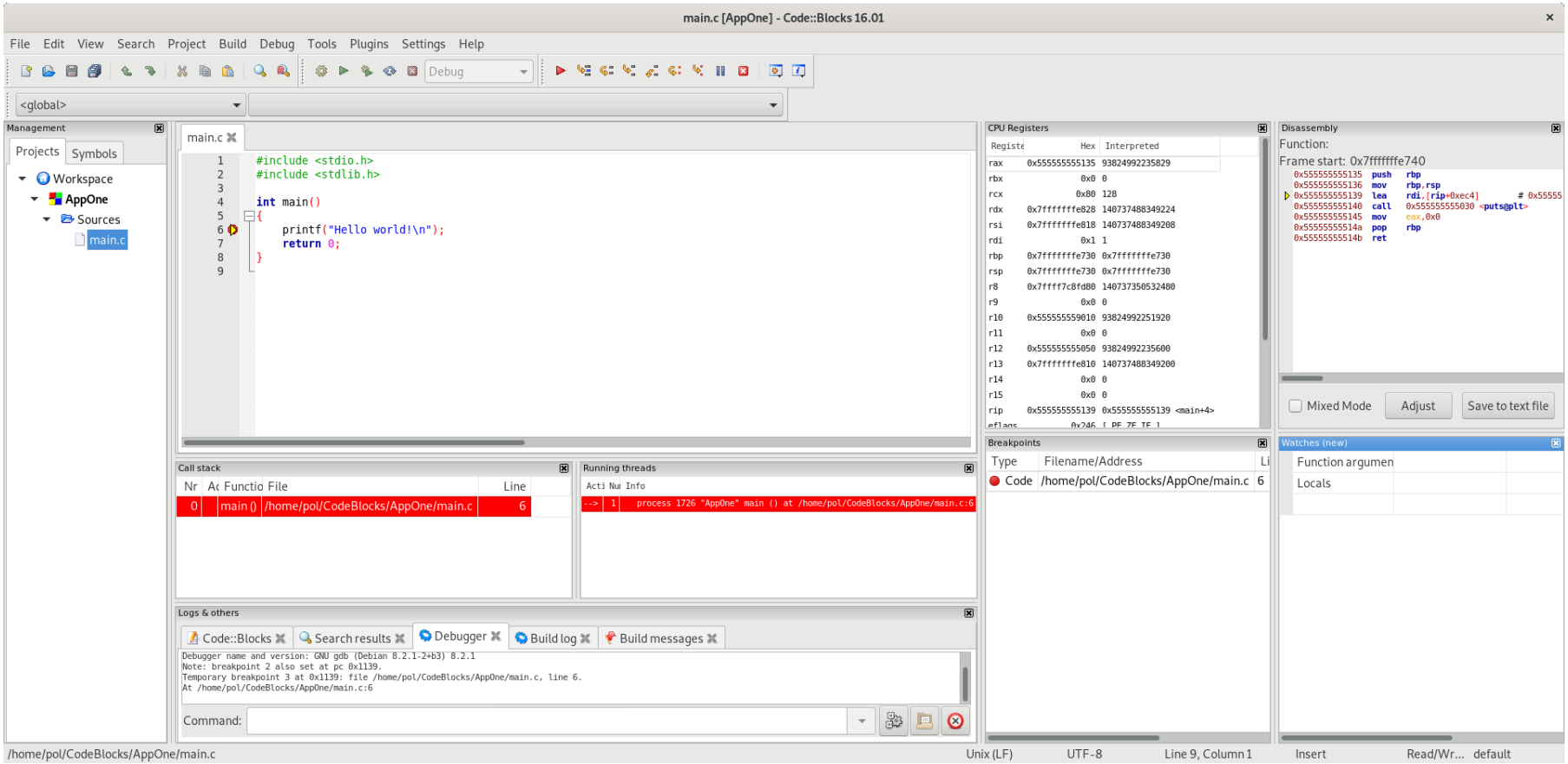


Kompilowanie, uruchamianie, debugowanie, śledzenie

Skróty klawiaturowe	
[F9]	budowanie i uruchomienie programu
[CTRL + F9]	budowanie programu
[F7]	uruchomienie do następnej linii kodu
[F8]	uruchomienie i kontynuowanie do kolejnego punktu przerwania
[SHIFT + F7]	kontynuowanie z rozwijaniem funkcji
[CTRL + F7]	kontynuowanie do powrotu z funkcji
[F5]	wstawianie i usuwanie punktów przerwania (*Breakpoint*)




main.c [AppOne] - Code::Blocks 16.01
File Edit View Search Project Build Debug Tools Plugins Settings Help
Debug
<global>
main() : int
Management
Projects Symbols
Workspace
AppOne
Sources
main.c
main.c
#include <stdio.h>
#include <stdlib.h>
int main()
{
printf("Hello world!\n");
return 0;
}
Logs & others
Code::Blocks Search results Debugger Build log Build messages
/home/pol/CodeBlocks/AppOne/main.c
Unix (LF) UTF-8 Line 6, Column 1 Insert Read/Wr... default


main.c [AppOne] - Code::Blocks 16.01
File Edit View Search Project Build Debug Tools Plugins Settings Help
Debug
<global>
Management
Projects Symbols
Workspace
AppOne
Sources
main.c
main.c
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main()
5 {
6 printf("Hello world!\n");
7 return 0;
8 }
9
CPU Registers
Registe Hex Interpreted
rax 0x555555555135 93824992235829
rbx 0x0 0
rcx 0x80 128
rdx 0x7fffffffe828 140737488349224
rsi 0x7fffffffe818 140737488349208
rdi 0x1 1
rbp 0x7fffffffe730 0x7fffffffe730
rsp 0x7fffffffe730 0x7fffffffe730
r8 0x7ffff7c8fd80 140737350532480
r9 0x0 0
r10 0x555555559010 93824992251920
r11 0x0 0
r12 0x555555555050 93824992235600
r13 0x7fffffffe810 140737488349200
r14 0x0 0
r15 0x0 0
rip 0x555555555139 0x555555555139 <main+4>
Disassembly
Function:
Frame start: 0x7fffffffe740
0x555555555135 push rbp
0x555555555136 mov rbp,rsp
0x555555555139 lea rdi,[rip+0xec4] # 0x55555
0x555555555140 call 0x555555555030 <puts@plt>
0x555555555145 mov eax,0x0
0x55555555514a pop rbp
0x55555555514b ret
Mixed Mode
Adjust
Save to text file
Call stack
Nr Ac Functio File Line
0 main () /home/pol/CodeBlocks/AppOne/main.c 6
Running threads
Acti Nu Info
--> 1 process 1726 "AppOne" main () at /home/pol/CodeBlocks/AppOne/main.c:6
Breakpoints
Type Filename/Address Li
Code /home/pol/CodeBlocks/AppOne/main.c 6
Watches (new)
Function argumen
Locals
Logs & others
Code::Blocks
Search results
Debugger
Build log
Build messages
Debugger name and version: GNU gdb (Debian 8.2.1-2+b3) 8.2.1
Note: breakpoint 2 also set at pc 0x1139.
Temporary breakpoint 3 at 0x1139: file /home/pol/CodeBlocks/AppOne/main.c, line 6.
At /home/pol/CodeBlocks/AppOne/main.c:6
Command:
/home/pol/CodeBlocks/AppOne/main.c
Unix (LF) UTF-8 Line 9, Column 1 Insert Read/Wr... default

Budowa programu (który nic nie robi)

```
int main()
{
  return 0;
}
```

Każdy program C++ musi posiadać funkcję o nazwie `main()`, stanowiącą punkt wejściowy programu. Od niej rozpoczyna się wykonanie programu.

Budowa programu:

- typ rezultatu funkcji,
- nazwa funkcji,
- parametry funkcji,
- ciało funkcji,
- instrukcja powrotu z podprogramu,
- wartość przekazywana w miejscu wywołania.

Biblioteki i dyrektywa #include

Symbole `EXIT_SUCCESS` i `EXIT_FAILURE` zalecane dla języka C, są zdefiniowane w bibliotece `stdlib.h`, która ma wersję mogąca pracować z kompilatorami C++ o nazwie `cstdlib`.

Dyrektywa `#include<>` powoduje włączenie do kodu źródłowego programu definicji i deklaracji zawartych w innych plikach źródłowych (często to dodatkowe biblioteki).

Dyrektywa `#include<>` jest realizowana przez **preprocesor** języka C++.

Pliki nagłówkowe w języku C++ mogą mieć rozszerzenia `*.h`, `*.hpp`, `*.hxx`.

Aktualnie w kodach źródłowych C++ nie pisze się rozszerzeń nazw plików nagłówkowych. Dodatkowo, nazwy plików nagłówkowych z języka C pisze się z prefiksem `c`.

Nic nie robiący program w języku C++:

```
#include <cstdlib>

int main()
{
  return EXIT_SUCCESS;
}
```

Różne kompilatory różnie podchodzą do powyższych zaleceń.

Standardowe strumienie

Standardowe kanały komunikacji między komputerem a otoczeniem (zwykle terminalem). Występują w Uniksie i systemach uniksopodobnych, w środowisku uruchomieniowym C/C++ i ich pochodnych.

Trzy podstawowe połączenia I/O:

- `stdin` – **standardowe wejście** programu, zwykle kojarzone z klawiaturą (*standard input*),
- `stdout` – **standardowe wyjście** programu, zwykle kojarzone z ekranem monitora (*standard output*),
- `stderr` – **standardowe wyjście błędów**, zwykle kojarzone z ekranem monitora (*standard error*).

`stdin` i `stdout` reprezentują normalny kanał komunikacji programu z użytkownikiem.

`stderr` zarezerwowany jest do wyświetlania komunikatów diagnostycznych programu.

Przykład **przekierowania** (redyrekcji) strumieni z poziomu systemu operacyjnego:

```
C:\> foo.exe > output.txt
```

```
C:\> foo.exe < input.txt
```

```
C:\> foo.exe < input.txt > output.txt
```

Strumienie w języku C++:

- `cin` – strumień reprezentujący standardowe wyjście, odpowiada strumieniowi `stdin` z C,
- `cout` – strumień reprezentujący standardowe wejście, odpowiada strumieniowi `stdout` z C,
- `cerr` – niebuforowany strumień wyjściowy błędów, odpowiada strumieniowi `stderr` z C,
- `clog` – buforowany strumień wyjściowy błędów, odpowiada strumieniowi `stderr` z C.

Aby skorzystać ze strumieni, należy włączyć plik nagłówkowy `iostream`.

W języku C++ strumienie są **obiektami**.

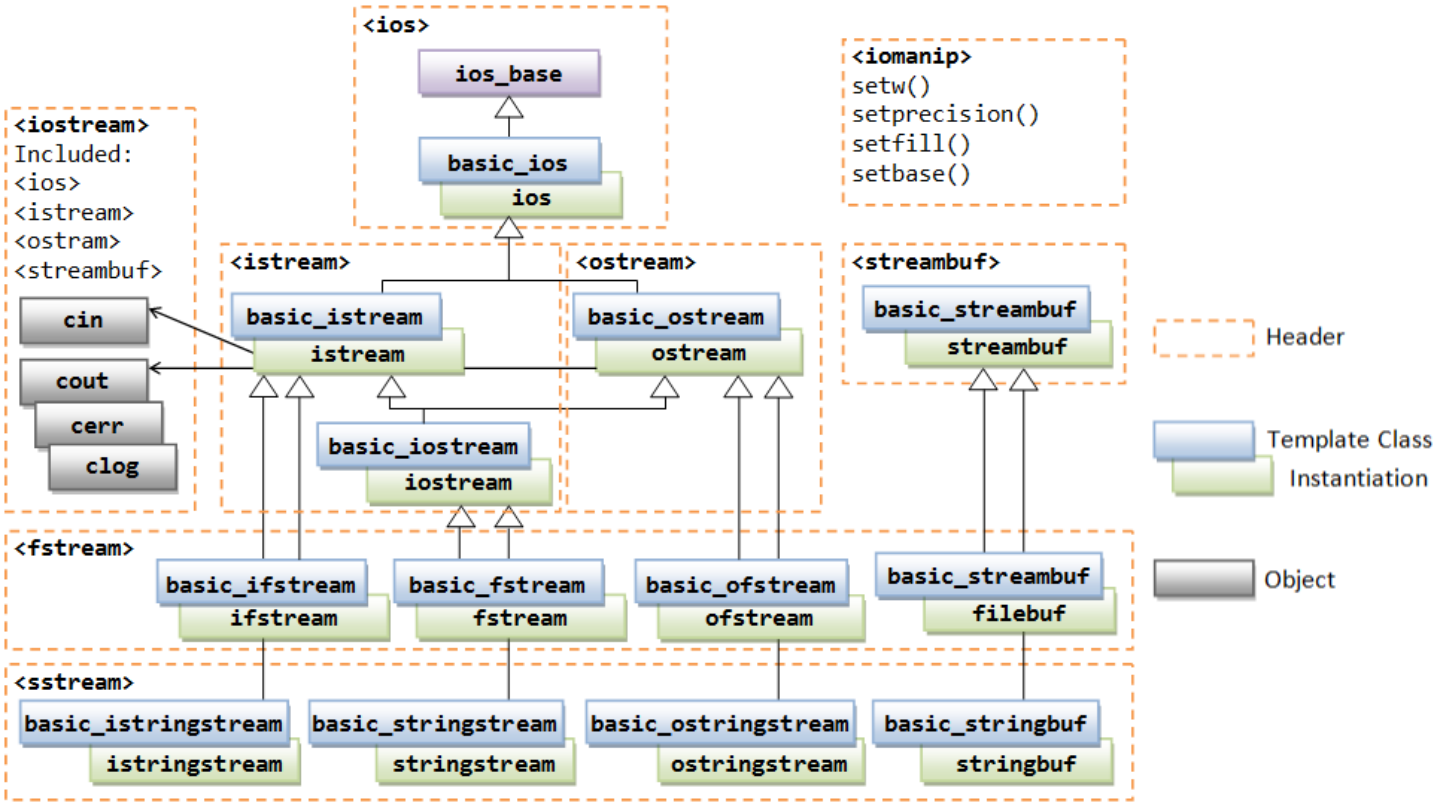
Poważny program w języku C++

```cpp
#include <iostream>

using namespace std;

int main()
{
  float cm, inch;

  cout << "Przeliczanie centymetrow na cale" << endl;
  cout << "Podaj wymiar w cm: ";

  cin >> cm;

  inch = 0.3937 * cm;

  cout << cm << "cm to " << inch << "\"" << endl;

  return 0;
}
```

```
Przeliczanie centymetrow na cale
Podaj wymiar w cm: 5
5cm to 1.968500"
```

Obsługa wejścia/wyjścia

Język C++, podobnie jak C nie posiada instrukcji do obsługi standardowego wejścia i wyjścia i wejścia.



Obsługa wejścia-wyjścia jest realizowana z wykorzystaniem zaawansowanych możliwości tego języka. Należą do nich: **klasy**, **klasy pochodne**, **przeciążanie funkcji** i **operatorów**, **funkcje wirtualne**, **szablony** i **wielodziedziczenie**.

Zarządzanie strumieniami i buforami może być dość skomplikowane, dlatego biblioteka `iostream` udostępnia kilka klas, których zadaniem jest ułatwienie zarządzania tymi elementami:

- klasa `streambuf` – zapewnia obszar pamięci na bufor wraz z metodami służącymi do wypełniania bufora, odwoływania się do zawartości bufora, opróżniania bufora, zarządzania pamięcią bufora,
- klasa `ios_base` – reprezentuje ogólne właściwości strumienia informujące o tym, czy jest otwarty, czy jest strumieniem tekstowym czy też binarnym,
- klasa `ios` – jest pochodną klasy `ios_base` i zawiera wskaźnik na obiekty klasy `streambuf`,
- klasa `ostream` – dziedziczy z klasy `ios` i udostępnia metody do obsługi wyjścia,
- klasa `istream` – dziedziczy z klasy `ios` i udostępnia metody do obsługi wejścia,
- klasa `iostream` – wielodziedziczy po klasach `istream` i `ostream`, zawiera metody do obsługi wejścia iak i wyjścia.

Aby skorzystać z takich udogodnień, należy użyć obiektów odpowiednich klas. Na przykład do obsługi wyjścia użyjemy obiektu klasy `ostream`, jak choćby `cout`.

Utworzenie takiego obiektu powoduje otwarcie strumienia, automatyczne utworzenia bufora i skojarzenie go ze strumieniem, a także udostępnienie funkcji składowych danej klasy.

```
#include <iostream>

int main()
{
  using namespace std;

  int a = 0, b = 0;

  cout << "Podaj dwie liczby:" << endl;

  cin >> a >> b;

  cout << "Suma " << a << " i " << b << " wynosi: " << a + b << "." << endl;

  return 0;
}
```

Obiekt `cout` jest zwykle łączony z operatorem `<<` (operator **wstawiania**).

Operator wstawiania jest **przeciążony** i współpracuje ze wszystkimi podstawowymi typami C++:

- `unsigned char`,
- `signed char`,
- `char`,
- `short`,
- `unsignet short`,
- `int`,
- `unsignet int`,
- `long`,
- `unsigned long`,
- `float`,
- `double`,
- `long double`.

Klasa `ostream` z obiektem `cout` zapewnia definicję funkcji operatorowej `operator << ()` dla każdego z typów danych.

W klasie `ostream` zdefiniowano też funkcje wstawiania dla typów wskaźnikowych:

- `const unsigned char *`,
- `const signed char *`,
- `const char *`,
- `void *`.

```
const char *str = "Ala ma psa.";
char name[20] = "Wiktor Wektor";

cout << "Halo!";
cout << name;
cout << str;
```

Operacje z użyciem obiektów pochodnych z klasy `ostream`, czyli na przykład `cout` są buforowane.

Mechanizm buforowania jest korzystny w przypadku gdy standardowe wejście jest skojarzone z plikiem, może być kłopotliwe (bardzo rzadkie przypadki) jeśli standardowe wyjście jest skojarzone z ekranem.

W większości implementacji języka C++ bufor skojarzony z ekranem jest opróżniany natychmiast oraz w przypadku oczekującej operacji wejścia.

```
cout << "Podaj liczbe: "
cin >> n;
```

Gdyby zaszła taka potrzeba, opróżnienie bufora można wymusić za pomocą jednego z dwóch manipulatorów: `endl` i `flush`.

```
cout << "Dzien dobry!" << flush;
cout << "Przetwarzanie, prosze czekac..." << endl;
```

Manipulatory są w rzeczywistości funkcjami. Można opróżnić bufor obiektu `cout` wywołując funkcję bezpośrednio:

```
flush(cout);
```

Klasa `ostream` przeciąża operator wstawiania `<<` w taki sposób, że wyrażenie `<< flush` jest zastępowane przez stosowne wywołanie funkcji `flush()`.

Operator wstawiania klasy `ostream` konwertuje wartości na postać tekstową:

- wartość **typu znakowego**, jeśli reprezentuje znak drukowalny, jest wyświetlana jako znak w polu o szerokości jednego znaku,
- liczbowe **typy całkowite** są wyświetlane jako całkowite liczby dziesiętne w polu o szerokości odpowiedniej, aby pomieścić daną liczbę i ewentualnie znak minusa,
- **łańcuchy** są wyświetlane w polu o szerokości równej długości danego łańcucha,
- **typy zmiennoprzecinkowe** są wyświetlane przy użyciu sześciu cyfr, zera końcowe są pomijane, liczba jest wyświetlana w notacji stałoprzecinkowej lub notacji naukowej, notacja naukowa jest stosowana, gdy wykładnik jest większy od 5 lub mniejszy od -4.

```
cout << "123456789012345678901\n";

char z = 'K';
int t = 273;

cout << z << "\n";
cout << t << "\n";
cout << -t << "\n";

double a = 1.200;

cout << a << "\n";
cout << (a + 1.0 / 9.0) << "\n";

double b - 1.67E2;

cout << b << "\n";

b += 1.0 / 9.0;

cout << b << "\n";
cout « (b * 1.0e4) << "\n";

double c = 2.3e-4;
cout << c << "\n";
cout << c / 10 << "\n";
```

```
12345678901234567890
K
273
-273
1.2
1.31111
167
167.111
1.6711le+006
0.00023
2.3e-005
```

Manipulatory służą też do **formatowania** standardowego wyjścia.

Manipulatory są używane do dwóch kategoriach: zmiana **sposobu prezentowania** wartości liczbowych i kontrolowanie **szerokości i położenia wypełnienia**.

Większość manipulatorów, które zmieniają format występują w parach: jeden manipulator ustawia stan formatu na nową wartość, a drugi go usuwa przywracając domyślne formatowanie.

Manipulatory zmieniające format strumienia zazwyczaj pozostawiają zmieniony stan dla wszystkich kolejnych operacji wyjścia.

manipulatory zdefiniowane w `iostream`	
`boolalpha`	wyświetla `true` i `false` jako napisy
`noboolalpha`	wyświetla `true` i `false` jako wartości numeryczne `1` i `0`
`showbase`	dodaje prefiks wskazujący podstawę systemu liczenia
`noshowbase`	nie dodaje prefiksu systemu liczenia
`showpoint`	zawsze wyświetla przecinek dziesiętny
`noshowpoint`	wyświetla przecinek dziesiętny, gdy wartość ma część ułamkową
`showpos`	wyświetla znak `+` w liczbach nieujemnych
`noshowpos`	nie wyświetla znaku `+` w liczbach nieujemnych
`uppercase`	wyświetla `0X` w systemie szesnastkowym i `E` w notacji naukowej
`nouppercase`	wyświetla `0x` w systemie szesnastkowym i `e` w notacji naukowej
`dec`	wyświetla wartości całkowite w systemie dziesiętnym
`hex`	wyświetla wartości całkowite w systemie szesnastkowym
`oct`	wyświetla wartości całkowite w systemie ósemkowym

manipulatory zdefiniowane w `iostream`	
`left`	dodaje znaki wypełnienia po prawej stronie wartości
`right`	dodaje znaki wypełnienia po lewej stronie wartości
`internal`	dodaje znaki wypełnienia między znakiem a wartością
`fixed`	wyświetla wartości zmiennoprzecinkowe w notacji dziesiętnej
`scientific`	wyświetla wartości zmiennoprzecinkowe w notacji naukowej
`hexfloat`	wyświetlaa wartości zmiennoprzecinkowe w systemie szesnastkowym (od C++ 11)
`defaultfloat`	resetuje format zmiennoprzecinkowy do domyślnego (od C++ 11)
`unitbuf`	opróżnia bufory po każdej operacji wyjściowej
`nounitbuf`	przywraca domyślne opróżnianie bufora
`skipws`	pomija białe znaki w operatorze wejściowym
`noskipws`	nie pomija białych znaków w operatorze wejściowym
`flush`	opróżnij bufor `ostream`
`ends`	wstawia `null`, a następnie opróżnia bufor
`endl`	wstawia znak nowego wiersza, a następnie opróżnia bufor

Przykład formatowania wartości logicznych.

```
cout << "default bool values: " << true << " " << false << endl;
cout << "alpha bool values: " << boolalpha << true << " " << false << endl;
```

```
default bool values: 1 0
alpha bool values: true false
```

Wartości całkowite w różnych systemach liczbowych.

```
cout << "default: " << 20 << ", " << 1024 << endl;
cout << "octal: " << oct << 20 << ", " << 1024 << endl;
cout << "hex: " << hex << 20 << ", " << 1024 << endl;
cout << "decimal: " << dec << 20 << ", " << 1024 << endl;
```

```
default: 20, 1024
octal: 24, 2000
hex: 14, 400
decimal: 20, 1024
```

Wartości całkowite w różnych systemach liczbowych, dołączony prefiks systemu liczenia.

```
cout << showbase;
cout << "default: " << 20 << ", " << 1024 << endl;
cout << "in octal: " << oct << 20 << ", " << 1024 << endl;
cout << "in hex: " << hex << 20 << ", " << 1024 << endl;
cout << "in decimal: " << dec << 20 << ", " << 1024 << endl;
cout << noshowbase;
```

```
default: 20, 1024
in octal: 024, 02000
in hex: 0x14, 0x400
in decimal: 20, 1024
```

```
cout << uppercase << showbase << hex;
cout << "printed in hexadecimal: " << 20 << ", " << 1024;
cout << nouppercase << noshowbase << dec << endl;
```

```
printed in hexadecimal: 0X14, 0X400
```

Formatowanie i precyzja wartości zmiennoprzecinkowych.

```
cout << "precision: " << cout.precision() << ", value: " << sqrt(2.0) << endl;
cout.precision(12);
cout << "precision: " << cout.precision() << ", value: " << sqrt(2.0) << endl;
cout << setprecision(3);
cout << "precision: " << cout.precision() << ", value: " << sqrt(2.0) << endl;
```

```
precision: 6, value: 1.41421
precision: 12, value: 1.41421356237
precision: 3, value: 1.41
```

```
cout << "default format: " << 100 * sqrt(2.0) << endl;
cout << "scientific: " << scientific << 100 * sqrt(2.0) << endl;
cout << "fixed decimal: " << fixed << 100 * sqrt(2.0) << endl;
cout << "hexadecimal: " << hexfloat << 100 * sqrt(2.0) << endl;
cout << "use defaults: " << defaultfloat << 100 * sqrt(2.0) << endl;
```

```
default format: 141
scientific: 1.414e+02
fixed decimal: 141.421
hexadecimal: 0x1.1ad7bc01366b8p+7
use defaults: 141
```

```
cout << 10.0 << endl;
cout << showpoint << 10.0 << endl;
cout << noshowpoint << endl;
```

```
10
10.0
```

Wcięcia, justowanie, wypełnienie.

```
int i = -16;
double d = 3.14159;

cout << "i: " << setw(12) << i << endl;
cout << "d: " << setw(12) << d << endl;

cout << left;
cout << "i: " << setw(12) << i << endl;
cout << "d: " << setw(12) << d << endl;

cout << right;
cout << "i: " << setw(12) << i << endl;
cout << "d: " << setw(12) << d << endl;

cout << internal;
cout << "i: " << setw(12) << i << endl;
cout << "d: " << setw(12) << d << endl;

cout << setfill('#');
cout << "i: " << setw(12) << i << endl;
cout << "d: " << setw(12) << d << endl;
cout << setfill(' ');
```

```
i:          -16
d:         3.14
i: -16
d: 3.14
i:          -16
d:         3.14
i: -         16
d:         3.14
i: -#########16
d: ########3.14
```

Klasa `ostream` zawiera również *niskopoziomowe* **metody** (funkcje) pozwalające na bezpośrednią obsługę wejścia i wyjścia z pominięciem formatowania.

funkcje jednoznakowe	
`is.get(ch)`	pobiera kolejny znak ze strumienia wejściowego i przypisuje do `ch`
`os.put(ch)`	umieszcza znak w strumieniu wyjściowym
`is.get()`	zwraca kolejny znak ze strumienia wejściowego po konwersji do `int`
`is.putback(ch)`	wstawia znak na początek strumienia wejściowego
`is.unget()`	umieszcza ostatni odczytany znak z powrotem w strumieniu wejściowym
`is.peek()`	odczytuje kolejny znak bez pobierania go

funkcje wieloznakowe	
`is.get(sink, size, delim)`	odczytuje łańcuch znaków i zapisuje je w tablicy znakowej `sink`
`is.getline(sink, size, delim)`	odczytuje dane ze strumienia do napotkania znaku nowej linii
`is.read(sink, size)`	odczytuje znaki ze strumienia i zapisuje w tablicy znakowej `sink`
`is.gcount()`	zwraca liczbę znaków odczytanych przez ostatnią operację wejścia
`os.write(source, size)`	wstawia określoną liczbę znaków do strumienia
`is.ignore(size, delim)`	pomija znaki w strumieniu wejściowym

Z czego jest zbudowany program?

W trakcie procesu kompilacji kod źródłowy jest dzielony na **jednostki leksykalne**.

Rozróżnia się naspępujące klasy jednostek leksykalnych:

- identyfikatory (*identifiers*),
- słowa kluczowe (*keywords*),
- stałe (*constants*),
- napisy (łańcuchy znaków, *string literals*),
- operatory (*operators*),
- separatory (*punctuators*).

Identyfikatory

Identyfikatory to ciągi liter, cyfr i znaków podkreślenia, muszą zaczynać się od litery lub znaku podkreślenia. Wielkie i małe litery są rozróżniane.

Kompilatory zwykle rozróżniają pierwsze 8 lub 32 znaki. Polskie znaki nie są traktowane jak litery i nie mogą być używane.

Identyfikatory są arbitralnie wybranymi nazwami dla **zmiennych**, **stałych**, **funkcji**, **typów danych** definiowanych przez programistę. Identyfikatory nie mogą być **słowami kluczowymi**.

```
maxvalue      mindelta      _5Pi
max_value     min_delta     _5_Pi
MaxValue      MinDelta      JW23
maxValue      minDelta      JW_23
```

Słowa kluczowe

Do języka C++ zostały przeniesione słowa kluczowe istniejące w języku C, dodano nowe:

```
and               and_eq            asm           atomic_cancel atomic_commit
atomic_noexcept   bitand            bitor         bool          catch
class             compl             const_cast    delete        dynamic_cast
explicit          export            false         friend        mutable
namespace         new               not           not_eq        operator
or                or_eq             private       protected     public
reflexpr          reinterpret_cast  static_cast   synchronized  template
this              throw             true          try           typeid
typename          using             virtual       wchar_t       while
xor               xor_eq
```

Zestaw słów kluczowych może być rozszerzany w zależności od kompilatora i środowiska programistycznego.

W kolejnych wersjach pojawiły się kolejne słowa kluczowe.

```
alignas   alignof   char16_t      char32_t      constexpr  decltype          ← C++11
noexcept  nullptr   static_assert thread_local
```

```
char8_t   concept   consteval     constinit     co_await   co_return         ← C++20
co_yield  requires
```

```
contract_assert                                                              ← C++26
```

Separatory

Nawiasy kwadratowe (*brackets*) `[ ]` wykorzystywane są do deklarowania i odwoływania się do jedno- i wielowymiarowych tablic.

Nawiasy okrągłe (*parentheses*) `( )` wykorzystywane są do grupowania wyrażeń, izolowania wyrażeń warunkowych, wskazują wywołanie funkcji i jej parametry.

Nawiasy klamrowe (*braces*) `{ }` oznaczają początek i koniec instrukcji złożonej, (blok).

Przecinek (*comma*) `,` rozdziela zwykle elementy na liście parametrów funkcji, występuje również w wyrażeniach przecinkowych.

Średnik (*semicolon*) `;` jest znakiem kończącym instrukcję. Każde wyrażenie w języku C (również wyrażenie puste) zakończone znakiem średnika jest interpretowane jako instrukcja wyrażeniowa.

Komentarze

Komentarze to fragmenty tekstu spełniające funkcje dowolnych objaśnień robionych przez programistów dla programistów.

Komentarze nie mogą występować w napisach i stałych znakowych. Komentarze są usuwane z tekstu źródłowego programu.

```
/* To jest komentarz jednoliniowy */

/*
Ten komentarz obejmuje
kilka linii
kodu
*/

// Komentarz jednowierszowy, bez znacznika konca
```

Typy zmiennych w języku C++

typ	wielkość	charakterystyka
`bool`	-	wartość logiczna
`char`	1	znak, liczba całkowita od `0` do `255`
`w_chart`	2	znak poza ASCII
`chart8_t`	1	znak, typ dedykowany do kodowania UTF-8
`chart16_t`	2	znak, typ dedykowany do kodowania UTF-16
`chart32_t`	4	znak, typ dedykowany do kodowania UTF-32
`int`	4	liczba całkowita
`float`	4	liczba rzeczywista
`double`	8	liczba rzeczywista długa

Modyfikatory: `signed`, `unsigned`, `short`, `long` znane z języka C mogą być analogicznie stosowane w języku C++.

Type	Size in bits	Format	Value range	
			Approximate	Exact
character	8	signed		−128 to 127
		unsigned		0 to 255
	16	UTF-16		0 to 65535
	32	UTF-32		0 to 1114111 (0x10ffff)
integer	16	signed	$\pm 3.27 \cdot 10^4$	−32768 to 32767
		unsigned	0 to $6.55 \cdot 10^4$	0 to 65535
	32	signed	$\pm 2.14 \cdot 10^9$	−2,147,483,648 to 2,147,483,647
		unsigned	0 to $4.29 \cdot 10^9$	0 to 4,294,967,295
	64	signed	$\pm 9.22 \cdot 10^{18}$	−9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
		unsigned	0 to $1.84 \cdot 10^{19}$	0 to 18,446,744,073,709,551,615

Type	Size in bits	Format	Value range	
			Approximate	Exact
binary floating-point	32	IEEE-754	• min subnormal: $\pm\ 1.401,298,4 \cdot 10^{-45}$ • min normal: $\pm\ 1.175,494,3 \cdot 10^{-38}$ • max: $\pm\ 3.402,823,4 \cdot 10^{38}$	• min subnormal: ±0x1p−149 • min normal: ±0x1p−126 • max: ±0x1.fffffep+127
	64	IEEE-754	• min subnormal: $\pm\ 4.940,656,458,412 \cdot 10^{-324}$ • min normal: $\pm\ 2.225,073,858,507,201,4 \cdot 10^{-308}$ • max: $\pm\ 1.797,693,134,862,315,7 \cdot 10^{308}$	• min subnormal: ±0x1p−1074 • min normal: ±0x1p−1022 • max: ±0x1.fffffffffffffp+1023
	80	x86	• min subnormal: $\pm\ 3.645,199,531,882,474,602,528 \cdot 10^{-4951}$ • min normal: $\pm\ 3.362,103,143,112,093,506,263 \cdot 10^{-4932}$ • max: $\pm\ 1.189,731,495,357,231,765,021 \cdot 10^{4932}$	• min subnormal: ±0x1p−16445 • min normal: ±0x1p−16382 • max: ±0x1.ffffffffffffffep+16383
	128	IEEE-754	• min subnormal: $\pm\ 6.475,175,119,438,025,110,924,438,958,227,646,552,5 \cdot 10^{-4966}$ • min normal: $\pm\ 3.362,103,143,112,093,506,262,677,817,321,752,602,6 \cdot 10^{-4932}$ • max: $\pm\ 1.189,731,495,357,231,765,085,759,326,628,007,016,2 \cdot 10^{4932}$	• min subnormal: ±0x1p−16494 • min normal: ±0x1p−16382 • max: ±0x1.ffffffffffffffffffffffffffffp+16383

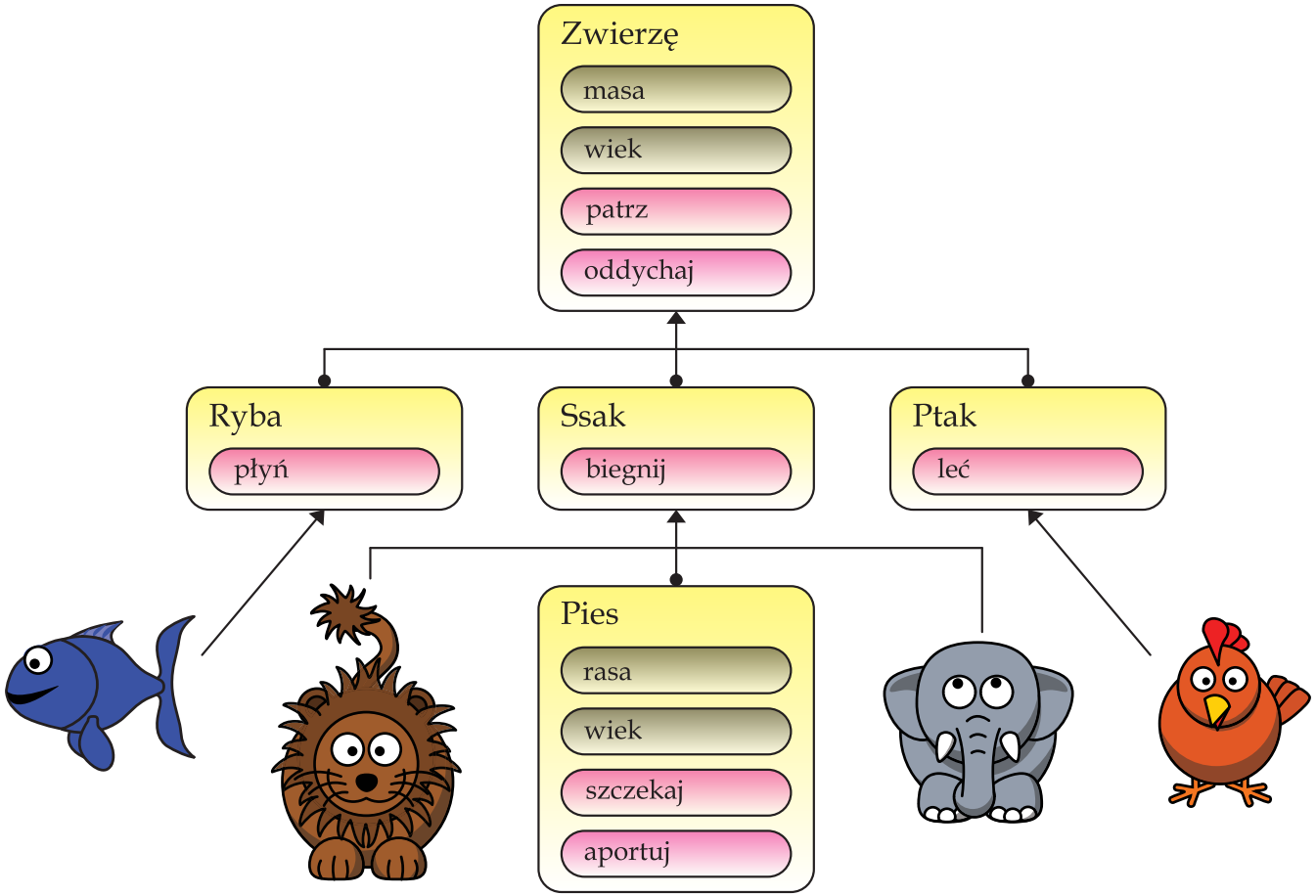
Operatory (arytmetyczne, logiczne etc.), **wyrażenia** (i instrukcje wyrażeniowe), **instrukcje** (pętle, sterowanie etc.), zasady definiowania zmiennych, zasadny **definiowania funkcji**, sposób definiowania i obsługi **tablic** są takie same jak w języku C.

Zapominalskich odsyłam do wykładu z **Podstaw programowania**.

Największym atutem programowania obiektowego jest zbliżenie programów komputerowych do naszego sposobu postrzegania rzeczywistości. Często nazywa się to zmniejszeniem **luki reprezentacji**.

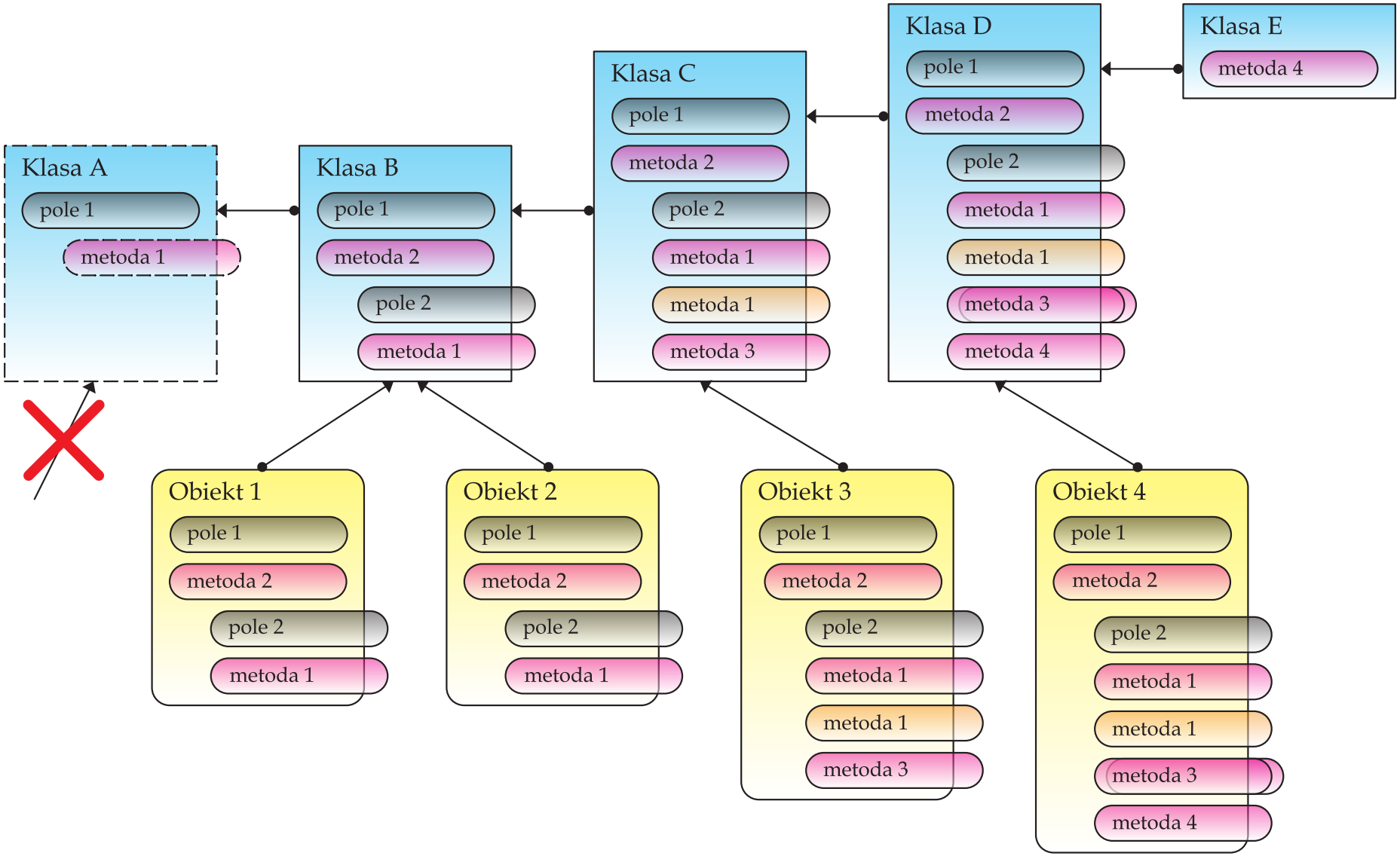
Wymyślając nowy lub analizując istniejący program obiektowy nasz mózg ma ułatwione zadanie. Dlatego ludzie są w stanie łatwiej zapanować nad kodem i tym samym tworzyć większe programy. Łatwiej jest również zrozumieć kod i pomysły innych programistów i tym samym współpracować w zespole oraz ponownie wykorzystywać istniejące rozwiązania.

Co więcej tego samego, naturalnego dla ludzi sposobu myślenia i tych samych pojęć można użyć przy analizie problemu, który ma być rozwiązany i projektowaniu jego programowego rozwiązania.

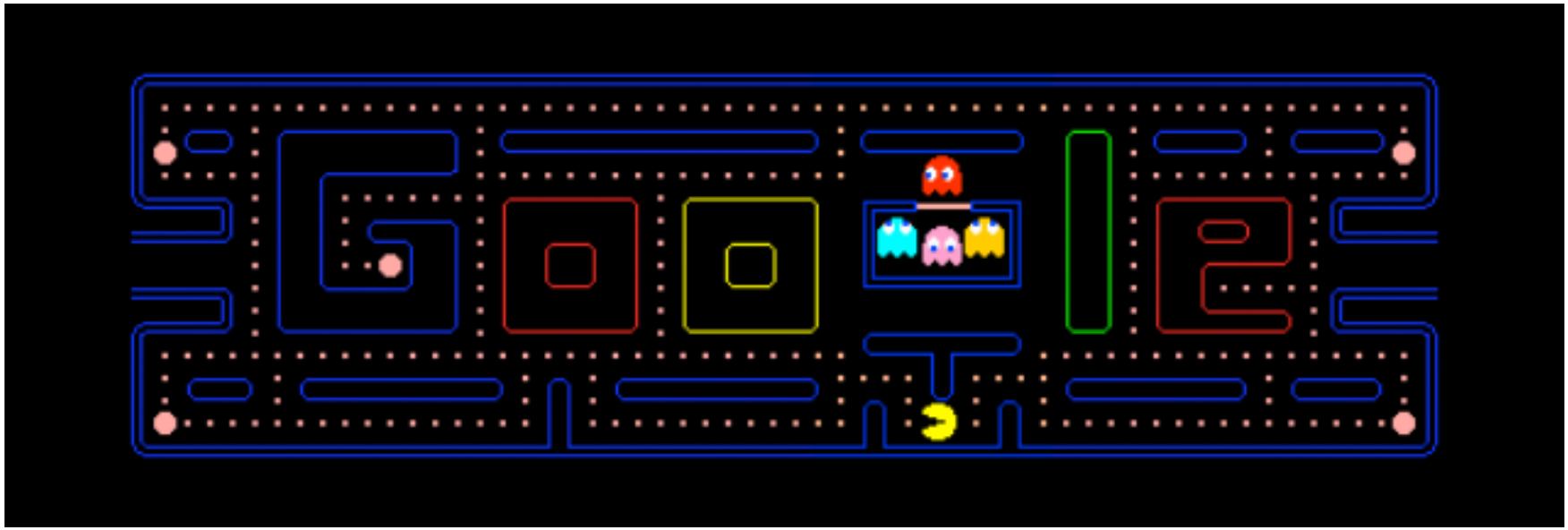

Zwierzę
masa
wiek
patrz
oddychaj
Ryba
płyń
Ssak
biegnij
Ptak
leć
Pies
rasa
wiek
szczekaj
aportuj

Arystoteles analizując rzeczywistość wprowadził pojęcia **formy** i **materii**. Formie w programowaniu obiektowym odpowiada **klasa** (*class*), materii jej **egzemplarz** (*instance*) wymiennie nazywany **obiektem** (*object*).

Klasyfikacja, czyli łączenie występujących w rzeczywistości obiektów w grupy – klasy, jest najbardziej naturalnym sposobem rozumienia rzeczywistości.


Klasa A
pole 1
metoda 1
Klasa B
pole 1
metoda 2
pole 2
metoda 1
Klasa C
pole 1
metoda 2
pole 2
metoda 1
metoda 1
metoda 3
Klasa D
pole 1
metoda 2
pole 2
metoda 1
metoda 1
metoda 3
metoda 4
Klasa E
metoda 4
Obiekt 1
pole 1
metoda 2
pole 2
metoda 1
Obiekt 2
pole 1
metoda 2
pole 2
metoda 1
Obiekt 3
pole 1
metoda 2
pole 2
metoda 1
metoda 1
metoda 3
Obiekt 4
pole 1
metoda 2
pole 2
metoda 1
metoda 1
metoda 3
metoda 4

Problem – Napisać program pozwalający na sterowanie Pacmanem (klasyczna gra), w najprostszej implementacji konsolowej. Powinien pozwalać na wyświetlanie wybranego znaku i sterowaniu przy użyciu klawiszy kursorów.



Pozycja dowolnego, ustalonego wcześniej znaku o kodzie code będzie opisana za pomocą współrzędnych x i y. Przesuwanie znaku polega na wymazywaniu znaku z poprzedniej pozycji i ponownym wyświetlaniu na nowej pozycji. Pozycja powinna być sprawdzana.

Realizacja proceduralna

Programista ma do dyspozycji:

- funkcję `clearScreen()`, czyszczącą ekran,
- funkcję `writeChar()`, wyświetlającą na ekranie znak na określonej pozycji,
- funkcję `getKey()`, której rezultatem jest kod klawisza sterującego (np. zmienna wyliczeniowa z wartościami typu: `KEY_ESC`, `KEY_UP`, `KEY_DOWN`, itd.).

```
int key, x = 1, y = 1, code = '*';
clearScreen();
do
  {
  writeChar(x, y, code);  ← wyświetl znak na zadanej pozycji

  key = getKey();         ← obsługa naciśnięcia klawisza
  writeChar(x, y, ' ');   ← nadpisanie znaku
  switch(key)
  {
    case KEY_UP:
      if(y > 1) --y;      ← idź wyżej
      break;
    case KEY_DOWN:
      if(y < 24) ++y;     ← idź niżej
      break;
    case KEY_LEFT:
      if(x > 1) --x;      ← idź w lewo
      break;
    case KEY_RIGHT:
      if(x < 80) ++x;     ← idź w prawo
      break;
  }
}
while(key != KEY_ESC);
```

Realizacja proceduralna charakteryzuje się wadami:

- brak powiązania pomiędzy zmiennymi opisującymi ten sam obiekt,
- brak powiązania pomiędzy funkcjami wyświetlającą i ukrywającą znak (prawidłowość parametrów),
- trudność z odczytaniem przeznaczenia kodu,
- brak powiązań między danymi a akcjami na rzecz obiektu.

Realizacja obiektowa

```
TPacman pac;
pac.x = 1; pac.y = 1; pac.code = '*';
clearScreen();
pac.show();
do
{
  switch(key = getKey())
  {
    case KEY_UP:
      pac.moveUp();
      break;
    case KEY_DOWN:
      pac.moveDown();
      break;
    case KEY_LEFT:
      pac.moveLeft();
      break;
    case KEY_RIGHT:
      pac.moveRight();
      break;
  }
}
while(key != KEY_ESC);
```

TPacman to nazwa **klasy** obiektów. Wszystkie obiekty tej klasy będą posiadały te same cechy i umiejętności.

Wszystkie informacje o obiekcie pamiętane są w jego **polach**. Polami klasy TPacman są x, y i code.

```
class TPacman
{
  public:
    int x;      ← pola
    int y;      ←
    int code;   ←

    void show();
    void hide();

    void moveUp();
    void moveDown();
    void moveLeft();
    void moveRight();
};
```

Oprócz pól, obiekt posiada także zaimplementowane funkcje będące jego składnikami. Funkcje te są nazywane **metodami** i są prototypowane w definicji klasy.

```
class TPacman
{
  public:
    int x;
    int y;
    int code;

    void show();  ← metody
    void hide();  ←

    void moveUp();      ← metody
    void moveDown();    ←
    void moveLeft();    ←
    void moveRight();   ←
};
```

Definicje metod umieszczane są zwykle na zewnątrz definicji klasy. Wtedy nazwa każdej metody jest kwalifikowana nazwą klasy.

```
void TPacman::show()
{
  writeChar(x, y, code);
}
```

```
void TPacman::hide()
{
  writeChar(x, y, ' ');
}
```

```
void TPacman::moveUp()
{
  hide();

  if(y > 1)
    --y;

  show();
}
```

```
void TPacman::moveDown()
{
  hide();

  if(y < 24)
    ++y;

  show();
}
```

```
void TPacman::moveLeft()
{
  hide();

  if(x > 1)
    --x;

  show();
}
```

```
void TPacman::moveRight()
{
  hide();

  if(x < 80)
    ++x;

  show();
}
```

Klasa określa właściwości i możliwości każdego obiektu.

W szczególności klasa zawiera specyfikację danych i metod, informację o typie i reprezentacji pól, implementację metod, informacje o powiązaniach z innymi klasami.

Obiekt jest abstrakcją pewnego konkretnego bytu ze świata rzeczywistego, reprezentującego **rzecz**, **pojęcie**, **zjawisko** lub pewny **byt programistyczny** nie mającey swojego odpowiednika w rzeczywistości.

Obiekt jet **instancją** klasy.

Podstawowymi cechami obiektu są:

- Tożsamość (*identity*).
- Stan (*state*).
- Zachowanie, działanie (*behavior*).

Tożsamość – pozwala na odróżnienie danego obiektu od innego. Programista przypisuje obiektowi **unikatową nazwę**, lub odwołuje się do obiektu za pomocą jego adresu w pamięci (wskaźnika).

Stan – opisuje obiekt. Stan obiektu jest opisany zestawem atrybutów i ich wartości. Atrybuty są wewnętrznymi danymi obiektu, czyli **polami** (*fields*. Wartości atrybutów mogą się zmieniać w czasie wykonania programu.

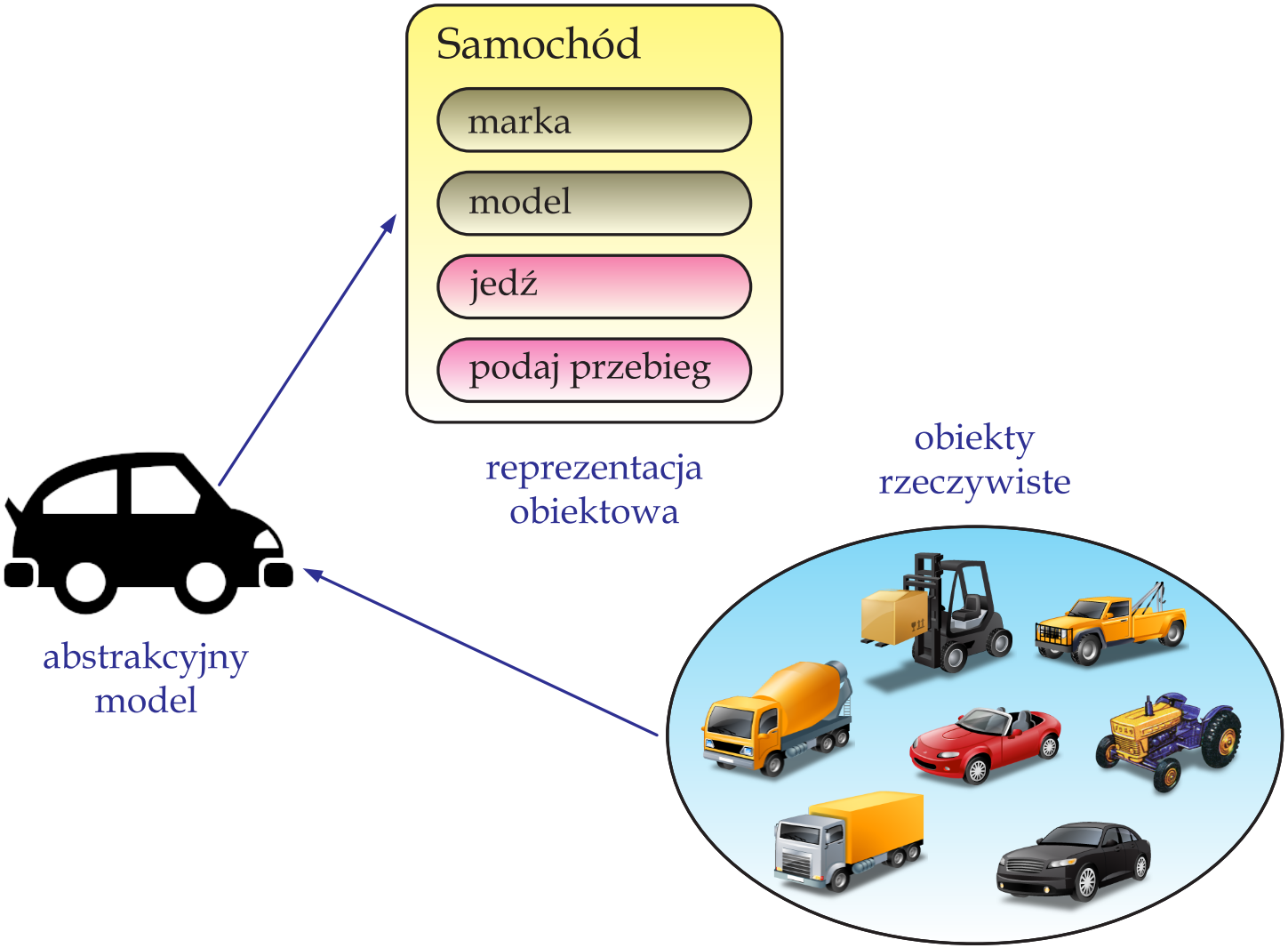
Zachowanie – zdolność do działania opisują operacje, które obiekt może wykonać. operacje są realizowane przez wykonanie **metod** (*methods*), czasem zwanych funkcjami składowymi (*member functions*).Metody mogą otrzymywać parametry i zwracać wartości, zwykle zmieniają stan obiektu. Metody są częścią obiektu.

Powszechnie uważa się, że najważniejszymi cechami programowania obiektowego są:

Abstrakcja (*abstraction*)
Każdy obiekt w systemie służy jako model abstrakcyjnego *wykonawcy* (bytu ze świata rzeczywistego), który może wykonywać pracę, opisywać i zmieniać swój stan oraz komunikować się z innymi obiektami w systemie.

Abstrakcja pozwala postrzegać modelowaną rzeczywistość bez wnikania w jej wewnętrzną struktur i bez ujawniania, w jaki sposób zaimplementowano dane cechy.

Obiekt jest dostawcą pewnych **informacji** i **usług**. Nie musimy wiedzieć jak obiekt coś robi, tylko co robi. Nie musimy też wiedzieć jak obiekt coś przechowuje, tylko co przechowuje.


Samochód
marka
model
jedź
podaj przebieg
reprezentacja obiektowa
obiekty rzeczywiste
abstrakcyjny model

```
class TPacman
{
  public:
    int x;
    int y;
    int code;

    void show();
    void hide();

    void moveUp();
    void moveDown();
    void moveLeft();
    void moveRight();
};
```

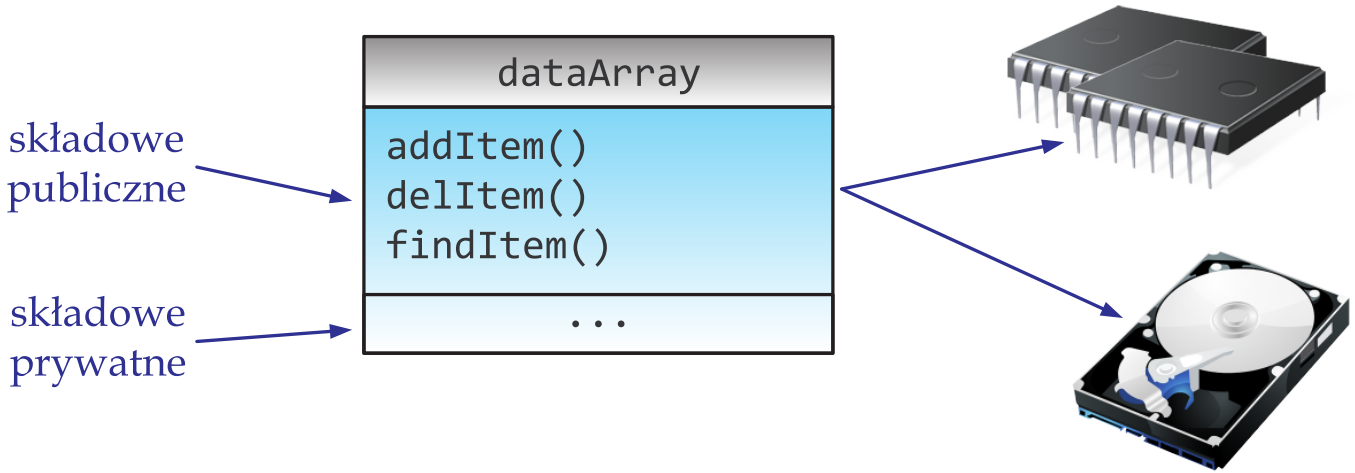
Enkapsulacja

Czyli ukrywanie implementacji, hermetyzacja. Zapewnia, że obiekt nie może zmieniać stanu wewnętrznego innych obiektów w nieoczekiwany sposób. Tylko własne metody obiektu są uprawnione do zmiany jego stanu.

Każdy typ obiektu prezentuje innym obiektom swój interfejs, który określa dopuszczalne metody współpracy.

Pewne języki osłabiają to założenie, dopuszczając pewien poziom bezpośredniego (kontrolowanego) dostępu do wnętrza obiektu. Ograniczają w ten sposób poziom abstrakcji.

Zmiana implementacji powinna być dla użytkownika klasy przezroczysta.




```
TPacman pac;

pac.x = 40;
pac.y = 12;
pac.code = '*';

cout << pac.x << endl;
cout << pac.y << endl;
cout << pac.code << endl;
```

```
TPacman pac;

pac.setX(40);
pac.setY(12);
pac.setCode('*');

cout << pac.getX() << endl;
cout << pac.getY() << endl;
cout << pac.getCode() << endl;
```

```
class TPacman
{
  public:
    int x, y, code;

    void show();
    void hide();

    void moveUp();
    void moveDown();
    void moveLeft();
    void moveRight();
};
```

```
class TPacman
{
  public:
    void setX(int newX);
    void setY(int newY);
    void setCode(int newCode);

    int getX();
    int getY();
    int getCode();

    void show();
    void hide();

    void moveUp();
    void moveDown();
    void moveLeft();
    void moveRight();

  private:
    int x, y, code;
};
```

Enkapsulacja pozwala na kontrolowanie poprawności danych przekazywanych obiektowi.

```
TPacman pac;

pac.setCode('*');
pac.setX(-400);
pac.setY(1200);

pac.show();
```

```
void TPacman::setX(int newX)
{
  if(isXOnScreen(newX))
    x = newX;
  else
  x = 1;  ← wartość domyślna
}
```

```
void TPacman::setY(int newY)
{
  if(isYOnScreen(newY))
    y = newY;
  else
    y = 1;  ← wartość domyślna
}
```

Opracujemy obiektową wersję programu do obliczania pól powierzchni płaskich figur geometrycznych.

Zacznijmy od klasy `TSquare`. Jej najistotniejszą cechą (*zasada abstrakcji*) będzie długość boku `side`.

Stosując **zasadę hermetyzacji** ukrywamy dane w części prywatnej i tworzymy metody zapewniające dostęp do pól danych. Przykład użycia klasy:

```
TSquare s;

s.setSide(10);
cout << s.getSide();
cout << s.area();
```

Metody klasy będziemy nieformalnie dzielić na:

- **akcesory** – funkcje umożliwiające pobieranie wartości pól,
- **modyfikatory** – funkcje umożliwiające zmianę wartości pól,
- **realizatory** – funkcje umożliwiające realizację usług na rzecz klasy.

Deklaracja klasy:

```
class TSquare
{
  public:
    void setSide(double newSide);
    double getSide();
    double area();

  private:
    double side;
};
```

Definicja metod (poza deklaracją klasy z użyciem operatora dostępu `::`):

```
void TSquare::setSide(double newSide)
{
  side = newSide;
}

double TSquare::getSide()
{
  return side;
}

double TSquare::area()
{
  return side * side;
}
```

Konstruktory

Próba użycia obiektu `TSquare` bez ustalonej wielkości boku będzie przynosiła nieprzewidywalne efekty:

```
TSquare s;

cout << a.area();
```

Aby zapobiec takim przypadkom, należy zdefiniować **konstruktor** – czyli specjalną metodę, która przygotowuje obiekt do jego użycia.

Konstruktor jest uruchamiany automatycznie przez kompilator. Nie ma zdefiniowanego typu rezultatu i nosi taką samą nazwę jak klasa.

Rodzaje konstruktorów:

- **konstruktor domyślny** (*default constructor*) – aktywowany, gdy tworzony jest obiekt bez jawnie określonych danych inicjalizujących,

```
TSquare s;
```

- **konstruktor ogólny** lub parametrowy (*general constructor*) – aktywowany, gdy tworzony obiekt ma jawnie koreślone dany inicjalizujące,

```
TSquare s(10);
```

- **konstruktor kopiujący** (*copy constructor*) – aktywowany, gdy tworzony jest obiekt inicjalizowany danymi z innego obiektu **tej samej klasy**,

```
TSquare a = s, b(s);
```

- **konstruktor rzutujący** (*cast constructor*) – aktywowany, gdy tworzony jest obiekt inicjalizowany danymi z innego obiektu **innej klasy**.

```
TSquare s(10);
TRectangle a = s, b(s);
```

Deklaracja domyślnego konstruktora:

```
class TSquare
{
  public:

    TSquare();

    void setSide(double newSide);
    double getSide();
    double area();

  private:
    double side;
};
```

Kontekst aktywowania konstruktora domyślnego:

```
TSquare a;

TSquare squares[5];
```

Intuicyjna realizacja konstruktora:

```
TSquare::TSquare()
{
  side = 1;
}
```

Realizacja konstruktora z listą inicjalizacyjną:

```
TSquare::TSquare(): side(1)
{
}
```

Listę inicjalizacyją podaje się po znaku dwukropka `:`, najpierw podajemy **nazwę pola**, a w nawiasach wyrażenie inicjalizujące (wartość pola).

Deklaracja konstruktora ogólnego:

```
class TSquare
{
  public:

    TSquare();
    TSquare(double defaultSide);

    void setSide(double newSide);
    double getSide();
    double area();

  private:
    double side;
};
```

Kontekst aktywowania konstruktora ogólnego:

```
TSquare a(2);
```

Intuicyjna realizacja konstruktora:

```
TSquare::TSquare(double defaultSide)
{
  side = defaultSide;
}
```

Realizacja konstruktora z listą inicjalizacyjną:

```
TSquare::TSquare(double defaultSide): side(defaultSide)
{
}
```

Konstruktora ogólny pozwala na inicjalizowanie obiektu z modyfikatorem `const`:

```
const TSquare cs;

cs.setSide(3);  ← błąd
```

```
const TSquare cs(3);
```

Metody klasy mogą być uzupełnione modyfikatorem `const`, wtedy nie będą mogły modyfikować obiektu, ale będą mogły być wywoływane na rzecz obiektów stałych:

```
class TSquare
{
  public:

    TSquare();
    TSquare(double defaultSide);

    void setSide(double newSide);
    double getSide() const;
    double area() const;

  private:
    double side;
};
```

```
double TSquare::getSide() const
{
  return side;
}
double TSquare::area() const
{
  return side * side;
}
```

Wykorzystanie operatora zakresu ::

Czy parametr funkcji `setSide( )` może się nazywać `side`? Zamiast:

```
void TSquare::setSide(double newSide)
{
  side = newSide;
}
```

chcielibyśmy:

```
void TSquare::setSide(double side)
{
  side = side;  ← ???
}
```

Oczywiście, taki zapis jest błędny, ponieważ parametr formalny funkcji przesłania w jej ciele pole gdy mają takie same nazwy.

W takim przypadku można użyć operatora zakresu:

```
void TSquare::setSide(double side)
{
  TSquare::side = side;
}
```

Podobny problem występuje w konstruktorze:

```
void TSquare::setSide(double newSide)
{
  side = newSide;
}
```

W takim przypadku można użyć operatora zakresu:

```
TSquare::TSquare(double side)
{
  TSquare::side = side;
}
```

Problem z przesłanianiem pól nie występuje, kiedy zastosujemy listę inicjalizującą:

```
TSquare::TSquare(double side): side(side)
{
}
```

Przeciążanie funkcji – dygresja

W obiekcie `TSquare` istnieją dwa identyfikatory `TSquare()`:

- konstruktor domyślny `TSquare()`,
- konstruktor ogólny `TSquare(double defaultSide)`.

Identyfikator `TSquare()` jest **przeciążony** (przeładowany). Formalnie, funkcja jest przeciążona, jeśli istnieje inna funkcja o tej samej nazwie ale różnej **sygnaturze**.

Sygnatura funkcji jest to nazwa funkcji, liczba argumentów, uporządkowany zbiór typów argumentów funkcji, klasa lub obszar nazw do którego funkcja należy.

Przeciążone funkcje i operatory nazywamy **polimorficznymi**

Przykład funkcji przeciążonych:

```
int add(int a, int b)
{
  return a + b;
}

double add(double a, double b)
{
  return a + b;
}

cout << add(1, 1);
cout << add(1.0, 1.0);
```

Deklaracja konstruktora kopiującego:

```
class TSquare
{
  public:

    TSquare();
    TSquare(double defaultSide);
    TSquare(TSquare &otherSquare);

    void setSide(double newSide);
    double getSide();
    double area();

  private:
    double side;
};
```

Kontekst aktywowania konstruktora kopiującego:

```
TSquare a(2);

TSquare b = a;
TSquare c(a);
```

Podobnie, dla typów prostych: `int i = 1, j = i;`

Intuicyjna realizacja konstruktora:

```
TSquare::TSquare(TSquare &otherSquare)
{
  side = otherSquare.side;
}
```

Realizacja konstruktora z listą inicjalizacyjną:

```
TSquare::TSquare(TSquare &otherSquare): side(otherSquare.side)
{
}
```

Problem z `const`:

```
const TSquare a(2);
TSquare b = a;  ← błąd, niejawna referencja do obiektu const
```

Rozwiązanie problemu z `const`:

```
TSquare::TSquare(const TSquare &otherSquare): side(otherSquare.side)
{
}
```

Mamy klasę od opisu prostokąta:

```
class TRectangle
{
  public:
    TRectangle();
    TRectangle(double width, double height);
    TRectangle(const TRectangle &otherRectangle);

    void setWidth(double width);
    void setHeight(double height);

    double getWidth() const;
    double getHeight() const;

    double area() const;

  private:
    double width, height;
};
```

Definicja konstruktorów i realizatora:

```
TRectangle::TRectangle()
: width(1), height(1)
{
}

TRectangle::TRectangle(double width, double height)
: width(width), height(height)
{
}

TRectangle::TRectangle(const Rectangle &otherRectangle)
: width(otherRectangle.width), height(otherRectangle.height)

double TRectangle::area() const
{
  return width * height;
}
```

Definicja modyfikatorów i akcesorów:

```
void TRectangle::setWidth(double width)
{
  TRectangle::width = width;
}

void TRectangle::setHeight(double height)
{
  Rectangle::height = height;
}

double TRectangle::getWidth() const
{
  return width;
}

double Rectangle::getHeight() const
{
  return height;
}
```

Definicja konstruktora rzutującego:

```
TRectangle::TRectangle(const TSquare &square)
: width(square.getSide()), height(square.getSide())
{
}
```

Konstruktor rzutujący odpowiedzialny jest za skopiowanie zawartości obiektu pewnej klasy do obiektu innej klasy na etapie inicjalizacji.

Kontekst aktywowania konstruktora rzutującego:

```
TSquare s(10);

TRectangle r(s);
```

Parametr domyślny to wartość określona na etapie **deklaracji funkcji**, która zostanie automatycznie wstawiona do parametru formalnego, jeżeli dana funkcja zostanie wywołana bez odpowiedniego parametru aktualnego.

Parametry domyślne można stosować z metodami klas jak i funkcji niezwiązanych z klasami.

Parametry domyślne muszą być zdefiniowane od końca listy parametrów.

```
void foo(int i, float f = 0, char c = 'A');

foo(10);
foo(20, 3.15);
foo(30, 22.1, 'Z');
```

W przypadku stosowania prototypów funkcji, wartości parametrów domyślnych określa się w prototypie.

```
void foo(int i, float f = 0, char c = 'A');
```

W definicji funkcji parametry domyślne nie występują.

```
void fun(int i, float f, char c)
{
}
```

Konstruktor domyślny jest **bezparametrowy** lub posiada wszystkie parametry będące **parametrami domyślnymi**.

Jednoczesne wystąpienie obu form spowoduje błąd kompilacji.

```
class TSquare
{
  public:

    TSquare(double defaultSide = 1);

    void setSide(double newSide);
    double getSide( );
    double area( );

  private:
    double side;
};
```

```
TSquare::TSquare(double defaultSide): side(defaultSide)
{
}
```

Porządkuje i wspomaga polimorfizm i enkapsulację dzięki umożliwieniu definiowania i tworzenia specjalizowanych obiektów na podstawie bardziej ogólnych.

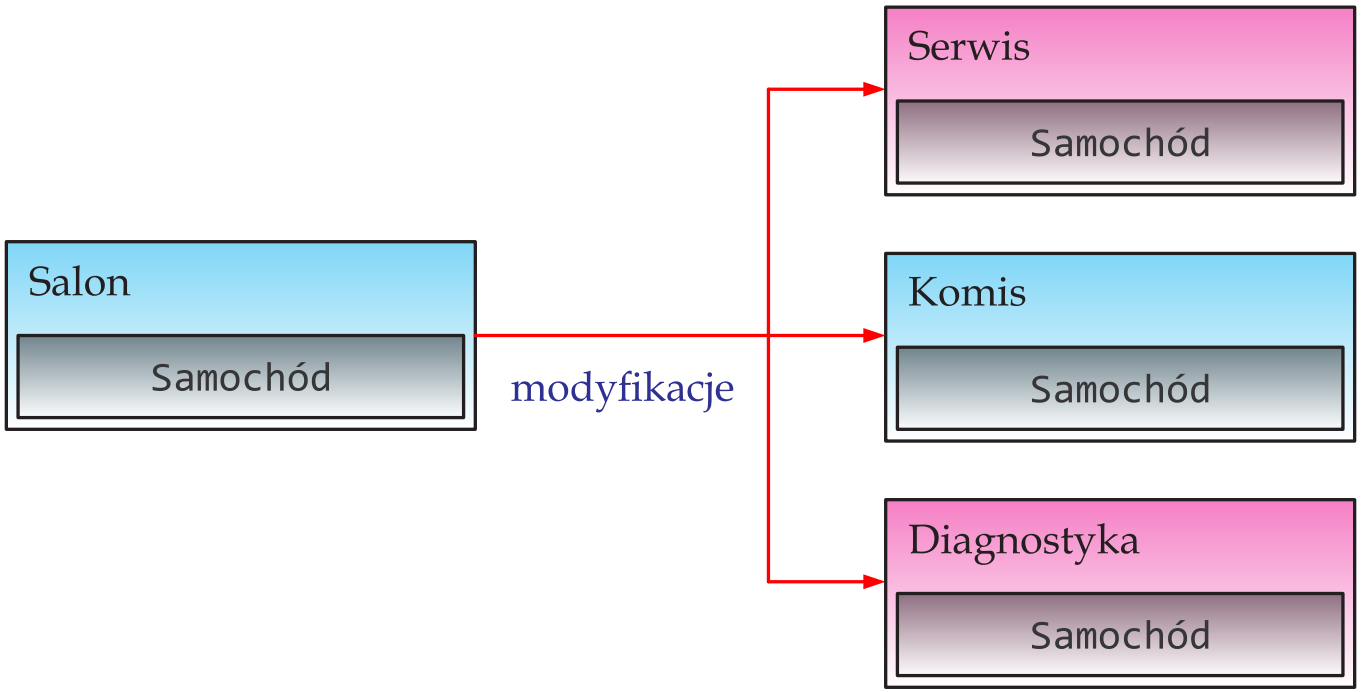
Dla obiektów specjalizowanych nie trzeba redefiniować całej funkcjonalności, lecz tylko tę, której nie ma obiekt ogólniejszy. W typowym przypadku powstają grupy obiektów zwane klasami, oraz grupy klas zwane drzewami. Odzwierciedlają one wspólne cechy obiektów.

Dziedziczenie to forma **ponownego wykorzystania** (*reuse*) i pozwala na:

- zwiększenie wydajności procesu programowania,
- poprawia jakość i niezawodność kodu, ułatwia rozwój i pielęgnację kodu.

Problemy spotykane przy ponownym wykorzystaniu kodu:

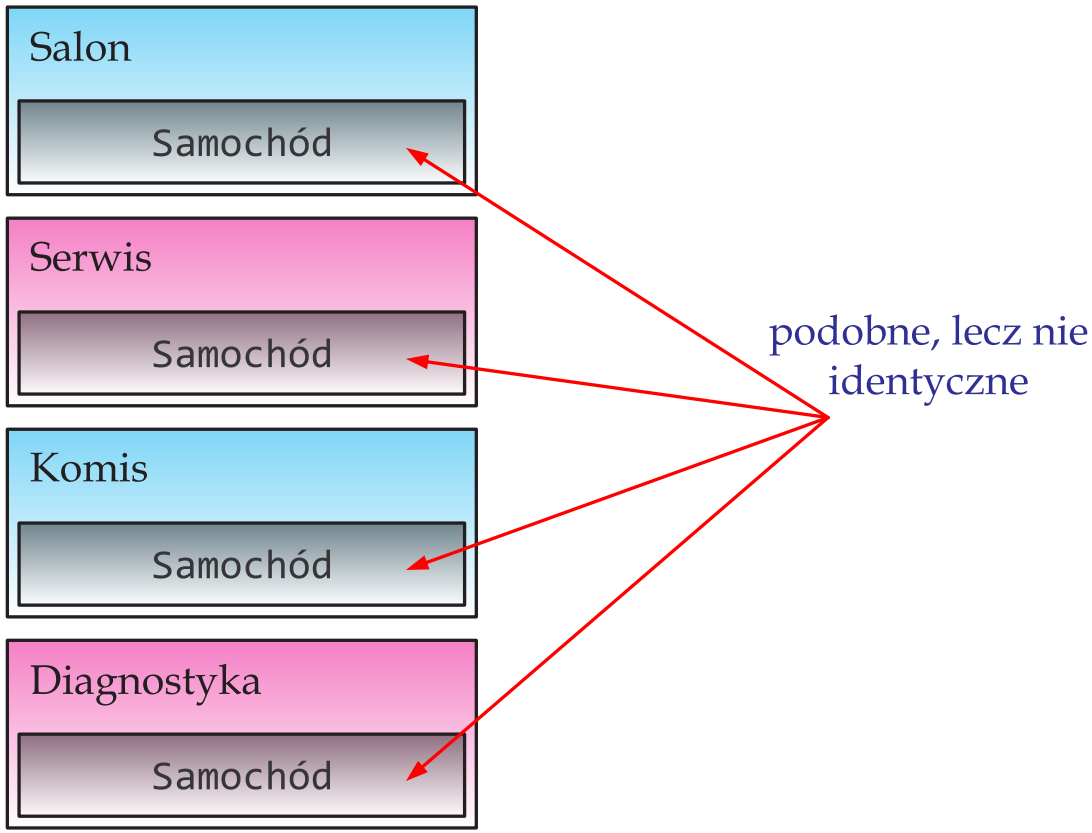
- powtarzalność nieidentycznych obiektów powoduje to, że musimy modyfikować istniejący kod, dostosowując go do specyfiki projektu,
- modyfikacje potrzebne z punktu widzenia jednego systemu mogą być nieużyteczne lub nieakceptowalne z punktu widzenia innego systemu,
- modyfikacje dostosowujące kod do specyfiki systemu zmuszają do utrzymywania różnych wersji tego samego kodu.



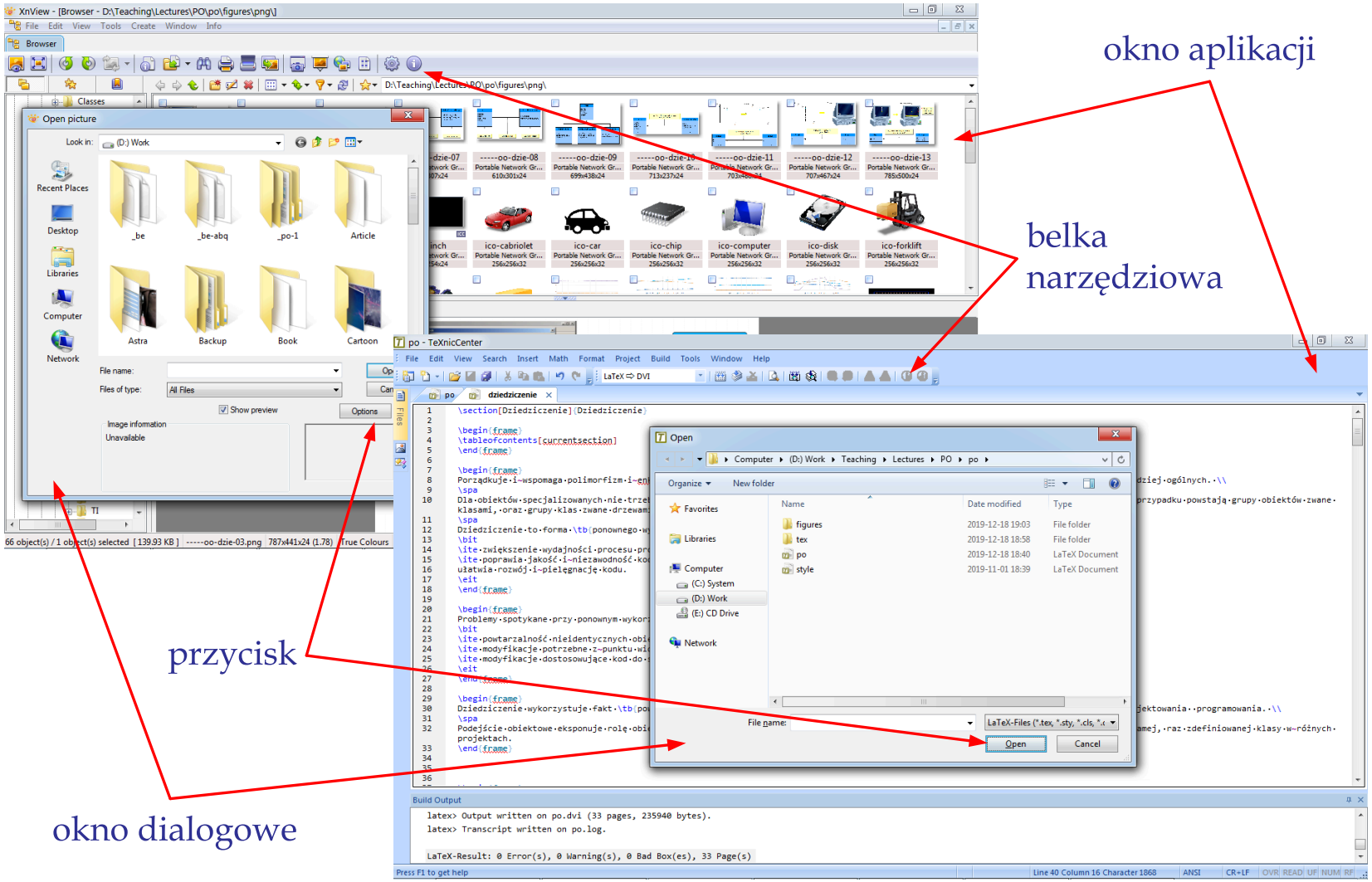
Dziedziczenie wykorzystuje fakt **powtarzalności elementów oprogramowania**. Powtarzalność taką można obserwować na etapie analizy, projektowania programowania.

Podejście obiektowe eksponuje rolę obiektu. W programowaniu obiektowym ponowne wykorzystanie kodu polega na wykorzystaniu obiektów tej samej, raz zdefiniowanej klasy w różnych projektach.

W obrębie obiektów dziedziny problemu powtarzają się obiekty podobne lecz bardzo rzadko identyczne.

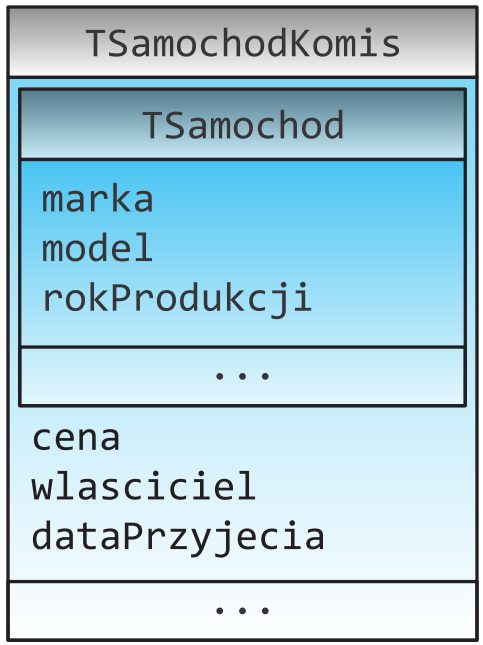


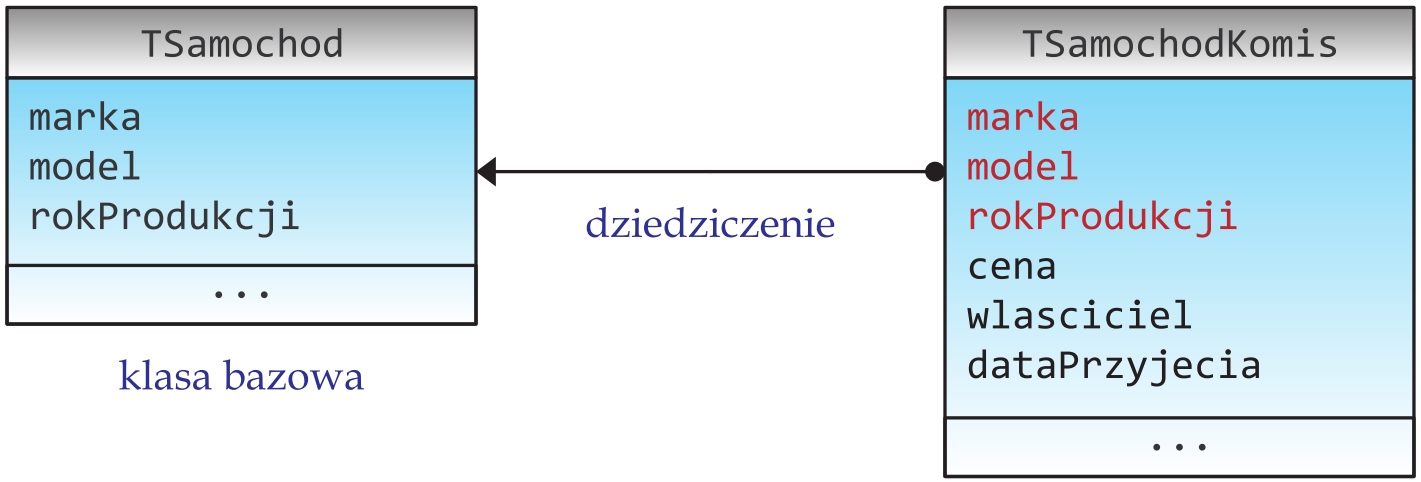
W obrębie obiektów implementacyjnych powtarzalność obiektów identycznych jest znacznie częstsza, co jednak nie jest regułą.

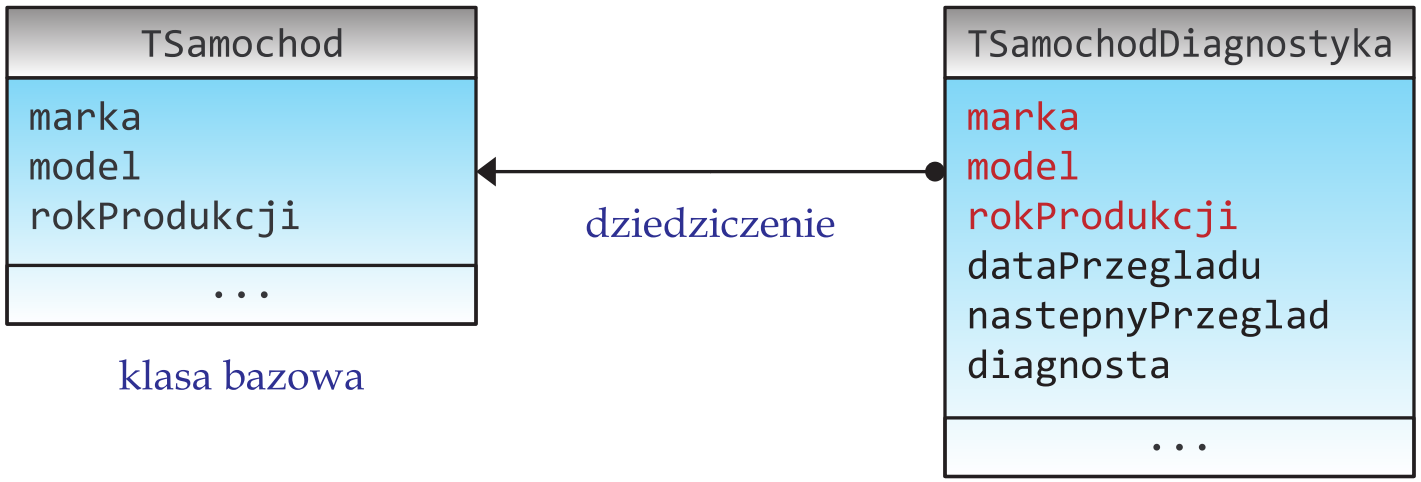


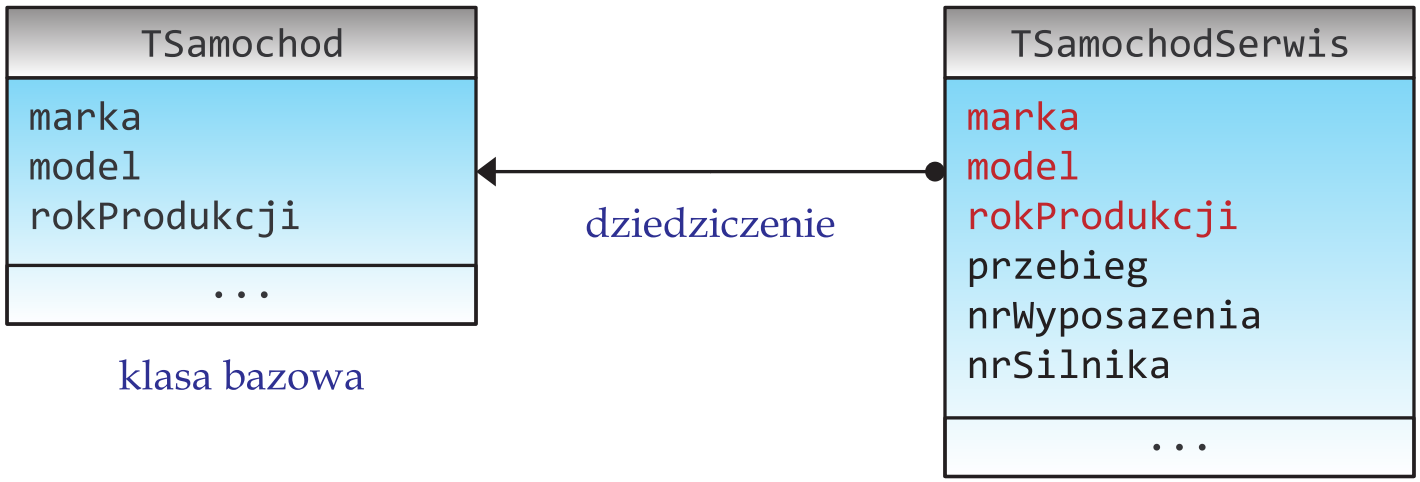
Dziedziczenie polega na **budowaniu nowych klas**, zwanych **klasami potomnymi**, na podstawie **klas już istniejących**, zwanych **klasami bazowymi**.

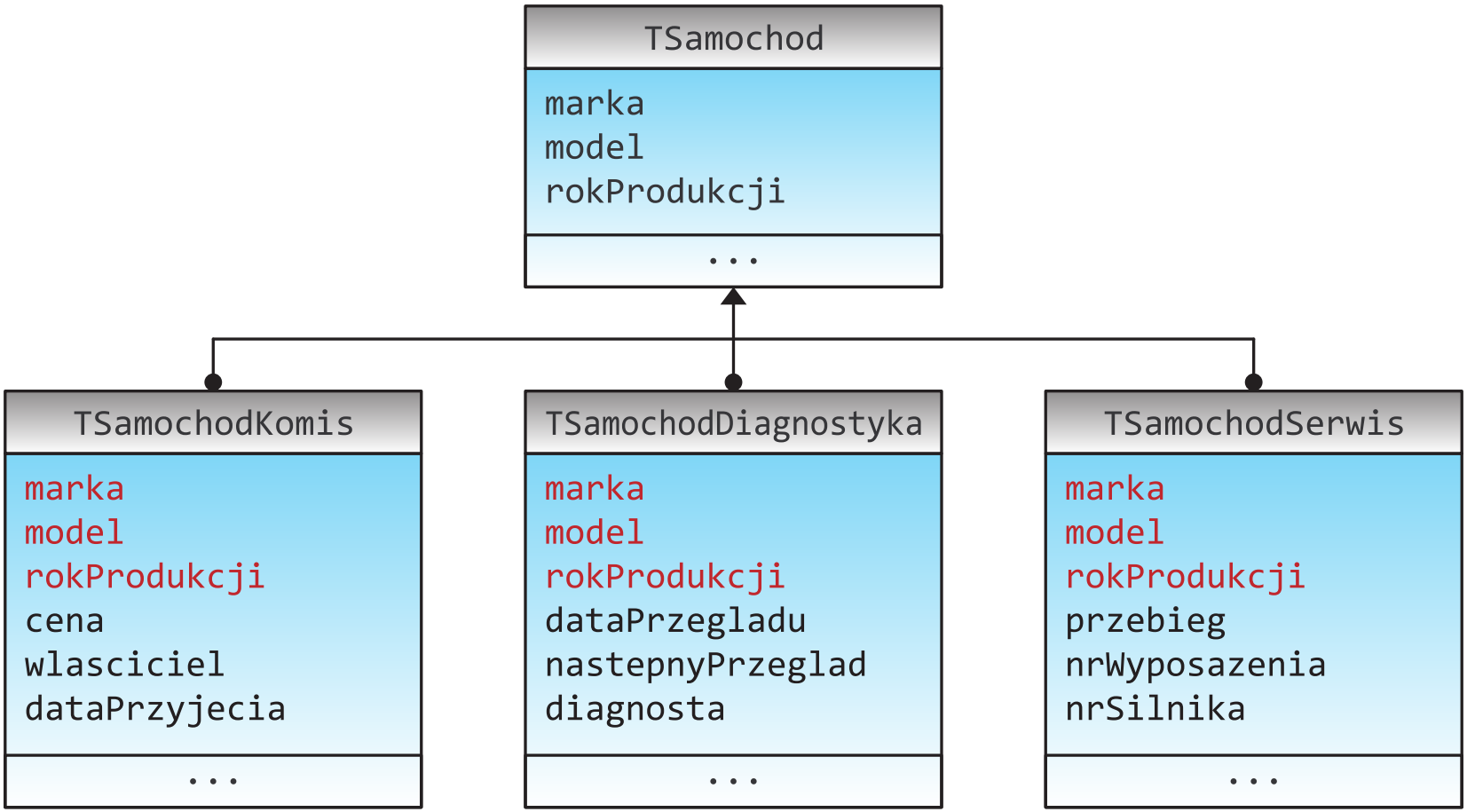
Każda klasa pochodna **dziedziczy wszystkie właściwości** klasy bazowej, **rozszerzając je o nowe** pola i metody.




TSamochod
marka
model
rokProdukcji
...
klasa bazowa
dziedziczenie
TSamochodKomis
marka
model
rokProdukcji
cena
wlasciciel
dataPrzyjecia
...
klasa potomna


TSamochod
marka
model
rokProdukcji
...
klasa bazowa
dziedziczenie
TSamochodDiagnostyka
marka
model
rokProdukcji
dataPrzegladu
nastepnyPrzeglad
diagnosta
...
klasa potomna


TSamochod
marka
model
rokProdukcji
...
klasa bazowa
dziedziczenie
TSamochodSerwis
marka
model
rokProdukcji
przebieg
nrWyposazenia
nrSilnika
...
klasa potomna

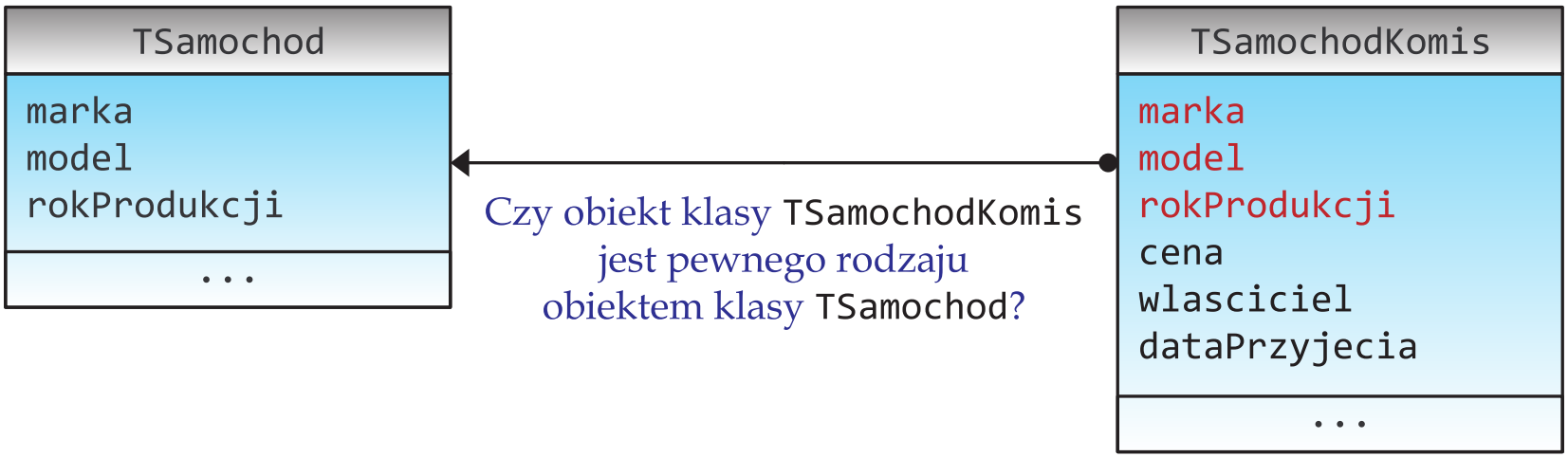

TSamochod
marka
model
rokProdukcji
...
TSamochodKomis
marka
model
rokProdukcji
cena
wlasciciel
dataPrzyjecia
...
TSamochodDiagnostyka
marka
model
rokProdukcji
dataPrzegladu
nastepnyPrzeglad
diagnosta
...
TSamochodSerwis
marka
model
rokProdukcji
przebieg
nrWyposazenia
nrSilnika
...

Jak zweryfikować poprawność wykorzystania dziedziczenia?

Reguła **is-a** (*is a kind of*) pozwala na sprawdzenie czy zachodzi związek dziedziczenia (specjalizacji–generalizacji) pomiędzy klasami.

Realizowane jest to poprzez sprawdzenie poprawności zdania:

- Czy obiekt klasy pochodnej jest pewnego rodzaju (is-a) obiektem klasy bazowej?



Przykład weryfikacji poprawności dziedziczenia.

Zadajemy dwa pytania:

- czy obiekt klasy `A` jest obiektem klasy `B`?
- czy obiekt klasy `B` jest obiektem klasy `A`?

Możliwe odpowiedzi:

- zawsze,
- nigdy
- czasami.

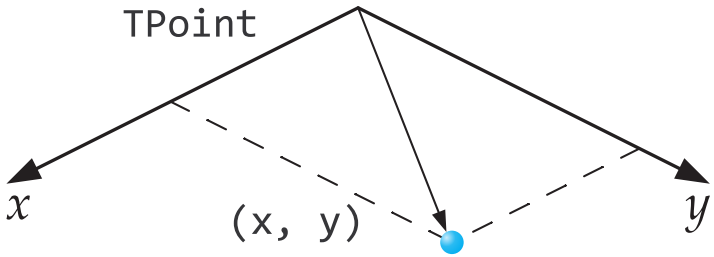
Interpretacja:

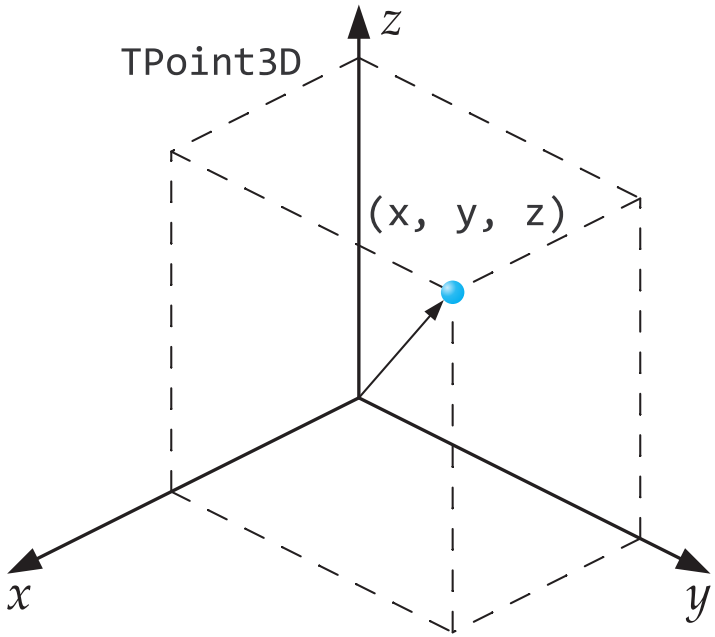
- dwie odpowiedzi nigdy: brak związku specjalizacja-generalizacja,
- dwie odpowiedzi zawsze: obiekty `A` i `B` są **synonimiczne** (np. dwie różne nazwy dla tej samej klasy)
- jeżeli: `A` is-a `B` = zawsze oraz `B` is-a `A` = czasami wtedy **A jest specjalizacją B**.

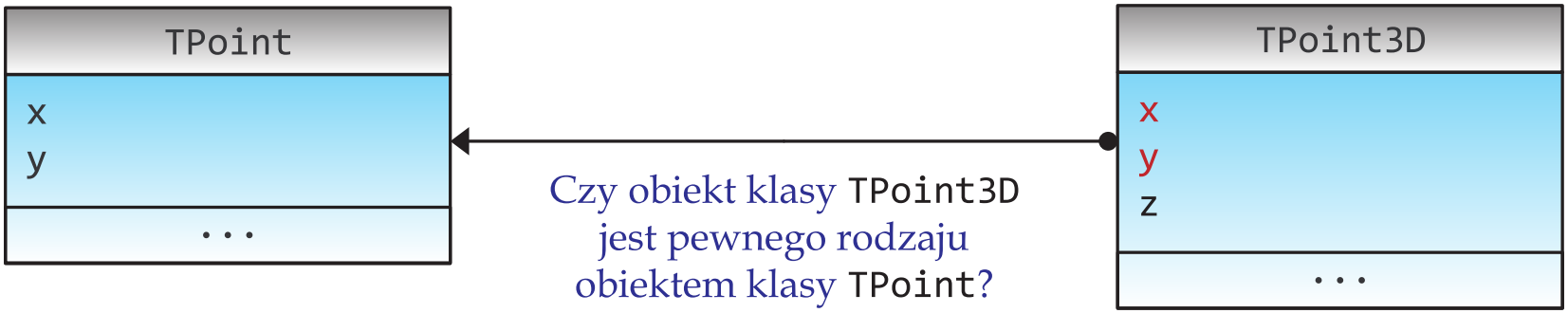
Czy klasa `TSamochodKomis` jest klasą pochodną klasy `TSamochod`:

- czy obiekt klasy `TSamochodKomis` jest obiektem klasy `TSamochod`? ⇐ **zawsze**
- czy obiekt klasy `TSamochod` jest obiektem klasy `TSamochodKomis`? ⇐ **czasami**

Dziedziczenie w praktyce







Przykład wykorzystania klasy `TPoint` i `TPoint3D`:

```
TPoint ptOne(2, 3);
TPoint3D ptTwo(2, 3, 4);

cout << "x = " << ptOne.getX() << endl;
cout << "y = " << ptOne.getY() << endl;

cout << "x = " << ptTwo.getX() << endl;
cout << "y = " << ptTwo.getY() << endl;
cout << "z = " << ptTwo.getZ() << endl;
```

Przykładowa definicja klasy bazowej `TPoint`:

```
class TPoint
{
  public:
    TPoint(): x(0), y(0) {}
    TPoint(int x, int y): x(x), y(y) {}

    void setX(int x) {TPoint::x = x;}
    void setY(int y) {TPoint::y = y;}

    int getX() const {return x;}
    int getY() const {return y;}

  private:
    int x, y;
};
```

Przykładowa klasa pochodna `TPoint3D`:

```
class TPoint3D : public TPoint
{
  public:
    TPoint3D(): TPoint(), z(0) {}
    TPoint3D(int x, int y, int z): TPoint(x, y), z(z) {}

    void setZ(int z) {TPoint3D::z = z;}

    int getZ() const {return z;}

  private:
    int z;
};
```

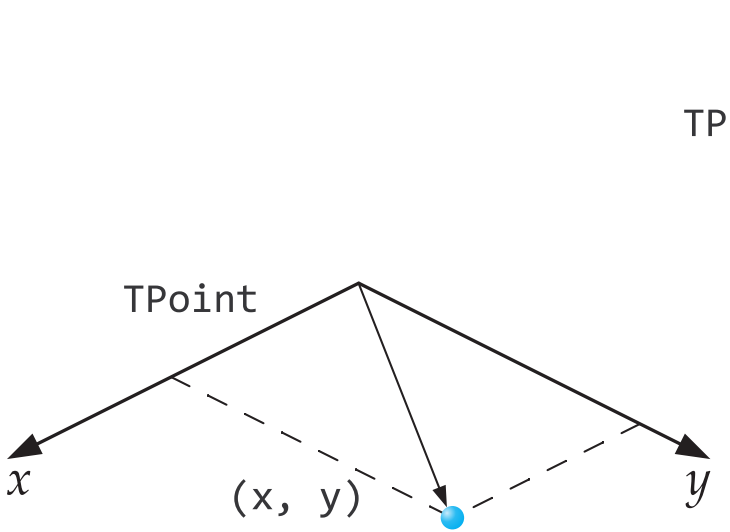
Dziedziczenie w **trybie publicznym** (`public TPoint`) zachowuje widoczność pól określoną w klasie bazowej.

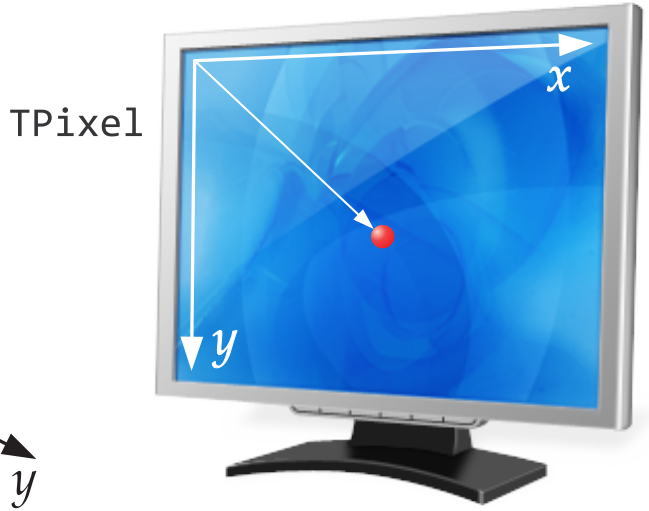
Konstruktor **klasy pochodnej** (`TPoint3D()`) aktywuje konstruktor **klasy bazowej** (`TPoint()`) umieszczony na liście inicjalizacyjnej, odbywa się to przed wywołaniem ciała konstruktora klasy pochodnej.

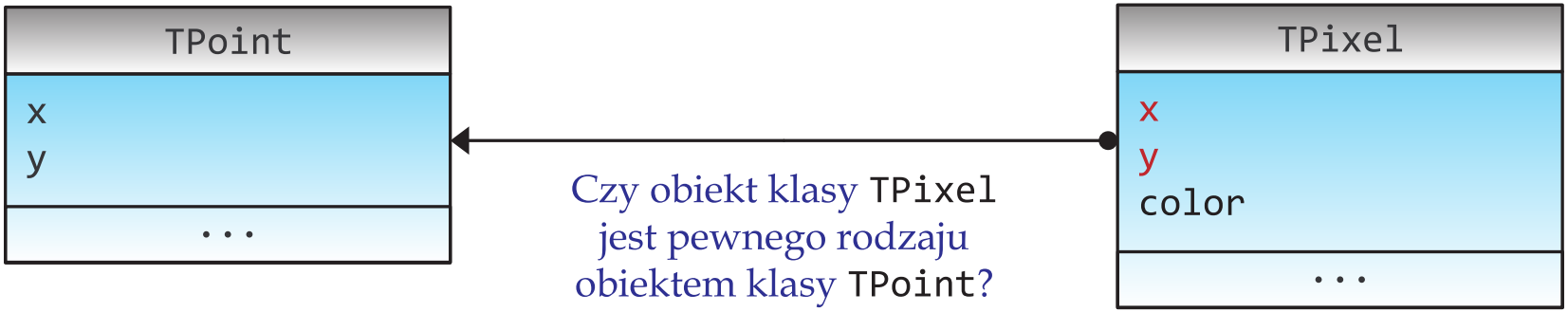
W klasie pochodnej definiujemy metody do obsługi **nowych pól** (`setZ()` i `getZ()`), obsługę **pól odziedziczonych** realizujemy z wykorzystaniem metod odziedziczonych.

Używamy konstruktorów klasy bazowej do zainicjowania pól odziedziczonych (`x` i `y`), nowe pola inicjują własne konstruktory (`z`).

Kolejna klasa pochodna







Czy obiekt klasy `TPixel` jest pewnego rodzaju obiektem klasy `TPoint`?

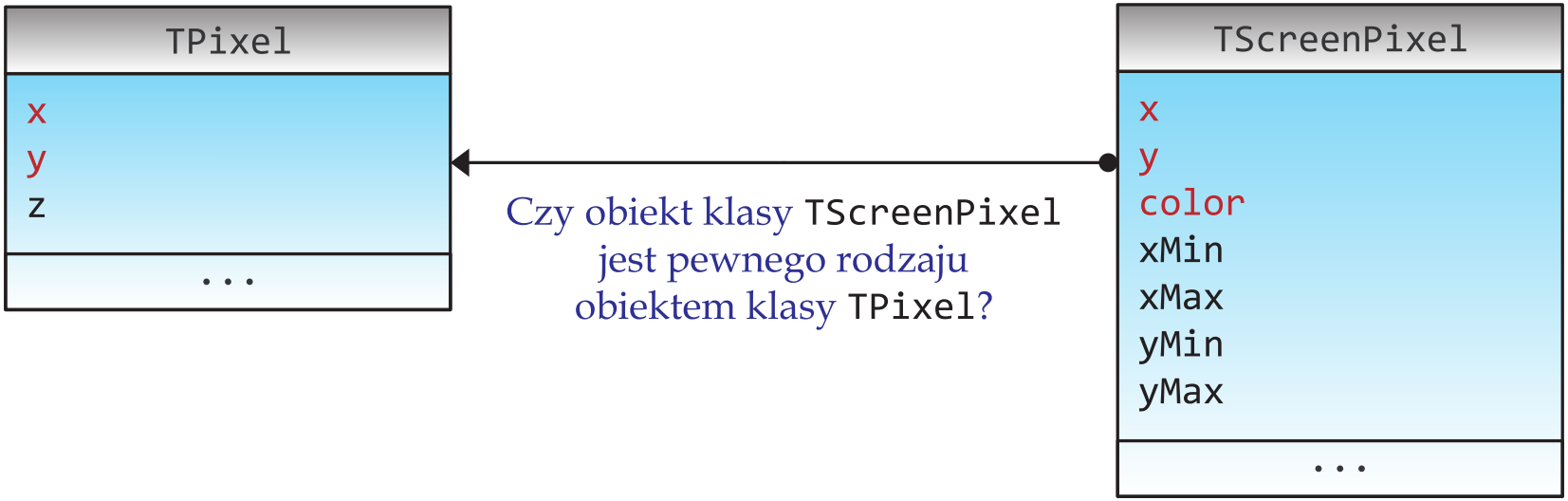
Przykład klasy pochodnej `TPixel`:

```
class TPixel : public TPoint
{
  public:
    TPixel(): TPoint(), color(0) {}
    TPixel(int x, int y, int color): TPoint(x, y), color(color) {}

    void setColor(int color) {TPixel::color = color;}

    int getColor() const {return color;}

    void put() const
    {
      putpixel(getX(), getY(), color);  ← funkcja z biblioteki graficznej
    }

  private:
    int color;
};
```

I jeszcze jedna klasa pochodna

TPixel

TScreenPixel



Przykład klasy pochodnej TScreenPixel:

```
class TScreenPixel : public TPixel
{
  public:
    TScreenPixel(): TPixel() {}
    TScreenPixel(int x, int y, int color): TPixel(x, y, color) {}

    void setX(int x)
    {
      TPoint::setX((x >= xMin && x <= xMax) ? x : 0);
    }

    void setY(int y)
    {
      TPoint::setY((y >= yMin && y <= yMax) ? y : 0);
    }

    static int xMin, xMax, yMin, yMax;
};

int TScreenPixel::xMin = 0;
int TScreenPixel::xMax = 799;
int TScreenPixel::yMin = 0;
int TScreenPixel::yMax = 599;
```

Pola statyczne dla danej klasy występują tylko raz, są współdzielone przez wszystkie obiekty tej klasy. Istnieją nawet wtedy, gdy nie został utworzony żaden obiekt klasy.

Pola statyczne muszą być zdefiniowane poza klasą, nadaje się im wedy wartość początkową.

Do pól statycznych klasy można odwoływać się bez obiektu:

```
TScreenPixel::xMax = screen.getMaxX();
TScreenPixel::yMax = screen.getMaxY();
```

Pola statyczne stanowią wspólną, współdzieloną pamięć wszystkich obiektów danej klasy.

Wykorzystywane są do zapamiętywania informacji, które są wspólne dla wszystkich obiektów i nie muszą być pamiętane osobno w każdym z obiektów.

Redefinicja metody w klasie pochodnej

```
class TPoint
{
  void setX(int x) {Point::x = x;}
};
```

```
class TScreenPixel : public TPixel
{
  void setX(int x)
  {
    TPoint::setX((x >= xMin && x <= xMax) ? x : 0);
  }
};
```

W klasie pochodnej można zmienić sposób działania metody odziedziczonej poprzez jej redefinicję.

Do metody odziedziczonej można dalej odwoływać się stosując nazwę klasy i operator zakresu `::`.

Składowe chronione

W klasie pochodnej nie ma bezpośredniego dostępu do pól prywatnych klasy bazowej.

```
class TPixel : public TPoint
{
  public:
    void put() const
    {
      putpixel(getX(), getY(), color);  ← getX(), nie x
    }
};
```

Przewidując, że pewna klasa będzie wykorzystywana jako klasa bazowa, można zadeklarować jej składowe (pola i metody) jako **chronione** `protected`.

Składowe zadeklarowane jako `protected` są dostępne dla obiektów wszystkich klas pochodnych (tak jak składowe `public`) i niedostępne dla obiektów innych, niezaprzyjaźnionych klas (tak jak składowe `private`).

Specyfikator protected działa jak `private`, z tym wyjątkiem, że obiekty klas pochodnych otrzymują dostęp do składowych `protected` klasy bazowej.

Deklaracja pól x i y jako chronionych:

```
class TPoint
{
  protected:
    int x, y;
};
```

```
class TPixel : public TPoint
{
  public:
    void put() const
    {
      putpixel(x, y, color);  ← dozwolone odwołanie
    }
};
```

Destruktory

Destruktor to funkcja wywoływana automatycznie przez kompilator bezpośrednio przed momentem, w którym obiekt przestaje istnieć.

Moment *śmierci* obiektu zależy od tego, w jaki sposób został on utworzony.

Obiekty **statyczne** giną po zakończeniu wykonania funkcji `main()`, obiekty **automatyczne** po wyjściu sterowania z bloku, w którym zostały zdefiniowane, obiekty **dynamicze** w momencie usunięcia ich z pamięci operatorem `delete`.

Destruktor to bezparametrowa funkcja, bez określonego rezultatu, o nazwie takiej, jak nazwa klasy poprzedzona znakiem tyldy `~`.

```
class TPoint
{
  public:
    ~TPoint();
};
```

```
class TScreenPixel
{
  public:
    ~TScreenPixel();
};
```

Klasa `TPoint` ze zmodyfikowanymi konstruktorem i destruktorem:

```
class TPoint
{
  public:
    TPoint(int x, int y): x(x), y(y)
    {
      cout << "TPoint(" << x << ", " << y << ")" << endl;
    }

    ~TPoint()
    {
      cout << "~TPoint()" << endl;
    }
};
```

Aktywowanie konstruktora i destruktora dla obiektu klasy `TPoint`:

```
cout << "Przed utworzeniem obiektu pt klasy TPoint" << endl;
{
  TPoint pt(2, 3);
  cout << endl << "Punkt ma wspolrzedne" << endl;
  cout << "x = " << pt.getX() << endl;
  cout << "y = " << pt.getY() << endl;
}
cout << endl << "Po usunieciu obiektu pt klasy TPoint" << endl;
```

```
Przed utworzeniem obiektu pt klasy TPoint

TPoint(2, 3)

Punkt ma wspolrzedne
x = 2
y = 3

~TPoint()

Po usunieciu obiektu pt klasy TPoint
```

Klasa TPoint3D ze zmodyfikowanymi konstruktorem i destruktorem:

```
class TPoint3D : public TPoint
{
  public :
    TPoint3D(int x, int y, int z): TPoint(x, y), z(z)
    {
      cout << "TPoint3D(" << x << ", " << y << ", " << z << ")"
        << endl;
    }

    ~TPoint3D()
    {
      cout << "~TPoint3D()" << endl;
    }
};
```

Aktywowanie konstruktora i destruktora dla obiektu klasy `TPoint3D`:

```
cout << "Przed utworzeniem obiektu pt klasy TPoint3D" << endl;
{
  TPoint3D p(2, 3, 4);
  cout << endl << "Punkt ma wspolrzedne" << endl;
  cout << "x = " << p.getX() << endl;
  cout << "y = " << p.getY() << endl;
  cout << "z = " << p.getZ() << endl;
}
cout << endl << "Po usunieciu obiektu pt klasy TPoint3d" << endl;
```

```
Przed utworzeniem obiektu pt klasy TPoint3D

TPoint(2, 3)

TPoint3D(2, 3, 4)

Punkt ma wspolrzedne
x = 2
y = 3
z = 4

~TPoint3D()

~TPoint()

Po usunieciu obiektu pt klasy TPoint3D
```

Polimorfizm

Słowo polimorfizm pochodzi od dwóch greckich słów: *poly* czyli **wiele** i *morphos* czyli **postać**.

Polimorfizm oznacza zatem wielopostaciowość. Polimorfizm w programowaniu obiektowym oznacza zdolność obiektów do różnych zachowań, w zależności od bieżącego kontekstu wykonania programu.

Referencje i kolekcje obiektów mogą dotyczyć obiektów różnego typu, a wywołanie metody dla referencji spowoduje zachowanie odpowiednie dla pełnego typu obiektu wywoływanego.

Inaczej mówiąc, są to mechanizmy pozwalające na używanie wartości, zmiennych i procedur na kilka różnych sposobów (wyabstrahowanie wyrażeń od konkretnych typów).

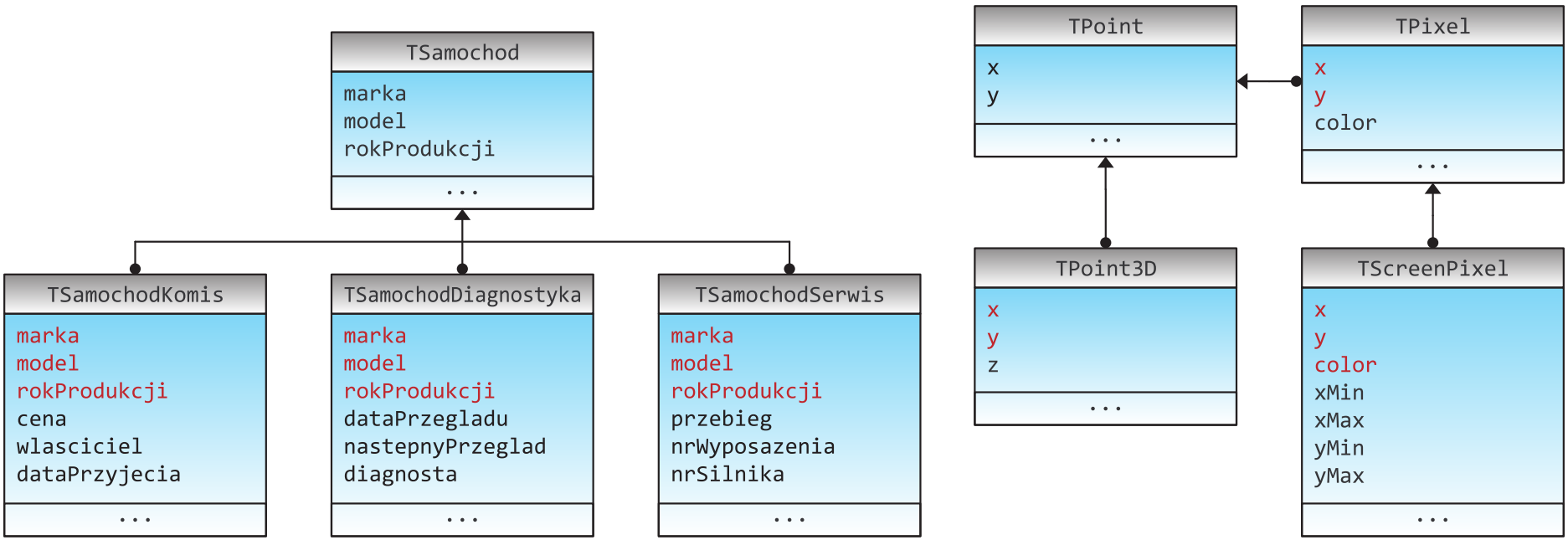
To jaka ma być **wykonana akcja**, ustalane jest na **etapie wykonania** programu, a wykonywane akcje mogą być różne.

W języku C++ wykonanie akcji polega na wywołaniu odpowiedniej funkcji. Polimorfizm w tym języku polega na tym, że wywoływane mogą być różne wersje tej samej funkcji.

Różne wersje tej samej funkcji **powstają w trakcie dziedziczenia**, kiedy to klasy pochodne w specjalny sposób redefiniują pewną funkcję składową.

Zdyscyplinowane stosowanie zasad abstrakcji, hermetyzacji i dziedziczenia prowadzi do powstania, często rozbudowanych, hierarchii klas.

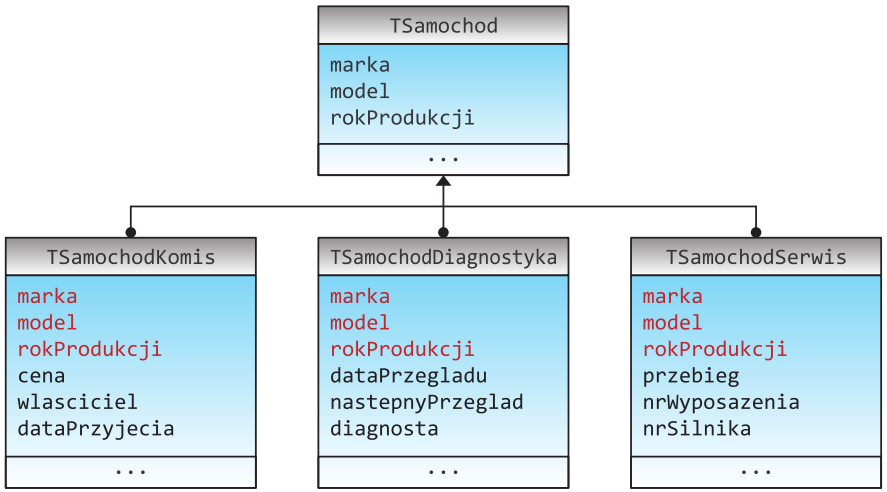
Obiekty takich klas są często do siebie podobne, ale nie jednakowe. Problem stanowi spójne i wygodne zarządzanie grupami obiektów różnych klas należących do tej samej hierarchii, zdefiniowanej dziedziczeniem.

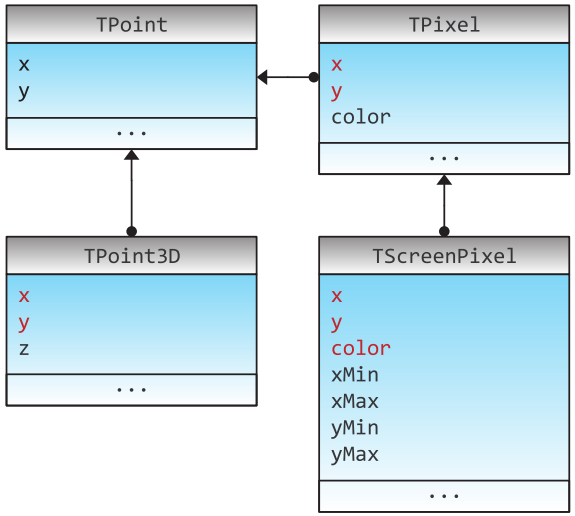


Obiekty spowinowacone mogą być traktowane w specjalny sposób.

```
TSamochod *auto;

auto = new TSamochodKomis;
auto->rokProdukcji = 2005;

delete auto;

auto = new TSamochodSerwis;
auto->rokProdukcji = 1999;

delete auto;
```

```
void pointInfo(TPoint &pt)
{
  cout << "x = " << pt.getX() << endl;
  cout << "y = " << pt.getY() << endl;
}

TPixel ptOne(2, 3, RED);
TPoint3D ptTwo(2, 3, 4);

pointInfo(ptOne);
pointInfo(ptTwo);
```

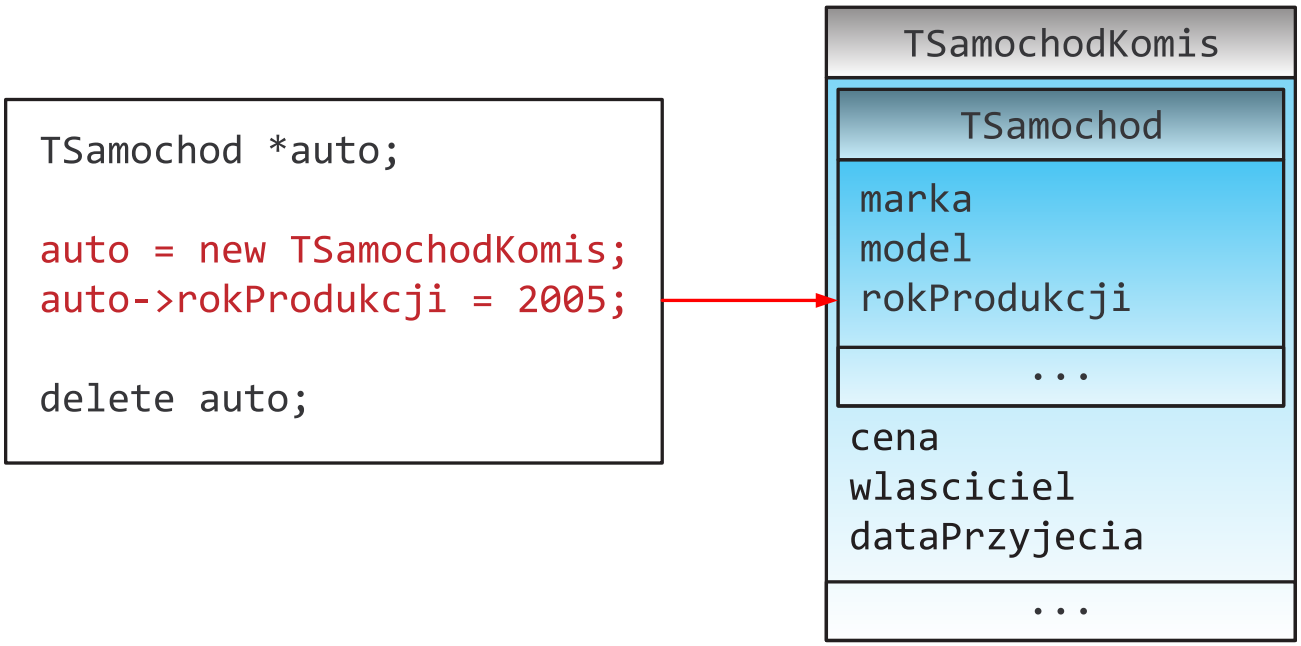




W języku C++ wolno do **wskaźnika klasy bazowej** przypisać **wskazanie obiektu klasy pochodnej**. Analogicznie jest w przypadku referencji.

Obiekt klasy pochodnej jest przecież pewnego rodzaju obiektem klasy bazowej (jest specjalizacją klasy bazowej), posiada zatem wszystkie pola i funkcje składowe.

Wolno zatem pośrednio odwoływać się do odziedziczonych pól i funkcji składowych.

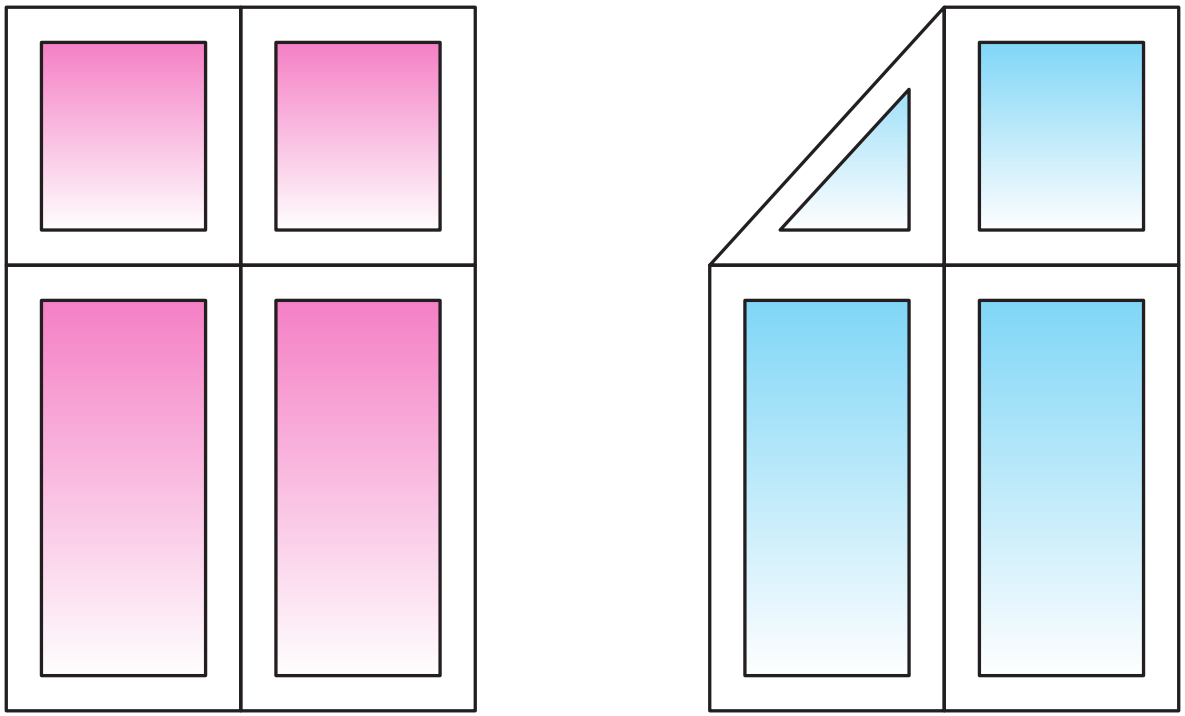


Polimorfizm w praktyce

Program wspomagający pracę technologa w firmie produkującej okna.

Zadaniem programu jest obliczanie:

- łącznego pola powierzchni wszystkich skrzydeł okna,
- przybliżonej, łącznej długości profili, użytych do produkcji każdego ze skrzydeł.



Stosując zasadę abstrakcji wyodrębniamy najistotniejsze cechy obiektów dla rozpatrywanego zagadnienia – szyby to figury geometryczne a okno to ich złożenie:

- łączna powierzchnia okna to, w przybliżeniu, suma pól figur opisujących szyby,
- łączna długość profili to, w przybliżeniu, suma obwodów figur opisujących szyby.

Realizacja programu sprowadza się do obliczeń pól powierzchni i obwodów obiektów, będących złożeniem elementarnych figur płaskich.

Nie powinniśmy mieć problemów z utworzeniem klas reprezentujących figury płaskie. Nie wiemy jak reprezentować i pracować z ich złożeniem.

Rozpoczniemy od utworzenia nieco *dziwnej* klasy – reprezentującej abstrakcyjną figurę geometryczną. Wyposażamy ją w funkcje obliczania pola i długości obwodu.

```
class TFigure
{
  public:
    TFigure() {}

    double area() const {return 0;}
    double perimeter() const {return 0;}
};
```

Klasa `TFigure` służyć będzie jak klasa bazowa dla specjalizowanych klas pochodnych, reprezentujących konkretne figury geometryczne.

Jej istotą jest stwierdzenie, że każda figura geometryczna powinna umieć wyznaczać swoje pole i obwód, w charakterystyczny dla siebie sposób.

Zatem każda z klas pochodnych, powinna redefiniować funkcje `area()` i `perimeter()`, w odpowiedni dla tych klas sposób.

Klasa reprezentująca **kwadrat** będzie teraz klasą pochodną w stosunku do klasy `TFigure`:

```
class TSquare : public TFigure
{
  public:
    TSquare( );
    TSquare(double side);

    void setSide(double side);
    double getSide( );

    double area( ) const;
    double perimeter( ) const;

  private:
    double side;
}
```

Redefinicja funkcji `area( )` i `perimeter( )` – obliczenia dla kwadratu:

```
double TSquare::area( ) const {return side * side;}
double TSquare::perimeter( ) const {return 4 * side;}
```

Klasa reprezentująca **prostokąt** będzie klasą pochodną w stosunku do klasy `TFigure`:

```
class TRectangle : public TFigure
{
  public:
    TRectangle( );
    TRectangle(double width, double height);

    void setWidth(double width);
    void setHeight(double height);
    double getWidth( ) const;
    double getHeight( ) const;

    double area( ) const;
    double perimeter( ) const;

  private:
    double width, height;
};
```

Redefinicja funkcji `area( )` i `perimeter( )` – obliczenia dla prostokąta:

```
double TRectangle::area( ) const {return width * height;}
double TRectangle::perimeter( ) const {return 2 * width + 2 * height;}
```

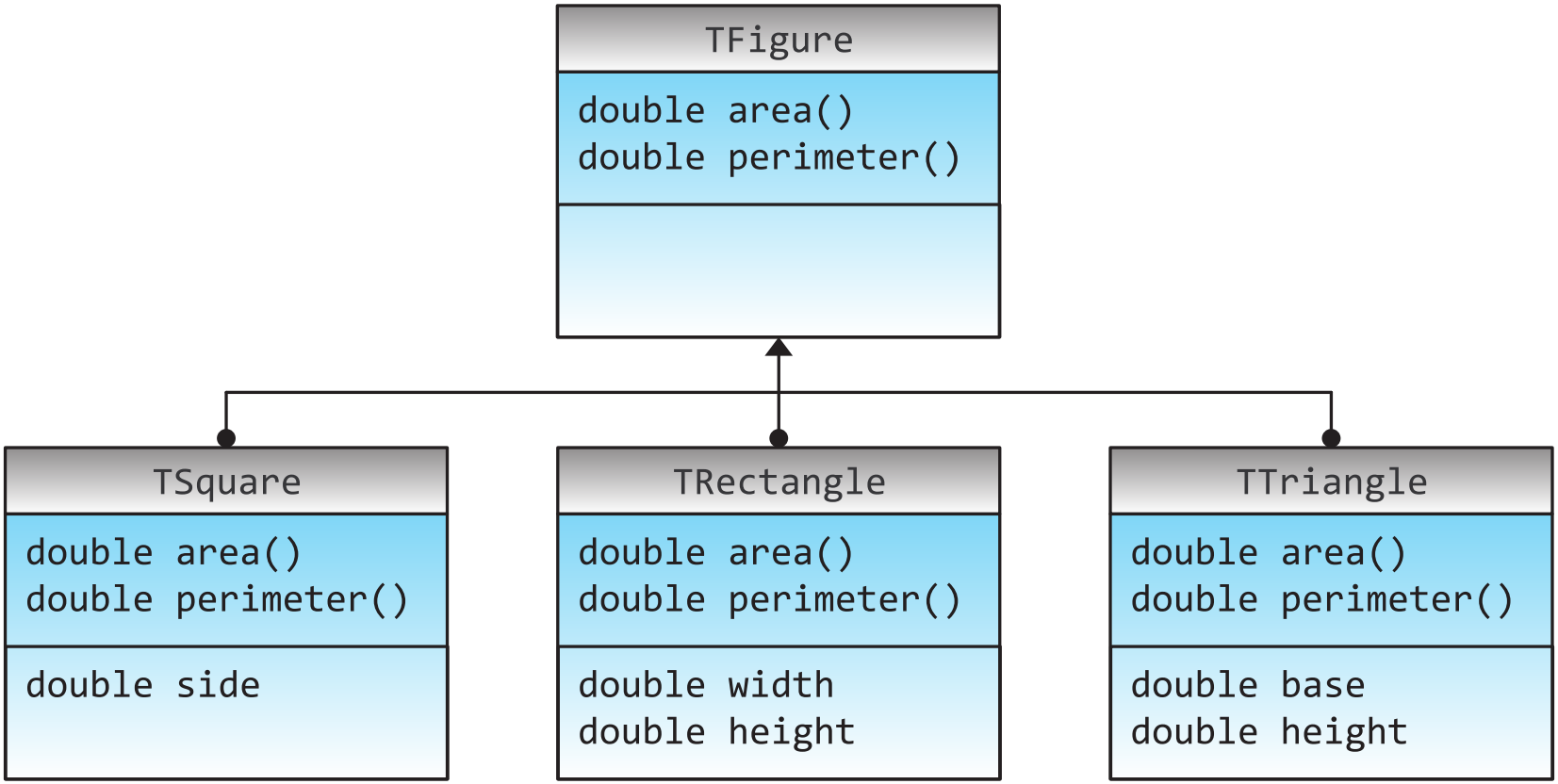
Klasa reprezentująca **trójkat** będzie klasą pochodną w stosunku do klasy `TFigure`:

```
class TTriangle : public TFigure
{
  public:
    TTriangle( );
    TTriangle(double base, double height);

    void setBase(double base);
    void setHeight(double height);
    double getBase( ) const;
    double getHeight( ) const;

    double area( ) const;
    double perimeter( ) const;

  private:
    double base, height;
};
```

Redefinicja funkcji `area( )` i `perimeter( )` – obliczenia dla trójkąta:

```
double TTriangle::area( ) const {return 0.5 * base * height;}
double TTriangle::perimeter( ) const {return sqrt(base * base +
  height * height) + base + height;}
```

Hierarchia klas figur płaskich



Załóżmy, że zdefiniowaliśmy wskaźnik do klasy bazowej `TFigure` i obiekty klas pochodnych:

```
TFigure *fig;

TSquare s(10);
TRectangle r(10, 20);
TTriangle t(10,10);
```

Wiemy już, że można przypisać do wskaźnika `fp` wskazanie na obiekt klasy pochodnej:

```
fig = &s;
fig = &r;
fig = &t;
```

Wolno zatem pośrednio odwoływać się do odziedziczonych pól i funkcji składowych (czy na pewno tak jak się spodziewamy?).

```
fig = &s;

cout << "Pole kwadratu wynosi: " << fig->area() << endl;
cout << "Obwod kwadratu wynosi: " << fig->perimeter() << endl;
```

Ale nie można (dlaczego?):

```
fig->setSide(100);  ← niedozwolone
```

Choć można dokonać *wymuszenia* stosując brutalne rzutowanie typów (niezalecane):

```
((TSquare *)fp)->setSide(2);
```

Wykorzystując wskaźnik do **klasy bazowej** można odwoływać się w legalny sposób do wszystkich odziedziczonych składowych **obiektu pochodnego**.

Stosując zwykłe rzutowanie (lub specjalne metody konwersji) można siłowo wymusić dostęp do składowych zdefiniowanych w klasie pochodnej.

W C++ istnieją inne, bezpieczniejsze metody konwersji typów (np. `static_cast<>()`, `dynamic_cast<>()`, `reinterpret_cast<>()`).

Rozbudujmy klasy o dodatkową funkcję określającą nazwę figury:

```
class TFigure
{
  public:
    string getName() const {return "Figura";}
};
```

```
class TSquare : public TFigure
{
  public:
    string getName() const {return "Kwadrat";}
};
```

```
class TRectangle : public TFigure
{
  public:
    string getName() const {return "Prostokat";}
};
```

```
class TTriangle : public TFigure
{
  public:
    string getName() const {return "Trojkat";}
};
```

Napiszmy uniwersalną funkcję wyświetlającą informacje o figurze:

```
void figureInfo(TFigure *fig)
{
  cout << fig->getName () << endl;
  cout << "Pole: " << fig->area() << endl;
  cout << "Obwod: " << fig->perimeter() << endl;
  cout << endl;
}
```

Stwórzmy obiekty klas pochodnych od klasy `TFigure`:

```
TFigure *fig;

TSquare s(10);
TRectangle r(10, 20);
TTriangle t(10,10);
```

I w końcu wywołujemy `figureInfo( )`:

```
figureInfo(&s);
figureInfo(&r);
figureInfo(&t);
```

I otrzymujemy...

```
Figura
Pole: 0
Obwod: 0

Figura
Pole: 0
Obwod: 0

Figura
Pole: 0
Obwod: 0
```

Nie działa!

Ale dlaczego?

Przypisanie konkretnego ciała funkcji do wywołania nazywa się wiązaniem (*binding*).

W języku C++ występują dwa rodzaje wiązania:

- **Wczesne wiązanie** (*early binding*) polega na przypisaniu konkretnej funkcji każdemu wywołaniu już na etapie kompilacji programu. Inna nazwa – **wiązanie statyczne** (*static binding*).
- **Późne wiązanie** (*late binding*) polega na przypisaniu konkretnej funkcji każdemu wywołaniu dopiero na etapie wykonania programu, w zależności od typu obiektu, którego wywołanie dotyczy. Inna nazwa to **dynamiczne wiązanie** (*dynamic binding*).

Jakie wiązanie stosuje kompilator?

```
TFigure *fig;
TSquare s(10);

fig = &s;
cout << "Pole kwadratu wynosi: " << fp->area() << endl;
cout << "Obwod kwadratu wynosi: " << fp->perimeter() << endl;
```

Wywołanie odbywa się za pośrednictwem wskaźnika do typu `TFigure`. Kompilator *przygląda* się definicji klasy i deklaracji wywoływanej funkcji:

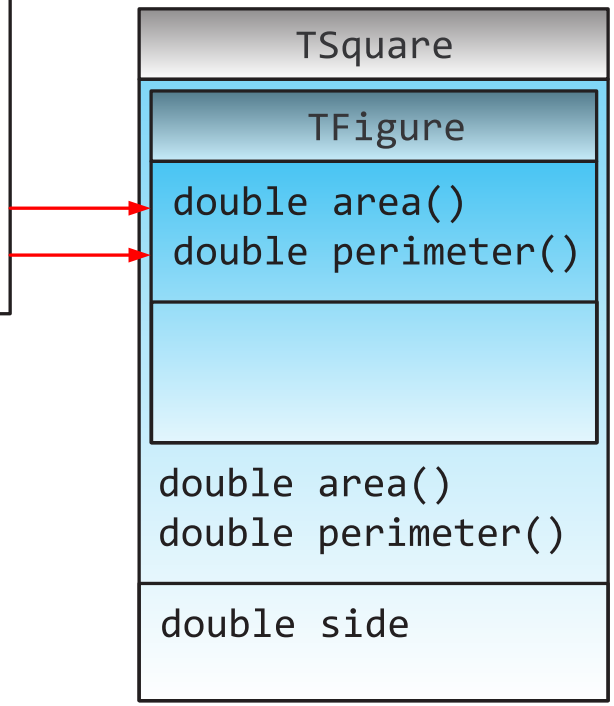
```
class TFigure
{
  string getName() const {return "Figura";}
  double area() const {return 0;}
  double perimeter() const {return 0;}
};
```

Wywoływane funkcje są *zwykłymi* funkcjami.

Takie funkcje są łączone statycznie. Kompilator wstawia wywołanie funkcji zdefiniowanych w klasie występującej w deklaracji wskaźnika `fig`. Wywoływane są zatem funkcje z klasy `TFigure`, niezależnie od klasy wskazywanego obiektu.

O tym która funkcja jest wywoływana decyduje kompilator na etapie kompilacji. Wersję funkcji determinuje **typ występujący w deklaracji zmiennej wskaźnikowej**.

```
TFigure *fig;
TSquare s(10);

fig = &s;

cout << fig->area() << endl;
cout << fig->preimeter() << endl;
```



Przyczyną problemów jest **wiązanie statyczne**. Mimo, że funkcja wywoływana jest dla obiektów klas potomnych względem `TFigure`, kompilator wstawi wywołanie funkcji zgodnie z typem występującym w deklaracji parametru `fig`.

Dokonajmy drobnej modyfikacji deklaracji funkcji składowych klasy `TFigure`:

```
class TFigure
{
  public:
    TFigure() {}

    virtual string getName() const {return "Figura";}

    virtual double area() const {return 0;}
    virtual double perimeter() const {return 0;}
};
```

Uruchamiamy ponownie...

```
Kwadrat
Pole: 100
Obwod: 40

Prostokat
Pole: 200
Obwod: 60

Trojkat
Pole: 50
Obwod: 34.1421
```

Działa poprawnie!

Funkcje wirtualne

Mimo, iż typ występujący w deklaracji wskaźnika to `TFigure`, wywołane zostaną funkcje należące do klasy **obiektu wskazywanego** przez wskaźnik `fig`. Przy wiązaniu **dynamicznym** istotny jest typ obiektu **wskazywanego**!

Wiązanie dynamiczne jest realizowane w języku C++ za pomocą funkcji **wirtualnych**. Funkcja wirtualna posiada w swojej deklaracji słowo kluczowe `virtual`.

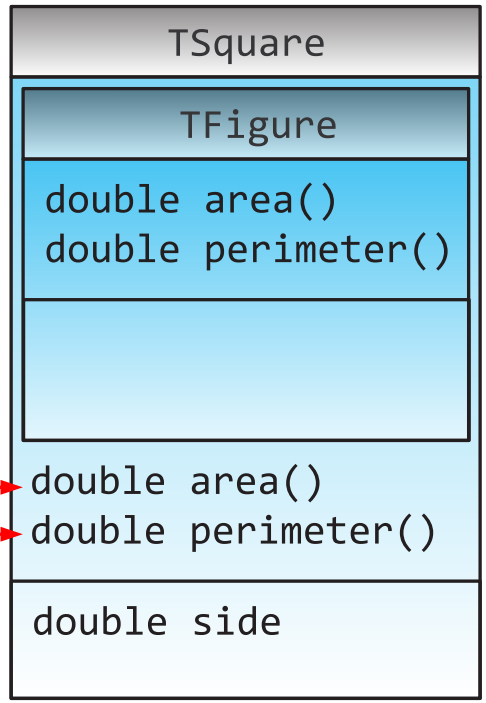
Jeżeli w klasie pochodnej funkcja wirtualna nie zostanie przedefiniowana, wszystko działa jak w przypadku innych funkcji. Jeżeli w klasie pochodnej funkcja wirtualna zostanie przedefiniowana, to zostanie ona wywołana gdy wskaźnik do klasy bazowej wskazuje na obiekt klasy pochodnej.

Wiązanie dynamiczne

O tym która funkcja jest wywoływana decyduje **typ obiektu wskazywanego**.

Funkcja wiązana dynamicznie musi być zadeklarowana jako wirtualna. Wystarczy, że słowo kluczowe `virtual` wystąpi w deklaracji klasy bazowej.

```
TFigure *fig;
TSquare s(10);

fig = &s;

cout << fig->area() << endl;
cout << fig->preimeter() << endl;
```



Klasa abstrakcyjna

Czysta wirtualność określa to, że metoda wirtualna z klasy bazowej nigdy nie powinna się wykonać.

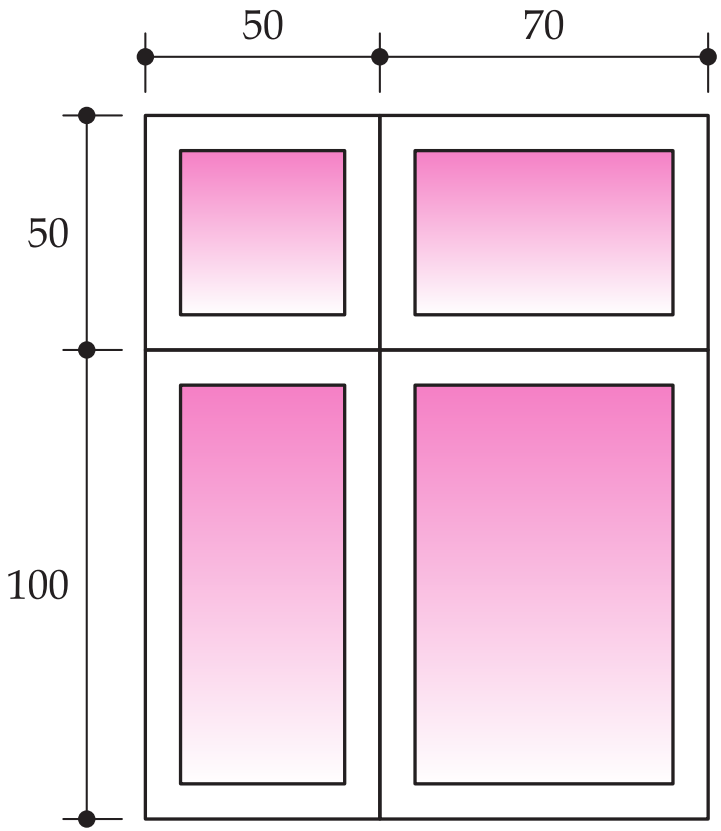
W efekcie klasa taka staje się **klasą abstrakcyjną**. Oznacza to, że iż nie jest możliwe stworzenie obiektu tej klasy.

Klasa służy jedynie temu, ab zdefiniować pewnego rodzaju interfejs i jest przeznaczona jedynie po to, by od niej dziedziczyć.

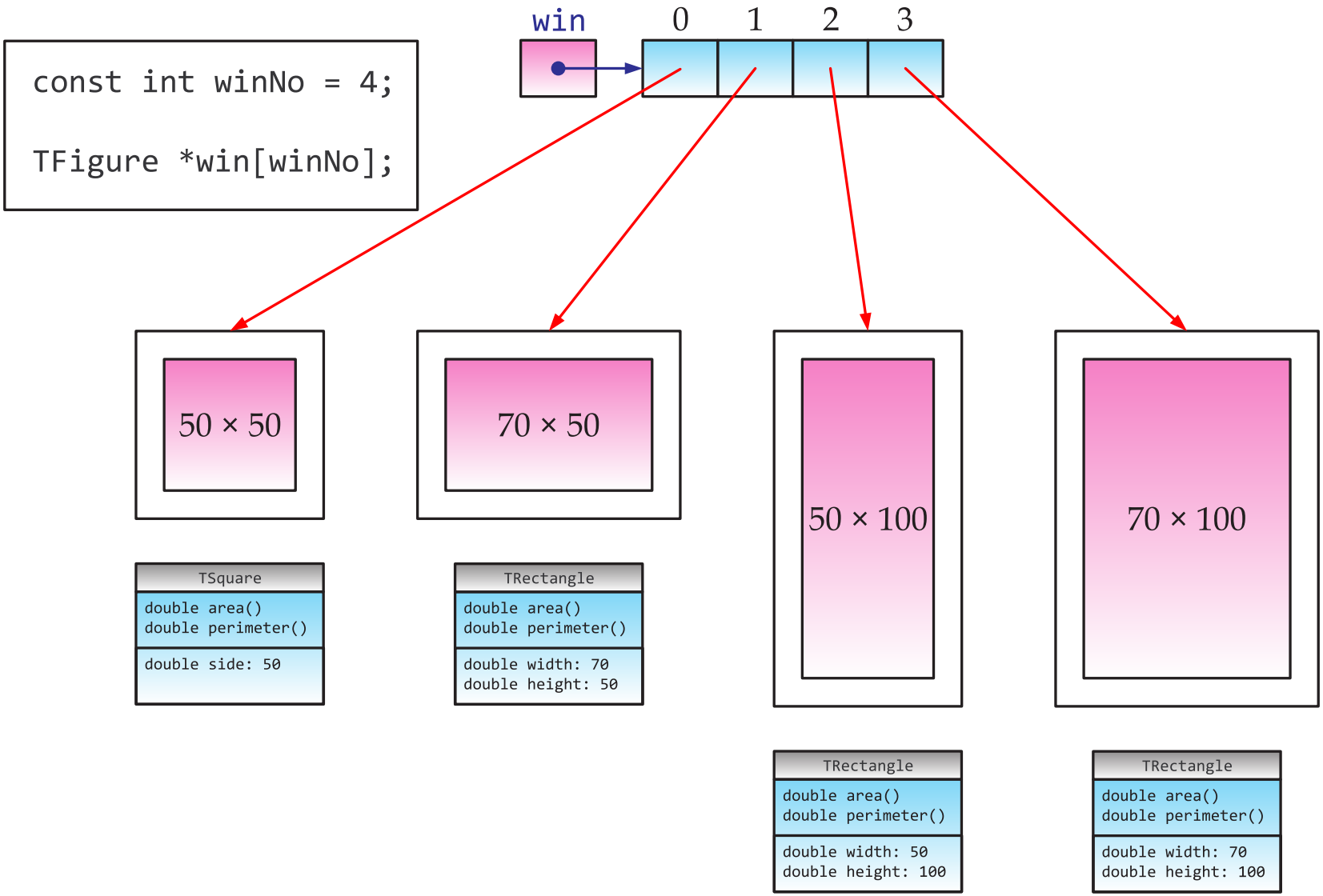
```
class TFigure
{
  public:
    virtual string getName( ) const = 0;

    virtual double area( ) const = 0;
    virtual double perimeter( ) const = 0;
};
```

Wykorzystanie późnego wiązania

Znany jest układ okna i wymiary skrzydeł. Jedno skrzydło jest kwadratowe, pozostałe są prostokątne. Należy wyznaczyć łączną powierzchnie szyb i długości wykorzystanych profili.



Reprezentowanie informacji o oknach



Definiowanie elementów, obliczenia, zwalnianie pamięci

```
const int winNo = 4;  ← określenie liczby elementów

TFigure *win[winNo];  ← tablica wskaźników na elementy okna

win[0] = new TSquare(50);          ← przydział pamięci
win[1] = new TRectangle(50, 70);   ←
win[2] = new TRectangle(50, 100);  ←
win[3] = new TRectangle(70, 100);  ←

calcAndInfo(win, winNo);  ← obliczenia i drukowanie wyników

for(int i = 0; i < winNo; delete win[i++]);  ← zwolenianie pamięci
```

Funkcja calcAndInfo()

```
void calcAndInfo(TFigure *win[], int winNo)
{
  double totalArea = 0;
  double totalPerimeter = 0;

  for(int i = 0; i < winNo; i++)
  {
    totalArea += win[i]->area();
    totalPerimeter += win[i]->perimeter();
  }

  cout << "Powierzchnia szyb: " << totalArea << endl;
  cout << "Dlugosc profili: " << totalPerimeter << endl;
}
```

Błędy i ich obsługa

W trakcie działania programu zawsze może dojść do powstania nietypowej sytuacji. Nie da się ich całkowicie wyeliminować. Można jednak przewidywać iż taka sytuacja wystąpi i się na nią odpowiednio przygotować.

Kontrolowanie poprawności wykonywanych operacji jest możliwe, jednak uciążliwe bez specjalnie przygotowanych do tego celu technik (przykład w C). Uciążliwość pisania sekwencji kontroli i obsługi błędów, budzi ochotę do użycia

instrukcji skoku `goto`, ale jest to co najmniej **bardzo nieeleganckie**.

```
if((file = fopen(name, "rt")) == NULL)
  return IN_FILE_ERROR;
else

  if((num_of_lines = count_lines(file)) = 0)
    return EMPTY_IN_FILE;
  else

    if((lines = malloc(num_of_lines * sizeof(char))) == NULL)
      return NO_MEM_ERROR;
    else

      if((buffer = malloc(MAX_LINE_LEN)) == NULL)
        return NO_MEM_ERROR;
      else

        while(fgets(buffer, file, MAX_LINE_LEN) != NULL)
        {
          ...
        }
...
```

W języku C przewidziano zestaw funkcji i makr do kontroli błędów i reagowania na ich powstanie:

- `assert()`,
- `signal()`,
- `raise()`,
- `setjmp()`,
- `longjmp()`.

Są one jednak rzadko stosowane – mało znane i wymagające uwagi.

Użycie sygnałów i nielokalnych skoków jest niebezpieczne w C++, nie zapewnia bowiem właściwego aktywowania np. destruktorów.

Błąd czyli sytuacja wyjątkowa

W języku C++ wprowadzono mechanizm pozwalający programiście na reagowanie na sytuacje błędne czy nietypowe – programista może wygenerować **wyjątek**.

Wyjątek (*exception*) to obiekt pewnej klasy. Wygenerowanie wyjątku polega na przekazaniu obiektu opisującego wyjątek z fragmentu kodu, w którym wystąpił problem, do fragmentu, w którym przewidziano jego obsługę.

```
class itemsError
{

};
```

Wygenerowanie wyjątku powoduje przerwanie wykonywania sprawiającego problemy kodu i przejście do obsługi sytuacji problematycznej. Obsługa ta może znajdować się w innym miejscu kodu.

```
int getItems()
{
  int numOfItems;
  cin >> numOfItems;

  if(numOfItems <= 0)
    throw itemsError();

  ...
}
```

Wyjątek jest obiektem, jego klasa określa typ sytuacji wyjątkowej. Obiekt może w sobie posiadać pola oraz funkcje składowe, pozwalające na sprecyzowanie informacji o zaistniałej sytuacji wyjątkowej.

```
try
{
  getItems();
}

catch(itemsError)
{
  cout << "Blad!";
}
```

Jeśli wyjątek nie zostanie obsłużony to program zostanie awaryjnie zakończony, dalsza część programu się nie wykona.

Zgłaszanie wyjątków

Wyjątki są generowane instrukcją `throw`. Po słowie kluczowym `throw` występuje dowolne wyrażenie pewnego typu. Wartość tego wyrażenia jest zgłaszana jako wyjątek. Wyjątki mogą być również typu wbudowanego.

```
if(numOfItems <= 0)
  throw 1;
```

```
if(numOfItems <= 0)
  throw "Liczba ujemna";
```

```
try
{
  getItems();
}

catch(int)
{
  cout << "Blad typu int";
}
```

```
try
{
  getItems();
}

catch(char const *)
{
  cout << "Blad typu char const*";
}
```

Tak zdefiniowana obsługa jest wrażliwa na różne typy wyjątków. W obrębie danego typu wyjątku, różne jego wartości nie są rozróżniane.

Co się dzieje po zgłoszeniu wyjątku?

Po wygenerowaniu wyjągtkiu indtrukcją `throw`, biblioteka czasu wykonania (*RTL*, *run time library*) wykonuje następujące kroki:

- pobiera wyjątek, określa jego typ,
- przeszukuje stos wywołań funkcji w poszukiwaniu takiej, która zawiera obsługę wyjątku tego typu (czyli odpowiednią instrukcję `catch`),
- jeżeli odpowiednia obsługa wyjątku zostanie znaleziona, stos jest w tym miejscu rozwijany (*unwind*) i wyjątek jest obsługiwany,
- jeżeli w trakcie tych poszukiwań nie zostanie znaleziony pasujący blok obsługi wyjątku, dochodzimy do punktu wejściowego programu, czyli funkcji `main()`,
- jeżeli tutaj też nie ma obsługi zgłoszonego wyjątku, program zostanie przerwany w trybie awaryjnym,
- w trakcie przechodzenia do kolejnych bloków na stosie wywołań, usuwane są wszystkie obiekty automatyczne, czemu towarzyszy aktywowanie ich destruktorów.

Propagacja wyjątku kończy się:

- jego obsługą w odpowiednim bloku `catch`, po czym wykonane zostaną kolejne instrukcje następujące po tym bloku,
- przerwaniem wykonania programu, jeśli wyjątek nie zostanie obsłużony.

Proces obsługi wyjątku jest jednokierunkowy. W jego trakcie niszczone są obiekty automatyczne, a sterowanie **nigdy nie wróci** automatyczne do miejsca zgłoszenia wyjątku.

Jeśli w trakcie usuwania obiektów automatycznych jakiś destruktor zgłosi wyjątek i nie obsłuży, następuje **awaryjne przerwanie** programu.

Awaryjne przerywanie programu

W przypadku wystąpienia nieobsłużonego wątku, wywoływana jest biblioteczna funkcja `terminate()`. Domyślnie funkcja ta wywołuje funkcję `abort()` pochodzącą ze standardowej biblioteki `stdlib.h`.

Funkcja `abort()` nie zapewnia aktywowania destruktorów dla obiektów globalnych i statycznych.

Można zmienić działanie funkcji `terminate()` poprzez zainstalowanie własnej funkcji kończącej, służy do tego funkcja `set_terminate()`.

Własna funkcja kończąca musi być bezargumentową funkcją o rezultacie typu `void`.

Rezultatem wywołania funkcji `set_terminate()` jest wskaźnik na poprzednio zainstalowaną funkcję.

Można zatem zapamiętać adres oryginalnej funkcji kończącej i przywrócić ją w razie konieczności.

Własna funkcja kończąca

```
#include <exception>
#include <iostream>
using namespace std;

void myTterminator()
{
  cout << "I'll be back!" << endl;
  updateErrorsLog();
  cin.get();
  exit(0);
}

void(*oldTerminator)() = set_terminate(myTerminator);

void throwFoo()
{
  throw 1;
}

int main()
{
  throwFoo();
  return 0;
}
```

Większa liczba rodzajów wątków

Wyjątki mogą być różnych typów. Można zatem rozróżniać rodzaje zgłaszanych sytuacji wyjątkowych.

```
class itemsNormalError
{
};

class itemsCriticalError
{
};
```

```
int getItems()
{
  int numOfItems;
  cin >> numOfItems;

  if(numOfItems <= 0)
    throw itemsNormalError();
  else
  {
    ...
    throw itemsCriticalError();
  }
}
```

Wyłapywanie wszystkich klas wyjątków.

```
class itemsNormalError
{
};

class itemsCriticalError
{
};
```

```
try
{
  getItems();
}

catch(itemsCriticalError)
{
  cout << "Blad krytyczny";
}

catch(itemsNormalError)
{
  cout << "Blad zwykly";
}
```

Wychwytywanie różnych klas wątków za jednym *zamachem*.

```
try
{
  getItems();
}

catch(...)
{
  cout << "Biore wszystkie!";
}
```

Wychwytywanie wybranych wątków i *hurtowa* obsługa reszty.

```
try
{
  getItems();
}

catch(itemsCriticalError)
{
  cout << "Blad krytyczny";
}

catch(...)
{
  cout << "I reszta";
}
```

Dane w wyjątkach

Do tychczas wątki były rozróżniane za pomocą ich typów. Inny typ wyjątku – inna sytuacja wyjątkowa.

Wyjątki to obiekty. Mogą posiadać pola, można je inicjować i odczytywać.
W polach obiektu stanowiącego wyjątek można przechowywać informację o różnych przyczynach powstania wyjątku.

```
class itemsError
{
  public:
    enum {CRITICAL, NORMAL};

    itemsError(int code): errCode(code) {}
    int getCode() const {return errCode;}

  private:
    int errCode;
};
```

Generowanie wyjątku jednego typu ze zróżnicowaniem kodu pamiętanego w polu klasy.

```
int getItems()
{
  int numOfItems;
  cin >> numOfItems;

  if(numOfItems <= 0)
    throw itemsError(itemsError::NORMAL);
  else
  {
    ...
    throw itemsError(itemsError::CRITICAL);
  }
}
```

Wyłapanie wyjątku i identyfikacja kodu błędu.

```
try
{
  getItems();
}

catch(itemsError &error)
{
  switch(error.getCode())
  {
    case itemsError::CRITICAL:
      cout << "Wyjatek krytyczny";
      break;

    case itemsError::NORMAL:
      cout << "Wyjatek zwykly";
      break;
    }
}
```

Eleganckie podejście obiektowe z hermetyzacją kodu błędu.

```
class itemsError
{
  public:
    enum {CRITICAL, NORMAL};

    itemsError(int code): errCode(code) {}
    int getCode() const {return errCode;}
    char * errorName();

  private:
    int errCode;
};

char * itemsError::errorName()
{
  switch( errCode )
  {
    case CRITICAL: return "Wyjatek krytyczny";
    case NORMAL  : return "Wyjatek zwykly";
    default      : return "Wyjatek nieznany";
  }
}
```

Wyjątek sam się przedstawia.

```
try
{
  getItems();
}

catch(itemsError &error)
{
  cout << itemsError.errorName();
}
```

Sterta

Sterta (*heap*) to wydzielony obszar pamięci:

- przeznaczony do przechowywania danych dynamicznych,
- kontrolowany **ręcznie** przez programistę,
- **ograniczony** pod względem rozmiaru.

Rozmiar sterty zmienia się wraz ze zmianą środowiska systemowego i kompilatora. Zwykle jest on jednak wystarczający dla typowych programów. Jednak przetwarzanie grafiki czy danych multimedialnych może wymagać ustawień specyficznych.

Typowy scenariusz wykorzystania sterty:

- przydział niezbędnej w danym momencie ilości pamięci – najpóźniej jak to możliwe,
- wykorzystanie przydzielonego obszaru,
- zwolnienie przydzielonej pamięci natychmiast, gdy nie jest już używana.

W języku C dynamiczny przydział i zwalnianie pamięci realizowały funkcje:

- `void * malloc(size_t size)`,
- `void * calloc(size_t nitems, size_t size)`,
- `void * realloc(void *ptr, size_t size)`,
- `void free(void *ptr)`.

W języku C++ dynamiczny przydział i zwalnianie pamięci realizować będzie:

- operator `new`,
- operator `delete`.

Przydzielanie i zwalnianie pamięci dla obiektu

Użycie operatora new powoduje przydzielenie pamięci dla obiektu w obszarze zwanym stertą, oraz inicjalizację tego obiektu z wykorzystaniem odpowiedniego konstruktora.

```
TSquare *p;
p = new TSquare;

cout << p->getSide();

delete p;
```

Można sterować rodzajem inicjalizacji.

```
Square *a = new Square;       ← konstruktor domyślny
Square *b = new Square(10);   ← konstruktor ogólny
Square *c = new Square(*b);   ← konstruktor kopiujący
```

Usuwanie obiektu. Operator delete wywołuje destruktor i zwalnia przydzieloną pamięć.

```
delete a;
delete b;
delete c;
```

- Operator `delete` może być stosowany wyłącznie dla obiektów utworzonych operatorem `new`.
- Mieszanie operatorów `new` i `delete` z funkcjami typu `malloc()` i `free()` daje niezdefiniowane rezultaty.
- Jeżeli operator `delete` zostanie użyty dla wskaźnika zerowego, nic się nie stanie a operacja ta nie jest błędna.
- Dwukrotne (lub wielokrotne) usunięcie tego samego obiektu jest błędem i prowadzi do niezdefiniowanych rezultatów.
- Odwoływanie się do obszaru pamięci, który został wcześniej zwolniony jest błędem zarówno w C jak i w C++.

Dobrą praktyką jest zerowanie wskaźnika tuż po zwolnieniu pamięci, pozwala to na kontrolowanie czy można odwołać się do obiektu wskazywanego w innym miejscu programu.

```
TSquare *p = new TSquare;
...
delete p;
p = 0;
```

```
if(p) // if(p != 0) lub if(p != NULL)
{
  p->setSide(100);
  ...
}
```

W języku C++ **można zakładać**, że wskaźnik pusty (pokazujący na nic) ma wartość **zero**. Korzystanie ze stałej `NULL` wymaga włączenia przynajmniej pliku nagłówkowego `cstddef`.

Przydzielanie pamięci dla tablic obiektów

Operator new może być wykorzystany do utworzenia tablicy obiektów.

Stosowanie tego operatora zapewnia, że dla każdego elementu tablicy zostanie aktywowany konstruktor domyślny.

```
TSquare *arrS = new TSquare[10];
```

Tak utworzona tablica może być wykorzystana w normalny dla tablic sposób.

```
for(int i = 0; i < 10; i++)
  cout << arrS[i].getSide();
```

Taką tablicę należy jednak usunąć w specyficzny sposób:

```
delete [] arrS;
```

Zapewni to uaktywnienie destruktora (o ile istnieje) dla każdego elementu tablicy.

Tablice dynamiczne mogą mieć rozmiar określony zmienną.

```
int numOfSquares;

cout << "Podaj liczbe kwadratow:";
cin >> numOfSquares;

Square *arrS = new Square[numOfSquares];

for(int i = 0; i < numOfSquares; i++)
  cout << arrS[i].getSide() << endl;

delete [] tabS;
```

Operatory `new` i `delete` mogą być stosowane dla typów wbudowanych.

```
int n;

cout << "Podaj liczbe elementow:";
cin >> n;

int *arrI = new int[n];

for(int i = 0; i < n; i++)
  cout << arrI[i] << endl;

delete [] arrI;
```

Należy uważać, aby nie utracić dynamicznie przydzielanych obszary pamięci.

```
int n;

cout << "Podaj liczbe elementow:";
cin >> n;

int *arrI = new int[n];

...
arrI = new int[10];  → wcześniej przydzielony obszar został utracony
...

for(int i = 0; i < n; i++)
  cout << arrI[i] << endl;

delete [] arrI;
```

Można temu częściowo zaradzić stosując wskaźnik, który nie może być później modyfikowany.

```
int *const tabI = new int[n];
```

Przydzielanie pamięci dla tablic wielowymiarowych

Przy alokacji tablic wielowymiarowych wszystkie wymiary poza pierwszym muszą być nieujemnymi wyrażeniami o wartości znanej na etapie kompilacji. Pierwszy wymiar może być zadany zmienną.

```
const int MAX_LINE_LEN = 256;
int numOfLines;

char (* lines)[MAX_LINE_LEN];

cout << "Podaj liczbe linii:";
cin >> numOfLines;

lines = new char[numOfLines][MAX_LINE_LEN];

for(int i = 0; i < numOfLines; i++)
{
  sprintf(lines[i], "Linia nr: %-2d", i + 1);
  cout << endl << lines[i];
}

delete [] lines;
```

Za mało pamięci

Jeżeli operator `new` nie potrafi znaleźć ciągłego, wolnego obszaru pamięci dla obiektu:

- sprawdza czy programista zdefiniował specjalną funkcję obsługi takiej sytuacji,
- jeżeli taka funkcja istnieje, jest ona wywoływana,
- w przeciwnym wypadku generowany jest wyjątek,
- dawniej rezultatem operatora była wartość zero.

Własną funkcję obsługi braku pamięci można zdefiniować wykorzystując funkcję `set_new_handler()` zdefiniowaną w pliku nagłówkowym `new`. Musi to być bezparametrowa funkcja o rezultacie `void`.

```
void out_of_memory()
{
  cerr << "Brak pamieci";
  exit(EXIT_FAILURE);
}
```

Sprawdź kiedy skończy się pamięć

```
#include <iostream>
#include <new>

using namespace std;

void out_of_memory()
{
  cerr << "Brak pamieci";
  exit(EXIT_FAILURE);
}

int main()
{
  set_new_handler(out_of_memory);

  for(;;)
    new int[100000];

  return 0;
}
```

Co to takiego przeciążanie?

Przeciążanie funkcji (*function overloading*) to tworzenie większej liczby funkcji o takiej samej nazwie.

Nazwa funkcji może być zatem użyta wielokrotnie do realizacji różnych czynności. Jest więc *przeciążona* dodatkowymi *obowiązkami*.

Kompilator zadba o dobranie właściwej wersji funkcji przeciążonej w zależności od kontekstu jej wywołania.

```
int add(int a, int b)
{
  return a + b;
}

double add(double a, double b)
{
  return a + b;
}
```

```
cout << endl << "Dodawanie int:" << add(1, 1);
cout << endl << "Dodawanie double:" << add(1.0, 1.0);
```

Przeciążanie funkcji jest możliwe, jeżeli **sygnatury** funkcji przeciążonych różnią się od siebie.

Aby sygnatury były inne, funkcje muszą się różnić typami parametrów, liczbą parametrów lub jednocześnie typami i liczbą parametrów.

Funkcje o tych samych nazwach i listach parametrów, różniące się tylko typem rezultatu, nie są przez kompilator rozróżniane i powodują wystąpienie błędu kompilacji.

Dlaczego sygnatura funkcji nie uwzględnia rezultatu funkcji?

```
int addAndPrint(int a, int b)
{
  cout << a + b;
  return a + b;
}

void addAndPrint(int a, int b)
{
  cout << a + b;
}
```

Możliwe wywołania, powodujące błąd kompilatora.

```
int result = addAndPrint(10, 20);
```

```
addAndPrint(10, 20);
```

Delikatne problemy i zaskoczenia

Dane są następujące funkcje.

```
void printData(long num)
{
  cout << "Liczba long: " << num << endl;
}
void printData(float num)
{
  cout << "Liczba float: " << num << endl;
}
void printData(double num)
{
  cout << "Liczba double: " << num << endl;
}
```

Poprawne wywołania.

```
long ln = 200;
float fn = 200;
double dn = 200;

printData(ln);
printData(fn);
printData(dn);
```

Wywołanie niepoprawne (błąd kompilatora):

```
printData(200);
```

Wywołanie poprawne ale zaskakujące:

```
printData(3.14);  ← liczba double
```

W przypadku, gdyby istniały tylko dwie wersje funkcji:

```
void printData(long num)
{
  cout << "Liczba long: " << num << endl;
}

void printData(float num)
{
  cout << "Liczba float: " << num << endl;
}
```

wywołanie `printData(3.14);` spowodowałoby błąd.

Zasady definiowania funkcji

Sygnatury i typ rezultatu są jednakowe, czyli deklaracja tej samej funkcji. Różnice w nazwach parametrów nie są znaczące.

```
void printData(long num);

void printData(long number);
```

Sygnatury są jednakowe, typy rezultatów są różne, kompilator potraktuje drugą deklarację jako niepoprawną redeklarację tej samej funkcji i zgłosi błąd. Przy doborze wersji funkcji przeciążonej typ rezultatu nie jest brany pod uwagę.

```
void printData(long num);

long printData(long num);
```

Nazwa zdefiniowana przez `typedef` nie oznacza nowego typu.

```
typedef int integer;

int printData(int num);

integer printData(integer num);
```

Sygnatury obu funkcji różnią są typami parametrów lub ich liczbą, mamy dwie wersje funkcji przeciążonej.

```
int printData(int num);

long printData(long num);
```

Rozróżnianie wersji funkcji

Poszczególne wersje funkcji przeciążonej różni sygnatura funkcji – nazwa funkcji i lista parametrów.

Określenie, która wersja funkcji przeciążonej ma być wywołana nazywa się **rozróżnianiem wersji** lub **rozpoznawaniem wywołania** (*function call resolution*).

Najważniejszym elementem rozpoznawania wywołania jest dopasowywanie parametrów (*argument matching*). Polega on na porównaniu parametrów aktualnych wywołania z parametrami formalnymi funkcji.

Proces dopasowania parametrów może skończyć się jednym z wariantów:

- udane dopasowanie,
- dopasowanie nie jest możliwe,
- dopasowanie jest niejednoznaczne.

Rodzaje dopasowania

Przeciążona funkcja.

```
void printData(long num);
void printData(float num);
void printData(double num);
```

Udane dopasowanie.

```
printData(10L);
printData(10.5f);
printData(10.5);
```

Dopasowanie nie jest możliwe.

```
printData("10");
```

Dopasowanie jest niejednoznaczne.

```
printData(10);
```

Istnieją cztery sposoby uzyskania ścisłego dopasowania, stosowane w następującej kolejności:

- ścisła zgodność,
- zgodność dzięki **awansowaniu**,
- zgodność dzięki **standardowym przekształceniom** typów,
- zgodność dzięki przekształceniom typów **definiowanym** przez programistę.

Ścisła zgodność

Typy parametru aktualnego i formalnego są takie same.

```
void printData(int);
void printData(char *);

printData(0);           ← zgodna z printData(int)

printData("Napis");    ← zgodna z printData(char *)

printData((char *)0);  ← zgodna z printData(char *)
```

Zgodność dzięki awansowaniu

Jeżeli nie występuje ścisła zgodność, to przed następną próbą dopasowania, typ parametru aktualnego jest **rozszerzany** wg. następujących zasad:

- parametr typu `char`, `unsigned char` lub `short` awansują do typu int; jeżeli rozmiary typu `int` i `short` są równe, parametr `unsigned short` awansuje do `unsigned int`, w rzeciwnym wypadku do `int`,
- parametr typu `float` awansuje do typu `double`,
- parametr typu wyliczeniowego awansuje do typu `int`.

```
void printData(int);
void printData(short);
void printData(long);

printData('a');   ← zgodna z printData(int) - a awansuje do int
printData(3.14);  ← brak możliwości awansu - niejednoznaczne
```

```
void printData(char);
void printData(int);
void printData(unsigned int);

unsigned char uc;
printData(uc);  ← zgodna z printData(int), uc awansuje do int
```

Zgodność dzięki standardowym przekształceniom typów

Jeżeli nie występuje ścisła zgodność, nawet po zastosowaniu awansowania parametru aktualnego, przed następną próbą dopasowania zastosowane zostaną standardowe **przekształcenia typów** wg. następujących zasad:

- każdy argument aktualny **typu liczbowego** będzie zgodny z dowolnym **innym typem** liczbowym parametru formalnego,
- każdy argument aktualny typu **wyliczeniowego** będzie zgodny z dowolnym **typem liczbowym** parametru formalnego,
- literał `0` będzie zgodny z parametrem formalnym **typu wskaźnikowego** jak i **typu liczbowego**,
- wskaźnik do dowolnego typu będzie zgodny z parametrem formalnym typu `void *`.

```
void printData(int);
void printData(float);

printData(3.14);  ← dwa możliwe przekształcenia - niejednoznaczne
```

```
void printData(int);

printData(3.14);  ← zgodna z printData(int) - przekształccenie standardowe
```

Zgodność dzięki przekształceniom typów definiowanym przez programistę – z operatorem konwersji typu

Jeżeli żaden z poprzednich sposobów nie doprowadził do dopasowania parametrów to dokonuje się zdefiniowanych przez programistę przekształceń typów (jeśli istnieją).

Definiowanie przekształceń typów dotyczy klas, dla nich można definiować **operatory przekształcenia typu**. Takie operatory pozwalają na zdefiniowanie sposobu konwersji obiektu pewnej klasy na obiekt zadanego typu.

```
class myInt
{
  public:
    operator int();  ← operator przeksztalcenia obiektu myInt do typu int
};

myInt i;

void printData(int);
void printData(float);

printData(i);  ← zgodna z printData(int) - wykorzystanie przeksztalcenia do int
```

Zgodność dzięki przekształceniom typów definiowanym przez programistę – z wykorzystaniem konstruktora

Kompilator w trakcie dopasowywania parametrów będzie używał **konwersji konstruktorowych**.

Polega ona na niejawnym utworzeniu obiektu tymczasowego i zainicjowaniu go odpowiednim konstruktorem.

```
class myInt
{
  public:
    myInt();
    myInt(int i);  ← konstruktor inicjowany typem int
};

void printData(myInt &num);

printData(1);  ← zgodna z printData(myInt &) - utworzenie
                  niejawnego obiektu tymczasowego i zainicjowanie
                  go konstruktorem myInt(int): printData(myInt(1))
```

Dzięki możliwości przeciążania funkcji:

- można stosować jednakowe, spójne nazwy funkcji mimo różnic w typach i liczbie parametrów,
- poszczególne wersje funkcji mogą być prostsze i łatwiejsze do napisania, być może niektóre wersje funkcji będą wyraźnie szybsze.

Dobór odpowiedniej wersji funkcji przeciążonej:

- odbywa się na etapie kompilacji i jest pamiętany w kodzie wynikowym,
- nie wpływa na efektywność wykonania programu,
- jest prosty i oczywisty w przypadkach jednoznacznych,
- może powodować błędy kompilacji kub nieprzewidziane zachowanie kodu w przypadku gdy parametry wywołania **nie pasują** do parametrów formalnych funkcji.

Przeciążanie operatorów na przykładzie

Należy opracować program realizujący obliczenia arytmetyczne dla liczb zespolonych. Do realizacji tych obliczeń wykorzystać klasę reprezentującą liczbę zespoloną `TComplex`.

Za liczbę zespoloną z przyjmuje się liczbę w następującej postaci:

$$z = a + i \cdot b,$$

gdzie i jest *jednostką urojoną* o właściwości: $i^2 = -1$.

Liczbę $a = \Re(z)$ nazywamy *częścią rzeczywistą*, zaś liczbę $b = \Im(z)$ *częścią urojoną* liczby zespolonej z.

Podstawowe operacje na liczbach zespolonych:

- dodawanie

$$(a + i \cdot b) + (c + i \cdot d) = (a + c) + i \cdot (b + d),$$

- odejmowanie

$$(a + i \cdot b) - (c + i \cdot d) = (a - c) + i \cdot (b - d),$$

- mnożenie

$$(a + i \cdot b)(c + i \cdot d) = (ac - bd) + i \cdot (bc + ad),$$

- dzielenie

$$\frac{(a + i \cdot b)}{(c + i \cdot d)} = \frac{(ac + bd)}{(b^2 + c^2)} + i \cdot \frac{(bc - ad)}{(b^2 + c^2)}.$$

Przechowywanie wartości w klasie

Klasa `TComplex` posiada konstruktor ogólny, który dzięki parametrom domyślnym staje się domyślnym, kopiujący nie jest konieczny.

```
class TComplex
{
  public:
    TComplex(double re = 0, double im = 0);

    double getReal() const;
    double getImag() const;

    void setReal(double re);
    void setImag(double im);

  private:
    double real;
    double imag;
};
```

```
TComplex::TComplex(double re, double im): real(re), imag(im)
{
}

double TComplex::getReal() const
{
  return real;
}

double TComplex::getImag() const
{
  return imag;
}

void TComplex::setReal(double re)
{
  real = re;
}

void TComplex::setImag(double im)
{
  imag = im;
}
```

Użycie klasy

Kreowanie obiektów.

```
TComplex a;         ← konstruktor TComplex() i domyślne parametry
TComplex b(2);      ← konstruktor TComplex(2) i parametr domyślny
TComplex c(1, 1);   ← konstruktor TComplex(1, 1)
```

Dodawanie liczb zespolonych – wersja prymitywna.

```
a.setReal(b.getReal() + c.getReal());
a.setImag(b.getImag() + c.getImag());
```

Czy **można tak**?

```
a = b + c;
a = b * c;
```

W zasadzie tak, ale aktualnie nie. Kompilator nie wie jak dodawać, mnożyć, dzielić czy odejmować liczby zespolone (manipulowanie obiektami). Należy go tego nauczyć – czyli związać z klasą `TComplex` zestawu operatorów, realizujących określone działania.

Operatory są w naturalny sposób przeciążone. Operatory potrafią wykonywać określone działania dla danych różnych typów.

```
int ia, ib, ic;
ic = ia + ib;

float fa, fb, fc;
fc = fa + fb;
```

Koncepcja przeciążania operatorów w języku C++ nie jest nowa. Nowością jest potraktowanie użycia operatora jako wywołania specjalnej funkcji, zwanej funkcją operatorową.

Programista może napisać dla większości operatorów specjalne funkcje, określające sposób działania danego operatora dla argumentów, z których przynajmniej jeden jest obiektem.

Zasady wykorzystania funkcji operatorowych:

- nie wolno zmieniać znaczenia operatora określonego dla wbudowanych typów danych,
- nie wolno budować nowych operatorów,
- przynajmniej jeden argument funkcji operatorowej musi być obiektem jakiejś klasy,
- nie wolno zmieniać zdefiniowanych pierwotnie reguł pierwszeństwa operatorów,
- musi być zachowana liczba argumentów operatora.

Np. przeciążenie operatora przypisania dla klasy `TComplex`:

```
TComplex a;
TComplex b(1, 2);

a = b;
```

Spełnia powyższe zasady.

Definicja i implementacja funkcji operatorowej dla operatora =

```
class TComplex
{
  public:
    void operator = (TComplex &z);
  ...
};
```

```
void TComplex::operator = (TComplex &z)
{
  real = z.real; // setReal(z.getReal());
  imag = z.imag; // setImag(z.getImag());
}
```

Jak używać, jak to działa?

```
TComplex a;
TComplex b(1, 2);

a = b;
```

Przypisanie `a = b` spowoduje wywołanie `a.operator = (b)`.

Na rzecz obiektu `a` zostanie wywołana funkcja `TComplex::operator = ()` z argumentem `b`.

Czy taki operator zawsze działa jak należy?

```
TComplex a;
TComplex b;
TComplex c(1, 2);

a = b = c;
```

Można się spodziewać, przypisanie `a = b = c` powinno zadziałać w taki sposób `a.operator = (b.operator = (c))`.

Czy implantacja operatora pozwala na takie działanie (co zwraca operator)?

Po poprawkach, operator będzie działał poprawnie w przypadku wielokrotnego przypisania.

```
TComples & TComplex::operator = (TComplex &z)
{
  real = z.real;
  imag = z.imag;

  return *this;
}
```

```
TComplex a;
TComplex b;
TComplex c(1, 2);

a = b = c;
```

Przypisanie `a = b = c` spowoduje wywołanie `a.operator = (b.operator = (c))`.

Operator przypisania warto zabezpieczyć przed przypisaniem do samego siebie.

```
TComplex a;

a = a;
```

```
TComplex & TComplex::operator = (TComplex &z)
{
  if(&z != this)
  {
    real = z.real;
    imag = z.imag;
  }

  return *this;
}
```

Dodanie modyfikatora `const` do parametru funkcji operatorowej pozwoli na współpracę z obiektami *stałymi*.

```
TComplex & TComplex::operator = (const TComplex &z)
{
  if(&z != this)
  {
    real = z.real;
    imag = z.imag;
  }

  return *this;
}
```

```
const TComplex a(0, 0);
TComplex b;

b = a;
```

Należy pamiętać, że operator przypisania to co innego niż konstruktor kopiujący. Czasem łatwo się pomylić.

```
TComplex a(1, 2);
TComplex a = b;  ← konstruktor kopiujący

b = a;           ← operator przypisania
```

Czy trzeba przeciążać operator przypisania i definiować konstruktor kopiujący?

Jeżeli operator przypisania nie jest jawnie zdefiniowany dla danej klasy, a jest potrzebny kompilatorowi, generuje on domyślny operator przypisania, realizujący kopiowanie **pole-po-polu**.

Jeżeli konstruktor kopiujący nie jest jawnie zdefiniowany dla danej klasy, a jest potrzebny kompilatorowi, realizuje on inicjalizację obiektu wykorzystując kopiowanie **pole-po-polu**.

Używać czy nie używać?

- stosowanie konstruktora kopiującego i przeciążonego operatora przypisania jest dobrą, programistyczną praktyką,
- dzięki jawnym definicjom programista ma kontrolę nad kopiowaniem wartości, występujących w wielu, czasem zaskakujących sytuacjach,
- mało klas jest tak prostych, że nie potrzebują konstruktora kopiującego ani przeciążonego operatora przypisania.

Klasa TComplex po poprawkach

```
class TComplex
{
  public:
    TComplex(double re = 0, double im = 0);
    TComplex(const TComplex &z);

    double getReal() const;
    double getImag() const;

    void setReal(double re);
    void setImag(double im);

    TComplex & operator = (const TComplex &z);

  private:
    double real;
    double imag;
};
```

```
TComplex::TComplex(double re, double im): real(re), imag(im) {}
TComplex::TComplex(const TComplex &z): real(z.real), imag(z.imag) {}

double TComplex::getReal() const {return real;}
double TComplex::getImag() const {return imag;}

void TComplex::setReal(double newReal) {real = re;}
void TComplex::setImag(double newImag) {imag = im;}

TComplex & TComplex::operator = (const TComplex &z)
{
  if(&z != this)
  {
    real = z.real;
    imag = z.imag;
  }
  return *this;
}
```

Dodawanie liczb zespolonych

```
TComplex a;
TComplex b(2);
TComplex c(1, 1);

a = b + c;
```

Wyrażenie `a = b + c` spowoduje wywołanie `a.operator = (b.operator + (c))`.

W C i C++ rezultatem funkcji nie powinien być **wskaźnik ani referencja do zmiennych automatycznych** funkcji. Te lokowane są zwykle na stosie i po zakończeniu działania funkcji **przestają istnieć**.

Możliwa implementacja operatora dodawania.

```
class TComplex
{
  public:
    TComplex operator + (const TComplex &z);
  ...
};
```

```
TComplex TComplex::operator + (const TComplex &z)
{
  TComplex temp;
  temp.real = real + z.real;
  temp.imag = imag + z.imag;

  return temp;
}
```

Przykłady użycia operatorów:

```
TComplex a;
TComplex b(2);

a + b;
```

Wyrażenie spowoduje wywołanie `a.operator + (b)`.

```
TComplex a;
TComplex b(2);
TComplex c(1, 1);

a = b + c;
```

Wyrażenie spowoduje wywołanie `a.operator = (b.operator + (c))`.

Wersja elegancka – bez dodatkowych zmiennych pomocniczych z wykorzystaniem konstruktora.

```
class TComplex
{
  public:
    TComplex operator + (const TComplex &z);
  ...
};
```

```
TComplex TComplex::operator + (const TComplex &z)
{
  return TComplex(real + z.real, imag + z.imag);
}
```

Odejmowanie liczb zespolonych

W sposób analogiczny jak dla dodawania.

```
TComplex a;
TComplex b(2);
TComplex c(1, 1);

a = b - c;
```

```
class TComplex
{
  public:
    TComplex operator - (const TComplex &z);
  ...
};
```

```
TComplex TComplex::operator - (const TComplex &z)
{
  return TComplex(real - z.real, imag - z.imag);
}
```

Zmiana znaku liczb zespolonych

Należy rozszerzyć klasę o kolejną funkcję operatorową (przeciążaną) dla operatora w wersji unarnej. Operator **nie posiada drugiego argumentu**.

```
TComplex a;
TComplex b(2, 2);

a = -b;
```

```
class TComplex
{
  public:
    TComplex operator - (const TComplex &z);
    TComplex operator - ();  ← unarny minus
  ...
};
```

```
TComplex TComplex::operator - ()
{
  return TComplex(-real, -imag);
}
```

Przykłady użycia operatorów:

```
TComplex a;
TComplex b(2);

a = -b;
```

Wyrażenie spowoduje wywołanie `a.operator = (b.operator - ())`.

```
TComplex a;
TComplex b(2);
TComplex c(1, 1);

a = b - c;
```

Wyrażenie spowoduje wywołanie `a.operator = (b.operator - (c))`.

Dla kompletności implementacji należy zdefiniować unarny operator dodawania:

```
class TComplex
{
  public:
    TComplex operator + (const TComplex &z);
    TComplex operator + ();  ← unarny plus
  ...
};
```

```
TComplex TComplex::operator + ()
{
  return TComplex(real, imag);
}
```

Przykłady użycia:

```
TComplex a;
TComplex b(2);

a = +b;
```

Wyrażenie spowoduje wywołanie `a.operator = (b.operator + ())`.

Operatory a różne typy

W przypadku różnych typów argumentów operator może nie działać.

```
TComplex z;
double d;

z + d;
d + z;
```

Wyrażenia spowodują wywołania:

`z.operator + (TComplex(d))`

`d.operator + (z)` ???

Funkcja operatorowa **nie musi** być (z pewnymi wyjątkami, np. przypisanie) zdefiniowana jako **funkcja składowa** a jako zwykła funkcja, posiadająca przynajmniej jeden argument będący obiektem.

Można to wykorzystać do definiowania funkcji operatorowych współpracujących z różnymi typami parametrów. Na przykład liczba zespolona (obiekt) i rzeczywista (typ wbudowany).

Dla jednoargumentowego operatora `#` i obiektu `0`:

```
#0;
```

wyrażenie spowoduje wywołanie:

- funkcji składowej `0.operator # ()`,
- funkcji zwykłej (nieskładowej) `operator #(0)`.

Dla dwuargumentowego operatora `#` i obiektów `A` i `B`:

```
A # B;
```

wyrażenie spowoduje wywołanie:

- funkcji składowej `A.operator # (Y)`,
- funkcji zwykłej (nieskładowej) `operator # (A, B)`.

Implementacja operatora dodawania w postaci zwykłej funkcji.

```
TComplex operator + (double n, const TComplex &z)
{
  return TComplex(n + z.getReal(), z.getImag());
}
```

```
TComplex a;
TComplex b(2);
double d;

d + b;
a = d + b;
```

Wyrażenia spowodują wywołania:

```
operator + (d, b)
a.operator = (operator + (d, b))
```

Nieskładowa funkcja operatorowa ma utrudniony dostęp do pól klasy.

```
TComplex operator + (double n, const TComplex &z)
{
  return TComplex(n + z.getReal(), z.getImag());
}
```

Aby funkcja nieskładowa miała wygodniejszy dostęp do pól klasy, może się z nią **zaprzyjaźnić** – modyfikator `friend`. Taka funkcja będzie miała bezpośredni dostęp do prywatnych pól klasy.

```
class TComplex
{
  friend TComplex operator + (double n, const TComplex &z);
  ...
};

TComplex operator + (double n, const TComplex &z)
{
  return TComplex(n + z.real, z.imag);
}
```

Kilka przykładów na różne okazje.

```
class TComplex
{
  friend TComplex operator + (double n, const TComplex &z);
  friend TComplex operator + (const TComplex &a, double n);
  friend TComplex operator + (const TComplex &a, const TComplex &b);
  ...
};
```

```
TComplex operator + (double n, const TComplex &z)
{
  return TComplex(n + z.real, z.imag);
}

TComplex operator + (const TComplex &z, double n)
{
  return TComplex(n + z.real, z.imag);
}

TComplex operator + (const TComplex &a, const TComplex &b)
{
  return TComplex(a.real + b.real, a.imag + b.imag );
}
```

Wykorzystanie funkcji:

```
TComplex a, b, c;
TComplex d(2);
TComplex e(2, 2);
double n;

a = n + c;
b = d + 2;
c = d + e;
```

Wyrażenia spowodują wywołania:

```
a.operator = (operator + (n, c)
b.operator = (operator + (d, double(2)))
c.operator = (operator + (d, e))
```

W przypadku *braku* odpowiednich funkcji operatorowych dla typów, np.przy deklaracji:

```
class TComplex
{
  friend TComplex operator + (const TComplex &a, const TComplex &b);
  ...
};
```

i użyciu:

```
TComplex a, b, c;
TComplex d(2);
TComplex e(2, 2);

a = 6 + c;
b = d + 9;
c = 3 + 5;
```

Wyrażenia spowodują wywołania:

```
a.operator = (operator + (TComplex(double(6)), c))
b.operator = (operator + (d, TComplex(double(9))))
c.operator = (operator + (TComplex(double(3)), TComplex(double(5))))
```

Kompilator będzie próbował rzutować typy argumentów, żeby dopasować odpowiednią funkcję.

Taki postępowanie może zakończyć się niepowodzeniem.

Wtedy można przeciążyć operator rzutowania typu ( ), aby mieć pewność, że typy są zmieniane w odpowiedni i kontrolowany sposób.

Wybór sposobu implementacji (funkcje operatorowe dla różnych typów albo definicja funkcji operatorowych rzutowania) zależy od programisty.

Operatory przedrostkowe i przyrostkowe

Można przeciążać operatory **przedrostkowe** i **przyrostkowe**. Wersje przedrostkowe definiuje się w poznany sposób.

Wersje przyrostkowe definiuje się przez funkcje operatorową z parametrem `int`. Parametr nie ma żadnego znaczenia praktycznego, jest istotny tylko ze względu na syntaktykę języka.

```
class TComplex
{
  public:
    TComplex & operator ++ ();    ← postać przedrostkowa
    TComplex operator ++ (int);   ← postać przyrostkowa

    TComplex & operator -- ();    ← postać przedrostkowa
    TComplex operator -- (int);   ← postać przyrostkowa
  ...
};
```

Przykład implementacji operatorów przedrostkowych:

```
TComplex & TComplex::operator ++ ()
{
  ++real;
  ++imag;

  return *this;  ← zwraca zmodyfikowany obiekt
}

TComplex & TComplex::operator -- ()
{
  --real;
  --imag;

  return *this;  ← zwraca zmodyfikowany obiekt
}
```

Wywołanie operatorów przedrostkowych:

```
TComplex a;
TComplex b(2);

a = ++b;
a = --b;
```

Wyrażenia spowodują wywołania:

```
a.operator = (b.operator ++ ())
a.operator = (b.operator -- ())
```

Przykład implementacji operatorów przyrostkowych:

```
TComplex TComplex::operator ++ (int)
{
  TComplex copy;

  copy = *this;  ← kopia obiektu, przypisanie
  ++real;
  ++imag;

  return copy;   ← zwraca kopię
}

TComplex TComplex::operator -- (int)
{
  TComplex copy(*this);  ← kopia obiektu, konstruktor kopiujący

  --real;
  --imag;

  return copy;           ← zwraca kopię
}
```

Wywołanie operatorów przyrostkowych:

```
TComplex a;
TComplex b(2);

a = b++;
a = b++;
```

Wyrażenia spowodują wywołania:

```
a.operator = (b.operator ++ (0))
a.operator = (b.operator -- (0))
```

operatory, które można przeciążać

`+`	`-`	`*`	`/`	`%`	`~`	`&`	`|`
`^`	`!`	`,`	`=`	`<`	`>`	`<=`	`>=`
`++`	`--`	`<<`	`>>`	`==`	`!=`	`&&`	`||`
`+=`	`-=`	`/=`	`%=`	`^=`	`&=`	`|=`	`*=`
`<<=`	`>>=`	`[]`	`()`	`->`	`->*`	`new`	`delete`

operatory, których przeciążać nie wolno

`::`	`.*`	`.`	`?:`

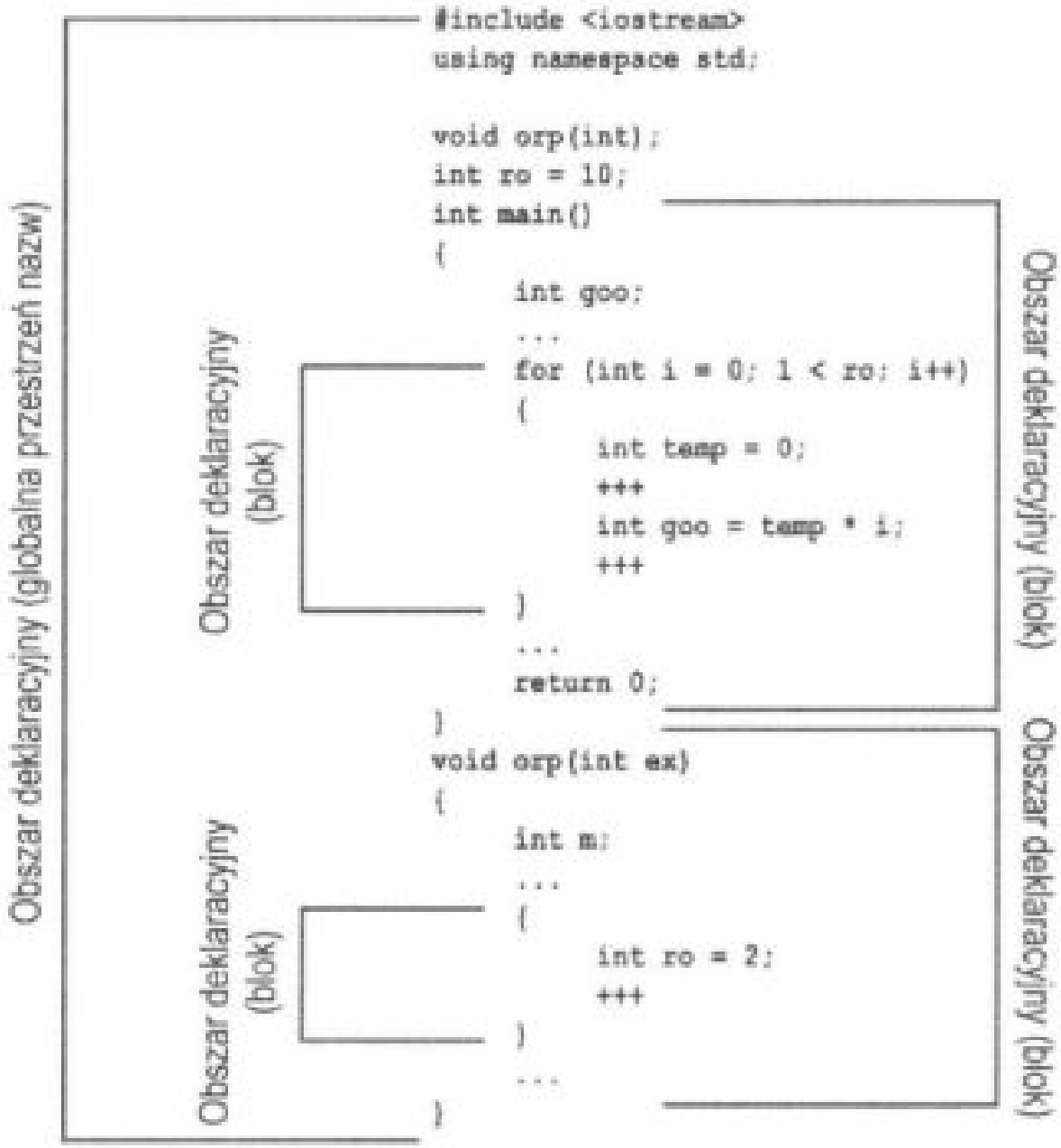
Tradycyjne przestrzenie nazw

Nazwy (identyfikatory) mogą odnosić się do zmiennych, funkcji, struktur, wyliczeń, klas i składowych struktur i klas.

Kiedy projekt programistyczny rozrasta się, rośnie też ryzyko kolizji tych nazw. Ryzyko to zwiększa się jeszcze bardziej, jeśli w projekcie wykorzystywane są biblioteki pochodzące z różnych źródeł.

Obszar deklaracyjny (*declarative region*) to taki obszar, w którym można umieszczać deklaracje: np.: zmienne globalne można deklarować poza wszelkimi funkcjami. Obszarem deklaracyjnym dla tego rodzaju zmiennych jest obszar pliku, w których są deklarowane.

Jeśli zmienna zostanie zadeklarowana w funkcji, jej obszar deklaracyjny będzie ograniczony do bloku, w którym została zadeklarowana.

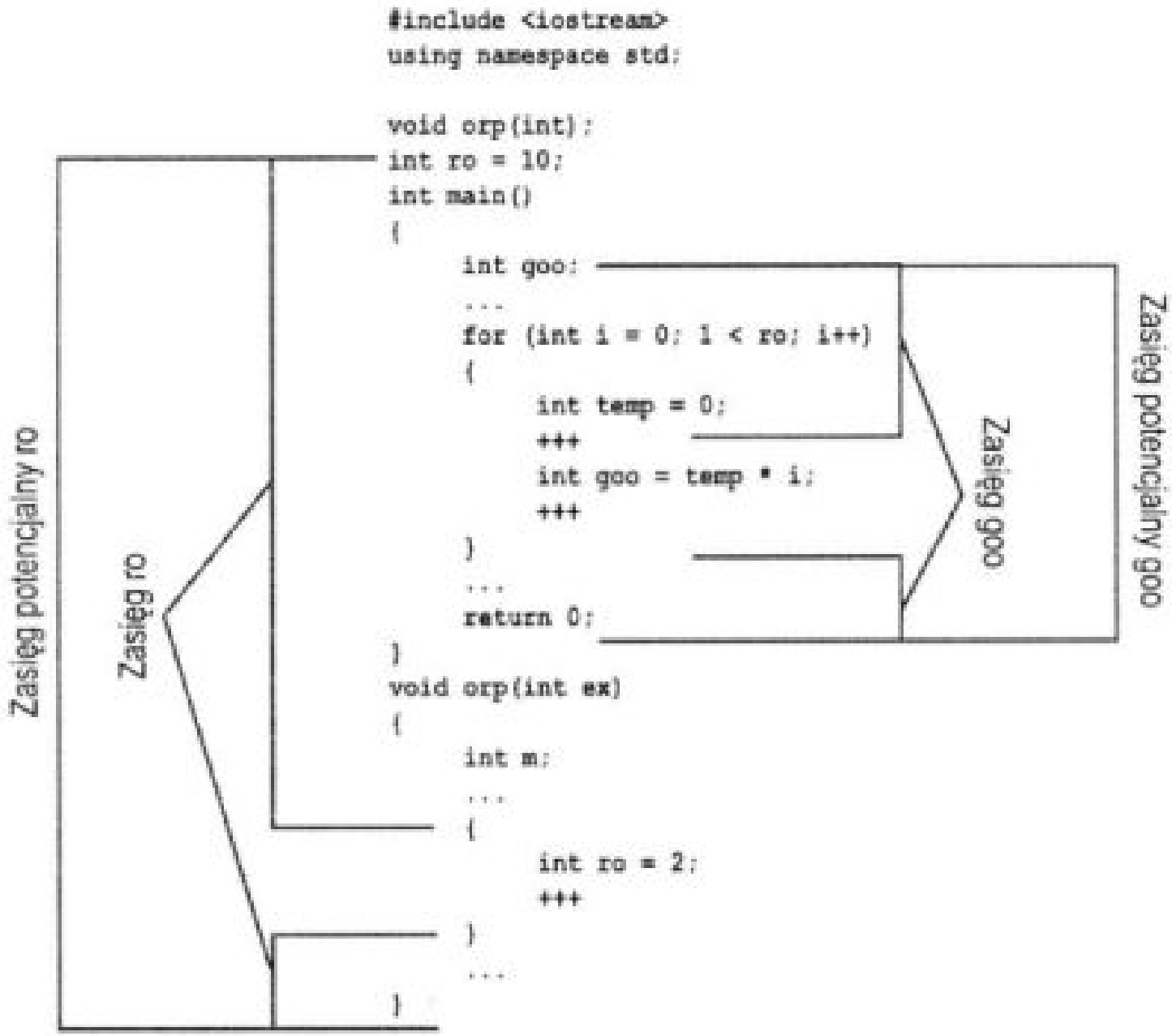

Obszar deklaracyjny (globalna przestrzeń nazw)
Obszar deklaracyjny (blok)
Obszar deklaracyjny (blok)
Obszar deklaracyjny (blok)
Obszar deklaracyjny (blok)
#include <iostream>
using namespace std;
void orp(int);
int ro = 10;
int main()
{
int goo;
...
for (int i = 0; 1 < ro; i++)
{
int temp = 0;
+++
int goo = temp * i;
+++
}
...
return 0;
}
void orp(int ex)
{
int m;
...
{
int ro = 2;
+++
}
...
}

Zasięg potencjalny (*potential scope*) zmiennej rozciąga się od miejsca deklaracji do końca obszaru dekoracyjnego. Zasięg potencjalny jest więc węższy od obszaru deklaracyjnego, bo można korzystać ze zmiennej przed miejscem jej definicji.

Zmienna nie musi jednak być widoczna w całym jej zasięgu potencjalnym. Może na przykład zostać przesłonięta przez inną zmienną o takiej samej nazwie zadeklarowanej w zagnieżdżonym obszarze deklaracyjnym.

Na przykład zmienna lokalna funkcji (dla mej obszarem deklaracyjnym jest funkcja) zakrywa zmienną globalną zadeklarowaną w tym samym pliku (jej obszar deklaracyjny to plik).

Ta część programu, dla której zmienna jest widoczna, to **zasięg zmiennej** i w takim znaczeniu stosowaliśmy ten termin dotychczas.


#include <iostream>
using namespace std;
void orp(int);
int ro = 10;
int main()
{
int goo;
...
for (int i = 0; i < ro; i++)
{
int temp = 0;
+++
int goo = temp * i;
+++
}
...
return 0;
}
void orp(int ex)
{
int m;
...
{
int ro = 2;
+++
}
...
}
Zasięg potencjalny ro
Zasięg ro
Zasięg goo
Zasięg potencjalny goo

Nowe mechanizmy przestrzeni nazw

C++ dodał możliwość tworzenia **nazwanych przestrzeni nazw**, wyznaczających nowego rodzaju obszary **dcklaracyjne**, których głównym zadaniem jest udostępnienie miejsca do deklaracji nazw.

Nazwy z jednej przestrzeni nazw nie kolidują z identycznymi nazwami deklarowanymi w innych przestrzeniach nazw.

Dodatkowo istnieją mechanizmy pozwalające poszczególnym częściom programu na korzystanie z nazw deklarowanych w przestrzeni nazw.

```
namespace jack
{
  void fetch();      ← prototyp funkcji
  int bucket;        ← deklaracja zmiennej
  double capacity;   ← deklaracja zmiennej
  struct well        ← deklaracja struktury
  {
    ...
  };
}
```

```
namespace jill
{
  double fetch;      ← deklaracja zmiennej
  int bucket();      ← prototyp funkcji
  double capacity;   ← deklaracja zmiennej
  struct hill        ← deklaracja struktury
  {
    ...
  };
}
```

Przestrzenie nazw mogą być lokowane na poziomie globalnym albo wewnątrz innych przestrzeni nazw, **nie mogą** jednak być **zagnieżdżane w blokach**.

Stąd nazwa zadeklarowana w przestrzeni nazw cechuje się domyślnie łączeniem zewnętrznym (chyba że odnosi się do stałej).

Poza przestrzeniami nazw definiowanymi przez programistę wyróżniona została **globalna** przestrzeń nazw. Odpowiada ona obszarowi deklaracyjnemu pliku.

To, co dotychczas było określane mianem **zmiennych globalnych**, możemy nazywać elementami **globalnej przestrzeni nazw**.

Nazwy w dowolnych przestrzeniach nazw nie kolidują ze sobą. Stąd nazwa `fetch` z przestrzeni `jack` może współistnieć z taką nazwą z przestrzeni `jill`.

Podobnie `hill` umieszczona w przestrzeni nazw `jill` może współistnieć z inną nazwą `hill` zewnętrzną wobec tej przestrzeni nazw.

Reguły obowiązujące deklaracje i definicje w przestrzeniach nazw są identyczne jak stosowne reguły definicji i deklaracji globalnych.

Przestrzenie nazw są **otwarte**, co oznacza, że można ją uzupełniać innymi nazwami.

```
namespace jill
{
  string goose(const string);
}
```

Podobnie z definicjami uzupełniającymi prototypy.

```
namespace jack
{
  void fetch()
  {
    ...
  }
}
```

Odwołanie do przestrzeni nazw

Najprostszym mechanizmem umożliwiającym odwołanie o nazw wewnątrz przestrzeni nazw jest operator zasięgu `::`.

```
jack::bucket = 5;
jill::hill mountain;
jack::fetch();
```

Identyfikator pozbawiony operatora zasięgu (np. `bucket`) to **nazwa niekwalifikowana**.

Identyfikatory poprzedzone operatorem zasięgu (np. `jack::bucket`) to nazwy **kwalifikowane**.

Deklaracja i dyrektywa `using`

Konieczność kwalifikowania wszystkich nazw jest kłopotliwa.

Język C++ przewiduje dwa dodatkowe mechanizmy upraszczające stosowanie przestrzeni nazw: **deklaracje** `using` i **dyrektywy** `using`.

Deklaracje `using` pozwalają na lokalne udostępnianie wybranych identyfikatorów przestrzeni nazw.

Dyrektywy `using` udostępniają zaś lokalnie całą zawartość przestrzeni nazw.

Deklaracja zawiera kwalifikowaną nazwę poprzedzoną słowem `using`.

```
using jill::fetch;
```

Deklaracja dodaje wskazaną nazwę do obszaru deklaracyjnego, w którym wystąpiła.

Po wystąpieniu takiej deklaracji można zamiast `jill::fetch` stosować nazwę `fetch`.

Deklaracja `using` wprowadza nazwę do obszaru deklaracyjnego, w którym wystąpiła. Użycie `using jill::fetch` w funkcji `main()` wprowadza nazwę `fetch` do obszaru deklaracyjnego wyznaczonego funkcją.

```
namespace jill
{
  double fetch;
  int bucket();
  double capacity;
  struct hill { ... };
}
```

```
char fetch;             ← deklaracja globalnej zmiennej fetch

int main()
{
  using jill::fetch;    ← wprowadzenie fetch do lokalnego obszaru deklaracyjnego
  double fetch;         ← błąd, nazwa fetch jest już zadeklarowana

  cin >> fetch;         ← odczyt wartości do jill::fetch
  cin >> ::fetch;       ← odczyt wartości do globalnej zmiennej fetch
}
```

Umieszczenie deklaracji `using` na zewnątrz funkcji wprowadza nazwę do globalnej przestrzeni nazw pliku.

```
namespace jill
{
  double fetch;
  int bucket();
  double capacity;
  struct hill { ... };
}
```

```
void foo();

using jill::fetch;  ← wprowadzenie fetch do globalnej przestrzeni nazw pliku

int main()
{
  cin >> fetch;     ← odczyt wartości do jill::fetch
  foo();
}

void foo()
{
  cout << fetch;    ← wyświetlenie jill::fetch
}
```

Dyrektywa `using` zawiera określenie przestrzeni nazw poprzedzone słowami `using namespace` i udostępnia wszystkie nazwy ze wskazanej przestrzeni nazw, które od tej pory można stosować z pominięciem operatora zasięgu.

```
using namespace jack;
```

Umieszczenie dyrektywy `using` na poziomie globalnym (poziomie pliku) czyni zawartość przestrzeni nazw dostępną globalnie (w pliku).

```
# include <iostream>

using namespace std;

int main()
{
 ...
}
```

Osadzenie dyrektywy `using` we wnętrzu funkcji udostępnia nazwy przestrzeni nazw w obrębie tej jednej funkcji.

```
# include <iostream>

int main()
{
  using namespace jack;
  ...
}
```

Stosowanie dyrektywy `using` celem wciągania kompletu nazw z przestrzeni nazw nie jest równoznaczne wielokrotnemu zastosowaniu deklaracji `using` dla poszczególnych nazw.

Dyrektywa bardziej przypomina **masowe zastosowanie** operatora zasięgu.

Po deklaracji `using` nazwa jest traktowana jak zadeklarowana w miejscu wystąpienia deklaracji `using`. Jeśli w danej funkcji zadeklarowana jest już nazwa, nie można takiej nazwy wciągnąć do funkcji deklaracją `using`.

Przy zastosowaniu dyrektywy `using` rozstrzyganie nazw odbywa się tak, jakby nazwy te zostały zadeklarowane w najmniejszym możliwym obszarze deklaracyjnym zawierającym zarówno przestrzeń nazw, jak i dyrektywę `using`.

Jeśli dyrektywa `using` zaimportuje nazwę, która jest już zdefiniowana w funkcji, nazwa lokalna będzie zakrywać nazwę importowaną z przestrzeni nazw, jak i ewentualną nazwę deklarowaną globalnie.

Wciąż można jednak odwołać się lokalnie do nazwy z przestrzeni nazw, stosując operator zasięgu.

```
namespace jill
{
  double fetch;
  int bucket();
  double capacity;
  struct hill { ... };
}
```

```
char fetch;                ← deklaracja globalnej zmiennej fetch
int main()
{
  using namespace jill;    ← import wszystkich nazw przestrzeni jill
  hill mountain;           ← utworzenie struktury typu jill::hill
  int water = bucket();    ← wywołanie funkcji jill::bucket
  double fetch;            ← poprawnie, zakrycie jill::fetch

  cin >> fetch;            ← odczyt wartości do lokalnej nazwy fetch
  cin >> ::fetch;          ← odczyt wartości do globalnej nazwy fetch
  cin >> jill::fetch;      ← odczyt wartości do nazwy jill::fetch
}

void foo()
{
  hill top;                ← błąd
  jill::hill crest;        ← poprawnie
}
```

W funkcji `main()` nazwa `jill::fetch` jest wciągana do lokalnej przestrzeni nazw. Nie posiada jednak lokalnego zasięgu, więc nie zakrywa globalnej wersji `fetch`.

Deklarowana lokalnie `fetch` zakrywa zarówno `jill::fetch`, jak i globalną wersję `fetch`.

Obie te zmienne są jednak dostępne przy użyciu operatora zasięgu.

Choć dyrektywa `using` w obrębie funkcji powoduje traktowanie importowanych nazw jako zadeklarowanych poza tą funkcją, wcale nic udostępnia tych nazw pozostałym funkcjom pliku.

Dlatego w funkcji `foo()` nic można wykonać niekwalifikowanego odwołania do nazwy `hill`.

Deklaracje `using` są **bezpieczniejsze** od **dyrektyw** `using`, ponieważ jawnie wymieniają udostępniane lokalnie nazwy.

Jeśli wciągana deklaracją nazwa koliduje z nazwą lokalną, kompilator da o tym znać.

Dyrektywa udostępnia lokalnie wszystkie nazwy przestrzeni nazw, nawet te zupełnie nam niepotrzebne.

Jeśli dojdzie do kolizji nazw, wersja lokalna zakryje wersję z przestrzeni nazw i nie będzie ze strony kompilatora żadnego ostrzeżenia.

Zagnieżdżanie przestrzeni zazw

Deklaracje przestrzeni nazw można zagnieżdżać.

```
namespace elements
{
  namespace hot
  {
    int fire;
  }

  float water;
}
```

Do zmiennej `fire` można się odwołać z użyciem operatora zasięgu.

```
elements::hot::fire = 998;
```

Można też użyć dyrektywy `using` i udostępnić nazwy wewnętrznej przestrzeni nazw.

```
using namespace elements::hot;

fire = 998;
```

W obrębie przestrzeni nazw można też stosować deklaracje i dyrektywy `using`.

```
namespace pieces
{
  using jill::fetch;
  using namespace elements;

  using std::cout;
  using std::cin;
}
```

Przykładowe odwołanie do `jill::fetch`, która jest składową składową przestrzeni nazw `pieces`.

```
std::cin >> pieces::fetch;
```

Ponieważ ta sama nazwa jest też elementem przestrzeni nazw `jill`, można równie dobrze stosować `jill::fetch`.

```
std::out << jill::fetch;
```

Można też (o ile nie ma kolizji) zastosować dyrektywę.

```
using namespace pieces;
cin >> fetch;
```

Dyrektywa `using` ma cechę **przechodniości**. Operator `#` jest przechodnim, kiedy `X # Y` i `Y # Z` implikuje `A # C`.

Przykładem operatora przechodniego jest operator większości `>`. Jeśli `X` jest większe od `Y`, a `Y` od `Z`, to na pewno `X` jest większe od `Z`.

Wynika z tego, że instrukcja:

```
using namespace pieces;
```

daje efekt w postaci lokalnego udostępnienia przestrzeni nazw `elements` za pośrednictwem dyrektywy `using` z przestrzeni nazw `pieces`, a więc jest równoważne sekwencji:

```
using namespace pieces;
using namespace elements;
```

Aliasy przestrzeni nazw

Dla przestrzeni nazw można zdefiniować aliasy.

```
namespace myFavoriteThing
{
  ...
}
```

```
namespace mft = myFavoriteThing;
```

Pozwala to ma upraszczanie stosowania zagnieżdżonych nazw.

```
namespace peh = pieces::elements::hot;

using peh::fire;
```

Nienazwane przestrzenie nazw

Jeżeli w deklaracji przestrzeni nazw zostanie pominięta nazwa, zostanie utworzona **nienazwana przestrzeń nazw**.

```
namespace
{
  int ice;
  int water;
}
```

Kod taki zachowuje się tak, jakby był uzupełniony **dyrektywą** `using`. Nazwy deklarowane w tej przestrzeni nazw znajdą się w potencjalnym zasięgu aż do końca obszaru deklaracyjnego zawierającego nienazwaną przestrzeń nazw.

Nazwy w nienazwanej przestrzeni nazw zachowują się więc jak nazwy globalne.

Skoro przestrzeń nie ma nazwy, nie można zastosować wobec niej jawnie **dyrektywy** `using` ani **deklaracji** `using`, udostępniających zawartość przestrzeni nazw poza nią.

W szczególności nie można stosować nazw z nienazwanej przestrzeni nazw w pliku innym niż plik zawierający deklarację tej przestrzeni nazw.

Takie cechy nienazwanych przestrzeni nazw czynią je alternatywą dla statycznych zmiennych podlegających łączeniu wewnętrznemu.

Standard języka C++ zniechęca do stosowania słowa `static` w przestrzeniach nazw i zasięgu globalnym (zniechęca, co oznacza, żc ich stosowanie jest zgodne z bieżącą wersją standardu, jednak w przyszłości najprawdopodobniej zostanie zabronione).

Dobre zwyczaje stosowania przestrzeni nazw:

- zamiast zewnętrznych zmiennych globalnych należy stosować zmienne w nazwanych przestrzeniach nazw,
- zamiast statycznych zmiennych globalnych należy stosowaźć zmienne w nienazwanych przestrzeniach nazw,
- tworząc bibliotekę funkcji lub klas, należy umieszczać jej elementy w przestrzeni nazw,
- dyrektywę `using` powinno się stosować jedynie w roli tymczasowego środka importu,
- nie powinno się stosować dyrektywy `using` w plikach nagłówkowych (na zachowanie programu wpływ może mieć kolejność włączania plików nagłówkowych); jeśli jest to konieczne, należy stosować dyrektywę `using` w pliku nagłówkowym za wszystkimi dyrektywami `include` preprocesora,
- do nazw z przestrzeni nazw powinno się odwoływać za pośrednictwem deklaracji i operatora zasięgu.